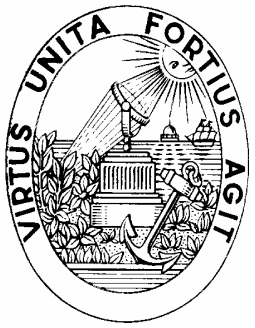
JOÃO CARLOS PASCOAL DE FARIA

Regras Activas Dirigidas pelos Dados para a Manutenção de Restrições de Integridade e Dados Derivados em Aplicações Interactivas de Bases de Dados



FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

1999

Regras Activas Dirigidas pelos Dados para a Manutenção de Restrições de Integridade e Dados Derivados em Aplicações Interactivas de Bases de Dados

João Carlos Pascoal de Faria

Dissertação apresentada à Faculdade de Engenharia da Universidade do Porto para a obtenção do grau de Doutor em Engenharia Electrotécnica e de Computadores

Investigação desenvolvida na Faculdade de Engenharia da Universidade do Porto e no Instituto de Engenharia de Sistemas e Computadores do Porto, sob a orientação do Doutor Raul Fernando de Almeida Moreira Vidal, Professor Associado da Faculdade de Engenharia da Universidade do Porto

Faculdade de Engenharia da Universidade do Porto

Setembro de 1999

O programa de investigação apresentado nesta dissertação foi apoiado pelo programa PRODEP, medida 4.2, acção de formação 12.9.4.

Resumo

Uma regra activa ou gatilho ("trigger") é um terno evento-condição-acção com o seguinte significado: se o evento ocorrer, avaliar a condição e, se esta for verdadeira, executar a acção. O termo "regra activa" surgiu na área das bases de dados activas, onde as regras activas são usadas para a imposição de restrições de integridade estática (nos estados válidos dos dados) ou dinâmica (nas transições de estado válidas), para o cálculo automático de dados derivados (calculados em função doutros), para o controlo e registo de acessos, para a imposição de regras de negócio, etc. Em alguns sistemas, é possível especificar prioridades absolutas ou relativas entre as regras, que condicionam a sua ordem de execução.

Uma aplicação interactiva de bases de dados é um programa de computador que permite a um utilizador visualizar e manipular dados armazenados numa bases de dados através de um interface com o utilizador baseado em formulários, relatórios e gráficos. Na área de desenvolvimento rápido de aplicações (de bases de dados) também se usam gatilhos (mais limitados) para a validação e possível correcção dos dados introduzidos pelo utilizador, para o cálculo automático de dados derivados (por exemplo, campos de formulários ou relatórios calculados em função doutros campos), para o controlo dinâmico de propriedades dos campos, para a extensão e redefinição de comportamentos automáticos embebidos nas ferramentas de desenvolvimento, etc.

Uma regra activa dirigida pelos dados é uma regra activa com eventos implícitos de um tipo restrito - eventos de modificação de dados - que podem ser inferidos a partir da parte de condição e/ou da parte de acção da regra, segundo certos pressupostos. As regras activas dirigidas pelos dados diferem das regras activas com eventos explícitos no modo de definição mas não no modo de execução, que é dirigido por eventos por razões de eficiência e de integração. Fórmulas de cálculo de dados derivados em folhas de cálculo ou em ferramentas de desenvolvimento rápido de aplicações e asserções genéricas em SQL, são exemplos de entidades que podem ser tratadas como regras activas dirigidas pelos dados.

A principal contribuição deste trabalho é a proposta de um tipo (ou modelo) de regras activas dirigidas pelos dados especialmente adaptado para a manutenção de dados derivados (calculados) e restrições de integridade (validações) em aplicações interactivas de bases de dados (nomeadamente em formulários de ecrã e relatórios), mais flexível e melhor integrado do que outros tipos de regras activas dirigidas pelos dados propostos anteriormente tanto na área de bases de dados como na área de desenvolvimento rápido de aplicações. Duas características importantes do modelo de regras proposto são a semântica de ponto fixo e a execução sequencial. A semântica de ponto fixo significa que uma regra deve ser executada (i.e., a condição deve ser avaliada e, se for verdadeira, a acção deve ser executada) sempre que dessa execução possa resultar uma alteração no estado dos dados. A execução sequencial é importante para garantir que as alterações causadas pela execução duma regra não são afectadas pela existência doutras regras. São estabelecidas regras de inferência dos eventos activadores de cada regra e das prioridades entre as regras, por forma a garantir a sua execução segura e eficiente, possivelmente em combinação com outros tipos de regras activas. São determinadas as condições a que um conjunto de regras do tipo proposto deve obedecer para garantir a terminação e o determinismo da sua execução. A natureza especializada destas regras permite obter condições menos conservadoras do que as que são conhecidas para regras activas genéricas. Mostra-se ainda como podem ser incorporadas no modelo de regras proposto as optimizações necessárias para lidar eficientemente com dados complexos, de que se destacam a diferenciação de regras orientadas a conjuntos e o encapsulamento de regras em objectos.

Finalmente, é descrita uma implementação concreta de um sistema de regras activas numa ferramenta de desenvolvimento rápido de aplicações desenvolvida no INESC, suportando regras dirigidas pelos dados e regras dirigidas por eventos, demonstrando as vantagens da abordagem proposta em relação às abordagens seguidas por outras ferramentas, apesar de alguns compromissos considerados na implementação.

Data-driven Active Rules for the Maintenance of Derived Data and Integrity Constraints in Interactive Database Applications

Abstract

An active rule or trigger is an event-condition-action rule with the following meaning: if the event occurs, evaluate the condition and, in case it evaluates to true, execute the action. Active rules have appeared in active database systems, where they are used for the enforcement of static or dynamic integrity constraints, the automatic calculation of derived data, the enforcement of business rules, etc. In some systems, it is possible to specify absolute or relative priorities among rules that condition the order by which multiple triggered rules are executed.

An interactive database application is a computer program that allows a user to query and manipulate data stored in a database through a user interface based on forms, reports, graphics, etc. In rapid application development tools, triggers with more limited capabilities are also used for data validation, for the automatic calculation of derived data, for the dynamic control of properties of data items, to extend or override automatic behavior embedded in those tools, etc.

A data-driven active rule is an active rule with implicit triggering events of a restricted type - data modification events - which can be inferred from the condition and/or the action of a rule, according to some assumptions. Data-driven active rules differ from event-driven active rules (i.e., active rules with explicit events) in the way they are defined, but not in the way they are executed; the execution of data-driven active rules is also event-driven for efficiency and integration reasons. Formulae for the calculation of derived data in spreadsheets or in rapid application development tools and generic assertions in SQL are examples of entities that may be treated as data-driven active rules.

The main contribution of the work reported in this thesis is the proposal of a model of data-driven active rules specially suited for the maintenance of derived data (calculated data) and integrity constraints (data validation) in interactive database applications (namely screen forms and reports), more flexible and with better integration than other types of data-driven active rules previously proposed. Two important characteristics of the proposed model are the fixpoint semantics and the sequential execution. The fixpoint semantics means that a rule should be executed (i.e., the condition should be evaluated and, in case it evaluates to true, the action should be executed) whenever a change in the state of the data may result from its execution. Sequential execution is important to guarantee that the changes caused by the execution of a rule are not affected by the existence of other rules. Inference rules are established to obtain the triggering events and priorities of data-driven active rules, for their efficient and safe execution, possibly in combination with other types of active rules. Conditions on sets of rules are determined that guarantee termination and determinism of rule execution. The specialized nature of the proposed rules allows the determination of less conservative conditions than the ones that are known for generic active rules. It is shown how to incorporate into the proposed model the optimizations required to deal efficiently with complex data, by means of the differentiation of set oriented rules and the encapsulation of rules within objects.

Finally, it is described a concrete implementation of an active rule system integrated in a rapid application development tool developed at INESC, supporting both data-driven and event-driven rules, demonstrating the advantages of the proposed approach by comparison to other tools, in spite of some compromises taken in the implementation.

Règles Actives Commandées par les Données pour la Maintenance des Données Dérivées et des Restrictions d'Intégrité en Applications Interactives de Bases de Données

Resumé

Une règle active ("trigger") est une triade événement-condition-action qui signifie: si l' évènement survient, évaluer la condition et, si celle-ci est vraie, exécuter l'action. Le terme "règle active" est apparu dans le domaine des Bases de Données Actives où les règles actives sont utilisées pour imposer des restrictions d'intégrité statique ou dynamique, pour le calcul automatique de données dérivées, pour imposer de règles de commerce, etc. Dans certains systèmes, il est possible de spécifier des priorités absolues ou relatives entre les règles, qui conditionnent leur ordre d'exécution.

Une application interactive de bases de données est un logiciel qui permet à l'utilisateur de visualiser et de manipuler des données emmaganisées dans un banc de données au moyen d'une interface avec l'utilisateur, basée sur des formulaires, des rapports et des graphiques. Dans le domaine du Développement Rapide d'Applications, on utilise également des règles actives pour la validation des données introduites par l'utilisateur, pour le calcul automatique de données dérivées (par exemple: des champs de formulaires ou de rapports calculés en fonction d'autres champs), pour le contrôle dynamique des propriétés des champs, pour l'extension et la redéfinition de comportements automatiques encapsulés dans les outils de développement, etc.

Une règle active commandée par les données est une règle active ayant des évènements implicites d'un type restreint – évènements de modifications de données – qui peuvent être inférés à partir de la condition et/ou de l'action de la règle, selon certains présupposés. Les règles actives commandées par les données diffèrent des règles actives avec des évènements explicites dans leur mode de définition, mais pas dans leur mode d'exécution qui, lui, est commandé par des évènements, pour des raisons d'efficacité et d'intégration. Des formules de calcul de données dérivées dans des tableurs ou dans des outils de développement rapide d'applications et assertions générales en SQL, sont des exemples d'entités qui peuvent être traitées comme des règles actives commandées par les données.

La contribution principale de ce travail est la proposition d'un type (ou modèle) de règles actives commandées par les données particulièrement adaptées à la maintenance des données dérivées (calculées) et des restrictions d'intégrité (validations), en applications interactives de bases de données (nommément des formulaires d'écran et des rapports), plus flexible et mieux intégré que d'autres types de règles actives commandées par les données précédemment proposés, aussi bien dans le domaine des bases de données que dans le domaine du développement rapide d'applications. Deux importantes caractéristiques du modèle de règles proposé sont la sémantique de point fixe et l'exécution séquentielle. La sémantique de point fixe signifie qu'une règle doit être exécutée (c'est-à-dire que la condition doit être évaluée et, si elle est vraie, l'action doit être mise en exécution) à chaque fois que cette exécution pourra produire une modification de l'état des données. L'exécution séquentielle est importante pour garantir que les modifications entrainées par l'exécution d'une règle ne se trouveront pas affectées par l'existance d'autres règles. On établit des règles d'inférence des évènements déclencheurs de chaque règle et des priorités entre les règles, de façon à garantir leur exécution sûre et efficace, éventuellement en association avec d'autres types de règles actives. On détermine les conditions auxquelles doit obéir un ensemble de règles du type proposé pour garantir l'achèvement et le déterminisme de leur exécution. La nature spécialisée de ces règles permet d'obtenir des conditions moins conservatrices que celles que l'on connaît pour des règles actives générales. On montre encore comment, au modèle de règles proposé, peuvent être incorporées les optimisations nécessaires pour manier efficacement des données complexes, à souligner la différentiation de règles orientées vers des ensembles et l'encapsulage de règles dans des objets.

On décrit une mise en oeuvre concrète d'un système de règles actives dans un outil de développement rapide d'applications développé à l'INESC qui admet des règles commandées par les données et des règles commandées par des évènements, démontrant les avantages de l'approche

proposée par rapport aux approches suivies par d’autres outils, malgré certains compromis adoptés lors de sa mise en oeuvre.

Agradecimentos

O autor gostaria de agradecer ao Prof. Raul Moreira Vidal a sua orientação e incentivo inestimáveis.

Gostaria de agradecer ao INESC-Porto, sobretudo na pessoa do Prof. Mário Jorge Leitão, pelo apoio logístico proporcionado e pelo contexto proporcionado para a ligação deste trabalho à prática.

Agradeço ainda a todos os colegas da FEUP e do INESC-Porto que me apoiaram durante a realização deste trabalho, em particular ao Prof. João Correia Lopes pela ajuda preciosa prestada ao nível do trabalho docente.

Queria também agradecer o apoio e a compreensão da minha esposa Maria e do meu filho Leonardo.

Índice

Lista de Abreviaturas

Português			Inglês		
BD	-	Base de Dados	DB	-	Database
SGBD	-	Sistema de Gestão de Bases de Dados	DBMS	-	Database Management System
SGBDA	-	Sistema de Gestão de Bases de Dados Activo	ADBMS	-	Active Database Management System
SGBDOO		Sistema de Gestão de Bases de Dados Orientado a Objectos	OODBMS		Object Oriented Database Management System
SGIU	-	Sistema de Gestão do Interface com o Utilizador	UIMS	-	User Interface Management System
DRA	-	Desenvolvimento Rápido de Aplicações	RAD	-	Rapid Application Development
PPR	-	Ponto de Processamento de Regras	RPP	-	Rule Processing Point
CFC	-	Componente Fortemente Conexo (de um grafo dirigido)	SCC	-	Strongly Connected Component
ECA	-	(regra) Evento-Condição-Acção	ECA	-	Event-Condition-Action (rule)
EA	.	(regra) Evento-Acção	EA	-	Event-Action (rule)
CA	-	(regra) Condição-Acção	CA	-	Condition-Action (rule)
			ODBC	-	Open Database Connectivity
			SQL	-	Structured Query Language
			UML	-	Unified Modeling Language

1 Introdução

1.1 Motivação

Uma aplicação interactiva de bases de dados é um programa de computador que permite a um utilizador visualizar e manipular dados armazenados numa ou mais bases de dados através de um interface com o utilizador baseado tipicamente em formulários, relatórios e gráficos, colectivamente designados *vistas da aplicação* [M97].

A figura seguinte mostra uma arquitectura típica de uma aplicação interactiva de bases de dados, na visão do autor.

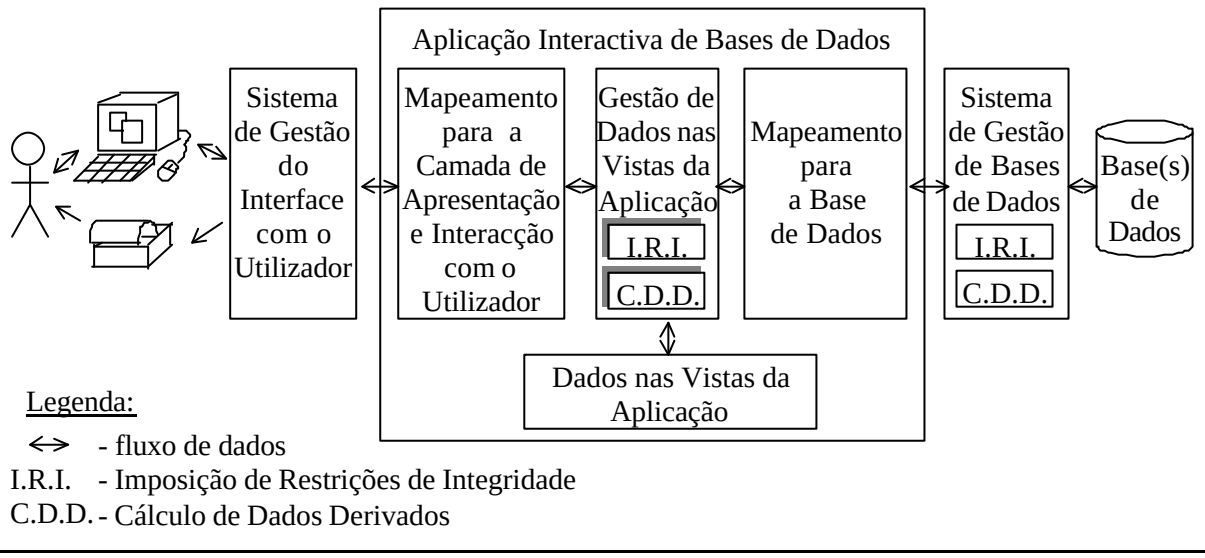


Fig. 1.1 Uma arquitectura típica de uma aplicação interactiva de bases de dados.

Normalmente, a base de dados (BD) é gerida por um sistema de gestão de bases de dados (SGBD) e a aplicação comunica com o SGBD através de uma linguagem de interrogação e manipulação de dados não procedimental, como o SQL ("Structured Query Language"), que oferece independência lógica e física dos dados [D95][R98]. Um SGBD fornece adicionalmente outros serviços valiosos para uma aplicação, de que se destacam:

- gestão de transacções (grupos indivisíveis de operações sobre a base de dados):
- recuperação automática de erros ou falhas (associado à gestão de transacções);
- controlo de concorrência (associado à gestão de transacções);
- imposição automática de restrições de integridade estáticas (restrições nos estados válidos dos dados) ou dinâmicas (restrições nas transições de estado válidas);
- cálculo automático de dados derivados materializados (armazenados);
- manutenção automática de réplicas (em bases de dados distribuídas).

Nos SGBD's actuais, sobretudo relacionais e objecto-relacionais [SM96], algumas restrições de integridade, dados derivados e réplicas são especificados de forma declarativa, e existe um mecanismo genérico - gatilhos ("triggers") ou regras activas - para tratar os casos em que a especificação declarativa não é possível ou satisfatória. Uma regra activa ou gatilho é, na sua forma mais geral, um terno evento-condição-acção com o seguinte significado: se o evento ocorrer, avaliar a condição e, se esta for verdadeira, executar a acção (automaticamente).

A comunicação entre a aplicação e o utilizador processa-se por intermédio de um sistema de gestão do interface com o utilizador (SGIU), tal como o MS-Windows ou o X-Windows, ou um sistema de mais alto nível.

Dentro da aplicação, podem-se distinguir três tarefas principais:

- Gestão de dados nas vistas da aplicação - Os dados visualizados numa vista da aplicação são mantidos em memória temporária da aplicação, em maior ou menor grau (só o registo corrente, todos os registos visíveis, todos os registos seleccionados, etc.). Assim, uma das tarefas da aplicação é a gestão destes dados (inserção, eliminação ou modificação de registos, esvaziamento dos dados, etc.), independentemente da forma como esses dados são visualizados pelo utilizador ou da forma como esses dados estão relacionados com a base de dados.
- Mapeamento para a camada de apresentação e interacção com o utilizador - As acções realizadas pelo utilizador sobre uma vista da aplicação, ao nível de abstracção suportado pelo SGIU (pressionamento de um botão, alteração do conteúdo de uma caixa de texto, etc.), são traduzidas para acções sobre os dados mantidos em memória temporária (eliminação de um registo, actualização de um campo, etc.). Em sentido inverso, os dados residentes em memória temporária são formatados e apresentados ao utilizador em janelas do ecrã ou na impressora.
- Mapeamento para a base de dados (camada de armazenamento persistente) - As alterações acumuladas na memória temporária da aplicação são enviadas para a base de dados. Em sentido inverso, a memória temporária é carregada com dados provenientes da base de dados.

Duas sub-tarefas importantes que estão intimamente relacionadas com a primeira tarefa acima mencionada (nomeadamente com a gestão de alterações nos dados das vistas da aplicação), são a imposição de restrições de integridade e o cálculo de dados derivados nas vistas da aplicação, sem o recurso ao SGBD. Uma aplicação tem de ser capaz de realizar estas sub-tarefas sem o recurso ao SGBD, por várias razões:

- Algumas restrições de integridade definidas na BD podem ser mapeadas para restrições de integridade numa vista da aplicação (sem o envolvimento doutros dados para além dos dados presentes nessa vista). Há toda a conveniência em verificar essas restrições de integridade localmente durante a introdução de dados, antes dos mesmos serem enviados para a BD (e, portanto, sem o envolvimento do SGBD).
- Dados duma vista da aplicação que podem ser calculados exclusivamente em função doutros dados introduzidos pelo utilizador na mesma vista, devem ser calculados localmente pela aplicação, sem o envolvimento do SGBD.
- Dados que podem ser calculados a partir de outros dados seleccionados para uma vista da aplicação (como por exemplo campos de sumário em relatórios), e que não estão sujeitos a restrições de selecção, podem ser calculados do lado da aplicação em vez de serem calculados do lado do SGBD, para não sobrecarregar as comunicações e o processamento de dados do lado da SGBD.
- Quando existe um canal de comunicação rápido entre a aplicação e o SGBD, e são tomadas medidas apropriadas de controlo de concorrência, as restrições de integridade que envolvem outros dados da BD para além dos que estão a ser introduzidos pelo utilizador numa vista da aplicação, podem também ser verificadas imediatamente durante a introdução de dados pelo utilizador. O mesmo se passa com os dados calculados em função doutros dados da BD, para além dos que estão a ser introduzidos pelo utilizador numa vista da aplicação.

- Mesmo numa aplicação interactiva de BD's, existem algumas vistas da aplicação que nada têm a ver com a BD. Restrições de integridade e dados derivados nessas vistas da aplicação têm de ser mantidos pela própria aplicação.

As aplicações interactivas de bases de dados são normalmente desenvolvidas com recurso a ferramentas de desenvolvimento rápido de aplicações (DRA) que incorporam automaticamente na aplicação (através de geradores de código, gestores da aplicação em tempo de execução, componentes de alto nível, etc.) o código necessário para a realização destas tarefas e sub-tarefas, com base em especificações de alto nível. A definição em simultâneo das especificações para as três tarefas acima indicadas (estrutura dos dados em memória, mapeamento para a camada de apresentação e interacção com o utilizador e mapeamento para a base de dados) contribui também de forma significativa para a rapidez de desenvolvimento. Para combinar automação com flexibilidade, muitas ferramentas suportam gatilhos do tipo evento-acção, através dos quais é possível estender ou substituir comportamentos automáticos (embebidos no código incorporado), ou especificar um comportamento onde não está previsto nenhum comportamento automático.

No entanto, as ferramentas actuais sofrem de várias limitações, de que se destacam as que se referem aos tipos de especificações declarativas de restrições de integridade e dados derivados suportados e aos tipos de gatilhos suportados:

- As especificações declarativas de dados derivados têm normalmente características muito semelhantes (na melhor das hipóteses) às que se encontram em folhas de cálculo. Normalmente, não é possível especificar declarativamente (sendo necessário recorrer a gatilhos):
  - itens de dados calculados condicionalmente, i.e., itens de dados que são calculados em algumas circunstâncias e introduzidos pelo utilizador noutras circunstâncias;
  - cálculo de vários itens de dados através da mesma fórmula, para evitar repetir cálculos dispendiosos, tais como os que envolvem o acesso a disco;
  - cálculo do mesmo item de dados por várias fórmulas não contraditórias, que surgem naturalmente quando são suportados os tipos de cálculos referidos nos dois pontos anteriores;
  - fórmulas (cíclicas) para conversão e correcção de dados introduzidos pelo utilizador (estas fórmulas também pode ser vistas como regras de imposição de restrição de integridade);
  - propriedades de itens de dados (tais como a obrigatoriedade e possibilidade de preenchimento) calculadas por fórmulas em função dos valores dos dados.
- As especificações declarativas de restrições de integridade (validações) são muito limitadas. Normalmente, não é possível especificar declarativamente (sendo necessário recorrer a gatilhos):
  - restrições em dados calculados;
  - regras de integridade do tipo condição-acção, em que a acção é a actualização de valores de itens de dados em função dos valores doutros dados para repor a integridade;
  - restrições multi-via ("multi-way constraints"), i.e. restrições com várias escolhas possíveis de itens de dados derivados (calculados) e primitivos (de entrada) (por exemplo, para impor a restrição $c=b-a$, permitir que o utilizador introduza quaisquer dois valores, sendo o terceiro valor calculado).
- A relação entre eventos e gatilhos é normalmente de um para um. Isto é, não podem existir dois gatilhos associados ao mesmo evento, e um gatilho não pode estar associado a mais do que um evento. Esta relação de um para um simplifica o processamentos dos gatilhos, mas tem inconvenientes. Por exemplo, se quisermos ou tivermos que manter através de gatilhos um campo calculado em função dos valores de n campos, podemos ter que escrever n gatilhos, associados aos eventos de modificação do valor de cada um dos n campos. Há assim uma

"dispersão de conhecimento" prejudicial de um ponto de vista de Engenharia de Software. Por outro lado, se quisermos manter através de gatilhos *m* campos calculados que dependem do valor de um dado campo, podemos ter que incluir num único gatilho associado ao evento de modificação do valor desse campo as expressões de cálculo desses *m* campos. Há neste caso uma "mistura de conhecimento" também prejudicial de um ponto de vista de Engenharia de Software. Se for admitida uma relação de muitos para muitos entre eventos e gatilhos, estes problemas desaparecem. A geração de gatilhos (como mecanismo de implementação) a partir de especificações de mais alto nível também fica facilitada.

- Normalmente, a acção de um gatilho é executada imediatamente quando o evento ocorre. Este modo de execução imediata simplifica o processamento de gatilhos mas pode originar ineficiências. Por exemplo, se tivermos um campo calculado da forma $c_1 = c_2 + c_3$, em que c_2 e c_3 são por sua vez calculados em função de um campo c_4, e se todos os cálculos forem processados através de gatilhos, uma única alteração de c_4 pode originar duas actualizações de c_1 (uma quando c_2 é actualizado em função de c_4 e outra quando c_3 é actualizado em função de c_4). Estes problemas seriam evitáveis com mecanismos de execução diferida, em maior ou menor grau.

Para resolver as limitações relacionadas com os tipos de gatilhos suportados, basta transpor para as ferramentas de DRA alguma da tecnologia de regras activas (ou gatilhos) desenvolvida na área de SGBD's comerciais ou de investigação. Já o mesmo não acontece com as limitações que se referem ao suporte de especificações declarativas de restrições de integridade e dados derivados.

1.2 Contribuições

A principal contribuição do trabalho apresentado nesta tese é uma proposta de um modelo de regras activas dirigidas pelos dados especialmente adaptado para a manutenção de dados derivados e restrições de integridade nas vistas das aplicações interactivas de bases de dados, suportando especificações declarativas de restrições de integridade e dados derivados sem as limitações referidas na secção anterior. O modelo proposto é mais flexível e melhor integrado com regras activas dirigidas por eventos do que outros modelos de regras activas dirigidas pelos dados propostos anteriormente na área de bases de dados.

Uma regra activa dirigida pelos dados é uma regra activa com eventos implícitos de um tipo restrito - eventos de modificação de dados - que podem ser inferidos a partir da parte de condição e/ou da parte de acção da regra, segundo certos pressupostos. Recorde-se que, na sua forma genérica, uma regra activa (ou gatilho) é um terno evento-condição-acção. As regras activas dirigidas pelos dados diferem das regras activas com eventos explícitos (dirigidas por eventos) no modo de definição mas não no modo de execução, que é dirigido por eventos por razões de eficiência e de integração. Fórmulas de cálculo de dados derivados e de validação de dados em folhas de cálculo ou em ferramentas de desenvolvimento rápido de aplicações, são exemplos de entidades que podem ser tratadas como regras activas dirigidas pelos dados.

Algumas características distintivas do modelo proposto de regras dirigidas pelos dados são:

- a semântica de ponto fixo, que significa que uma regra deve ser executada (i.e., a condição deve ser avaliada e, se for verdadeira, a acção deve ser executada) sempre que dessa execução possa resultar uma alteração no estado dos dados;
- a ausência de qualquer limitação ao tipo de alterações no estado dos dados que podem ser efectuadas por uma regra, havendo apenas a obrigação de cada regra ser determinística;
- a ausência de qualquer limitação aos conjuntos de regras que se podem definir (por exemplo, a relação entre itens de dados derivados e regras que derivam os seus valores pode ser de muitos para muitos);
- a forte integração entre regras dirigidas pelos dados para a imposição de restrições de integridade e para o cálculo de dados derivados;

- a forte integração com regras dirigidas por eventos, porque as regras dirigidas pelos dados são essencialmente traduzidas para regras dirigidas por eventos (com prioridades) para efeito da sua execução.

Apesar de motivado por uma finalidade concreta, o modelo de regras é apresentado de uma forma muito genérica para potenciar a sua aplicação a outras áreas.

São estabelecidas regras de inferência dos eventos activadores de cada regra e das prioridades entre as regras, por forma a garantir a sua execução segura e eficiente, possivelmente em combinação com regras activas dirigidas por eventos. Esses critérios garantem, em muitos casos, que cada regra é executada no máximo uma vez em qualquer ponto de processamento de regras. Garantem também que as restrições de integridade são verificadas o mais cedo possível.

São determinadas as condições a que um conjunto de regras do tipo proposto deve obedecer para garantir a terminação e o determinismo da sua execução. A natureza especializada destas regras permite obter condições menos conservadoras do que as que são conhecidas para regras activas genéricas.

Mostra-se ainda como podem ser incorporadas no modelo de regras proposto as optimizações necessárias para lidar eficientemente com dados complexos, de que se destacam a diferenciação de regras orientadas a conjuntos e o encapsulamento de regras em objectos.

Uma contribuição prática é a re-engenharia de uma ferramenta de desenvolvimento de aplicações desenvolvida no INESC-Porto, com algoritmos e soluções práticas para a integração de regras dirigidas pelos dados e regras dirigidas por eventos, e casos de uso concretos dessas regras.

Existem também algumas contribuições laterais, com possível interesse noutras áreas:

- um algoritmo eficiente para achar um conjunto de arestas de realimentação ("feedback edge set") de um grafo dirigido com apenas uma aresta de realimentação por cada ciclo do grafo (no anexo 1), com benefícios comprovados para a ordenação de certos conjuntos de regras activas dirigidas pelos dados (comprovação no capítulo 5) e para a ordenação de regras dedutivas (comprovação em [RSS90]);
- a demonstração de algumas propriedades de grafos (no anexo 2) que podem ser usadas na análise do impacto da ordem de aplicação das regras no número de iterações necessárias para atingir um ponto fixo para todas as regras, e também na análise do impacto da ordem de aplicação das equações no número de iterações necessárias na resolução iterativa de um sistema de equações;
- algoritmos de escalonamento multi-critério (com base em prioridades absolutas ou relativas de diferentes forças) que podem ser usados noutros sistemas de regras (no capítulo 8).

1.3 Organização da tese

No capítulo 2, analisa-se o suporte para especificações declarativas de restrições de integridade e dados derivados e o suporte para gatilhos ou regras activas, tanto em ferramentas de DRA como em SGBD's representativos. A análise de ferramentas concretas de DRA permite também concretizar o cenário descrito na figura 1.1.

No capítulo 3 definem-se e justificam-se os aspectos essenciais do modelo de regras activas dirigidas pelos dados proposto nesta tese.

No capítulo 4 estabelecem-se os critérios de activação (eventos activadores) das regras dirigidas pelos dados, por forma a garantir a sua execução segura e eficiente.

No capítulo 5 estabelecem-se os critérios de ordenação das regras dirigidas pelos dados, na forma principalmente de prioridades absolutas ou relativas com diferentes forças, para garantir a sua execução eficiente.

No capítulo 6 determinam-se as condições a que um conjunto de regras deve obedecer para garantir a terminação e o determinismo do processamento de regras, e obtêm-se alguns resultados úteis para a análise do impacto da ordem de execução das regras na velocidade de terminação. As condições e resultados obtidos podem ser usadas na análise de conjuntos de regras concretos e servem também para justificar os critérios introduzidos no capítulo 5.

No capítulo 7 mostra-se como podem ser incorporadas no modelo de regras proposto optimizações necessárias para lidar eficientemente com dados complexos, de que se destacam a diferenciação de regras orientadas a conjuntos e o encapsulamento de regras em objectos.

Nos capítulos 8 e 9 descreve-se uma implementação concreta de um sistema de regras activas numa ferramenta de desenvolvimento rápido de aplicações de bases de dados desenvolvida no INESC-Porto, suportando tanto regras dirigidas pelos dados como regras dirigidas por eventos. São abordados aspectos de integração e de implementação. O componente nuclear do sistema de regras é o motor de regras descrito no capítulo 8. O motor de regras baseia-se num modelo de dados orientado a objectos limitado, sem suporte directo para objectos compostos e herança. São descritos algoritmos que implementam alguns critérios de activação e de ordenação identificados nos capítulos 4 e 5.

Finalmente, no capítulo 10, extraem-se conclusões do trabalho efectuado e identificam-se tópicos para trabalho futuro.

Alguns resultados laterais importantes do trabalho desenvolvido são apresentados nos anexos 1 e 2. No anexo 3 apresenta-se um artigo aceite para publicação nas actas do XIV Simposium Brasileiro de Bases de Dados - SBBD'99 que sintetiza os resultados mais importantes do trabalho desenvolvido.

2 Revisão do estado da arte

Neste capítulo analisa-se o suporte para especificações declarativas de restrições de integridade e dados derivados e o suporte para gatilhos ou regras activas, tanto em ferramentas de desenvolvimento rápido de aplicações como em SGBD's representativos.

2.1 Conceitos básicos

2.1.1 Restrições de integridade

As restrições de integridade dividem-se em estáticas e dinâmicas.

Uma *restrição de integridade estática*, ou simplesmente *restrição estática*, é uma condição a que têm que obedecer os estados válidos de um sistema (base de dados, objecto, conjunto de objectos, etc.). Um exemplo de uma restrição de integridade estática é a que diz que a data de casamento de uma pessoa não pode ser anterior à data de nascimento. Normalmente, admite-se que as restrições de integridade estática sejam violadas em estados transitórios que ocorrem no decurso de operações ou transacções (grupos indivisíveis de operações) de alteração de estado.

Uma *restrição de integridade dinâmica*, ou simplesmente, *restrição dinâmica*, é uma condição a que têm que obedecer as transições de estado válidas. Um exemplo de uma restrição de integridade dinâmica é a que diz que o estado civil de uma pessoa não pode passar de casado para solteiro. Normalmente, admite-se que as restrições de integridade dinâmicas sejam violadas por transições elementares que ocorrem no decurso de uma transacção, bastando que a transacção no seu todo, vista como uma transição de estado de mais alto nível, obedeça a essas restrições.

Podem ainda distinguir-se restrições de integridade "*built-in*" de restrições de integridade *genéricas* [ZCF+97]. As primeiras são especificadas através de construções especiais da linguagem de definição de dados utilizada (e possivelmente impostas ao nível da organização física dos dados), enquanto que as segundas são especificadas através de condições (ou asserções) genéricas. No caso do modelo relacional, as restrições de integridade da chave e de integridade referencial [D95] são restrições de integridade "built-in".

Uma restrição de integridade pode ter associada, implícita ou explicitamente, uma *acção de reparação* ("repair action"), a executar quando a restrição é violada. A acção de reparação implícita mais comum é abortar e desfazer a operação ou transacção em que ocorreu a violação.

As restrições de integridade também são chamadas *regras de integridade* por vários autores [D95][ZCF+97].

Dos vários tipos de restrições de integridade acima referidos, este trabalho incide principalmente sobre restrições de integridade estáticas genéricas, possivelmente com uma acção de reparação associada.

2.1.2 Dados derivados

Um *dado derivado* (por oposição a primitivo) é um dado calculado em função de outros dados.

Um dado derivado pode ser *virtual* (i.e., não armazenado) ou *materializado* (i.e., armazenado). Um dado derivado virtual é sempre calculado a pedido ("on demand"). Um dado derivado materializado tem de ser recalculado sempre que os dados usados no seu cálculo são alterados. O recálculo pode ser efectuado na mesma transacção em que ocorreram as alterações ("on update"), ou pode ser efectuado só quando o valor do dado derivado é necessário ("on demand").

Um exemplo de um dado derivado é um total duma linha duma factura, calculado em função do preço unitário e da quantidade. Exemplos de dados derivados virtuais e materializados em SQL são vistas ("views") e instantâneos ("snapshots"), respectivamente.

A relação existente entre um dado derivado materializado "on update" e os dados usados no seu cálculo pode ser vista como uma restrição de integridade estática (normalmente de igualdade). A acção de reparação associada à restrição de integridade é o recalculo do dado derivado.

Normalmente as restrições de integridade estáticas envolvem apenas dados primitivos, mas também podem ser definidas restrições de integridade que envolvem dados derivados, materializados ou mesmo virtuais. É claro que a imposição eficiente deste género de restrições de integridade é mais complexa.

Dos vários tipos de dados derivados acima referidos, este trabalho incide principalmente sobre dados derivados materializados "on update".

2.1.3 Transacções

Uma transacção é um grupo indivisível de operações com as seguintes propriedades “ACID” [GR93]:

- *Atomicidade* - As alterações de estado agrupadas numa transacção são atómicas: ou tudo acontece ou nada acontece.
- *Consistência* - Uma transação é uma transformação de estado correcta: o estado atingido no final da transacção obedece às restrições de integridade estáticas (supondo que o estado inicial também obedece), e a transacção no seu todo, vista como uma transição de estado, obedece às restrições de integridade dinâmicas.
- *Isolamento* - Mesmo que as transacções sejam executadas concorrentemente, a uma transacção *T* parece que qualquer outra transacção executou totalmente antes ou totalmente depois de *T*.
- *Durabilidade* - Assim que uma transacção termina com sucesso, as alterações produzidas pela transacção sobrevivem a falhas.

Atendendo à propriedade de consistência, é natural que a imposição de restrições de integridade e o cálculo de dados derivados materializados estejam intimamente associados ao processamento de transacções.

2.1.4 Regras activas ou gatilhos

O termo "regra activa" surgiu na área de *sistemas de gestão de bases de dados activos* (SGBDA's) [D88]. Os SGBD's tradicionais são *passivos*, no sentido de que executam apenas as operações solicitadas pelos utilizadores ou programas de aplicação. Em contrapartida, os SGBD's mais recentes são *activos*, no sentido de que realizam eles próprios certas operações automaticamente em resposta a certos eventos que ocorrem ou a certas condições que são satisfeitas [WC96a]. Este *comportamento reactivo* é especificado por regras activas. O termo "gatilho" é mais usado (com o mesmo significado) em SGBDA's comerciais. Formas de gatilhos mais limitadas encontram-se em ferramentas de DRA.

Na sua forma mais geral, uma *regra activa* ("active rule") ou *gatilho* ("trigger") é um terno evento-condição-acção com o seguinte significado: se o evento ocorrer, avaliar a condição e, se a condição for satisfeita, executar a acção [WC96a]. Uma regra com estas três partes também é chamada uma *regra ECA* [WC96a]. Diz-se que a ocorrência do evento *activa* a regra ("triggers the rule"). Executar uma regra é avaliar a condição e, se esta for satisfeita, executar a acção.

O evento ou a condição podem ser suprimidos, mas não os dois simultaneamente. No caso da condição ser suprimida, tem-se uma regra evento-acção, ou *regra EA*. Os gatilhos que se encontram nas ferramentas de DRA são normalmente deste tipo. No caso do evento ser suprimido (o que só é possível em muito poucos sistemas, como por exemplo o sistema Ariel [H92]), tem-se uma regra condição-acção ou *regra CA*, semelhante a uma *regra de produção* (padrão-acção) em Inteligência Artificial.

Os eventos podem ser *simples* (primitivos) ou *compostos*. A forma de composição normalmente suportada é a composição "ou". Os eventos simples mais comunemente suportados são eventos de modificação de dados. Outros eventos suportados por alguns sistemas são eventos de consulta de dados (no sistema POSTGRES [PS96], por exemplo), eventos transaccionais (*commit*, *rollback*, etc.), eventos temporais, eventos do interface com o utilizador (em ferramentas de desenvolvimento rápido de aplicações), etc. Alguns eventos podem ter associados valores que podem ser usados na parte de condição ou de acção. Por exemplo, um evento que se refere à inserção de um registo pode ter associado os valores do registo que está a ser inserido.

As operações suportadas na parte de acção incluem sempre operações de manipulação e consulta de dados e operações para assinalar erros.

A ocorrência do evento, a avaliação da condição e a execução da acção podem estar separados no tempo e em termos transaccionais, o que é especificado por *modos de acoplamento*. Por exemplo, no sistema HiPAC [DBC96] pode-se definir um modo de acoplamento entre o evento e a condição (E-C) e outro modo de acoplamento entre a condição e a acção (C-A). Em qualquer dos casos (E-C ou C-A) o modo de acoplamento pode ser:

- *imediato*: (a avaliação da condição ou a execução da acção, conforme o caso) ocorre imediatamente a seguir, dentro da mesma transacção;
- *diferido*: ocorre no fim da transacção corrente;
- *desacoplado* ("detached"): ocorre numa transacção separada, subdividindo-se em:
  - *desacoplado dependente* - a transacção separada só é lançada se a transacção original for concluída com sucesso;
  - *desacoplado independente* - a transacção separada é lançada mesmo que a transacção original aborte.

Por exemplo, para efeito da imposição de restrições de integridade genéricas, é conveniente um modo de acoplamento diferido entre o evento e a condição, e um modo de acoplamento imediato entre a condição e a acção.

Os sistemas de regras activas diferem no modo de tratar as seguintes situações:

- *Activação múltipla*, isto é, activação de várias regras pelo mesmo evento. Algumas opções possíveis são:
  - executar as várias regras *sequencialmente*, por uma ordem escolhida pelo sistema ou definida pelo utilizador (através de prioridades absolutas ou relativas); esta é a opção mais comum;
  - executar as várias regras *concorrentemente*.
- A*ctivação em cascata*, isto é, activação duma regra causada por um evento gerado por outra regra. Algumas opções possíveis são:
  - executar as regras activadas em cascata *recursivamente* (sem esperar pela conclusão da execução da regra causadora da activação em cascata);
  - executar as várias activadas em cascata *iterativamente* (depois de terminada a execução da regra causadora da activação em cascata).

Uma visão geral actualizada das opções existentes em diferentes sistemas é dada em [PD99].

Algumas aplicações sugeridas para regras activas em SGBD's são:

- imposição de restrições de integridade genéricas;
- cálculo automático de dados derivados materializados;

- imposição de regras do negócio (por exemplo, gerar automaticamente uma ordem de compra quando o stock de um artigo desce abaixo de um certo limite);
- manutenção automática de réplicas (em bases de dados distribuídas);
- implementação de regras de autorização;
- auditoria ("auditing") e registo ("logging") de eventos;
- gestão de fluxos de trabalho (*workflow management*) que envolvem transferência de dados entre tarefas.

As aplicações que mais interessam no contexto deste trabalho são as duas primeiras.

2.2 Ferramentas de desenvolvimento rápido de aplicações de bases de dados

Relativamente a duas ferramentas de desenvolvimento rápido de aplicações de bases de dados representativas, analisam-se de seguida os seguintes aspectos:
- estrutura de dados das vistas da aplicação;
- suporte para especificações declarativas de restrições de integridade (validações) e dados derivados (calculados);
- suporte para gatilhos.

2.2.1 Oracle Developer/2000

O Oracle Developer/2000 (ou simplesmente Developer/2000) é a ferramenta da Oracle Corporation para a construção/geração de aplicações de bases de dados constituídas essencialmente por formulários ("forms"), relatórios ("reports") e gráficos ("graphics"), colectivamente designados vistas da aplicação ("application views") em [M97]. O Developer/2000 engloba um conjunto de produtos que anteriormente apareciam separados: Oracle Forms (sucessor do SQL*Forms), Oracle Reports (sucessor do SQL*Reports) e Oracle Graphics.

O Developer/2000 é uma ferramenta representativa e bem conceituada na sua categoria, embora seja usada principalmente em conexão com o próprio SGBD Oracle. No entanto, as aplicações desenvolvidas com o Developer/2000 podem também funcionar em conexão com outros SGBD's via ODBC ("Open Database Connectivity", norma da Microsoft que uniformiza o interface por SQL embebido em C para diferentes SGBD's).

Descrevem-se de seguida algumas características importantes desta ferramenta no contexto deste trabalho. As características referem-se à versão 2 da ferramenta, conforme descrita principalmente em [M97], com alguns detalhes recolhidos de [OF93].

2.2.1.1 Formulários

Em termos da sua estrutura de dados, um formulário divide-se em um ou mais *blocos de dados* ("data blocks"). Um bloco de dados contém um registo ou um conjunto de registos com a mesma estrutura, isto é, com os mesmos *itens* (anteriormente designados *campos*). Apenas a estrutura dos registos (lista de itens) e o número de registos que o bloco pode conter são especificados em tempo de desenvolvimento.

Um bloco de dados pode ser de um dos seguintes tipos:
- *bloco de dados com tabela-base* ("base-table data block") - corresponde a uma tabela ou vista na base de dados e gere um certo conjunto de registos correspondentes a linhas na tabela ou vista; na versão 2, um bloco deste tipo também pode corresponder a um procedimento armazenado ("stored procedure") na base de dados;
- *bloco de controlo* ("control block") - não corresponde a uma tabela ou vista, e os seus registos não correspondem a linhas de tabelas ou vistas na base de dados; geralmente, um bloco de

controlo tem um único registo, com itens calculados ou de entrada de dados tratados "manualmente".

A função principal de um bloco de dados do primeiro tipo é proporcionar um interface para uma tabela ou, na versão 2, um procedimento armazenado na base de dados, através do qual o utilizador pode interrogar (com "query-by-example") e manipular uma parte da base de dados. O Developer/2000 gera automaticamente os comandos SQL para interrogar e manipular a base de dados com base nas propriedades dos blocos de dados e dos seus itens. As propriedades mais importantes para esse efeito são o nome da tabela, vista ou "stored procedure" correspondente ao bloco, e os nomes das colunas (da tabela ou vista) ou argumentos (da "stored procedure") correspondentes aos itens do bloco com correspondência na base de dados (chamados "database items"). Outras propriedades importantes de um bloco são as cláusulas WHERE e ORDER BY a inserir nas expressões de selecção (SELECT) em SQL, e a lista de operações permitidas (SELECT, INSERT, UPDATE, DELETE). Note-se que a parte de WHERE duma expressão de selecção inclui também condições de junção geradas automaticamente pelo Developer/2000 e, possivelmente, condições de selecção definidas dinamicamente pelo utilizador através das facilidades de "query-by-example".

Normalmente os blocos de dados de um formulário estão relacionados entre si. O Developer/2000 gere automaticamente relações mestre-detalhe ("master-detail") entre blocos de dados de um formulário. Uma relação mestre-detalhe é uma relação entre um bloco de dados mestre ("master data block") e um bloco de dados de detalhe ("detail data block") em que o conjunto de registos do bloco de dados de detalhe (chamados registos de detalhe ou "detail records") está relacionado com o registo corrente do bloco de dados mestre (chamado registo mestre ou "master record"). A propriedade mais importante da relação é a condição de junção (normalmente equi-junção) entre os itens dos dois blocos de dados. Um formulário com blocos relacionados dessa forma é chamado formulário mestre-detalhe ("master-detail form"). Aparentemente, um bloco de dados de detalhe pode ter mais do que um bloco de dados mestre, e pode, por sua vez, assumir o papel de mestre em relação a outros blocos de dados.

Desde a versão 2 do Designer/2000 é possível definir certos itens calculados de uma forma extremamente simples. Um item calculado representa um valor calculado em função doutros itens. Existem dois tipos de itens calculados suportados automaticamente pelo Designer/2000:

- Itens com modo de cálculo *fórmula* ("formula items") - São itens calculados em função doutros itens do mesmo registo, por uma fórmula em PL/SQL. Sempre que ocorre alguma alteração num registo, o que inclui o carregamento ("fetch") do registo em resultado de uma interrogação, o Developer/2000 recalcula os itens calculados por fórmulas.
- Itens com modo de cálculo *sumário* ("summary items") - São itens, normalmente com um único valor, calculados por uma função de sumarização (SUM, MIN, MAX, COUNT, etc.) aplicada ao conjunto de valores doutro item ("summarized item") de um bloco ("summarized block") do mesmo formulário. Quando se instrui o Designer/2000 para carregar ("fetch") para o bloco de dados sumarizado todos os registos resultantes de uma interrogação (com opção "Query All Records"), o cálculo pode ser baseado nos valores residentes no próprio bloco. Caso contrário, o cálculo deve ser baseado directamente na base de dados (aquando duma interrogação), para o que é necessário activar a opção "Precompute Summaries". Posteriormente, o Developer/2000 actualiza automaticamente o valor do item de sumário sempre que é alterado, inserido ou removido um registo no bloco de sumarização.

Estes itens calculados não podem ser "database items". Aparentemente, não se admitem itens calculados por fórmulas de forma cíclica. Em situações mais complexos ("database items" que são ao mesmo tempo calculados, itens calculados só em algumas situações, etc.) é necessário usar gatilhos.

Há diversas validações que são efectuadas automaticamente pelo Designer/2000 a partir de propriedades de alto nível dos itens de dados, tais como:

- a máscara de formatação de um item de texto;
- a obrigatoriedade de preenchimento de um item;

- o tipo de dados e comprimento máximo (em alguns casos) de um item;
- o intervalo de valores permitidos num item;
- a lista de valores permitidos num item.

Outras validações são programadas através de gatilhos.

Um formulário pode ter parâmetros. Um parâmetro de um formulário é uma variável do formulário à qual se atribui um valor através de um argumento passado quando se lança o formulário. Os parâmetros podem ser usados, por exemplo, como variáveis em blocos de código em SQL ou PL/SQL associados ao formulário. A utilização de parâmetros é útil para promover a reutilização. Os parâmetros são só de entrada; parâmetros de saída podem ser simulados através de variáveis globais.

2.2.1.2 Relatórios

No Developer/2000, um relatório é uma forma de apresentação ("display") de dados orientada à página. Enquanto que o propósito de um formulário é o de permitir a gestão interactiva dos dados, o propósito de um relatório é o de formatar um grande conjunto de dados de uma forma facilmente legível, e não o de permitir a gestão dos dados.

Os vários tipos de relatórios suportados pelo Developer/2000 diferem mais quanto à forma como os dados são formatados do que propriamente quanto à estrutura dos dados a formatar.

Os dados a formatar são definidos, em primeiro lugar, por uma ou mais interrogações ("queries"). Cada interrogação é uma expressão de selecção (SELECT) em SQL. A cada interrogação corresponde um conjunto de colunas do relatório com:

- "database columns" - colunas de resultado da interrogação;
- "summary columns" - colunas que acumulam informação de sumário de conjuntos de registos;
- "formula columns" - colunas calculadas em função doutras colunas do relatório através de blocos de código (funções) em PL/SQL;
- "placeholder columns" - colunas preenchidas doutra forma, normalmente através de gatilhos.

Podem-se agrupar os registos (linhas) produzidos por uma interrogação em grupos de registos com igual valor numa coluna ou conjunto de colunas, chamadas colunas de quebra ("break columns"). Normalmente, isso interessa para efeito de sumarização. Este grupos podem, por sua vez, ser agrupados em grupos de nível mais alto. Generalizando, cada registo individual pode ser visto como um grupo de registos do nível mais baixo possível, e o conjunto de todos os registos pode ser visto como um grupo de registos do nível mais alto possível. No modelo de dados do relatório, as colunas são subdivididas em grupos de colunas de níveis correspondentes aos níveis de agrupamento dos registos. Em cada grupo de colunas encontram-se as colunas com igual valor (ou um único valor) dentro de um grupo de registos do nível correspondente. O grupo de colunas de nível mais baixo é chamado o "record group" e os grupos de colunas de níveis intermédios são chamados "break groups". O grupo de colunas de nível mais alto pode ser chamado "query group". Por exemplo, a tabela hierárquica da figura seguinte corresponderia a um relatório hierárquico com três grupos de colunas, definidos no cabeçalho da tabela. As colunas de sumário têm o nome em itálico e a coluna de quebra tem o nome sublinhado.

Q_C_A	**G_Artigo**			**G_Compra_Artigo**		
Despesa Total	**Nome**	***Quantidade Total***	***Preço Total***	**Data**	**Quantidade**	**Preço**
	Resma papel A4			1/1/99	1	600$
		3	1600$	2/1/99	2	1000$
	Pasta Arquivo A4			1/1/99	2	600$
2800$		4	1200$	3/3/99	2	600$

Fig. 2.1 Exemplo de tabela hierárquica com agrupamentos.

São suportados relatórios matriciais ("matrix reports" ou "crosstab reports"), com totais marginais tanto na vertical como na horizontal, resultantes de agrupamentos de registos nas duas direcções, possivelmente com vários níveis de agrupamento em cada direcção. No modelo de dados de um relatório matricial é criado um grupo especial ("cross-product group") correspondente ao produto cartesiano ("cross-product") dos grupos horizontais e verticais, ou seja, correspondente aos cruzamentos de linhas com colunas.

No caso de um relatório com múltiplas interrogações ("multiple query"), podem estabelecer-se ligações ("data links") entre colunas das diferentes interrogações, semelhantes às relações mestre-detalhe que se estabelecem entre diferentes blocos de dados de um formulário.

Tal como um formulário, um relatório pode ter parâmetros. Os parâmetros podem ser definidos pelo sistema ou pelo utilizador (programador). Um relatório com parâmetros pode ter um formulário associado ("parameter form") para a introdução dos valores dos parâmetros pelo utilizador final.

Conforme é observado em [M97], as definições de relatórios e formulários tendem a fundir-se. É possível tornar os relatórios interactivos até um certo grau, e é possível efectuar formatações relativamente sofisticadas nos formulários.

2.2.1.3 Gráficos

Um gráfico ("graphic" ou "chart") é uma representação pictórica de dados, geralmente agregados. Os dados representados num gráfico são normalmente produzidos através duma expressão de selecção (SELECT) em SQL. No caso comum de gráficos que apresentam dados agregados, os dados devem ser produzidos já em forma agregada pela expressão de selecção.

Tal como os formulários e os relatórios, os gráficos podem ter parâmetros.

Podem-se inserir gráficos em relatórios e formulários, para o que é necessário alguma programação através de gatilhos.

2.2.1.4 Gatilhos

A definição de formulários e relatórios é refinada através de gatilhos.

Um gatilho é um bloco de código em PL/SQL (linguagem de programação estruturada baseada em SQL com extensões procedimentais) que se associa a um determinado objecto (formulário, bloco de dados de formulário, item de formulário, ou relatório) para executar quando ocorre um certo evento nesse objecto. O nome do gatilho é o nome do evento (isto é possível porque existe uma relação de zero ou um para um entre gatilhos e eventos em cada objecto). Frequentemente, os termos *gatilho* e *evento* são usados como sinónimos.

Os eventos podem dividir-se em:

- eventos "built-in" - são eventos gerados internamente durante o processamento de formulários e relatórios (processamento de interrogações, navegação, transacções, etc.);
- eventos do teclado ("key triggers") - correspondem ao pressionamento de teclas;
- eventos do interface ("interface event triggers") - correspondem a eventos normalmente gerados pelo sistema de janelas (para além dos eventos do teclado), tais como o pressionamento de um botão, a activação ou desactivação de uma janela, a expiração de um temporizador ("timer"), etc.;
- eventos definidos pelo utilizador - são eventos com nomes definidos pelo utilizador (programador) e sinalizados explicitamente no código PL/SQL.

São os eventos/gatilhos do primeiro tipo que interessam mais aqui. A programação através de gatilhos do primeiro tipo exige a compreensão prévia do modelo de execução das aplicações geradas com o Designer/2000, o qual é descrito através de um conjunto de processos de vários tipos (navegacionais, de interrogação, de validação, de processamento de transacções, etc.). Esses processos explicam os pontos onde são sinalizados os eventos "built-in". Os gatilhos disparados por esses eventos servem para aumentar e, em alguns casos, redefinir, os comportamentos pré-definidos descritos nesses processos.

Um gatilho assinala uma falha através da instrução '`RAISE Form_Trigger_Failure`". O efeito depende do evento em causa, mas normalmente corresponde ao cancelamento do processo que gerou o evento.

São consideradas diversas restrições e medidas para evitar ciclos infinitos devidos a gatilhos, como por exemplo:

- os gatilhos que são executados durante os processos de navegação estão impedidos de invocar rotinas que desencadeiam navegação (chamadas "restricted built-ins");
- as alterações efectuadas pelos gatilhos que são executados durante os processos de validação são consideradas imediatamente validadas e não desencadeiam novas execuções de gatilhos.

Indicam-se de seguida alguns eventos/gatilhos úteis para a manutenção de restrições de integridade e itens calculados em formulários:

- `Post-Query` - Dispara após o carregamento ("fetch") de um registo para um bloco durante o processamento de uma interrogação. Pode ser usado para popular itens de controlo ou itens noutros blocos, calcular valores acumulados, calcular estatísticas sobre os registos retornados por uma interrogação, etc. Também pode ser usado para rejeitar registos seleccionados (com "`RAISE Form_Trigger_Failure`"). Pode-se utilizar o comando SELECT mas não os comandos de manipulação de dados (INSERT, UPDATE e DELETE) de SQL. Note-se que os casos de utilização deste gatilho diminuíram significativamente com o suporte automático para itens calculados introduzido na versão 2 do Designer/2000.

- `When-Validate-Item` - Dispara durante o processo de validação de um item alterado, após as validações automáticas terem sido concluídas com sucesso. Pode ser usado para efectuar validações ao nível de um item para além das que já são efectuadas automaticamente. Eventuais erros são assinalados através de mensagens para o utilizador e de "`RAISE Form_Trigger_Failure`". Também pode ser usado para propagar alterações ocorridas num item para outros itens (itens calculados). Pode-se utilizar o comando SELECT mas não os comandos de manipulação de dados (INSERT, UPDATE e DELETE) de SQL.

- `When-Validate-Record` - Dispara durante o processo de validação de um registo alterado num bloco, após as validações automáticas terem sido concluídas com sucesso. É normalmente usado para efectuar validações que envolvem mais do que um item. De resto é semelhante ao anterior. O seguinte exemplo é retirado de [OF93] (o caracter "`:`" precede os nomes de blocos ou itens de formulários):

```
BEGIN
  IF :Experiment.Start_Date > :Experiment.End_Data THEN
    MESSAGE('Your date range ends before it starts!');
    RAISE Form_Trigger_Failure;
  END IF;
END;
```

- `When-New-Form-Instance` - Dispara no arranque de um formulário. Pode ser usado para inicializar variáveis globais e efectuar outras inicializações.
- `When-Create-Record` - Dispara quando é criado um novo registo num bloco, o que acontece, por exemplo, quando o operador pressiona uma tecla apropriada. É normalmente usado para definir valores por omissão complexos para alguns itens (já que valores por omissão simples podem ser definidos como propriedades estáticas dos itens). Também pode ser usado para manter incrementalmente itens de sumário ou rejeitar a inserção de registos (com "`RAISE Form_Trigger_Failure`").
- `When-Remove-Record` - Dispara quando o operador ou a aplicação limpa ("clears") ou elimina um registo. Pode ser usado para manter incrementalmente itens de sumário.

Normalmente os gatilhos que permitem redefinir comportamentos pré-definidos têm o nome começado por "`On`". Por exemplo, o gatilho `On-Insert` dispara onde o comportamento normal seria a execução de um comando INSERT de SQL, durante o processo de propagação de alterações ("post") para a base de dados. Pode-se associar a este evento um bloco de código em PL/SQL a executar em vez do comando INSERT de SQL.

Muitos dos gatilhos não se aplicam durante o modo de introdução do critério de interrogação ("enter query mode"), mas apenas em modo "normal".

Pode-se associar a um objecto de mais alto nível (formulário ou bloco de dados) um gatilho disparado por um evento que de facto ocorre em objectos de mais baixo nível (blocos de dados ou itens). Nesse caso, o gatilho é herdado pelos objectos de mais baixo nível, podendo também ser redefinido (ou aumentado) para alguns objectos de mais baixo nível (por gatilhos com o mesmo nome associados a esses objectos).

A título de conclusão, note-se que a manutenção de itens calculados através de gatilhos pode ser relativamente trabalhosa e sujeita a erro, porque pode ser necessário repetir o mesmo bloco de código em vários gatilhos (por exemplo, vários gatilhos do tipo `When-Validate-Item`), e cada gatilho pode ter de tratar de vários itens calculados. As facilidades de manutenção automática de certos itens calculados introduzidas na versão 2 constituem um progresso significativo, mas há casos que têm de continuar a ser tratados através de gatilhos.

2.2.2 Microsoft Access

O Microsoft Access (ou simplesmente Access) é um produto comercial muito popular que permite desenvolver rapidamente aplicações de bases de dados constituídas principalmente por formulários e relatórios (colectivamente chamados interfaces). O Microsoft Access também inclui um motor de base de dados próprio (o Microsoft Jet Database Engine), que não impede a conexão a outros SGBD's, e facilidades interactivas de definição, interrogação e manipulação directa da base de dados que não são abordadas aqui. A descrição que se segue é baseada na consulta da documentação "online" do Microsoft Access 97. Recentemente foi lançado o Microsoft Access 2000, que não afecta significativamente os aspectos tratados aqui.

2.2.2.1 Formulários

Os formulários gerados com o Microsoft Access têm estrutura, propriedades e comportamento muito semelhantes aos dos formulários gerados com o Oracle Developer/2000.

Os formulários do Microsoft Access não têm a estrutura de blocos dos formulários do Designer/2000. Em contrapartida, um formulário ("main form") pode ter sub-formulários ("subforms"). Um sub-formulário (de 1º nível) pode, por sua vez, ter os seus próprios sub-formulários (de 2º nível), mas estes já não podem ter sub-formulários (de 3º nível). Normalmente, a relação que existe entre um formulário e um sub-formulário é uma relação mestre-detalhe ou pai-filho, semelhante à que se verifica entre os blocos de dados de um formulário no Designer/2000. Um formulário com sub-formulários é chamado um formulário hierárquico. Um formulário sem sub-formulários é chamado um formulário plano ("flat").

Normalmente, um formulário é um interface para um conjunto de dados previamente definido - uma tabela ou interrogação ("query") - que constitui a fonte de registos ("record source") do formulário. Um formulário deste tipo é chamado um "bound form". Os controlos de um formulário (objectos gráficos como "text boxes", "check boxes", "combo boxes", etc.) que estão ligados a campos ("fields") da tabela ou interrogação em que se baseia o formulário são chamados "bound controls". Os campos da tabela ou interrogação em que se baseia o formulário são chamados campos do formulário. O facto de um formulário ser baseado numa interrogação em vez duma tabela não impede o seu uso para alteração de dados, mesmo quando a interrogação em que se baseia o formulário envolve mais do que uma tabela. Em particular, está grandemente automatizada a manipulação, através de um único formulário, de registos de uma tabela base que são estendidos com campos de tabelas referenciadas por chaves estrangeiras da tabela base (também chamadas "lookup tables").

Diferentemente do que se passa no Designer/2000, em cada formulário (plano ou hierárquico) só pode estar alterado um registo de cada vez. Quando se elimina um registo num formulário, o Access pede confirmação e elimina imediatamente o(s) registo(s) correspondente(s) na base de dados. Quando se insere ou altera um registo e se passa a outro registo, o Access salva automaticamente o registo inserido ou alterado. Este modo de funcionamento simplifica grandemente a validação de dados, mas é limitativo em algumas aplicações.

Podem-se definir facilmente *controlos calculados*, isto é, controlos que mostram o resultado de uma expressão em vez do valor dum dado armazenado. Basta indicar a expressão. Os tipos mais importantes de controlos calculados são:

- Controlos do nível do registo (com um valor variável de registo para registo), cujo valor é calculado em cada registo em função dos valores doutros controlos ou campos do formulário (no mesmo registo), incluindo controlos e campos calculados;
- Controlos do nível do formulário (com um único valor para todo o formulário), cujo valor é calculado por uma função de agregação (SUM, MAX, MIN, etc.) aplicada a uma expressão em campos ou controlos do formulário, incluindo campos calculados mas não controlos calculados (esta limitação não é grave uma vez que se podem usar campos calculados).

Um campo calculado é um campo de uma interrogação que é calculado por uma expressão em função doutros campos.

No caso de formulários com sub-formulários, é possível, com algumas restrições, definir controlos calculados num formulário em função de controlos doutro formulário.

Cálculos mais complexos podem ser efectuados através de procedimentos e macros executados em resposta à ocorrência de eventos.

Podem-se definir *regras de validação* de dados ao nível de cada controlo de um formulário. Uma regra de validação é uma condição acompanhada opcionalmente de uma mensagem de erro ("validation text") a afixar no caso da condição não ser satisfeita. A regra de validação de um controlo pode referir outros controlos. A regra de validação de um controlo é aplicada apenas quando o valor do controlo é alterado pelo utilizador. Portanto, não tem efeito em controlos calculados nem funciona como filtro de selecção. Validações mais complexas podem ser efectuadas através de procedimentos e macros associados a eventos. Um formulário herda automaticamente as regras de validação definidas ao nível das tabelas (aplicáveis a um registo de cada vez) e dos campos das tabelas em que se baseia o formulário. A regra de validação de registos só é testada imediatamente antes de salvar um registo.

Note-se que a validação de dados fica grandemente facilitada pelo facto de só poder estar alterado um registo de cada vez, o que pode ser limitativo em algumas aplicações.

Existem outras propriedades que restringem os dados que podem ser introduzidos e as alterações que podem ser efectuadas pelo utilizador (a considerar antes de usar regras de validação): máscaras de formatação, lista de operações permitidas, etc.

2.2.2.2 Relatórios

No Microsoft Access existe uma forte integração entre formulários e relatórios. É possível salvar um formulário como um relatório e é possível imprimir um formulário. Como seria de esperar, os relatórios não podem ser usados para entrada de dados.

Tal como no Designer/2000, é possível agrupar os registos de forma hierárquica, para tornar o relatório mais legível ou para calcular sub-totais por grupo. O agrupamento dos registos no Microsoft Access está estreitamente ligado com o critério de ordenação dos registos e com o "layout" do relatório. Os registos podem ser ordenados por uma lista de até 10 campos ou expressões, por valores ascendentes ou descendentes (só no caso de campos). Cada campo ou expressão dessa lista, no todo ou em parte (primeiros n caracteres de uma string, ano de uma data, etc.), pode ser usado para definir um nível de agrupamento de registos. Para clarificar ideias, e usando a analogia com SQL, suponhamos que o critério de ordenação é da forma ORDER BY c_1, c_2, ..., c_n. Então, é possível definir n agrupamentos de registos da forma GROUP BY c_1, ..., c_{i-1}, $f_i(c_i)$, para valores de i (nível de agrupamento) desde 1 até n. A função $f_i(c_i)$ dá um valor em função de c_i. No "layout" do relatório podem definir-se secções de cabeçalho e rodapé para cada nível de agrupamento. No máximo, podem existir as seguintes secções, pela ordem indicada: cabeçalho do relatório, cabeçalho de página, cabeçalho de grupo de nível 1, ..., cabeçalho de grupo de nível n, detalhe, rodapé, rodapé de grupo de nível n, ..., rodapé de grupo de nível 1, rodapé de página, rodapé do relatório. Um cabeçalho/rodapé de grupo de nível i só é impresso no primeiro/último registo de cada grupo de registos de nível i.

Podem-se colocar controlos calculados em qualquer secção de um relatório. Um controlo de uma secção (cabeçalho ou rodapé) de nível i pode ser calculado por uma função de agregação aplicada a campos de nível $> i$, ou por uma expressão simples (sem funções de agregação) envolvendo controlos ou campos de nível i ou nível $< i$ (por exemplo, para apresentar uma percentagem em relação ao total de um grupo). Isto também se aplica em relação a controlos colocados no cabeçalho ou rodapé do relatório (caso em que o nível é 0) e a controlos colocados na secção de detalhe (caso em que o nível é $n+1$).

Um relatório pode incluir sub-relatórios e sub-formulários, até dois níveis.

São suportados relatórios matriciais ("crosstab reports"), mas é necessário alguma programação em muitos casos (nomeadamente no caso, relativamente comum, em que os nomes de colunas provêm dos dados seleccionados).

2.2.2.3 Gráficos

O Microsoft Access suporta vários tipos de gráficos ("charts"). Um gráfico apresenta dados produzidos por uma expressão de selecção em SQL. Os gráficos podem ser inseridos em formulários e relatórios.

2.2.2.4 Eventos

O comportamento de formulários e relatórios é refinado através da definição de procedimentos ("event procedures") ou macros que são executados automaticamente em reposta à ocorrência de eventos. A cada evento num objecto (relatório, formulário, controlo, etc.) só pode estar associada uma macro ou um procedimento. Uma macro é uma sequência de acções simples (possivelmente com argumentos) ou de pares condição-acção. Os procedimentos são escritos em Visual Basic for

Applications (VBA). Os procedimentos são mais flexíveis do que as macros mas exigem mais conhecimentos de programação.

Existem vários tipos de eventos: eventos do rato, eventos do teclado, eventos de janelas, eventos relacionados com o foco de entrada, eventos relacionados com a manipulação de dados, com a abertura e fecho de formulários, etc.

Os eventos mais importantes para efeito de validação de dados são:

- `BeforeUpdate` (ao nível de um formulário) - Ocorre imediatamente antes de serem salvas as alterações num registo modificado ou adicionado. Tipicamente é usado para efectuar validações que envolvem mais do que um controlo, e que não interessa efectuar imediatamente aquando da alteração do conteúdo de cada controlo. Nas macros, a acção que cancela o evento é `CancelEvent`.
- `BeforeUpdate` (ao nível de um controlo de um formulário) - Ocorre quando se altera o conteúdo de um controlo e se move o foco de entrada para outro controlo ou é necessário salvar o registo (nesse momento, o texto que está a ser editado pelo utilizador é efectivamente introduzido no controlo, actualizando o seu valor). Não ocorre em controlos calculados. Tipicamente é usado para efectuar validações que não podem ser expressas pela regra de validação do controlo, tais como validações que envolvem controlos doutros formulários, validações com mensagens de erro variáveis ou validações que envolvem funções definidas pelo utilizador (em VBA). Nas macros, a acção que cancela o evento é `CancelEvent`.
- `OnDelete` (ao nível de um formulário) - Ocorre quando um registo é eliminado, mas antes da eliminação ser confirmada e efectivamente executada.

Para efeito de definição do valor de um controlo ("unbound") ou propriedade ("locked", "enabled", etc.) em função de valores doutros controlos, podem-se usar os eventos anteriores, ou eventos que ocorrem imediatamente a seguir aos anteriores (como `AfterUpdate` e `AfterDelConfirm`). A acção a usar nas macros é `SetValue`, com dois argumentos: o nome do controlo ou propriedade do controlo ou do formulário, e a expressão que determina o valor a atribuir. Note-se que as alterações causadas com `SetValue` não causam a ocorrência (pelo menos imediata) do evento `BeforeUpdate` ou `AfterUpdate`, pelo que não é fácil propagar cálculos em cascata.

2.2.3 Outras ferramentas

As ferramentas anteriores destinam-se especificamente a desenvolver aplicações de bases de dados.

Uma abordagem alternativa para o desenvolvimento rápido de aplicações de bases de dados consiste na utilização de linguagens de programação de uso genérico, orientadas a objectos, que incorporam facilidades de construção de interfaces ("interface builders") - caso do Visual Basic, Delphi (baseada em Object-Pascal), Power Builder (baseado em C++), Java, etc. Nesses ambientes, é possível desenvolver rapidamente aplicações de bases de dados com base em componentes de interface especialmente concebidas para esse efeito (como "data base grids", etc.). No entanto, os automatismos para a manutenção de restrições de integridade e dados derivados não são superiores aos encontrados nas duas ferramentas analisadas.

2.3 Restrições de integridade e gatilhos ou regras activas em sistemas de bases de dados

2.3.1 Restrições de integridade declarativas em SQL-92

A norma SQL-92 define vários tipos de restrições de integridade ("constraints") que se podem classificar nas seguintes categorias (segundo [CW96]):

- restrições em tabelas;
- restrições de integridade referencial;
- asserções genéricas.

Estas restrições de integridade são chamadas declarativas, para distinguir das restrições de integridade mantidas de forma mais procedimental através de gatilhos (só disponíveis em SQL3 ou em SGBD's comerciais).

2.3.1.1 Restrições em tabelas

Como parte da instrução `CREATE TABLE`, podem especificar-se restrições dos seguintes tipos:

- `NOT NULL` - a coluna especificada não aceita valores nulos;
- `UNIQUE` - a coluna ou combinação de colunas especificada constitui uma chave ("unique key") da tabela, no sentido de que não podem existir duas linhas com a mesma combinação de valores nessas colunas, exceptuando linhas com valores nulos em todas as colunas da chave;
- `CHECK <condição>` - o valor da condição especificada tem de ser verdadeiro ou desconhecido (devido a valores nulos) em qualquer linha da tabela;
- `PRIMARY KEY` - a coluna ou combinação de colunas especificada constitui uma chave primária da tabela; uma chave primária combina as propriedades de `UNIQUE` e `NOT NULL`; cada tabela só pode ter uma chave primária; por omissão, as referências para uma tabela são para a chave primária (ver adiante).

As restrições numa tabela são verificadas imediatamente nas linhas inseridas pelo comando `INSERT` e nas linhas actualizadas pelo comando `UPDATE` (dependendo das colunas restringidas e das colunas actualizadas). Se uma restrição for violada, é desfeito ("rolled back") o comando SQL que causou a violação e é sinalizado um erro.

2.3.1.2 Restrições de integridade referencial

Uma restrição de integridade referencial especifica que determinadas colunas de uma tabela *referenciante* só podem tomar combinações de valores existentes em determinadas colunas de uma tabela *referenciada*. A restrição de integridade referencial pode ser declarada na instrução `CREATE TABLE` da tabela referenciante com a seguinte sintaxe:

```
<foreign key clause> ::=
  FOREIGN KEY (<referencing columns>)
  REFERENCES <table name> [<referenced columns>]
  [<foreign key actins>]

<foreign key action> ::= <event> <action>

<event> ::= ON UPDATE | ON DELETE

<action> ::= CASCADE | SET DEFAULT | SET NULL | NO ACTION
```

Usa-se a seguinte notação: `[ ]` - opcional, `|` - alternativa, `{}` - agrupamento.

Por omissão, as colunas referenciadas são a chave primária da tabela referenciada

A integridade referencial é verificada no fim de cada instrução SQL capaz de a violar, da seguinte forma:

- São proibidas inserções (`INSERT`) e actualizações (`UPDATE`) na tabela referenciante (a tabela em que é definida a restrição) que violem a integridade referencial (a instrução SQL que as provocou é desfeita).

- As eliminações (`DELETE`) e actualizações (`UPDATE`) na tabela referenciada que violam a integridade referencial são tratadas de acordo com a acção indicada:

  `CASCADE` - propaga as eliminações e actualizações para a tabela referenciante;

  `SET NULL` - coloca valores nulos nas colunas referenciantes;

  `SET DEFAULT` - coloca valores por omissão nas colunas referenciantes;

  `NO ACTION` - proíbe essas actualizações ou eliminações (a instrução SQL é desfeita).

2.3.1.3 Asserções genéricas

Uma asserção é uma restrição de integridade genérica envolvendo uma ou mais tabelas. A sintaxe da definição de uma asserção em SQL-92 é:

```
<SQL-92 assertion> ::=
  CREATE ASSERTION <constraint name>
  CHECK (<condition>)
  [<constraint evaluation>]

<constraint evaluation> ::=
  [NOT] DEFERRABLE  [INITIALLY {DEFERRED | IMMEDIATE}]
```

A condição (restrição propriamente dita) é um predicado arbitrário em SQL.

Uma asserção pode ser avaliada/verificada num dos seguintes modos:

- `IMMEDIATE` - A restrição é verificada imediatamente após a execução de qualquer operação SQL capaz de a violar. Se a condição não se verificar, apenas essa operação SQL é desfeita ("rolled back").
- `DEFERRED` - A restrição é verificada imediatamente antes de uma transacção em que foram executadas operações SQL capazes de violar a restrição fazer "commit". Se a condição não se verificar, a transacção é desfeita ("rolled back"). Este modo é útil para permitir que restrições sobre várias tabelas e/ou linhas de tabelas sejam violadas transitoriamente no decurso das transacções.

Por comparação, as restrições de tabelas são verificadas sempre em modo imediato.

Se uma asserção for especificada como `DEFERRABLE`, então pode-se escolher outro modo de avaliação na transacção corrente, em vez do modo especificado com `INITIALLY`, com a instrução:

```
SET CONSTRAINTS <constraint names> {IMMEDIATE | DEFERRED}
```

Note-se que poucos SGBD's comerciais suportam asserções genéricas, muito provavelmente devido à dificuldade em as verificar eficientemente. Por exemplo, o Oracle8 não as suporta.

2.3.1.4 Evoluções em SQL3

A norma SQL3 prevê os mesmos tipos de restrições de integridade da norma SQL-92, com pequenas alterações.

Em SQL3 (e em alguns SGBD's comerciais como o Oracle8), a cláusula prevista para especificar o modo de avaliação das asserções (`IMMEDIATE` ou `DEFERRED`) passa a estar disponível em todos os tipos de restrições.

Foram também introduzidas na definição das asserções algumas das opções existentes na definição de gatilhos, nomeadamente:

- é possível especificar asserções do nível do tuplo (`FOR EACH ROW`);
- é possível especificar as operações de manipulação de dados que desencadeiam a verificação imediata da asserção.

2.3.2 Gatilhos em SQL3

Uma das novidades principais da norma SQL3 em relação à norma SQL-92 é a introdução de gatilhos. A descrição que se segue é baseada em [KMC98].

A sintaxe da definição de um gatilho é:

```
<SQL3 trigger definition> ::=
  CREATE TRIGGER <trigger name>
  <trigger activation time>
  <trigger event> ON <table name>
  [REFERENCING <old or new values alias list>]
  [<trigger granularity>]
  [WHEN (<condition>)]
  <trigger action>

<trigger activation time> ::= BEFORE | AFTER

<trigger granularity> ::= FOR EACH ROW | FOR EACH STATEMENT

<trigger event> ::= INSERT | DELETE | UPDATE [OF <column name
list>]

<old or new values alias> ::=
   OLD [AS] <identifier>
  |NEW [AS] <identifier>
  |OLD_TABLE [AS] <identifier>
  |NEW_TABLE [AS] <identifier>

<trigger action> ::=
  <SQL procedure statement>
  | <SQL procedure atomic block>
```

Os gatilhos seguem o paradigma das regras ECA:

- evento (obrigatório) - é a execução duma operação de manipulação de dados específica (INSERT, DELETE ou UPDATE) numa tabela específica;
- condição (opcional) - é um predicado SQL arbitrário, podendo conter interrogações (SELECT);
- acção (obrigatória) - é uma instrução ou bloco atómico de instruções SQL estendido com facilidades procedimentos, excluindo instruções relativas a esquemas, conexões, sessões e transacções (não se pode invocar ROLLBACK mas pode-se assinalar uma excepção).

É possível associar vários gatilhos à mesma operação e tabela.
Se um gatilho lançar uma excepção, a operação activadora é desfeita ("rolled back"), mas a transacção em que ela ocorre não é abortada.

O tempo de activação do gatilho indica o momento em que o gatilho é activado/executado em relação à execução da operação activadora:

- `BEFORE` - o gatilho é activado/executado antes da execução da operação activadora ou, mais precisamente, depois de ter sido calculado o efeito da operação activadora (conjunto de linhas afectadas, com valores antigos e novos sempre que aplicáveis), mas antes desse efeito ser reflectido no estado corrente da base de dados;
- `AFTER` - o gatilho é activado/executado depois da execução da operação activadora, num momento em que o estado corrente da base de dados já reflecte o efeito da operação activadora.

A granularidade do gatilho indica o número de vezes que o gatilho é activado/executado em resposta a uma execução da operação activadora:

- FOR EACH ROW - o gatilho é activado/executado uma vez para cada linha afectada (inserida, eliminada ou actualizada) pela execução da operação activadora;
- FOR EACH STATEMENT (granularidade por omissão) - o gatilho é activado/executado uma única vez pela execução da operação activadora (mesmo que nenhuma linha seja afectada).

A cláusula REFERENCES permite associar nomes às seguintes tabelas ou linhas de tabelas de transição, para as poder usar (só para leitura) na parte de condição ou de acção:

- OLD_TABLE - tabela lógica acessível em gatilhos FOR EACH STATEMENT quando a operação activadora é DELETE ou UPDATE, contendo os valores antigos de todos as linhas afectadas (eliminadas ou actualizadas) pela execução da operação activadora;
- NEW_TABLE - tabela lógica acessível em gatilhos FOR EACH STATEMENT quando a operação activadora é INSERT ou UPDATE, contendo os novos valores de todas as linhas afectadas (inseridas ou actualizadas) pela execução da operação activadora;
- OLD - linha (tuplo) lógica acessível em gatilhos FOR EACH ROW quando a operação activadora é DELETE ou UPDATE, contendo os valores antigos duma linha afectada (eliminada ou actualizada) pela execução da operação activadora;
- NEW - linha (tuplo) lógica acessível em gatilhos FOR EACH ROW quando a operação activadora é INSERT ou UPDATE, contendo os novos valores duma linha afectada (inserida ou actualizada) pela execução da operação activadora.

Estas tabelas e linhas de tabelas de transição estão disponíveis tanto em gatilhos AFTER como em gatilhos BEFORE, e mantêm-se inalteradas durante a execução de cada gatilho.

O modelo de execução dos gatilhos (em particular, o momento preciso da execução dos gatilhos AFTER e BEFORE e o significado preciso das tabelas de transição) complica-se grandemente ao lidar com os seguintes aspectos:

- combinação com a verificação das restrições de integridade declarativas - a estratégia seguida consiste genericamente em incluir já nas tabelas de transição as alterações causadas pelas acções de reposição de integridade referencial (CASCADE, SET NULL, SET DEFAULT), e em aplicar os gatilhos AFTER só depois de todas as restrições de integridade declarativas (com verificação imediata) terem sido tratadas;
- activação múltipla, isto é, vários gatilhos activados pela mesma operação na mesma tabela no mesmo momento (AFTER ou BEFORE) - a estratégia seguida consiste genericamente em executar os vários gatilhos sequencialmente pela ordem em que foram definidos, seja qual for a sua granularidade; um gatilho FOR EACH ROW é executado para todas as linhas afectadas antes de se passar ao gatilho seguinte (pelo menos conceptualmente); um gatilho não vê (nas suas tabelas de transição) as transições causadas pelos gatilhos executados antes dele, apesar do estado corrente da base de dados as reflectir;
- activação em cascata, isto é, um gatilho executar uma operação que activa outros gatilhos (ou mesmo outra instância do mesmo gatilho) - a estratégia seguida consiste genericamente em executar imediatamente (recursivamente) os gatilhos activados em cascata e só depois concluir a execução do gatilho activador; um gatilho não vê (nas suas tabelas de transição) as transições causadas pelas operações por ele invocadas ou pelos gatilhos activados em cascata, apesar do estado corrente da base de dados as reflectir.

A lógica para um gatilho não ver as transições posteriores à activação do gatilho é que essas transições podem ser processadas imediatamente por outras instâncias do mesmo gatilho. De qualquer forma, é de esperar que o desfasamento potencial entre as transições e o estado corrente da base de dados vistos por um gatilho seja uma fonte de problemas! Por outro lado, as estratégias acima descritas podem impedir optimizações muito importantes (com por exemplo a execução de todos os gatilhos FOR EACH ROW para cada linha afectada antes de passar à linha seguinte).

Apesar da ordem de execução dos gatilhos estar bem definida, o processamento dos gatilhos pode ser não determinístico, porque não é definida a ordem pela qual um gatilho `FOR EACH ROW` processa as linhas afectadas. Em consequência disso, o estado final da base de dados, após a execução duma operação de manipulação de dados que desencadeia a execução de gatilhos, pode depender não só do estado inicial da base de dados, das restrições de integridade declarativas existentes, dos gatilhos existentes e da ordem de definição dos gatilhos, mas também da forma como o SGBD está implementado.

Exemplo

Seja uma base de dados com o seguinte esquema relacional (com chaves primárias sublinhadas e referências para outras tabelas indicadas por setas):

```
Artigo(codigo, nome, quant_inicial, quant_actual)
Movimento(cidigo_artigo→Artigo, data, quant_entrada)
```

Para manter actualizada a quantidade actual de cada artigo em face dos movimentos realizados (em que `quant_entrada` pode ser positivo ou negativo), podem-se definir os seguintes gatilhos:

```
CREATE TRIGGER propagaInsMov
AFTER INSERT ON Movimento
REFERENCING NEW AS NewMov
FOR EACH ROW
  UPDATE Artigo
    SET quant_actual = quant_actual + NewMov.quant_entrada
    WHERE codigo = NewMov.codigo_artigo;

CREATE TRIGGER propagaUpdMov
AFTER UPDATE OF codigo_artigo, quant_entrada ON Movimento
REFERENCING NEW AS NewMov, OLD AS OldMov
FOR EACH ROW
BEGIN ATOMIC
  UPDATE Artigo
    SET quant_actual = quant_actual - OldMov.quant_entrada
    WHERE codigo = OldMov.codigo_artigo;
  UPDATE Artigo
    SET quant_actual = quant_actual + NewMov.quant_entrada
    WHERE codigo = NewMov.codigo_artigo;
END

CREATE TRIGGER propagaDelMov
AFTER DELETE ON Movimento
REFERENCING OLD AS OldMov
FOR EACH ROW
  UPDATE Artigo
    SET quant_actual = quant_actual - OldMov.quant_entrada
    WHERE codigo = OldMov.codigo_artigo;
```

2.3.3 Gatilhos em Oracle8

A descrição que se segue é baseada na consulta da documentação "online" do Oracle8 Server SQL Reference Release 8.0. A sintaxe da definição de um gatilho é indicada na figura seguinte.

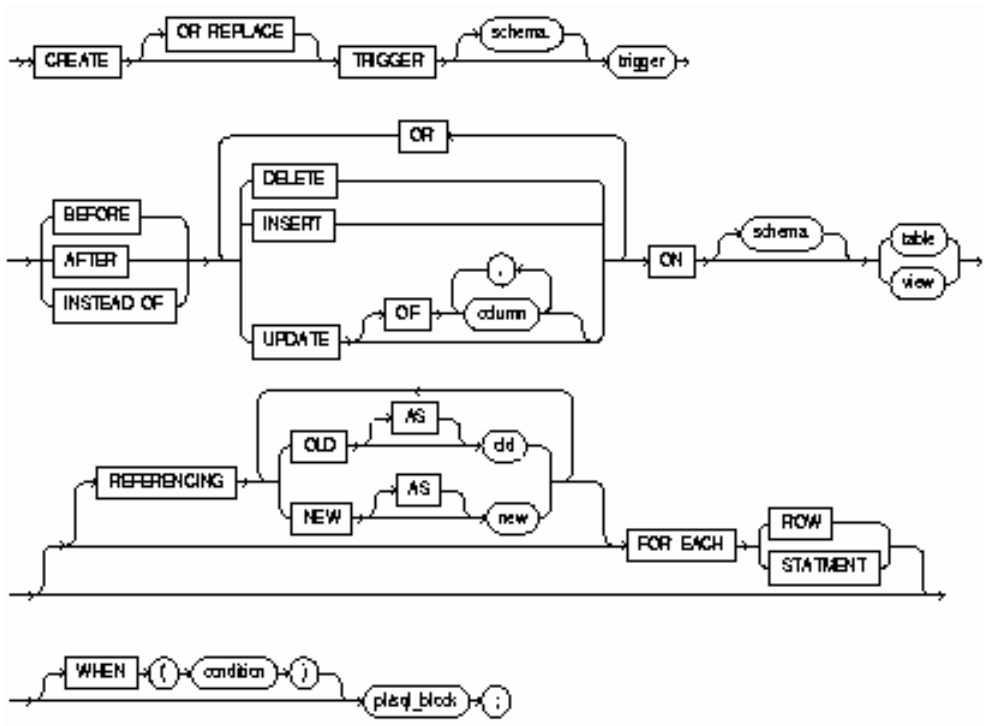


Fig. 2.2 Sintaxe da definição de um gatilho no Oracle8.

As principais diferenças em relação a SQL3 são:

- Um gatilho pode monitorar várias operações sobre a mesma tabela.
- Quando um gatilho monitora mais que uma operação, predicados especiais `INSERTING`, `DELETING`, `UPDATING` e `UPDATING(column_name)` podem ser usados na parte de acção (bloco PL/SQL) para determinar que operação está a decorrer e, no caso de `UPDATE`, que colunas estão a ser actualizadas.
- Todos os gatilhos `FOR EACH ROW` são aplicados em cada linha afectada, antes de passar à linha seguinte (o que simplifica grandemente e torna mais eficiente o processamento de gatilhos deste tipo).
- Não existem as tabelas de transição `NEW_TABLE` e `OLD_TABLE`, pelo que os gatilhos `FOR EACH STATEMENT` só vêm o estado corrente da base de dados (o que simplifica grandemente o processamento de gatilhos deste tipo, mas também diminui a sua utilidade).
- A parte de condição só é suportada em gatilhos `FOR EACH ROW`, e restringe-se a um predicado simples na linha afectada, não podendo incluir interrogações (`SELECT`).
- Os nomes `OLD` e `NEW` podem ser usados (precedidos de `":"` na parte de acção) sem necessidade de criar sinónimos com a cláusula `REFERENCING`.
- No caso de activação múltipla, a ordem de execução dos gatilhos é indeterminada. Por essa razão, é aconselhada a fusão de gatilhos que possam ser activadas ao mesmo tempo num único gatilho.
- No caso de activação em cascata, o nível de recursividade do processamento de gatilhos está limitado por `MAX_OPEN_CURSORS` (pelo menos era assim no Oracle7 segundo [CW96]).
- Os gatilhos podem ser desactivados (“disabled”) e reactivados (“enabled”).
- Podem-se criar gatilhos do tipo `INSTEAD OF` em vistas, os quais servem para implementar de uma forma definida pelo utilizador (através do bloco de código em PL/SQL) as operações de manipulação de vistas. Estes gatilhos são sempre `FOR EACH ROW` e, no caso de `UPDATE`,

não se podem indicar as colunas. Os gatilhos `AFTER` e `BEFORE` só são aplicáveis a tabelas e os gatilhos `INSTEAD OF` só são aplicáveis a vistas.

Exemplo

Para obter o mesmo efeito dos gatilhos em SQL3 apresentados anteriormente, basta escrever o seguinte gatilho em Oracle8:

```
CREATE TRIGGER propagaMov
AFTER INSERT OR DELETE OR UPDATE OF codigo_artigo,
quant_entrada
ON Movimento
FOR EACH ROW
BEGIN
  IF INSERTING OR UPDATING THEN
    UPDATE Artigo
      SET quant_actual = quant_actual + :NEW.quant_entrada
      WHERE codigo = :NEW.codigo_artigo;
  END IF;
  IF DELETING OR UPDATING THEN
    UPDATE Artigo
      SET quant_actual = quant_actual - :OLD.quant_entrada
      WHERE codigo = :OLD.codigo_artigo;
  END IF;
END;
```

2.3.4 Regras activas no sistema Starbust

O sistema Starbust é um protótipo de SGBD extensível desenvolvido na IBM, e que tem influenciado a evolução do SGBD comercial DB2, também da IBM. O sistema Starbust suporta regras activas orientadas a conjuntos com a seguinte sintaxe [WF90][W96]:

```
<Starbust rule> ::=
  create rule <rule name> on <table name>
  when <triggering-operations>
  [if <condition>]
  then <action-list>
  [precedes <rule-list>]
  [follows <rule-list>]

<triggering-operation> ::=
  inserted | deleted | updated[(<column-names>)]

<condition> ::= <SQL select statement>

<action> ::=  <any database operation>
```

Uma regra em Starbust tem as três partes essenciais duma regra ECA:

- evento - a execução duma das operações de manipulação de dados especificadas na parte de "**`when`**" (`inserted` corresponde à operação `insert`, etc.) sobre a tabela indicada na parte de "**`on`**";
- condição (opcional) - instrução de selecção em SQL indicada na cláusula "**`if`**" que se considera falsa quando o resultado da selecção é vazio;
- acção - a execução de uma sequência de comandos SQL especificados na cláusula "**`then`**", incluindo possivelmente comandos de definição de dados e "`rollback`".

Além disso, é possível especificar prioridades (ou precedências) relativas entre as regras através das cláusulas "**precedes**" e "**follows**". Se duas regras `r1` e `r2` se encontram activadas num dado momento, e "`r1 precedes r2`" ou "`r2 follows r1`", a regra `r1` é executada primeiro. No caso de nenhuma das regras ter precedência sobre a outra, directa ou indirectamente (por transitividade), tem precedência a regra criada há mais tempo (como acontece em SQL3). As prioridades relativas são implicitamente transitivas e não podem ter ciclos.

Tanto na parte de condição como na parte de acção duma regra é possível usar *tabelas de transição*. Dada uma regra sobre uma tabela *T*, as tabelas de transição são:

- **`inserted`** - tabela lógica acessível numa regra com operação activadora `inserted`, contendo os tuplos que foram inseridos em *T* causando a activação da regra;
- **`deleted`** - tabela lógica acessível numa regra com operação activadora `deleted`, contendo os tuplos que foram eliminados de *T* causando a activação da regra;
- **`new-updated`** - tabela lógica acessível numa regra com operação activadora `updated`, contendo os novos valores dos tuplos actualizados em *T* causando a activação da regra;
- **`old-updated`** - tabela lógica acessível numa regra com operação activadora `updated`, contendo os valores antigos dos tuplos actualizados em *T* causando a activação da regra (a cada tuplo em `old-updated` corresponde um tuplo em `new-updated` com o mesmo identificador interno).

Estas tabelas de transição desempenham papel semelhante (mas não idêntico, como veremos a seguir) às tabelas `OLD_TABLE` e `NEW_TABLE` de SQL3.

As regras activadas são executadas em modo *diferido* no fim da transacção (imediatamente antes de "commit"), ou quando é explicitamente invocado o processamento de regras com o comando "`process rules`". Em qualquer dos casos, as regras activadas são executadas sequencialmente, incluindo no mesmo ciclo as regras activadas em cascata (regras activadas em resultado da execução doutras regras). Em cada iteração, é escolhida (do conjunto de regras activadas que inclui as regras activadas em cascata) uma regra para execução de acordo com a prioridade relativa e a ordem de criação das regras, conforme se explicou acima. A execução duma regra consiste na avaliação da condição seguida da execução da acção no caso do condição ser verdadeira. Se uma regra invoca `rollback`, o processamento de regras é interrompido e a transacção é desfeita ("rolled back").

Quando uma regra é executada, as suas tabelas de transição descrevem o *efeito líquido* das alterações (inserções, eliminações e actualizações de tuplos) ocorridas desde o início da transacção (no caso da 1ª execução da regra nessa transacção) ou desde o início da última execução da mesma regra (no caso contrário), até ao início da execução corrente da regra. Para produzir o efeito líquido, as alterações (transições) elementares são combinadas da seguinte forma:

- se um tuplo é inserido e depois actualizado, o efeito líquido é a inserção do tuplo actualizado;
- se um tuplo é actualizado e depois eliminado, o efeito líquido é a eliminação do tuplo original;
- se um tuplo é actualizado mais do que uma vez, o efeito líquido é uma única actualização do valor original para o valor mais recente;
- se um tuplo é inserido e depois eliminado, não aparece no efeito líquido.

Desta forma, garante-se que cada tuplo alterado (identificado pelo seu identificador interno) só aparece numa das tabelas de transição (excepto no caso de `old-updated` e `new-updated`, em que aparece em ambas as tabelas) e, dentro dessa tabela de transição, só aparece uma vez. Dado que, em caso de execução repetida duma regra, as tabelas de transição só contabilizam as alterações ocorridas desde o início da última execução, a mesma alteração (inserção, eliminação ou actualização de um tuplo) não é "vista" mais do que uma vez pela mesma regra.

Os próprios eventos activadores são considerados em termos líquidos. Uma regra só se considera activada se pelo menos uma das suas tabelas de transição não for vazia (razão pela qual as operações

activadoras têm os nomes das tabelas de transição). Assim, uma regra pode ser desactivada ("untriggered") por eventos de alteração de dados.

Na opinião do autor, as principais vantagens do sistema de regras do Starbust são:

- a simplicidade conceptual do modelo de execução;
- a flexibilidade permitida pela execução diferida (sem se proibir o processamento de regras a seguir à execução de qualquer instrução SQL) e pela orientação a conjuntos.

A simplicidade resulta principalmente do facto das regras activadas em cascata serem executadas sequencialmente (iterativamente) em conjunto com as regras activadas directamente, e não recursivamente como acontece em SQL3. O problema do potencial desfasamento entre as tabelas de transição e o estado corrente da base de dados vistos por um gatilho em SQL3 não se põe com as regras do sistema Starbust. Uma vez que em cada momento só está uma regra em execução no sistema Starbust, é muito mais fácil analisar o comportamento dum conjunto das regras no sistema Starbust, do que é analisar o comportamento dum conjunto de gatilhos em SQL3 (devido à execução recursiva).

Em contrapartida, na maioria dos casos não será possível atingir com regras no sistema Starbust a mesma eficiência que se consegue com gatilhos `FOR EACH ROW` em Oracle8 (em SQL3, a conclusão não é tão clara).

Exemplo [W96]

Considere-se o seguinte esquema relacional:

```
dept(dept-no,mgr-no)
emp(emp-no,name,salary,dept-no)
```

A regra seguinte elimina os departamentos e os empregados de departamentos cujos "managers" foram eliminados:

```
create rule cascade on emp
when deleted
then delete from emp
     where dept-no in
        (select dept-no from dept
         where mgr-no in (select emp-no from deleted));
     delete from dept
     where mgr-no in (select emp-no from deleted)
```

Note-se que a regra se activa a si própria quando um empregado eliminado é "manager" doutro departamento, obrigando a nova execução da regra (após a execução anterior terminar).

2.3.5 Regras dirigidas pelos dados no projecto PARDES

No projecto PARDES [E93][EGS93] e suas evoluções [GE95][G95], é proposta uma abordagem de mais alto nível (comparativamente às abordagens baseadas em regras ECA ou EA) para a manutenção de "regras dirigidas pelos dados".

Uma *regra dirigida pelos dados* ("data-driven rule") é definida como uma regra que é activada pela modificação de itens de dados numa base de dados. São apontadas três aplicações principais de regras dirigidas pelos dados:

- Manutenção de dados derivados armazenados - Itens de dados derivados podem ser mantidos através de regras (*de derivação*) dirigidas pelos dados que calculam os valores de itens de dados derivados em função dos valores doutros itens de dados, e são activadas pela modificação de qualquer item de dados que participa na derivação.

- Imposição de restrições de integridade - Uma restrição de integridade pode ser mantida por uma regra (*de restrição*) dirigida pelos dados que desencadeia uma acção de reparação apropriada (abortar, derivar o valor de itens de dados, etc.) quando uma condição é violada, e é activada pela modificação de qualquer item de dados que participa na condição.
- Invocação de operações externas - Em muitas situações interessa invocar automaticamente uma operação externa que consulta a base de dados (sem a alterar) sempre que certos itens de dados consultados por essa operação são modificados. Essa activação pode ser implementada por uma regra dirigida pelos dados que invoca a operação, e é activada pela modificação de qualquer item de dados consultado pela operação (desde que esses itens sejam declarados por exemplo como argumentos da invocação).

Em contrapartida, uma *regra dirigida por eventos* ("event driven rule") é definida como uma regra que é activada pela ocorrência de eventos (como acontece com as regras ECA ou EA), em vez de ser activada pela modificação de itens de dados. Esta distinção entre regras dirigidas por eventos e regras dirigidas pelos dados não é clara, porque a modificação de um item de dados também constitui um evento. Para clarificar ideias, parece não haver contradição com a documentação consultada se considerarmos que uma regra dirigida pelos dados é uma regra de um dos três tipos acima mencionados (de derivação, de restrição ou de invocação duma operação externa) que é activada *implicitamente* pela modificação de itens de dados.

São apontadas diversas deficiências na manutenção de restrições de integridade e dados derivados através de regras ECA ou EA, relacionadas essencialmente com o baixo nível de abstracção das regras ECA ou ECA.

Para ultrapassar essas deficiências, é proposta a seguinte abordagem de mais alto nível:

- Um item de dados derivado (também chamado PDI - "persistent derived information") é especificado por uma expressão de atribuição (também chamada derivação) directamente executável que calcula o valor do item em função dos valores doutros itens. Essa expressão de atribuição é tratada como uma regra de derivação dirigida pelos dados. A expressão é executada se for modificado o valor de qualquer item de dados que participa no lado direito da expressão de atribuição. Os itens de dados derivados não podem ser actualizados directamente pelo utilizador. Opcionalmente, podem existir do lado direito da atribuição diferentes expressões para diferentes condições mutuamente exclusivas (não é claro se as condições têm de cobrir todos os casos, ou se, pelo contrário, o valor do atributo derivado pode não estar definido em alguns casos).
- Uma restrição de integridade é especificada por uma expressão booleana (a restrição propriamente dita), que é tratada como uma regra de restrição dirigida pelos dados. A expressão é avaliada se for modificado o valor de qualquer item de dados que participa na expressão; se o resultado for falso, a transacção é abortada. Em [G95], pode-se indicar um "estabilizador" ("stabilizer") na forma duma expressão de atribuição destinada a repor a integridade, a qual é executada em vez de abortar a transacção.

Exemplos de derivações e restrições que são tratadas como regras dirigidas pelos dados são indicados na figura seguinte, em que se subentende um modelo de dados orientado a objectos.

Classes and properties:

Class = Customer
Properties = Customer-id, Status, Credit, Debit (*derived*), Balance (*derived*), Max-Order-Price

Class = Product
Properties = Product-id, Price, Manufactured-Quantity, Avaliable (*derived*)

Class = Order

Attributes = Order-id, Customer-id (*reference to Customer*), Order-Price (*derived*),
Items: *set of* (Product-id (*reference to Product*), Quantity)

Derivations (computational invariantes [E93] / derive operations [G95]):

D1: Order-Price := sum(Quantity * Price)

D2: Debit := sum(Order-Price) * 0.8 when Status = preferred
sum(Order-Price) otherwise

D3: Balance := Credit - Debit

D4: Available := Manufactured-Quantity - sum(Quantity)

Constraints (logical invariantes [E93] / enforce operations [G95]):

C1: Available ≥ 0

C2: Balance > -2 * average(Order-Price)

C3: Credit ≥ 0
stabilizer = (Credit := 0)

C4: Order-Price ≤ Max-Order-Price

Fig. 2.3 Exemplo de definição do esquema de uma base de dados com derivações e restrições tratadas como regras dirigidas pelos dados no projecto PARDES (adaptado de [E93] e [G95]).

Nas expressões das restrições e derivações há detalhes de "matching" intra-classe e inter-classe que podem ser inferidos pelo sistema, de acordo com os seguintes princípios:

- "Matching" intra-classe - Duas propriedades da mesma classe que participam numa regra referem-se implicitamente à mesma instância da classe. Por exemplo, a derivação D3 da figura anterior pode ler-se:
  $\forall c \in$ Customer, Balance(c) := Credit(c) - Debit(c)
- "Matching" inter-classe - Duas propriedades de classes diferentes que participam numa regra referem-se implicitamente a instâncias das respectivas classes relacionadas pelas referências expressas no esquema. Por exemplo, a restrição C4 da figura anterior pode ler-se:
  $\forall o \in$ Order, $\forall c \in$ Customer, Customer-id(o) = Customer-id(c) $\Rightarrow$
  Order-Price(o) ≤ Max-Order-Price(c)
- No caso de operações de agregação, o "matching" é efectuado localmente à operação de agregação (este princípio não é apresentado explicitamente mas pode inferir-se dos exemplos consultados). Por exemplo, a primeira linha da derivação D2 da figura anterior ler-se-á:
  $\forall c \in$ Customer, Debit(c) :=
  sum(Order-Price(o) | $o \in$ Order $\wedge$ Customer-id(c) = Customer-id(o)) * 0.8
  where Status(c) = preferred

São consideradas as seguintes premissas (restrições) importantes:

- unicidade - cada item de dados derivado aparece do lado esquerdo de exactamente uma derivação (não é claro se isto também se aplica aos estabilizadores);
- não reflexividade - o valor de um item de dados não pode ser derivado directa ou indirectamente a partir do seu próprio valor (no entanto, pode ser derivado a partir do seu valor antigo existente no início da transacção).

Estas restrições destinam-se a garantir a terminação e o determinismo da execução das regras (na realidade, não bastam estas restrições para garantir o determinismo, porque a ordem de verificação das

regras de restrição pode afectar o estado final alcançado). No entanto, é bom notar que estas restrições impedem uma utilização mais alargada de regras dirigidas pelos dados.

Em [G95], a restrição de unicidade é relaxada, o que está relacionado com o facto de um item de dados poder ter múltiplos valores alternativos; no entanto, quando se lê o valor dum item de dados, tem de se seleccionar um dos valores alternativos, o que explica que a garantia de determinismo seja abandonada em [G95].

A execução das regras baseia-se num grafo de dependências que traduz as dependências dirigidas pelos dados existentes entre os vários itens de dados. Esse grafo é usado para determinar que regras é necessário executar, e por que ordem, em face das modificações ocorridas na base de dados. A ordem de execução escolhida é uma ordem topológica do grafo de dependências (a ordem topológica existe porque a premissa de não reflexividade garante que o grafo de dependências é acíclico).

É sugerido que o sistema deve optimizar a execução de cada regra (nomeadamente quando estão envolvidas operações de agregação ou propriedades de várias classes) em face das modificações elementares ocorridas na base de dados, mas não é apresentado um método sistemático para o fazer. Por exemplo, o sistema deveria ser capaz de inferir as seguintes acções relativamente à derivação D1 da figura anterior:

- quando é inserido um item *i* relativo a um produto *p* e uma ordem *o*:
  adicionar *Quantity(i)*Price(p)* a *Order-Price(o)*;
- quando é eliminado um item *i* relativo a um produto *p* e uma ordem *o*:
  subtrair *Quantity(i)*Price(p)* a *Order-Price(o)*;
- quando é modificada a quantidade dum item *i* relativo a um produto *p* e uma ordem *o*:
  adicionar *(Quantity(i)-old(Quantity(i))) * Price(p)* a *Order-Price(o)*;
- quando é modificado o preço dum produto *p*,
  para cada item *i* relativo a esse produto,
  adicionar *Quantity(i)* (Price(p)-old(Price(p)))* a *Order-Price(o)*.

2.3.6 Restrições e gatilhos no sistema Ode

O Ode [GJ91][GJ96] é um SGBD orientado a objectos baseado no paradigma de objectos de C++ [S97]. O primeiro interface para o sistema Ode é a linguagem de programação de bases de dados O++, que é uma extensão da linguagem C++ com facilidades de definição e manipulação de objectos persistentes, gestão de transacções, e associação de restrições e gatilhos a objectos.

2.3.6.1 Objectos persistentes e transacções

Na linguagem O++, um objecto persistente é acedido por um apontador com prefixo `persistent` que armazena o identificador único e imutável (identidade) do objecto persistente, como em:

```
persistent Emp *e;    // Emp é o nome da classe
```

Contrariamente aos objectos voláteis, os objectos persistentes são alocados numa área de armazenamento persistente, onde continuam a existir mesmo depois do programa que os criou terminar.

As transacções em O++ têm a forma dum bloco com prefixo `trans`, como em:

```
trans { .... }
```

As transacções são abortadas com a instrução `tabort`. Abortar uma transacção implica desfazer ("rollback") as alterações ocorridas nos objectos persistentes. As transacções proporcionam as propriedades de atomicidade, isolamento e durabilidade relativamente aos objectos persistentes (e só a esses).

São suportadas restrições "hard" e "soft" e ainda construções especiais para suportar integridade referencial e integridade relacional (de associações). Estas construções espaciais não são abordadas aqui.

2.3.6.2 Restrições "hard"

As restrições "hard" são especificadas na secção "`constraint:`" da definição de uma classe, da seguinte forma:

```
constraint:
        constraint1 : handler1
        ...
        constraintn : handlern
```

$constraint_i$ é uma expressão Booleana que referencia componentes da classe e $handler_i$ é uma instrução que é executada quando a restrição é violada (i.e., quando $constraint_i$ tem valor `false`). Se $handler_i$ não for especificado, a transacção é abortada com `tabort` quando a restrição é violada.

As restrições "hard" são verificadas no fim das chamadas de construtores e de funções-membro públicas da classe não-constantes (mas não no fim das chamadas de destrutores). Uma função-membro constante é uma função-membro com qualificador "`const`", que está por isso proibida de alterar o estado do objecto. Embora o acesso directa a dados-membro públicos duma classe não seja proibida, é da responsabilidade do programador assegurar que esses acessos não violam as restrições, pois não é efectuada nenhuma verificação das restrições para esses acessos.

Se uma restrição associada a um objecto não é satisfeita e não existe nenhum "handler" associado a essa restrição, a transacção é abortada. Se existe um "handler" associado à restrição, o "handler" é executado e a restrição é reavaliada. Se a restrição continuar a não ser satisfeita, a transacção é abortada.

Conforme foi referido acima, a granularidade da verificação das restrições "hard" é ao nível da função-membro pública. Isto tem duas vantagens importantes: os objectos estão sempre num estado consistente (excepto possível durante uma operação de actualização) e a implementação da verificação das restrições é simplificada.

Exemplo

```
class supplier {
    ...
    Name state;
    ...
  constraint:
    state == Name("NY") || state == Name(""):
      printf("Invalid Supplier State \n");
};
```

Se a restrição for violada, é afixada a mensagem indicada. Uma vez que o "handler" não repõe a consistência, a transacção é abortada. Acções não relacionadas como a base de dados, como a instrução `printf`, não são desfeitas.

2.3.6.3 Restrições "soft"

As restrições "soft" são especificadas na secção "`soft constraint:`" da definição de uma classe, com a mesma sintaxe das restrições "soft".

As restrições "soft" são verificadas apenas no fim da transacção (portanto, em modo diferido). Uma restrição "soft" será normalmente usada quando envolve outros objectos.

Exemplo

```
class person {
     ...
     persistent person *spouse;
     ...
   soft constraint:
     (spouse == NULL) || (this == spouse->spouse):
};
```

Note-se que se esta restrição fosse especificada como "hard", nunca seria possível registar o casamento ou o divórcio de duas pessoas.

2.3.6.4 Conceito de restrição intra-objecto e restrição inter-objecto

No sistema Ode são introduzidas as noções de restrição intra-objecto e restrição inter-objecto.

Uma restrição é chamada *intra-objecto* se:
1. está associada a um objecto específico, e
2. a condição associada à restrição é avaliada apenas quando o objecto é actualizado.

Caso contrário, a restrição é chamada *inter-objecto*. Uma restrição intra-objecto pode referenciar outros objectos tanto na parte de condição como na parte de acção ("handler"). No entanto, a restrição não é verificada quando os objectos referenciados são actualizados.

As restrições suportadas pelo sistema Ode são intra-objecto, principalmente por razões de obediência à filosofia de orientação a objectos.

Mesmo assim, é reconhecida a importância das restrições inter-objecto. Uma solução sugerida para tratar restrições inter-objecto é a conversão de cada restrição inter-objecto para várias restrições intra-objecto equivalentes (automaticamente ou pelo programador).

Exemplo

O seguinte pedaço de código especifica que o salário de um empregado não pode exceder o salário do respectivo chefe ("manager"). Trata-se de uma restrição inter-objecto envolvendo dois objectos: um empregado e um chefe. Esta restrição inter-objecto é implementada por duas restrições intra-objecto, uma associada ao empregado e outra ao chefe.

```
class manager;

class employee {
     ...
     persistent manager *mgr;
     float sal;
   public:
     ...
     float salary() const { return sal; }
 constraint:
     sal < mgr->salary;
};

class manager: public employee {
```

```
    persistent employee *emp<MAX>; // <MAX> - conjunto
    int sal_greater_than_all_employees();
    ...
constraint:
    sal_greater_than_all_employees();
};

int manager::sal_greater_than_all_employees()
{
  persistent employee *e;
  for (e in emp) // for - iterador
    if (e->salaray() > salary())
      return 0;
  return 1;
}
```

2.3.6.5 Gatilhos

O sistema Ode suporta gatilhos associados a objectos. A terminologia e sintaxe seguida é diferente da habitual. Para simplificar a apresentação, são omitidos alguns pormenores julgados menos importantes no contexto deste trabalho, nomeadamente gatilhos temporizados.

Os gatilhos são especificados na secção "`trigger:`" da definição de uma classe, com a seguinte sintaxe:

```
trigger:
  ...
  [perpetual] T(parameter-decl): event-expression ==> [mode] action
  ...
```

T é o nome do gatilho. Um gatilho pode ter parâmetros que são indicados quando o gatilho é activado (ver adiante) e podem ser usados na parte de acção. A acção é uma instrução em C++.

O modo pode ser:

- `independent` (valor por omissão) - a acção é executada imediatamente numa transacção separada independente;
- `immediate` - a acção é executada imediatamente na mesma transacção (como nas restrições "hard");
- `deferred` - a acção é executada no fim da mesma transacção (como nas restrições "soft").

Um gatilho pode ser activado ("activated"), disparado ("fired") e desactivado ("deactivated"). Quando um objecto é criado todos os seus gatilhos se encontram desactivados. Um gatilho *T* é activado explicitamente para um objecto específico através de uma chamada do tipo:

object-id `->` *T*`(`*arguments*`)`

São permitidas várias activações do mesmo objecto (possivelmente com argumentos diferentes).

Um gatilho activado dispara automaticamente quando o seu predicado (indicado por "*event-expression*") se torna verdadeiro; nesse momento, a sua acção é escalonada para execução, conforme o modo.

Existem dois tipos de gatilhos: "*once-only*" e "*perpetual*". Um gatilho "once-only" é automaticamente desactivado cada vez que dispara, enquanto que um gatilho "perpetual" mantém-se activado (apenas é necessário activá-lo explicitamente a primeira vez).

A linguagem de eventos do sistema Ode é muito rica. Os eventos básicos são do tipo `before` *op* ou `after` *op* em que *op* pode ser:

- `create` - criação dum objecto (chamada do construtor)
- `delete` - destruição dum objecto (chamada do destrutor)
- `update` - actualização dum objecto (chamada de função-membro pública com permissão para alterar o objecto)
- `read` - leitura dum objecto (chamada de função-membro pública que acede ao objecto só para leitura)
- `access` - acesso a um objecto (chamada de função-membro pública)
- `tbegin` - início de transacção
- `tcomplete` - fim da execução do código da transacção, antes de "commit"
- `tcommit`
- `tabort`
- qualquer função-membro

Existem vários operadores para combinar eventos. Pode-se também combinar um evento com uma expressão booleana ("mask predicate") no estado do objecto ou nos argumentos do evento (no caso do evento corresponder a uma chamada de função com argumentos), obtendo assim o mesmo efeito duma regra ECA.

Exemplo

A restrição "hard" exemplificada anteriormente equivale ao seguinte gatilho:

```
class supplier {
    ...
    Name state;
    ...
  public:
    ...
    supplier() { this->CheckState(); }
      // activa o gatilho no constructor
    ...
  trigger:
    perpetual CheckState()
      (after create | after update) // evento(s)
      && (state == Name("NY") || state == Name("")) //
condição
      ==> immediate // modo de acoplamento
      { printf("Invalid Supplier State \n"); tabort; } //
acção
};
```

2.3.6.6 Herança, conflitos e ordem de execução

Tal como acontece com as outras propriedades duma classe, as restrições e os gatilhos são herdadas pelas classes derivadas.

Duas restrições definidas numa classe podem entrar em conflito (supõe-se que isso acontece quando o "handler" duma restrição viola outra restrição). Nesse caso, a transacção é abortada. Embora isso não seja claro na documentação consultada, supõe-se que essa situação só é detectada se o "handler" actualizar os dados através duma função-membro pública.

Não existem facilidades para controlar a ordem de verificação das restrições e de execução dos gatilhos (não é garantido sequer que seja seguida a ordem da definição).

2.4 Conclusões

Pode-se concluir que restrições de integridade genéricas e dados derivados materializados são especificados através de regras de vários tipos (usando o termo "regra" em sentido abrangente):

- regras dirigidas por eventos (com eventos explícitos):
  - *regras ECA* - Encontram-se em SGBD's mas não em ferramentas de desenvolvimento de aplicações. Note-se que quando o modo de acoplamento entre a condição e a acção é imediato (caso mais comum), uma regra ECA é praticamente equivalente a uma regra EA, porque a condição pode ser embebida na parte de acção. Servem para especificar de forma flexível regras de imposição de restrições de integridade e de cálculo de dados derivados.
  - *regras EA* - Encontram-se nas ferramentas de desenvolvimento de aplicações e em SGBD's. Os eventos suportados são muito variáveis. Servem também para especificar de forma flexível regras de imposição de restrições de integridade e de cálculo de dados derivados.
- regras dirigidas pelos dados (com eventos de manipulação de dados implícitos):
  - *regras C* - Especificações declarativas de restrições de integridade, com eventos e acções implícitos. É o caso de asserções e "check constraints" em SQL, restrições sem estabilizadores no sistema PARDES, restrições sem "handlers" no sistema Ode e regras de validação no Microsoft Access. A condição define sempre uma restrição de integridade. A acção é implicitamente abortar e desfazer a transacção ou a operação que viola a restrição.
  - *regras A* - Especificações declarativas (no sentido de não ser necessário especificar os eventos) de dados derivados. Caso de derivações no projecto PARDES e fórmulas de cálculo de dados derivados nas ferramentas de desenvolvimento de aplicações. As especificações declarativas encontradas nas ferramentas e sistemas analisadas sofrem das limitações referidas no capítulo 1.
  - *regras CA* - Especificações declarativas de dados derivados condicionalmente ou restrições de integridade com acção de reparação explícita. Caso, por exemplo, de restrições com estabilizadores no sistema PARDES, derivações condicionais nos sistemas sucessores do projecto PARDES e restrições com "handlers" no sistema Ode.

As ferramentas de desenvolvimento rápido de aplicações dispõem de gatilhos do tipo *EA* muito limitados:

- a relação entre eventos e gatilhos é de um para um (um gatilho só monitora um evento e um evento só é monitorado por um gatilho), o que tem os inconvenientes já explicados no capítulo 1;
- em cada gatilho, o modo de acoplamento entre o evento e a acção é sempre imediato, o que tem os inconvenientes já explicados no capítulo 1.

Em contrapartida, os SGBDA's comerciais ou de investigação dispõem de gatilhos mais flexíveis, que podem ser transpostos com relativa facilidade para as ferramentas de desenvolvimento rápido de aplicações.

No que se refere ao suporte para especificações declarativas de restrições de integridade, existe alguma semelhança entre as restrições em tabelas especificadas com "check constraints" em SQL e as regras de validação suportadas pelas ferramentas de desenvolvimento rápido de aplicações (como o Microsoft Access). Apesar de a norma SQL3 prever a especificação de asserções genéricas (sem eventos explícitos), envolvendo possivelmente várias tabelas, a maioria dos SGBD's comerciais não as implementam, provavelmente por dificuldade de optimização. No caso das ferramentas de desenvolvimento de aplicações, não são suportadas especificações declarativas de asserções genéricas (locais a um formulário ou a um grupo de formulários relacionados entre si).

Nas ferramentas de desenvolvimento de aplicações analisadas o suporte para especificações declarativas de dados derivados (simples) é superior ao que se encontra na maioria nos SGBD's analisados, e encontra-se a um nível muito semelhante ao que se encontra no projecto PARDES, sofrendo das limitações já referidas no capítulo 1. São principalmente essas limitações que se pretendem resolver com o modelo de regras que se propõe a seguir.

3 Regras activas dirigidas pelos dados com semântica de ponto fixo

3.1 Introdução

Neste capítulo preconiza-se um modelo de regras (regras activas dirigidas pelos dados com semântica de ponto fixo) particularmente adequado para a manutenção de restrições de integridade e dados derivados em formulários de ecrã simples, e em outros interfaces para o utilizador presentes em aplicações interactivas de bases de dados igualmente simples.

Por formulários de ecrã simples entendem-se formulários de ecrã uni-registo, constituídos por um conjunto fixo de campos atómicos. Por conseguinte, a simplicidade reside no modelo de dados. Este caso é tratado em primeiro lugar porque:

- as soluções encontradas para o tratamento de restrições de integridade e dados derivados em formulários de ecrã simples são uma boa base de partida para o caso de formulários e outros interfaces para o utilizador mais complexos (a tratar no capítulo 7);
- alguns problemas de análise e processamento de regras não têm a ver com a complexidade do modelo de dados, e são mais facilmente estudados no quadro de um modelo de dados simples.

Mesmo em formulários de ecrã com um modelo de dados simples, encontram-se já necessidades relativamente complexas ao nível de manutenção de dados (ou propriedades) derivados e restrições de integridade, de que se destacam as seguintes:

i) itens de dados calculados em função doutros itens de dados calculados;
ii) itens de dados calculados condicionalmente, i.e. itens de dados que são calculados em algumas circunstâncias e introduzidos pelo utilizador noutras circunstâncias;
iii) cálculo de vários itens de dados através da mesma fórmula, para evitar repetir cálculos dispendiosos, tais como os que envolvem o acesso a disco;
iv) cálculo do mesmo item de dados por várias fórmulas não contraditórias, que surgem naturalmente quando são suportados os tipos de derivações referidos nos dois pontos anteriores;
v) fórmulas para conversão e correcção de dados introduzidos pelo utilizador;
vi) variação dinâmica de propriedades dos campos em função dos valores dos dados, tais como a obrigatoriedade e possibilidade de preenchimento;
vii) restrições ao domínio;
viii) restrições de integridade genéricas tanto em dados primitivos como em dados derivados;
ix) restrições multi-via ("multi-way constraints"), i.e. restrições com várias escolhas possíveis de itens de dados derivados (calculados) e primitivos (de entrada) (e.g., para impor a restrição $c=b-a$, permitir que o utilizador introduza quaisquer dois valores, sendo o terceiro valor calculado);
x) definição procedimental de cálculos complexos.

3.1.1 Insuficiência das abordagens puramente dirigidas por eventos

A generalidade das ferramentas actuais de desenvolvimento rápido de aplicações (de que são exemplos representativos as ferramentas analisadas no capítulo 2) proporcionam mecanismos dirigidos por eventos - gatilhos da forma evento-acção (EA) - que podem ser usados para manter todos os tipos de dados derivados e restrições de integridade normalmente encontrados em aplicações reais. Os gatilhos são flexíveis, mas obrigam o programador a determinar os eventos em resposta aos quais os itens de dados derivados devem ser recalculados e as restrições de integridade verificadas. Mesmo assim, o tratamento de restrições de integridade genéricas é problemático.

Uma abordagem mais flexível, mas igualmente dirigida por eventos, ocorre na maioria dos SGBDA's, onde dados derivados e restrições de integridade podem ser mantidos através de regras activas ou gatilhos da forma evento-acção (EA) ou evento-condição-acção (ECA), conforme vimos no capítulo 2. Nestes sistemas, as restrições de integridade genéricas, mesmo em dados calculados, são mais facilmente suportadas devido à existência de mecanismos de "rollback".

3.1.2 Procura de abordagens dirigidas pelos dados

As ferramentas de desenvolvimento de aplicações suportam especificações declarativas de dados derivados seguindo um paradigma muito semelhante ao das folhas de cálculo. No entanto, não dispõem, em geral, da flexibilidade necessária para suportar a maioria dos tipos de derivações e restrições acima mencionados.

Algumas ferramentas de construção de interfaces gráficos para o utilizador utilizam um paradigma de alto nível - satisfação de restrições pelo modelo perturbacional - que poderia ser usado para suportar de forma elegante algumas das características acima mencionadas, especialmente restrições multi-via [SMBB93]. No entanto, os tipos de relações entre os dados que são suportados são limitados (lineares e pouco mais).

No âmbito dos SGBDA's, para evitar as desvantagens de uma abordagem puramente dirigida por eventos, têm sido propostas diversas abordagens para a manutenção de dados derivados e restrições de integridade, de que se destacam as seguintes:

- A geração automática de regras activas (ECA) a partir de restrições e derivações expressas numa linguagem declarativa de alto nível, tal como é proposto no sistema Starbust [CW90] [CW91]. Esta abordagem foi concebida principalmente para a manutenção incremental de vistas materializadas (armazenadas) e restrições de integridade, sendo a manutenção de restrições de integridade essencialmente reduzida à manutenção de vistas materializadas.
- A definição directa de derivações prontas a executar como regras dirigidas pelos dados, tal como é proposto no projecto PARDES, conforme vimos no capítulo 2. Estas regras podem ser vistas como regras activas incondicionais (só com a parte de acção) com eventos implícitos inferidos pelo sistema a partir da parte de acção (o que só é possível devido a esta ter uma finalidade estabelecida e uma sintaxe simples). Esta abordagem é especialmente adaptada para a manutenção de atributos derivados e restrições nos valores dos atributos.
- A definição de regras activas da forma condição-acção (CA), com eventos implícitos inferidos pelo sistema a partir da parte de condição, como acontece no sistema Ariel [H92]. Restrições de integridade são impostas de forma muito simples através de regras cuja condição é a violação (negação) da restrição e cuja acção é "rollback". Esta abordagem é pouco prática, no entanto, para exprimir derivações incondicionais, semelhantes às que se encontram nas folhas de cálculo e no projecto PARDES, devido à necessidade de se especificar a parte de condição quando os eventos são omitidos.

3.1.3 Abordagem preconizada

Embora estas três últimas abordagens sejam complementares, as duas últimas são as que oferecem um melhor equilíbrio entre a natureza declarativa (com eventos implícitos) e procedimental (com acções explícitas), e servem de ponto de partida para a abordagem preconizada aqui.

As regras dirigidas pelos dados do projecto PARDES sofrem, no entanto, de limitações muito semelhantes às encontradas nas folhas de cálculo, limitações essas que são eliminadas no modelo aqui proposto. Para suportar de forma uniforme derivações condicionais e incondicionais, preconizam-se regras condicionais com um significado ligeiramente diferente do habitual em SGBDA's. Enquanto que habitualmente se considera que a acção deve ser executada sempre que a condição é verdadeira, no modelo proposto a acção deve ser executada sempre que a condição é verdadeira *e* a execução da acção pode resultar na alteração do estado dos itens de dados (isto só faz sentido, claro, em regras destinadas à manutenção de dados derivados e restrições de integridade).

As características fundamentais do modelo de regras proposto, entendido no quadro das regras activas, e que justificam o título deste capítulo, são:

- o facto de as mesmas serem dirigidas pelos dados, isto é, o facto dos eventos serem inferidos pelo sistema, e o facto de até mesmo a parte de condição poder ser omitida;
- o facto das regras terem semântica de ponto fixo, no sentido de que uma regra deve ser executada desde que daí possa resultar uma alteração do estado dos dados.

Um modelo de regras deve contemplar as seguintes componentes [DGG95], que serão objecto dos capítulos seguintes:

- um modelo de conhecimento, i.e., um formalismo para definir ou representar regras em conjunto com um formalismo para definir ou representar dados (modelo de dados);
- um modelo de execução, que define como é que as regras são executadas.

Embora os requisitos do modelo de regras proposto provenham de um domínio de aplicação bem definido - formulários de ecrã simples, o modelo é definido de forma relativamente genérica e abstracta para:

- potenciar a sua aplicação a outros domínios;
- facilitar a compreensão e o raciocínio;
- facilitar a sua comparação e cruzamento com outros modelos de regras.

3.2 Modelo de dados

Qualquer modelo de regras pressupõe um modelo de dados, isto é, um formalismo para descrever dados e operações sobre esses dados. Para tornar o modelo de regras genérico e abstracto, considera-se um modelo de dados também genérico e abstracto.

Assim, para efeitos de processamento de regras, os itens de dados controlados pelas regras são representados abstractamente por *variáveis de estado*. Pressupõe-se que o conjunto de variáveis de estado é *finito*.

As variáveis de estado podem representar campos de formulários e propriedades desses campos variáveis dinamicamente (como a possibilidade e obrigatoriedade de preenchimento), mas também podem representar dados mais complexos, como registos de ecrã, tabelas de ecrã, ficheiros, etc. A estrutura interna das variáveis de estado é ignorada de momento. O refinamento do modelo de regras, no sentido de levar em conta a estrutura interna de variáveis de estado complexas, será abordado no capítulo 7.

O conjunto de valores possíveis das variáveis de estado é chamado *espaço de estados*. As variáveis de estado são as *dimensões* do espaço de estados.

Formalmente, utiliza-se a seguinte notação:

V - designa o conjunto de variáveis de estado

S - designa o espaço de estados
$x, y, z, \ldots$ - designam variáveis de estado
S_x - designa o conjunto de valores (estados) possíveis da variável de estado x

Tem-se então que

$S = S_x \times S_y \times S_z \times \ldots$

isto é, S é o produto cartesiano dos conjuntos de valores possíveis das variáveis de estado.

Para efeito de processamento de regras, as operações primitivas sobre variáveis de estado que interessa considerar são a consulta do estado (leitura de valor) e actualização do estado (escrita ou atribuição de valor). Na prática, estas operações podem ser especializadas de muitas maneiras, mas só nos interessa aqui a sua natureza genérica.

3.3 Definição de regras

Para tornar o modelo de regras genérico e abstracto, não se define uma linguagem de definição de regras prática. Apenas se especifica a sua forma geral, poder expressivo e significado das regras nela expressas. Para ser possível apresentar alguns exemplos, define-se uma representação abstracta. Consideram-se primeiro as regras destinadas à manutenção de dados derivados, e de seguida as regras destinadas à manutenção de restrições de integridade.

3.3.1 Definição de regras de derivação

As regras destinadas à manutenção de dados derivados são designadas *regras de derivação*. Na linha do proposto no projecto PARDES, as regras de derivação dirigidas pelos dados definem o valor de itens de dados (representados por variáveis de estado) em função do valor doutros itens de dados ou constantes, consultando e actualizando *explicitamente* os seus valores.

Assim, uma linguagem prática de definição de regras deste tipo, deve dispor de primitivas para consultar (ler) e actualizar (escrever) os valores das variáveis de estado, condicional ou incondicionalmente.

A única exigência que se coloca ao nível da definição de regras de derivação é a seguinte: as regras devem ser individualmente *determinísticas*, no sentido de que, quando uma regra r é executada a partir de um estado s, o estado final s' atingido no final da execução de r é função apenas de s, não dependendo de estados passados ou do tempo.

Supõe-se ainda que, quando uma regra é executada, apenas o seu *efeito líquido* é importante. Isto é, quando uma regra r é executada a partir de um estado s, somente o estado inicial s e o estado final s', atingido no fim da execução de r, são importantes. Estados intermédios não são considerados importantes. Efeitos laterais (mensagens para o utilizador, etc.) podem existir, mas supõe-se que desempenham um papel secundário. Regras que produzem o mesmo efeito líquido são consideradas semanticamente equivalentes.

Dados os pressupostos anteriores, um regra r pode ser definida abstractamente por uma *função* de S em S (função de transformação de estado), com o mesmo nome da regra, que descreve o seu efeito líquido. O estado final atingido no final da execução de r a partir de um estado s é denotado $r(s)$.

Entende-se que uma regra de derivação deve ser executada sempre que, da sua execução, possa resultar a alteração do estado dos itens de dados, em termos líquidos, isto é, por comparação do seu estado inicial e final. Por outras palavras, uma regra de derivação r deve ser executada num estado s, se s não for um ponto fixo de r. Formalmente, usando a noção de regra como função no espaço de estados, um *ponto fixo* de uma regra r é um estado s tal que $r(s)=s$. As regras de derivação restringem, implicitamente, os estados válidos do sistema aos estados que obedecem à condição $r(s)=s$. Por essa razão, esta condição é chamada a *restrição imposta* por r (implicitamente), e os pontos fixos de r são também os estados *consistentes* com r.

Considera-se ainda que a ordem por que as regras são definidas não é relevante.

3.3.1.1 Representação abstracta

Para apresentar alguns exemplos utiliza-se a seguinte representação abstracta: uma regra *r* é representada por uma conjunção, para diferentes *y*'s, de fórmulas incondicionais ou condicionais do tipo *y'=f(X)* ou *p(Z)* Þ *y'=f(X)*, respectivamente. Nestas fórmulas, *y* designa uma variável de estado, *p(Z)* é um predicado arbitrário definido por uma condição num conjunto *Z* de variáveis de estado e *f(X)* é uma função arbitrária definida por uma expressão num conjunto *X* de variáveis de estado. Condições comuns podem ser postas em evidência. Referências a variáveis com pelicas, como em *y'*, designam valores finais, enquanto que referências sem pelicas, designam valores iniciais. Os símbolos "Þ"e "=" têm o significado lógico habitual. Por omissão, considera-se que as variáveis de estado mantêm o seu valor (*pressuposto de quiteude*). Isto é, as variáveis de estado cujo valor final não é definido, têm um valor final igual ao valor inicial; as variáveis de estado cujo valor final é definido condicionalmente, têm um valor final igual ao valor inicial quando a condição é falsa. Ver exemplos na figura 3.1.

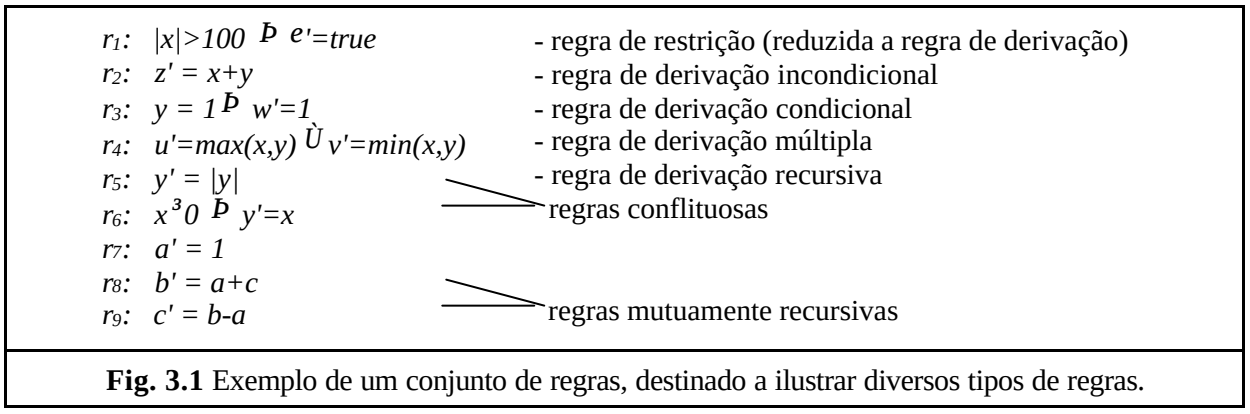


Fig. 3.1 Exemplo de um conjunto de regras, destinado a ilustrar diversos tipos de regras.

Esta representação tem diversas vantagens.

Em primeiro lugar, para definir completamente a função de *S* em *S* correspondente a uma regra, é suficiente adicionar as seguintes condições: para cada variável *v* cujo estado final não é referenciado, adicionar a condição *v'=v*; para cada variável *y* cujo estado final é definido na forma condicional *p(Z)* Þ *y'=f(X)*, adicionar a condição Ø*p(Z)* Þ *y'=y*, em que "Ø" representa a negação lógica.

Exemplo

Considerando um conjunto de variáveis de estado *V={x, y, z}*, a regra r_2 da figura 3.1, pode ser estendida da seguinte forma:

r_2: *z' = x+y* Ù *y' = y* Ù *x' = x*

o que corresponde à seguinte função no espaço de estados:

r_2: *S* ® *S*

(x, y, z) ⇝ *(x', y', z')=(x, y, x+y)*

Em segundo lugar, qualquer regra representada nesta notação pode ser traduzida para uma linguagem prática, referenciando exactamente as mesmas variáveis de estado em modos correspondentes. Referências a valores finais correspondem a referências para escrita, enquanto que referências a valores iniciais correspondem a referências para leitura. Podem ser necessárias variáveis locais auxiliares para armazenar valores iniciais de algumas variáveis, como mostra o exemplo seguinte.

Exemplo

Supondo variáveis inteiras, a seguinte regra (com uma condição comum em evidência):

r: x > y Þ *(x' = y* Ù *y' = x)*

pode ser traduzida pela seguinte função em C (notar a utilização da variável local auxiliar `oldx`):

```
void r() {int oldx; if (x > y) {oldx = x; x = y; y = oldx;}}
```

Finalmente, para obter a condição que define os estados consistentes com uma regra de derivação, basta remover as pelicas.

Exemplo
Os estados consistentes com a regra:
r: x > y Þ (x' = y Ù y' = x)
são os que obedecem à condição:
x > y Þ (x = y Ù y = x)
que é equivalente a:
x £ y Ú (x = y Ù y = x)
ou simplesmente:
x £ y

3.3.1.2 Tipos de regras de derivação

Para além do determinismo, não se impõe, à partida, nenhuma restrição ao tipo de regras de derivação que podem ser expressas, individualmente e em conjunto. Em particular, admitem-se regras dos tipos a seguir indicados.

Uma regra que deriva o valor de mais do que uma variável de estado, como a regra r_4 na figura 3.1, é chamada regra de derivação *múltipla* (por oposição a *simples*).

Uma regra de derivação múltipla pode ter algumas derivações condicionais e outras incondicionais. A regra em si diz-se *condicional* (por oposição a incondicional) se contiver uma condição (não universal) comum a todas as derivações, que pode ser posta em evidência. As regras condicionais podem ser expressas na forma condição-acção. No entanto, diferentemente do que acontece habitualmente nos sistemas de regras activas que suportam regras do tipo condição-acção, como no sistema Ariel [H92], não existe aqui a obrigação de executar a acção sempre que a condição se verifica, mas tão só quando a condição é verdadeira e a execução da acção pode resultar numa alteração de estado.

Regras que derivam valores para a mesma variável, desejavelmente de forma não contraditória, são chamadas regras *conflituosas*. Exemplos de regras conflituosas não contraditórias:

- regras com condições mutuamente exclusivas, tais como:
  r_{10}*: x > 0 Þ y' = 1* e r_{11}*: x £ 0 Þ y' = -1*;
- regras com acções idênticas, tais como:
  r_{12}*: x < 0 Þ e' = true* e r_{13}*: y < 0 Þ e' = true*;
- regras com acções cumulativas, tais como:
  r_{14}*: z' = z È x* e r_{15}*: z' = z È y*;
- regras que actualizam diferentes partições da mesma variável de estado, tais como:
  r_{16}*: z' =z È {1}* e r_{17}*: z' = z-{2}* (supondo que *z* representa um conjunto).

Regras que derivam um valor para uma variável (no final da execução da regra) usando o valor da mesma variável (no início da execução da regra) são chamadas recursivas, ou mais precisamente, *auto-recursivas*. Exemplos de regras auto-recursivas úteis:

- regras para conversão ou correcção de entradas, do género da regra r_5 na figura 3.1;
- regras usadas em métodos iterativos, tal como:
  r_{18}*: y' = exp(-y)*.

Conjuntos de duas ou mais regras que, em conjunto, derivam um valor para uma variável usando o valor da mesma variável, pelo menos aparentemente, são chamadas *mutuamente recursivas*. Exemplos de regras mutuamente recursivas úteis:

- regras com condições mutuamente exclusivas, tais como:
  r_{19}: $x = 0 \Rightarrow y' = z$ e r_{20}: $x \neq 0 \Rightarrow z' = y$;
- regras que impõem a mesma restrição de várias maneiras, tais como as regras r_8 e r_9 na figura 3.1.

Definições mais precisas de regras recursivas e conflituosas serão apresentadas no capítulo 4.

3.3.2 Definição de regras de restrição

As regras que obrigam a repor um estado anterior consistente quando uma restrição de integridade é violada, são designadas *regras de restrição*. Estas regras devem ser sempre condicionais, em que a condição é precisamente a violação de uma restrição de integridade. Também são chamadas "abort rules" por alguns autores [ZCF+97]. Estas regras impõem uma restrição de integridade de forma explícita, enquanto que as regras de derivação o fazem de forma implícita.

Uma linguagem prática de definição de regras deve dispor de uma primitiva para abortar o processamento de regras e, subsequentemente, repor o sistema num estado anterior consistente com todas as regras.

As regras de restrição são de seguida tratadas como (reduzidas a) regras de derivação que activam uma variável de estado especial e (de erro), de valor booleano, que é monitorada pelo sistema de controlo (ver regra r_1 na figura 3.1). A exigência de determinismo estabelecida para as regras de derivação, aplica-se também às regras de restrição após a redução a regras de derivação. A redução a regras de derivação induz, no entanto, uma pequena imprecisão de linguagem: os estados consistentes com uma regra de restrição não são exactamente os pontos fixos da regra de derivação equivalente, mas sim os estados que não violam a restrição.

Podem também existir *regras mistas*, isto é, regras simultaneamente de derivação e de restrição. A existência de uma primitiva explícita para abortar é útil para suportar este género de regras. No caso de regras puramente de restrição, a acção de abortar poderia ser implícita, bastando indicar a restrição de integridade.

Exemplo

r: $(x \neq 0 \Rightarrow y' = 1 / x) \wedge (x = 0 \Rightarrow e' = true)$

3.4 Modelo de execução

Descrevem-se de seguida as opções fundamentais tomadas relativamente ao modo como as regras devem ser executadas.

Em primeiro lugar, preconiza-se que a execução das regras (destinadas à manutenção de restrições de integridade e dados derivados) esteja *associada à execução das transacções*, tirando partido das propriedades de atomicidade e preservação de consistência das mesmas (componentes A e C das propriedades ACID [GR93]). Por um lado, a gestão de transacções pressupõe um mecanismo de recuperação de falhas ("rollback") para garantir a propriedade de atomicidade, o qual pode e deve ser usado para suportar as regras de restrição. Por outro lado, para garantir a propriedade de preservação de consistência, deve existir um ponto de processamento de regras pelo menos no final de cada transacção ("on commit"), podendo existir pontos de processamento de regras intermédios. Entende-se por ponto de processamento de regras (PPR) um ponto no tempo em que é invocado um algoritmo de processamento (execução) de regras.

É de notar que, no domínio de aplicação que mais nos interessa - formulários de ecrã, as transacções incidem, não nos dados armazenados de forma persistente e partilhada numa BD, mas sim nos dados armazenados de forma temporária e privativa na memória de trabalho da aplicação. Nesse contexto, as propriedades de isolamento (I) e durabilidade (D) não são importantes, o que, em nossa opinião, não impede a utilização do conceito de transacção.

Em segundo lugar, preconiza-se que a execução das regras seja *dirigida por eventos*, como é característico dos sistemas de regras activas. Isto é, as regras dirigidas pelos dados, definidas só com a parte de acção (A) ou também com a parte de condição (CA), são executadas como regras activas completas, da forma EA ou ECA respectivamente, sendo os eventos inferidos pelo sistema. A execução dirigida por eventos é eficiente em tempo e em espaço, e é importante para a integração das regras dirigidas pelos dados com regras activas provenientes doutras fontes. Note-se que os eventos que activam as regras têm de ser continuamente monitoradas pelo sistema de controlo, e não só nos PPR's.

Em terceiro lugar, preconiza-se um modo de execução *diferida*, por oposição a imediata. Num modo de execução imediata, uma regra é executada imediatamente assim que ocorre um evento activador ("triggering event"). Num modo de execução diferida, a execução da regra é diferida para um momento posterior, que, de acordo com o 1º ponto, será normalmente o fim da transacção em que ocorre o evento. Usando a terminologia habitual de sistemas de regras activas, o modo de acoplamento preconizado entre o evento (implícito) e a condição é diferido, enquanto que o modo de acoplamento preconizado entre a condição e a acção é imediato. A execução diferida, apesar de exigir um algoritmo mais sofisticado, é importante para suportar restrições de integridade que envolvem itens de dados que são actualizadas separadamente, e para minimizar o número de execuções das regras.

Em quarto lugar, preconiza-se um modo de execução *sequencial*, em vez de concorrente ou encaixada. A execução sequencial é importante para garantir que as regras são atómicas umas em relação às outras. Assim, o efeito líquido da aplicação de uma regra é independente da existência de outras regras. A possível execução concorrente (em paralelo) equivalente a uma execução sequencial (em série) é matéria de optimização que não se exclui, mas não é abordada aqui. Uma vez que as regras podem ser representadas abstractamente como funções, a execução sequencial pode ser representada abstractamente como composição de funções.

Note-se que a execução sequencial implica a execução diferida, pelo menos na presença de activação *múltipla* (várias regras activadas pelo mesmo evento) e em *cascata* (regras activadas por eventos gerados por outras regras). No caso de activação em cascata, uma vez que a execução encaixada é afastada, as regras activadas em cascata não podem ser executadas imediatamente quando o evento activador ocorre. No caso de activação múltipla, se o processamento de regras for invocado imediatamente quando o evento ocorre, uma vez que a execução concorrente é afastada, essas regras não podem ser executadas imediatamente.

Finalmente, para tornar o problema mais facilmente tratável, supõe-se que, durante a execução das regras, os conjuntos de regras e de variáveis de estado não são alterados e as únicas alterações ao estado das variáveis de estado são produzidas por regras.

Assim, adopta-se o algoritmo habitual de processamento sequencial de regras activas:

Algoritmo 3.1 (*processamento sequencial de regras*):

1. Enquanto existirem regras activadas e a variável de erro *e* não tiver sido activada,
    1.1. Seleccionar uma regra activada *r*. No caso de existir mais do que uma regra activada, seleccionar uma delas de acordo com algum critério de resolução de conflitos.
    1.2. Executar a regra *r*, i.e., avaliar a condição e, no caso de ser satisfeita, executar a acção.
2. Se a variável de erro *e* tiver sido activada, repor o sistema no último estado consistente.

Para completar a definição do modelo de execução, falta definir os critérios de activação das regras e de resolução de conflitos (ou de ordenação), que serão objecto dos capítulos 4 e 5. Na presença de certos tipos de regras, o algoritmo acima descrito pode não terminar e pode ser não determinístico. No

capítulo 6, são determinadas condições suficientes a que um conjunto de regras deve obedecer para garantir a terminação e o determinismo do algoritmo 3.1.

3.5 Relaxamento de algumas restrições

Algumas restrições impostas ao nível da definição das regras podem ser relaxadas sem comprometer o formalismo apresentado.

3.5.1 Regras que consultam o estado das variáveis de estado no início da transacção

Foi dito que as regras não podem referenciar estados passados. Ora, sob determinadas condições a discutir adiante, regras que consultam o estado das variáveis de estado no início da transacção podem ser enquadradas no formalismo apresentado.

Regras deste tipo são úteis para impor *restrições de integridade dinâmicas*, isto é, restrições às transições de estado válidas, entre o estado inicial e final de cada transacção.

Note-se que o estado das variáveis de estado (pelo menos das variáveis alteradas) no início da transacção já tem de ser guardado para efeito de 'rollback", pelo que não se introduz nenhum peso adicional no processamento de regras.

Seja a seguinte notação:

old(x) - valor da variável de estado *x* no início da transacção

changed(x) - mesmo que *x ¹ old(x)*

Conceptualmente, no final de uma transacção bem sucedida, é reposta a igualdade *x=old(x)* para todas as variáveis de estado, através da atribuição *old(x) ¬ x*. No final de uma transacção abortada, a mesma igualdade é reposta através da atribuição *x ¬ old(x)*.

Diz-se que uma regra *r* é *segura* se a restrição imposta por *r* não é violada quando se faz *old(x) ¬ x* para todas as variáveis de estado. Isto garante que, após terminar o processamento de regras (num ponto fixo para todas as regras) e terminar a transacção respectiva (momento em que se repõe *old(x)=x*), o início da transacção seguinte é um ponto fixo para *r*.

Se todas as regras são seguras, o estado das variáveis de estado no início da transacção pode ser tratado como uma constante em cada PPR, pelo que, essencialmente, pode ser ignorado para efeito do processamento de regras.

Exemplos

A regra seguinte proíbe transacções que mais do que duplicam o valor de *x*:

*r_1: x > 2*old(x) Þ e' = true*

A restrição imposta por esta regra é: *x £ 2*old(x)*. Esta condição continua a ser verdadeira quando se substitui *old(x)* por *x* (supondo que era anteriormente verdadeira) desde que *x³0*. Portanto, esta regra só é segura se *x* aceita apenas valores não negativos.

A regra seguinte proíbe transacções que alteram o valor de *x* para um valor superior ao de *y* (mas não proíbe transacções que alteram o valor de *y* para um valor inferior ao de *x*):

r_2: changed(x) Ù x>y Þ e' = true

A restrição imposta por esta regra é: *x=old(x) Ú x£y*. Quando se substitui *old(x)* por *x*, a restrição continua obviamente a ser satisfeita, pelo que a regra é segura.

A regra seguinte desfaz alterações da variável *x* que violam uma determinada restrição (faz uma espécie de "rollback" local):

r_3: x > 100 Þ x' = old(x)

A restrição imposta por esta regra é *x£100 Ú x=old(x)*. Quando se substitui *old(x)* por *x*, a restrição continua obviamente a ser satisfeita, pelo que a regra é segura.

A regra seguinte mantém um contador do nº de vezes (nº de transacções em) que uma variável *x* é alterada:

r_4: *changed(x)* Þ *c' = old(c)+1*

A restrição imposta por esta regra é $x=old(x) \acute{U} c=old(c)+1$. Quando se substitui *old(x)* por *x* e *old(c)* por *c*, a restrição continua a ser satisfeita, pelo que a regra é segura. Note-se que, uma vez que *old(c)* é tratado como uma constante, a regra não é auto-recursiva em *c*.

A regra seguinte actualiza uma variável *y* em função de *x*, apenas quando *x* é alterado (não impede, portanto, alterações directas de *y*):

r_5: *changed(x)* Þ *y'=x*

A restrição imposta por esta regra é $x=old(x) \acute{U} y=x$. Quando se substitui *old(x)* por *x*, a restrição continua a ser satisfeita, pelo que a regra segura.

A seguinte regra não é segura:

r_6: *Øchanged(x)* Þ *y'=x*

A restrição imposta por esta regra é $x \neq old(x) \acute{U} y=x$. Quando se substitui *old(x)* por *x*, a restrição continua a ser satisfeita apenas no caso em que *y=x*, pelo que a regra não é segura e, portanto, não é suportada.

3.5.2 Regras que consultam a data e hora actuais

Foi dito que as regras não podem referenciar o tempo. Ora, sob determinadas condições, regras que consultam a data e hora actuais podem ser enquadradas no formalismo apresentado.

Regras que referenciam a data e hora actuais podem ser úteis para animar os interfaces para o utilizador, para recolher dados automaticamente de fontes externas a intervalos de tempo regulares, etc.

Para suportar regras deste tipo, a data e hora actuais podem ser representadas por uma ou mais variáveis de estado que são actualizadas periodicamente através de alarmes ou temporizadores. A única restrição que se impõe, tal como acontece em relação às actualizações causadas pelo utilizador, é que essas variáveis de estado não devem ser actualizadas durante o processamento de regras.

4 Activação das regras

Neste capítulo desenvolvem-se os critérios de activação das regras definidas no capítulo anterior, isto é, os eventos que as activam e desactivam, os quais se enquadram numa visão dinâmica das regras como sistemas reactivos. Esses critérios são baseados num levantamento prévio das dependências de entrada e saída de dados existentes entre as regras e as variáveis de estado, que correspondem a uma visão funcional das regras como sistemas de transformação de dados.

4.1 Dependências entre regras e variáveis

4.1.1 Variáveis de entrada e variáveis de saída de uma regra

As variáveis cujo valor é consultado (lido) por uma regra *r* são chamadas *variáveis de entrada* de *r*, ou simplesmente *entradas* de *r*. As variáveis cujo valor é actualizado (escrito) por uma regra *r* são chamadas *variáveis de saída* de *r*, ou simplesmente *saídas* de *r*.

Utiliza-se a seguinte notação:

i-vars(r) - conjunto de variáveis de entrada ("input") da regra *r*
o-vars(r) - conjunto de variáveis de saída ("output") da regra *r*
vars(r) - conjunto de variáveis de entrada *ou* saída da regra *r* (variáveis *referenciadas* por *r*)
io-vars(r) - conjunto de variáveis de entrada *e* saída da regra *r*

Verificam-se as seguintes relações:

$vars(r) = i\text{-}vars(r) \cup o\text{-}vars(r)$
$io\text{-}vars(r) = i\text{-}vars(r) \cap o\text{-}vars(r)$

Estando as regras numeradas, também se usa a seguinte notação abreviada:

X_i - $i\text{-}vars(r_i)$
Y_i - $o\text{-}vars(r_i)$
V_i - $vars(r_i)$

Na representação abstracta das regras, as variáveis de entrada são aquelas cujo valor inicial é referenciado (sem pelica), enquanto que as variáveis de saída são aquelas cujo valor final é referenciado (com pelica). Numa linguagem de regras prática, os conjuntos de variáveis de entrada e saída podem ser obtidos por uma simples análise sintáctica.

Exemplo
A regra

$r: x > z \rightarrow y' = x$

tem

$i\text{-}vars(r) = \{x, z\}$, $o\text{-}vars(r) = \{y\}$, $vars(r) = \{x, y, z\}$, $io\text{-}vars(r) = \{\}$.

Note-se que o nível de descrição das regras com a expressão de cada regra mais as suas variáveis de entrada e saída é muito semelhante ao nível de descrição das dependências entre ficheiros nas “makefiles” entendidas pelo comando "make" em Unix.

4.1.2 Influência da forma sintáctica

As entradas e saídas de cada regra podem ser determinadas por uma simples análise sintáctica, o que lhes confere interesse prático. Infelizmente, e mesmo na representação abstracta, duas regras semanticamente equivalentes (isto é, com o mesmo efeito líquido), mas escritas de forma diferente, podem referenciar conjuntos diferentes de variáveis ou as mesma variáveis em modos diferentes. A própria escolha entre a forma incondicional e a forma condicional pode afectar as variáveis referenciadas, conforme mostram os exemplos seguintes.

Exemplo 1

As regras seguintes (em que r_6 provém da figura 3.1) têm o mesmo efeito líquido:

$$r_6: x \geq 0 \Rightarrow y'=x \qquad r_{6a}: y' = \begin{cases} x, \text{ se } x \geq 0 \\ y, \text{ se } x < 0 \end{cases} \qquad r_{6b}: x \geq 0 \vee y < x \Rightarrow y'=x$$

No entanto, $i\text{-}vars(r_6)=\{x\} \neq i\text{-}vars(r_{6a})=\{x,y\} = i\text{-}vars(r_{6b})$. A forma do lado esquerdo é a mais simples e a que referencia menos variáveis. A forma do meio é incondicional. A forma da direita tem a condição o mais restritiva possível (negação da restrição imposta pela regra).

Exemplo 2

As regras seguintes (em que r_2 provém da figura 3.1) têm o mesmo efeito líquido:

$r_2: z'=x+y$ $\qquad$ $r_{2a}: z \neq x+y \Rightarrow z'=x+y$

No entanto, $i\text{-}vars(r_2)=\{x, y\} \neq i\text{-}vars(r_{2a})=\{x, y, z\}$. A forma da esquerda (incondicional) é a mais simples e a que referencia menos variáveis. A forma da direita tem a condição mais restritiva possível (negação da restrição imposta pela regra).

Exemplo 3

As regras seguintes têm o mesmo efeito líquido:

$r: x>y \Rightarrow y'=x$ $\qquad$ $r': y'=max(x,y)$

As entradas e saídas mantêm-se, mas a referência a y passa para o lado da condição, o que pode ser relevante para o critério de activação. A forma da esquerda tem a condição o mais restritiva possível (negação da restrição imposta pela regra), enquanto que a forma da direita é incondicional.

4.1.3 Formas canónicas

Quando não é possível rescrever uma regra r, por forma a referenciar menos variáveis de entrada ou de saída mantendo o efeito líquido, diz-se que r está em *forma canónica*. Por exemplo, todas as regras da figura 3.1 estão em forma canónica, incluindo as regras r_2 e r_6 acima transcritas.

Interessa que as regras sejam escritas em forma canónica, referenciando assim conjuntos *mínimos* de entradas e saídas, para tornar mais precisos os métodos de processamento (activação e ordenação) e de análise de regras baseados no conhecimento das variáveis de entrada e de saída. Além disso, as formas canónicas são, em geral, mais simples.

Em alguns casos, como no 1º exemplo acima, só existe a forma canónica condicional. Noutros casos, como no 2º exemplo acima, só existe a forma canónica incondicional. Noutros casos, como no exemplo 1 acima, existem formas canónicas dos dois tipos. Esta é uma boa justificação para suportar tanto regras condicionais como regras incondicionais.

Em alguns casos, em linguagens de regras práticas, não é conveniente escrever as regras em forma canónica, por razões de eficiência.

Exemplo

A regra

$r: y'=x+z \vee w'=(x+z)/2$

pode ser traduzida de forma eficiente pela seguinte função em linguagem C [KR88]:

```
void r()
{ y = x + z; w = y / 2; }
```

Notar que a variável `y` é acedida para leitura.

Mesmo quando as regras não são escritas em forma canónica, os conjuntos de variáveis de entrada e saída que interessa considerar no processamento e análise de regras, são os conjuntos mínimos necessários para obter o mesmo efeito líquido. Duas medidas complementares que podem ser usadas para obter os conjuntos mínimos de variáveis de entrada e saída nesses casos são:

- Efectuar uma análise de fluxo de dados durante a compilação das regras para detectar operações de leitura e escrita que poderiam ser evitados pela rescrita dessas regras. É o caso da leitura de uma variável depois da escrita na mesma variável, porque poderia ser evitada com o recurso a uma variável local auxiliar.
- Colocar ao dispor do programador mecanismos para os explicitar.

4.1.4 Unicidade e significado dos conjuntos mínimos de entradas e saídas

A forma canónica para obter um determinado efeito líquido não é única, mas, felizmente, os conjuntos mínimos de variáveis de entrada e saída necessários para produzir um determinado efeito líquido já o são, sendo função apenas do efeito líquido em causa (como não podia deixar de ser), de acordo com o teorema seguinte.

Teorema 4.1: Os conjuntos mínimos de variáveis de entrada e saída de uma regra *r* que podem ser obtidos rescrevendo *r*, mantendo o seu efeito líquido, são únicos. Além disso, dependem apenas da função de *S* em *S* (com o mesmo nome da regra) que descreve o efeito líquido de *r*, de acordo com as seguintes expressões (traduzidas esquematicamente na figura 4.1):

$$min\text{-}o\text{-}vars(r) = \{y \in V: \exists s \in S: r(s).y \neq s.y\}$$

$$min\text{-}i\text{-}vars(r) = \{x \in V: \exists y \in V: \exists s,t \in S: s.(V-\{x\}) = t.(V-\{x\}) \wedge r(s).y \neq s.y \wedge r(s).y \neq r(t).y\}.$$

(Nestas expressões, o símbolo "." denota a projecção de um tuplo num componente ou num conjunto de componentes.)

Demonstração:

Saídas: A condição "$\exists s \in S: r(s).y \neq s.y$" diz apenas que o valor de *y* é alterado quando a regra *r* é aplicada, para pelo menos um estado inicial *s*. Na representação abstracta, é óbvio que só têm de ser referenciados os valores finais das variáveis que obedecem a esta condição. Uma vez que uma regra definida na representação abstracta pode ser implementada numa linguagem prática com referencia às mesmas variáveis em modos correspondentes (referências a valores finais correspondem a referências para escrita, enquanto que referências a valores iniciais correspondem a referências para leitura), conclui-se que só têm de ser referenciadas para escrita as variáveis que obedecem à condição indicada. Reciprocamente, só as variáveis que são referenciadas para escrita podem obedecer à condição indicada, de acordo com o pressuposto de quietude também considerado na representação abstracta.

Entradas: Suponhamos que existe uma regra *r'* semanticamente equivalente a *r*, que não lê o valor de uma variável *x*. Para qualquer variável $y \in V$ (incluindo *x*), a decisão de escrever algum valor em *y* e o valor escrito, não dependem do valor inicial de *x*, mas apenas do valor inicial de *V-{x}*. Para alguns valores iniciais de *V-{x}*, o valor final de *y* é sempre igual ao seu valor inicial (para qualquer valor inicial de *x*), enquanto que para os restantes valores de *V-{x}*, o valor final de *y* é sempre o mesmo (para qualquer valor inicial de *x*). Formalmente, isto pode ser expresso da seguinte forma:

$$(1)\quad \forall y \in V, \forall k \in range(V-\{x\}), [\forall s,t \in S, s.(V-\{x\}) = t.(V-\{x\}) = k \Rightarrow r(s).y = r(t).y] \vee [\forall s' \in S, s'.(V-\{x\}) = k \Rightarrow r(s').y = s'.y]$$

Reciprocamente, se para alguns valores iniciais de *V-{x}*, o valor final de *y* é sempre igual ao seu valor inicial (para qualquer valor inicial de *x*), enquanto que para os restantes valores de *V-{x}*, o valor final de *y* é sempre o mesmo (para qualquer valor inicial de *x*), então é possível implementar *r* sem ler

x. Uma expressão de derivação possível, na representação abstracta, é do tipo $p(V\text{-}\{x\}) \Rightarrow y' = f(V\text{-}\{x\})$, com *p* e *f* definidos da seguinte forma: para aqueles valores iniciais de $V\text{-}\{x\}$ a que corresponde sempre o mesmo valor final de *y* (para qualquer valor inicial de *x*), o predicado *p* toma o valor verdadeiro e a função *f* toma esse valor de *y*; para aqueles valores iniciais de $V\text{-}\{x\}$ a que correspondem múltiplos valores finais de *y* (para diferentes valores iniciais de *x*), mas sempre iguais ao valor inicial de *y*, o predicado *p* toma o valor falso e a função *f* toma um valor arbitrário.

Assim, conclui-se que existe uma regra *r'* semanticamente equivalente a *r* que não lê o valor de uma variável *x* se e só se a condição (1) se verificar. Equivalentemente, a variável *x* tem de ser lida (i.e., tem de pertencer ao conjunto de entradas) de qualquer regra semanticamente equivalente a *r*, se e só se a seguinte condição se verificar (negação da condição (1)):

$$(2)\quad \exists y \in V: \exists k \in range(V\text{-}\{x\}): [\exists s,t \in S: s.(V\text{-}\{x\}) = t.(V\text{-}\{x\}) = k \wedge r(s).y \neq r(t).y] \wedge \\ \wedge\ [\exists s' \in S: s'.(V\text{-}\{x\}) = k \wedge r(s').y \neq s'.y]$$

A variável *k* pode ser removida desta expressão, resultando a seguinte expressão equivalente:

$$(2')\quad \exists y \in V: \exists s,t,s' \in S: s.(V\text{-}\{x\}) = t.(V\text{-}\{x\}) = s'.(V\text{-}\{x\}) \wedge r(s).y \neq r(t).y \wedge r(s').y \neq s'.y$$

A condição $r(s).y \neq r(t).y$ implica que $r(s').y \neq r(s).y$ ou $r(s').y \neq r(t).y$. Assim, a expressão (2) implica (escolhendo $s''=s$ ou $s''=t$, respectivamente):

$$(2'')\quad \exists y \in V: \exists s',s'' \in S: s'.(V\text{-}\{x\}) = s''.(V\text{-}\{x\}) \wedge r(s').y \neq r(s'').y \wedge r(s').y \neq s'.y$$

A recíproca também é verdadeira (escolhendo $s=s'$ e $t=s''$). Assim, conclui-se que as expressões (2') e (2'') são equivalentes. A expressão (2'') é equivalente à condição apresentada no enunciado do teorema, bastando substituir *s'* por *s* e *s''* por *t*. Δ

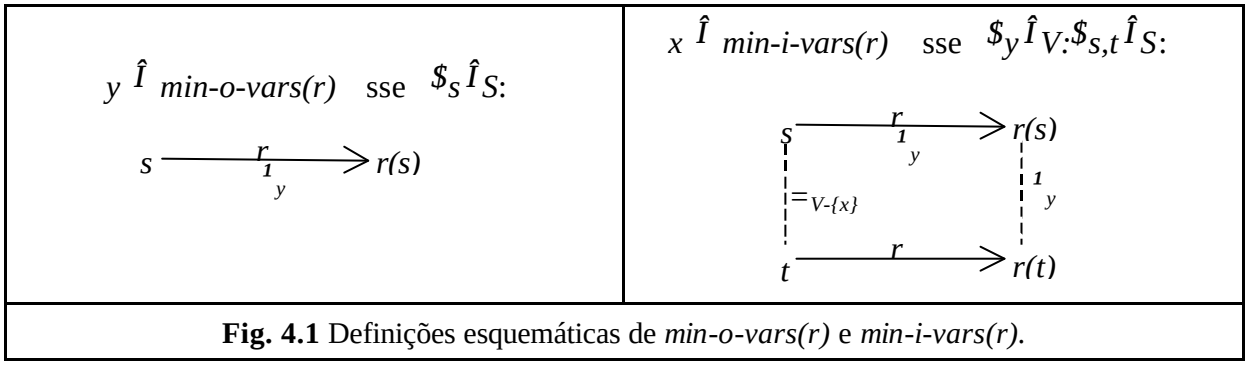


Fig. 4.1 Definições esquemáticas de *min-o-vars(r)* e *min-i-vars(r)*.

As expressões apresentados neste teorema podem ser lidas da seguinte forma:

- Uma variável *y* faz parte de *min-o-vars(r)* se e só se o valor de *y* é alterado em termos líquidos (por comparação dos valores iniciais e finais) quando *r* é executada, para pelo menos um estado inicial.
- Uma variável *x* faz parte de *min-i-vars(r)* se e só se existem dois estados iniciais *s* e *t*, diferentes apenas em *x*, tal que os estados finais correspondentes (no fim da execução de *r*) diferem no valor de pelo menos uma variável *y*, e o valor dessa variável *y* é alterado para pelo menos um dos estados iniciais (digamos *s*). No caso de variáveis *x* e *y* distintas, a condição $r(s).y \neq s.y$ é redundante, e a definição é óbvia; basicamente, o que se diz é que o valor final de *y* depende funcionalmente do valor inicial de *x*. No caso de *x* e *y* serem a mesma variável, a definição é mais subtil.

Este teorema é importante porque estabelece uma ponte entre a sintaxe (neste caso, as variáveis referenciadas para leitura ou escrita) e a semântica de uma regra (neste caso, o seu efeito líquido descrito por uma função no espaço de estado), servindo, mais adiante, para demonstrar propriedades que são baseados nessas variáveis.

4.1.5 Grafo de dependências entre regras e variáveis (grafo r-v)

As relações de entrada e saída entre regras e variáveis podem-se representar graficamente através de um *grafo de dependências entre regras e variáveis*, designado abreviadamente *grafo r-v*, e definido da seguinte forma:

- As regras e variáveis de estado são os vértices do grafo.
- Uma aresta *v* ® *r*, dirigida de uma variável *v* para uma regra *r*, significa que *v* é uma variável de entrada de *r*.
- Uma aresta *r* ® *v*, dirigida de uma regra *r* para uma variável *v*, significa que *v* é uma variável de saída de *r*.

A figura 4.2 mostra o grafo r-v correspondente ao conjunto de regras da figura 3.1. É usada a notação dos diagramas de fluxos de dados, tal como é apresentada em [R91], porque as regras podem ser vistas como processos, as variáveis de estado podem ser vistas como depósitos de dados e as relações de entrada e saída entre regras e variáveis podem ser vistas como fluxos de dados.

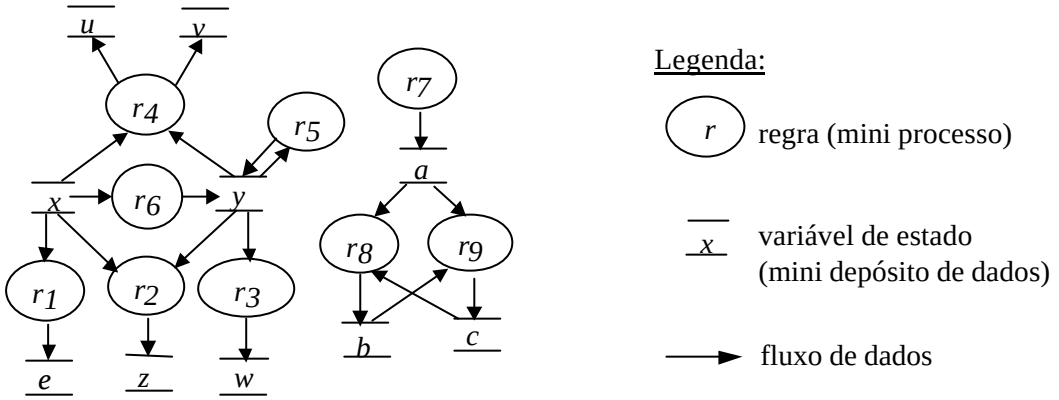


Fig. 4.2 Grafo de dependências entre regras e variáveis relativo ao conjunto de regras da fig. 3.1.

O grafo r-v tem as seguintes características:

- é dirigido;
- é simples, isto é, não tem arestas paralelas nem lacetes;
- é bipartido, isto é, há vértices de dois tipos (regras e variáveis) e todas as arestas ligam vértices de tipos diferentes.

O grafo r-v pode ter ciclos, que assinalam a presença de regras recursivas. As regras que pertencem a um ciclo do grafo r-v com duas ou mais regras são chamadas *mutuamente recursivas*. É o caso das regras r_8 e r_9 na figura 4.2. Uma regra que pertence a um ciclo do grafo r-v com uma regra apenas é chamada *auto-recursiva*. É o caso da regra r_5 na figura 4.2. Uma regra que pertence a pelo menos um ciclo do grafo r-v é chamada *recursiva*. Estas definições são mais precisas do que as dadas no capítulo 3.

O grafo r-v é a fonte principal de informação em que se baseiam os critérios de activação e ordenação de regras a definir mais adiante, como acontece também no projecto PARDES (em rigor, só para a definição do critério de ordenação é que interessa a visão geral do conjunto de regras dada pelos grafos de dependências).

4.1.6 Grafo de dependências entre regras (grafo r-r)

Uma imagem global das interferências entre regras, por via das variáveis por elas manipuladas, é dada por uma *grafo de dependências entre regras*, designado abreviadamente *grafo r-r*, e definido da seguinte forma:

- As regras são os vértices do grafo.

- Uma aresta *r ® r'*, entre regras diferentes, significa que *r* actualiza uma variável lida por *r'*, isto é, *o-vars(r) Ç i-vars(r') ¹ {}*.
- Um lacete *r ® r*, significa que *r* actualiza uma variável também lida por *r*, isto é, *io-vars(r) ¹ {}*.

O grafo r-r pode ter ciclos, que assinalam a presença de regras recursivas.

Um exemplo de um grafo r-r é dado na figura 4.3.

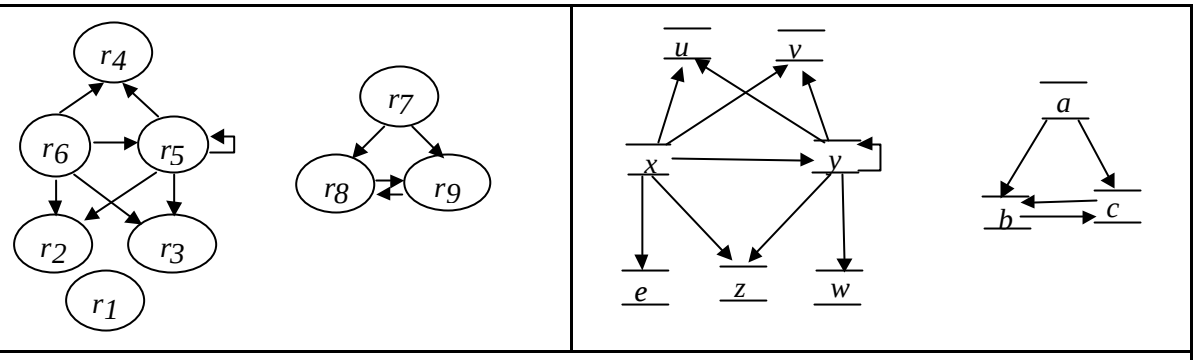


Fig. 4.3 Grafo de dependências entre regras (à esquerda) e grafo de dependência entre variáveis (à direita) relativos ao conjunto de regras das figuras 3.1 e 4.2.

4.1.7 Grafo de dependências entre variáveis (grafo v-v)

Uma imagem global das dependências entre variáveis, por via das regras que as manipulam, é dada por uma *grafo de dependências entre variáveis*, designado abreviadamente *grafo v-v*, e definido da seguinte forma:

- As variáveis de estado são os vértices do grafo.
- Uma aresta *x ® y*, em que *x* e *y* podem denotar a mesma variável, significa que existe pelo menos uma regra *r* com variável de entrada *x* e variável de saída *y* (pode-se dizer que *y* depende de *x* por intermédio da regra *r*).

Ver um exemplo na figura 4.4.

4.1.8 Grafo de interferências entre regras

O grafo r-r traduz interferências entre regras do tipo escrita-leitura, mas não interferências do tipo escrita-escrita, que por vezes interessa considerar (nomeadamente para a análise de terminação e determinismo do processamento de regras). Os vários tipos de interferências entre regras podem ser traduzidos por um *grafo de interferências* entre regras, definido da seguinte forma:

- As regras são os vértices do grafo.
- Uma aresta *r ® r'*, entre regras diferentes, significa que *r* actualiza uma variável referenciada (lida ou actualizada) por *r'*, isto é, *o-vars(r) Ç vars(r') ¹ {}*.
- Um lacete *r ® r*, significa que *r* actualiza uma variável também lida por *r*, isto é, *io-vars(r) ¹ {}*.

Note-se que este grafo é idêntico ao grafo r-r, acrescido de arestas que ligam regras com variáveis de saída comuns, isto é, regras conflituosas. Os ciclos no grafo de interferências assinalam a presença de regras recursivas ou conflituosas.

O processo de obtenção do grafo de interferências a partir do grafo r-v é ilustrado na figura 4.4. Um exemplo de um grafo de interferências é dado na figura 4.5.

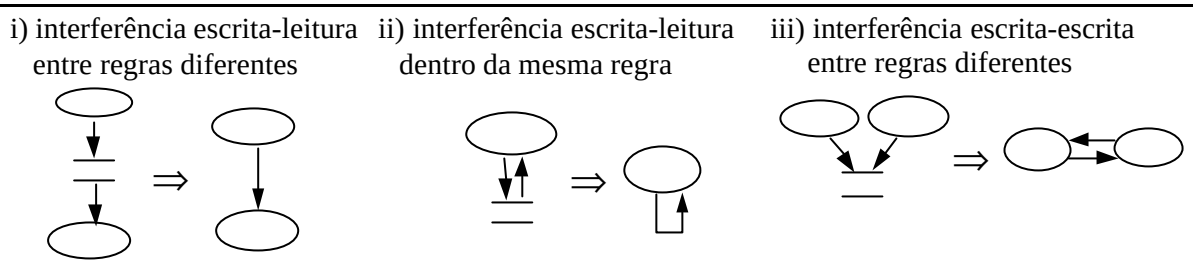


Fig. 4.4 Construção do grafo de interferências entre regras a partir do grafo r-v.

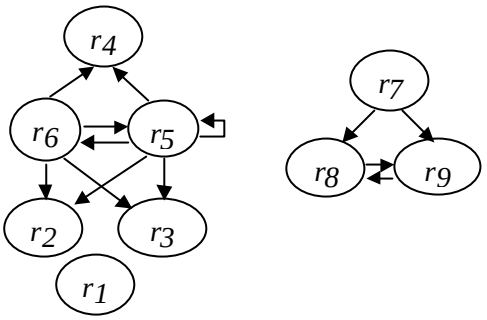


Fig. 4.5 Grafo de interferências entre regras correspondente ao conjunto de regras da fig. 3.1.

4.2 Critérios de activação

Nesta secção são apresentados critérios de activação (isto é, eventos de activação e desactivação) das regras dirigidas pelos dados, do tipo A (só acção) ou CA (condição-acção), necessários para a sua tradução para regras EA ou ECA.

No âmbito das regras dirigidas pelos dados, o primeiro objectivo de um critério de activação é garantir a terminação do algoritmo de processamento de regras sequencial (algoritmo 3.1) depois e só depois de ter sido atingido um ponto fixo para todas as regras. O segundo objectivo de um critério de activação é evitar execuções de regras desnecessárias.

É apresentado primeiro um critério básico, que exige apenas o conhecimento (de natureza essencialmente sintáctica) das variáveis de entrada e de saída de cada regra, e de seguida são discutidos diversos melhoramentos que exigem informação adicional, de natureza sintáctica ou semântica.

4.2.1 Critério de activação básico

Define-se de seguida um critério de activação conservador, baseado apenas no conhecimento estático das variáveis de entrada e saída de cada regra e na monitorização dinâmica dos eventos de alteração de estado das variáveis de estado. A atribuição de um novo valor a uma variável igual ao valor anterior da mesma variável não é considerada uma alteração.

Critério 4.1 (*activação de regras*):

i) Todas as regras "nascem" activadas.

ii) As regras são desactivadas imediatamente antes de serem executadas.

iii) Uma regra *r* que está a ser executada é activada quando uma variável de entrada da regra é alterada. Esta alteração tem de ser causada pela própria regra, porque se proíbem alterações de estado não provocadas pelas regras e as regras são executadas sequencialmente. A variável alterada tem de ser também, como é óbvio, uma variável de saída da regra.

iv) Uma regra *r* que não está a ser executada é activada quando uma variável de entrada ou de saída da regra é alterada (por outras regras ou pelo utilizador).

Justificação:

i) Óbvio, por falta de informação em contrário. Costuma ser implícito nos sistemas de regras activas.
ii) Ver ponto seguinte. Costuma ser implícito nos sistemas de regras activas.
iii) A decisão de que valores são atribuídos a que variáveis quando uma regra *r* é executada depende apenas dos valores iniciais das variáveis de entrada de *r*. Se os valores das variáveis de entrada de *r* não forem alterados durante a sua execução, uma segunda execução de *r* causaria a atribuição exactamente dos mesmos valores às mesmas variáveis. Assim não há necessidade de voltar a executar *r*, razão pela qual pode permanecer desactivada.
iv) Suponhamos que uma regra *r* permaneceu desactivada no final da sua execução. Enquanto nenhuma variável referenciada por *r* (variável de entrada ou de saída) for alterada, uma nova execução de *r* causaria a atribuição dos mesmos valores às mesmas variáveis, sem causar portanto qualquer alteração de estado. Assim, não existe necessidade de executar *r* outra vez, razão pela qual pode permanecer desactivada.

Uma demonstração mais formal de que este critério é correcto, é trivial em face dos resultados da secção seguinte.

As variáveis referidas no ponto iii) são chamadas *variáveis de auto activação* ("self-triggering") de *r*, denotadas *st-vars(r)*. As variáveis referidas no ponto iv) são chamadas *variáveis de activação externa* (*"*external triggering"*)* de *r*, denotadas *et-vars(r)*.

O critério apresentado pode ser traduzido pelo diagrama de estados da figura seguinte. No diagrama não se indicam as transições que ocorrem quando uma transacção é abortada; nessa eventualidade, é necessário repor o estado dos dados e das regras.

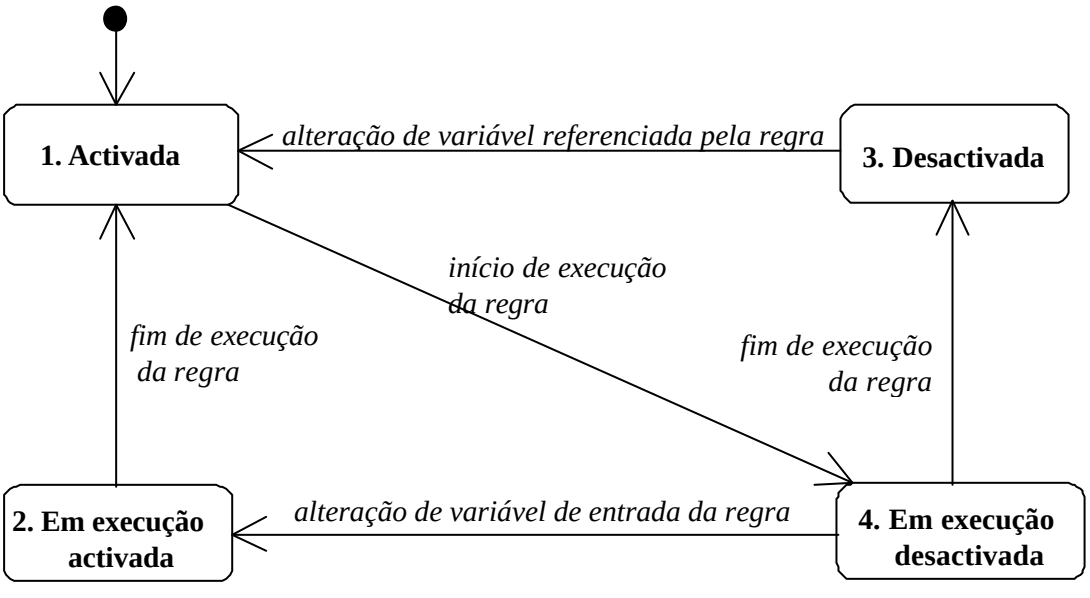


Fig. 4.6 Diagrama de estados de uma regra de acordo com o critério de activação básico, em UML.

4.2.2 Optimizações estáticas baseadas na minimização dos conjuntos de variáveis activadoras

Os conjuntos de variáveis activadoras apresentadas anteriormente podem ser demasiados conservadores. Analisamos de seguida, de um ponto de vista teórico, as seguintes questões: Quais são

os conjuntos mínimos de variáveis activadoras que podem ser considerados e que ainda asseguram uma activação correcta? Como é que estão relacionadas com o efeito líquido, descrito por uma função no espaço de estados? Como é que estão relacionados com os conjuntos mínimos de variáveis de entrada e de saída? Indicam-se também algumas optimizações que podem ser efectuadas na prática.

4.2.2.1 Conjuntos mínimos de variáveis de activação externa

Para ser correcto, qualquer conjunto U de variáveis de activação externa de uma regra r deve obedecer à seguinte condição: se nenhuma variável em U é alterada quando ocorre uma transição de um estado s para um estado t, e s é um ponto fixo de r, então t também é um ponto fixo de r. Formalmente, $\forall s,t \in S,\ s.U=t.U \wedge r(s)=s \Rightarrow r(t)=t$.

É óbvio que só a alteração de variáveis referenciadas na restrição imposta por uma regra r (condição que dá os seus pontos fixos) pode mudar o estado de r de um estado não produtivo (um ponto fixo) para um estado produtivo. Assim, o conjunto mínimo de variáveis de activação externa de uma regra r contém precisamente as variáveis referenciadas na restrição imposta por r. Formalmente,

Teorema 4.2: Qualquer regra r tem um conjunto mínimo único de variáveis de activação externa dado por:

$$min\text{-}et\text{-}vars(r) = \{v \in V: \exists s,t \in S:\ s.(V-\{v\}) = t.(V-\{v\}) \wedge r(s) = s \wedge r(t) \neq t\}.$$

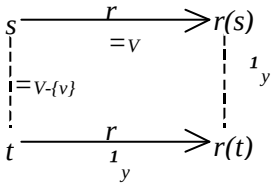
Demonstração: Resulta da definição de conjunto correcto de variáveis de activação externa e da discussão anterior. Δ

Obviamente, a restrição imposta por uma regra r pode ser expressa num subconjunto das variáveis referenciadas por r. Assim, não é de estranhar que:

Teorema 4.3: Para qualquer regra r, $min\text{-}et\text{-}vars(r) \cap min\text{-}i\text{-}vars(r) \cup min\text{-}o\text{-}vars(r)$.

Demonstração: Suponhamos que $v \in min\text{-}et\text{-}vars(r)$ e $v \notin min\text{-}i\text{-}vars(r)$. Então, y e v têm de denotar variáveis distintas no diagrama seguinte:

$v \in min\text{-}et\text{-}vars(r)$ sse $\exists y \in V: \exists s,t \in S$:



Consequentemente, o diagrama mostra que $v \in min\text{-}o\text{-}vars(r)$ (basta trocar s com t e comparar com o diagrama da figura 4.1). Δ

Exemplo

Seja a seguinte regra:

$r:\ y>0 \Rightarrow [(x' = x+1) \wedge (y' = 0)]$

Pelo critério de activação básico, teríamos $et\text{-}vars(r) = vars(r) = \{x, y\}$.

Retirando as pelicas, obtém-se a restrição imposta implicitamente por r:

$y>0 \Rightarrow [(x = x+1) \wedge (y = 0)]$

que pode ser simplificada para

$y \leq 0$.

Assim, toma-se $et\text{-}vars(r) = \{y\}$.

4.2.2.2 Conjuntos mínimos de variáveis de auto activação

Para ser correcto, qualquer conjunto U de variáveis de auto activação de uma regra r deve obedecer à seguinte condição: se nenhuma variável em U é alterada quando r é executada, o estado atingido no final dessa execução é um ponto fixo de r. Formalmente, $\forall s \in S,\ r(s).U=s.U \Rightarrow r^2(s)=r(s)$.

Infelizmente, uma regra pode ter mais do que um conjunto mínimo de variáveis de auto activação, como demonstra o exemplo seguinte.

Exemplo

A regra

$r: y>1 \Rightarrow [(x' = x+1) \vee (y' = y/2)]$

tem dois conjuntos mínimos de variáveis de auto activação: $\{x\}$ e $\{y\}$.

Pelo menos, e como seria de esperar, pode-se garantir que:

Teorem 4.4: Se U é um conjunto mínimo de variáveis de auto activação de uma regra r, então $U \cap \textit{min-i-vars}(r) \neq \emptyset \cap \textit{min-o-vars}(r)$.

Demonstração: Temos de demonstrar que $U \cap \textit{min-o-vars}(r)$ e que $U \cap \textit{min-i-vars}(r)$. A primeira parte é óbvia, porque somente as variáveis que são alteradas quando r é executada podem ser incluídas em U. Mostra-se de seguida que qualquer variável $x \in U$ tem de pertencer a $\textit{min-i-vars}(r)$. Uma vez que U é um conjunto mínimo de variáveis de auto activação de r, a seguinte condição tem de se verificar para qualquer variável $x \in U$:

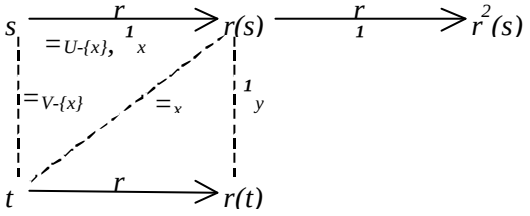
$\exists s \in S: r(s).(U-\{x\}) = s.(U-\{x\}) \wedge r(s).x \neq s.x \wedge r^2(s) \neq r(s)$

Seja t um estado obtido a partir de s da seguinte forma:

$t.(V-\{x\}) = s.(V-\{x\}) \wedge t.x = r(s).x$.

Esta condição, em conjunto com a condição $r(s).(U-\{x\}) = s.(U-\{x\})$, implica que $t.U = r(s).U$. Suponhamos que $r(t) = r(s)$. Então $r(t).U = r(s).U = t.U \wedge r^2(t) \neq r(t)$. Isto é impossível porque U é um conjunto correcto de variáveis de auto activação. Assim $r(t) \neq r(s)$, ou, equivalentemente, $\exists y \in V: r(s).y \neq r(t).y$. As conclusões obtidas até ao momento podem ser traduzidas pelo seguinte diagrama:

$\forall x \in U, \exists s,t \in S, \exists y \in V$:



Para provar que x pertence a $\textit{min-i-vars}(r)$, de acordo com a expressão do teorema 4.1, temos de provar que $r(s).y \neq s.y$ (ou equivalentemente $r(t).y \neq t.y$, trocando s com t). Se y e x denotam a mesma variável, a conclusão é imediata porque $r(s).x \neq s.x$. Se x e y denotam variáveis distintas, então tem de ser $s.y = t.y$ (porque s e t diferem apenas em x); isto, em conjunto com $r(s).y \neq r(t).y$, implica que $r(s).y \neq s.y$ ou $r(t).y \neq t.y$. Assim, conclui-se que $x \in \textit{min-i-vars}(r)$. Δ

4.2.2.3 Optimizações estáticas para regras condicionais

No caso de regras condicionais, basta considerar para $\textit{st-vars}(r)$ as variáveis que são simultaneamente entradas e saídas da parte de acção (i.e., pode-se ignorar a parte de condição). Este critério exige a determinação separada das variáveis de entrada da parte de acção e da parte de condição.

Exemplo

Seja a seguinte regra:

$r: x>y \Rightarrow y' = x$

Pode-se tomar *st-vars(r)={}*. Pelo critério básico, ter-se-ia *st-vars(r)={y}*.

4.2.2.4 Optimizações estáticas para regras idempotentes

Uma regra *r* diz-se *idempotente* quando $r^2(s)=r(s)$ para qualquer estado *s*. Isto significa que um ponto fixo é atingido numa única execução da regra. Uma regra idempotente tem, obviamente, um conjunto mínimo de variáveis de auto activação vazio, e vice-versa. Sabendo-se que uma regra *r* é idempotente, toma-se *st-vars(r)={}*.

Regras com conjuntos disjuntos de variáveis de entrada e de saída (i.e., regras que não são auto-recursivas) são trivialmente idempotentes. Regras condicionais em que a execução da acção torna sempre falsa a condição, também são idempotentes. No caso geral, é necessária uma análise semântica para determinar se uma regra é idempotente. Por exemplo, a regra r_5 na figura 3.1 é idempotente (as outras regras na mesma figura são trivialmente idempotentes).

Quase todas as regras com interesse prático são idempotentes. Devido à importância desta propriedade e à dificuldade da sua determinação automática, deve existir um mecanismo ao dispor do programador de regras para indicar que uma dada regra é idempotente.

4.2.2.5 Optimizações estáticas baseadas numa análise de fluxo de dados

Numa linguagem prática, e com base apenas numa análise de fluxo de dados, podem-se excluir de *st-vars(r)* as variáveis que não são actualizadas depois de serem lidas (na parte de acção da regra).

Exemplo

Suponhamos que a seguinte regra

r: (x>0 Þ y' = x) Ù (x£0 Þ x' = y)

é traduzida para a seguinte função em C

```
void r() {if (x>0) y = x; else x = y;}
```

Neste caso, *i-vars(r) = o-vars(r) = {x, y}*. Pelo critério básico, *st-vars(r)={x, y}*. Uma vez que *y* não é actualizado depois de ser lido, pode ser removido de *st-vars(r)*. Na realidade, uma análise semântica revela que a regra é idempotente, pelo que se poderia tomar *st-vars(r)={}*.

4.2.3 Optimizações dinâmicas baseadas no valor da condição

Claramente, as variáveis de activação externa de uma regra condicional podem ser restringidas dinamicamente da seguinte forma:

- Se uma regra *r* é executada e a condição é falsa, tomar subsequentemente para *et-vars(r)* as variáveis referenciadas na parte de condição.
- Se uma regra *r* é executada e a condição é verdadeira, tomar subsequentemente para *et-vars(r)* as variáveis referenciadas na parte de acção.

Inicialmente, o valor da condição não é conhecido, mas isso não é problema porque se considera que as regras nascem activadas. Estas optimizações estão traduzidas no diagrama da figura 4.7.

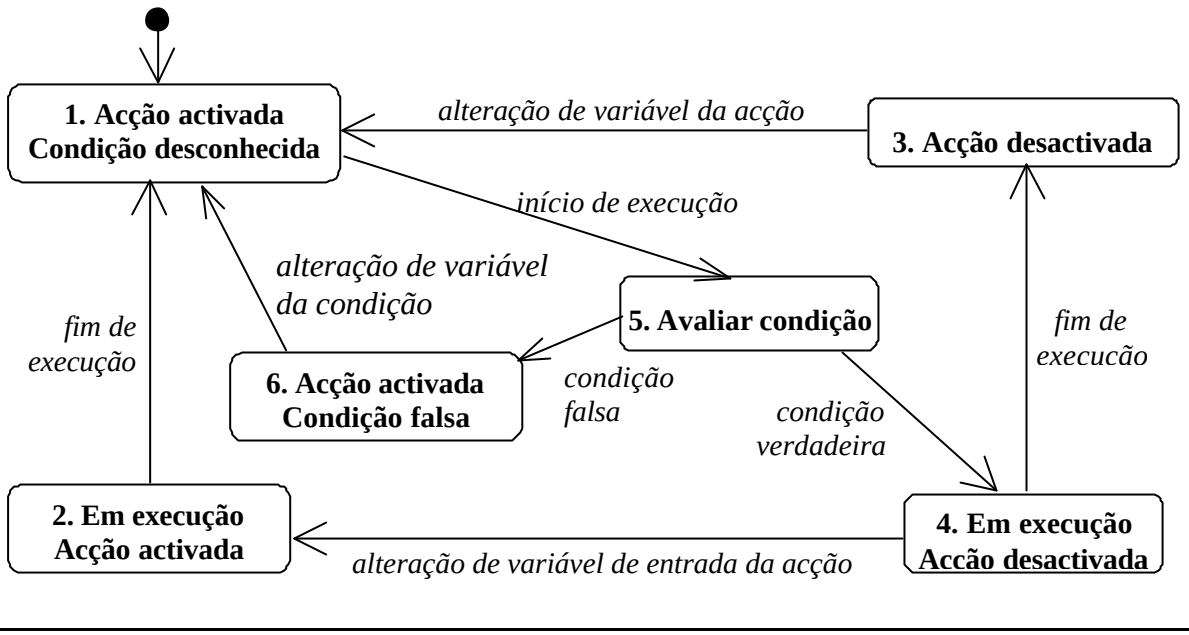


Fig. 4.7 Diagrama de estados optimizado para regras condicionais.

Numa linguagem prática, é necessário que as partes de condição e de acção possam ser avaliadas separadamente, e que não haja passagem de informação da parte de condição para a parte de acção (nomeadamente através de variáveis locais).

Exemplo

Seja a seguinte regra:

r: y=1 Þ w' = 1

Se *r* for executada e a condição for verdadeira, toma-se subsequentemente $et\text{-}vars(r_3)=\{w\}$. Caso contrário, toma-se $et\text{-}vars(r_3)=\{y\}$.

Esta optimização tem, no entanto, as seguintes desvantagens:

- obriga a manter informação de estado junto com as regras de forma permanente entre transacções, o que é pouco conveniente no caso de regras intra-objecto a ver mais adiante;
- a implementação deste critério em sistemas de regras activas conhecidos só parece possível nos poucos sistemas que permitem misturar eventos primitivos e condições através de eventos compostos, como é o caso do sistema Ode (conforme vimos na secção 2.3.6.5).

4.2.4 Optimizações dinâmicas baseadas na monitorização de eventos de leitura e escrita

Na prática, pode-se optimizar o critério de activação básico, se forem monitorados os eventos de leitura (consulta) e escrita (actualização) de dados, para além dos eventos de alteração de dados, dispensando até o conhecimento estático das variáveis de entrada e de saída de cada regra.

A ideia é associar a cada regra conjuntos dinâmicos de entradas e saídas que evoluem da seguinte forma:

- Inicialmente, os conjuntos dinâmicos de entradas e saídas estão vazios.
- Os conjuntos dinâmicos de entradas e saídas são esvaziados no início da execução de cada regra.
- Uma variável *x* é adicionada ao conjunto dinâmico de entradas de uma regra *r* quando é lida durante a execução de *r*, mas só no caso de não estar já no conjunto dinâmicos de saídas de *r* (i.e., ignoram-se leituras depois de escritas).
- Uma variável *y* é adicionada ao conjunto dinâmico de saídas de uma regra *r* quando é escrita durante a execução de *r* (antes de tratar o possível evento de alteração).

- No caso de regras condicionais, se a condição for verdadeira, o conjunto dinâmico de entradas é esvaziado antes de executar a acção.

O critério básico ainda se aplica, mas com base nos conjuntos dinâmicos, em vez dos conjuntos estáticos.

Este critério tem, no entanto, as seguintes desvantagens:

- exige a monitorização de leituras e escritas (embora a monitorização de escritas já tinha de ser feita de alguma forma para detectar as alterações);
- a variação dinâmica dos eventos activadores dificulta a implementação deste critério em sistemas de regras activas conhecidos (em que os eventos activadores de cada regra são fixos, embora possam ser compostos);
- não é facilmente extensível para regras intra-objecto (a ver adiante), porque seria necessário manter conjuntos de variáveis diferentes para instâncias diferentes da mesma regra.

Assim, este critério só é útil no caso de não ser possível efectuar uma análise sintáctica prévia das regras.

4.2.5 Activação apenas por alteração de variáveis de entrada

Na prática, é geralmente desnecessário considerar a activação externa das regras por alteração das suas variáveis de saída, bastando considerar a activação (tanto externa como interna) por alteração das variáveis de entrada, pelas seguintes razões:

- Geralmente, convém impedir, por um sistema de permissões, que o utilizador altere o valor de itens de dados que são calculados por regras. No caso de itens de dados calculados condicionalmente, a própria permissão de alterar esses itens de dados pode ser calculada pela mesma regra, conforme ilustra o 1º exemplo a seguir.

- No caso de regras auto-recursivas, nomeadamente no caso de regras de conversão ou correcção de entradas, não faz sentido impedir que o utilizador altere as variáveis calculadas recursivamente. Mas não é necessário considerar a activação dessas regras por alteração de variáveis de saída, porque se trata também de variáveis de entrada.

- No caso de regras mutuamente recursivas que impõem a mesma restrição de diferentes maneiras, basta a execução de uma das regras para impor a restrição. Como interessa preservar, sempre que possível, os valores introduzidos pelo utilizador, não interessa executar a regra que recalcula uma variável alterada pelo utilizador, mas sim a regra (ou uma das regras) que lê o valor introduzido pelo utilizador, conforme ilustra o 2º exemplo a seguir.

- As regras conflituosas (regras que actualizam a mesma variável de estado) com interesse prático não contêm definições contraditórias que exijam a activação de uma regra quando uma variável de saída é alterada por outra regra, conforme ilustram os exemplos dados no capítulo 3.

Por conseguinte, deve existir um mecanismo ao dispor do programador de regras que lhe permita dizer que uma dada regra (ou todas as regras) não é activada pela alteração das suas variáveis de saída.

Exemplo 1

Suponhamos que *locked(x)* designa uma variável de estado, de valor booleano, que indica a proibição de o utilizador alterar o valor de um item de dados representado pela variável de estado *x*.

Para impedir que o valor de *x* seja alterado pelo utilizador quando é calculado por uma regra *r*, pode-se escrever a regra na forma geral:

r: (condição Þ x' = expressão) Ù (locked(x)' = condição)

Traduzindo para uma função em C (note-se que "=" é o operador de atribuição):

```
void r() {if (locked(x) = condição) x = expressão;}
```

Exemplo 2

As duas regras seguintes (transcritas da figura 3.1), impõem a mesma restrição de diferentes maneiras:

r_8: $b' = a+c$

r_9: $c' = b-a$

No caso do utilizador alterar o valor de b, interessa activar apenas a regra r_9 (que lê b), para que o valor de c seja calculado em função de b. Não interessa activar também a regra r_8 (que escreve em b) porque, se esta for executada em primeira lugar, a operação do utilizador é perdida. Acontece uma situação simétrica quando o utilizador altera o valor de c.

4.2.6 Política de efeito líquido

Para garantir que apenas o efeito líquido das regras é importante, é necessário seguir uma das seguintes *políticas de efeito líquido*:

- Proibir regras que produzem alterações de estado que se cancelam mutuamente na mesma execução.
- As alterações que se cancelam mutuamente, ocorridas durante a execução de uma regra, são detectadas e ignoradas pelo mecanismo de monitorização de eventos. Esta política é mais segura do que a anterior, mas exige que se guarde de alguma forma o valor antigo (no início da execução da regra) de cada variável alterada durante a execução de uma regra. Estes valores antigos podem ser libertados no fim da execução da regra.
- Para efeito da activação de cada regra, só são consideradas as alterações ocorridas em sentido líquido (i.e., ignorando alterações que se cancelam mutuamente) após o início da última execução dessa regra, caso tenha já sido executada na transacção em causa, ou após o início da transacção, no caso contrário. Esta política é a mais comum em SGBDA's que implementam uma política de efeito líquido, como nos sistemas Starbust [W96] e Chimera [ZCF+97]. Pode conduzir a menos activações de regras do que a política anterior, mas exige que se guarde muito mais informação, pelo que não é de considerar a não ser que essa informação seja útil por outros motivos.

Se não for seguida uma destas políticas (ou outra semelhante), o algoritmo de processamento de regras pode não terminar, mesmo depois de se ter atingido um ponto fixo para todas as regras.

5 Ordenação das regras

Neste capítulo desenvolvem-se os critérios de ordenação das regras definidas no capítulo 3. Esses critérios são baseados principalmente no conhecimento das dependências de entrada e saída de dados existentes entre as variáveis de estado e as regras, dependências essas já referidas no capítulo 4. São identificados vários critérios complementares ou alternativos, com diferentes forças. É demonstrada a relevância desses critérios, principalmente em termos de eficiência mas também em termos de semântica do processamento de regras, de forma genérica ou para conjuntos de regras que obedecem a certas propriedades.

5.1 Introdução

A ordem pela qual as regras são consideradas para execução no algoritmo de processamento de regras, quando há mais do que uma regra activada, pode ter impacto em termos de *semântica* (a terminação do processamento de regras, e em caso de terminação, o estado final atingido) e em termos de *eficiência* do processamento de regras (medida, na falta de mais informação, pelo número de vezes que cada regra é executada).

Assim, interessa definir critérios de ordenação das regras destinados a promover boas propriedades do processamento de regras, em termos de semântica e de eficiência.

Habitualmente, em sistemas de regras activas, as regras são ordenadas, de forma total ou parcial, através de prioridades absolutas ou prioridades relativas (também chamadas precedências).

Num esquema de *prioridades absolutas* é associada uma prioridade numérica a cada regra. Durante o processamento de regras, uma regra activada r não pode ser seleccionada para execução (não é candidata a execução) se existir outra regra activada r' com prioridade numérica superior à de r. Se existirem várias regras com a mesma prioridade, a ordenação estabelecida é parcial; caso contrário, a ordenação é total, o que implica que só pode existir uma regra candidata a execução em cada momento.

Num esquema de *prioridades relativas* são estabelecidas precedências entre pares de regras, por afirmações do género "r' tem precedência sobre r". Também se usa a notação $r' > r$ com o mesmo significado. Durante o processamento de regras, uma regra activada r não pode ser seleccionada para execução (não é candidata a execução) se existir outra regra activada r' com precedência sobre r. Estas precedências podem ser consideradas *transitivas* ou não. Suponhamos que são definidas as precedências $r'' > r'$ e $r' > r$, e que o conjunto de regras activadas num dado passo do processamento de regras é $\{r'', r\}$. Se as precedências forem consideradas transitivas, a precedência $r'' > r$ é implicitamente implicada pelas anteriores, pelo que só a regra r'' pode ser seleccionada para execução. Se as precedências não forem consideradas transitivas, qualquer das regras pode ser seleccionada para execução. Em qualquer caso, exige-se sempre que as precedências sejam *acíclicas*, para garantir que, seja qual for o conjunto de regras activadas, é sempre possível seleccionar uma regra para execução.

O esquema de prioridades relativas é mais flexível, porque as prioridades absolutas podem ser traduzidas para prioridades relativas, enquanto que o inverso não é verdadeiro (pelo menos sem introduzir precedências para além das especificadas). Em contrapartida, as prioridades absolutas são úteis para definir de forma expedita prioridades entre grupos de regras. Assim, ambos os esquemas são úteis.

Para facilitar a execução das regras dirigidas pelos dados em combinação com outras regras activas, convém utilizar esquemas de prioridades deste género. Idealmente, no caso de terem impacto em termos de semântica, as prioridades estabelecidas devem ter um significado claro para o programador de regras.

5.2 Ordenação pelo princípio calcular antes de usar

Intuitivamente, o princípio "calcular antes de usar" deve ser seguido sempre que possível. A ideia é a seguinte: se uma variável *x* é calculada por uma regra *r*, e é usada (lida) por outra regra *r'*, então deve-se primeiro calcular o valor de *x*, executando a regra *r*, antes de usar o valor de *x*, executando a regra *r'*. Se o grafo r-r for acíclico, isto é, se não existirem regras recursivas, é sempre possível aplicar este princípio, que corresponde à execução das regras por ordem topológica do grafo r-r. O princípio não é aplicável, no entanto, entre regras recursivas.

Assim, preconiza-se o seguinte esquema de prioridades relativas:

Critério 5.1: Uma regra *r* tem precedência sobre uma regra *r'* (i.e., *r* > *r'*) se existir um caminho de *r* para *r'*, e nenhum caminho em sentido contrário, no grafo r-r (ou, equivalentemente, no grafo r-v).

Conforme foi explicado anteriormente, a precedência significa que, se ambas as regras estiverem activadas, a regra *r'* não pode ser seleccionada para execução. A inexistência de um caminho em sentido contrário, de *r'* para *r*, é exigida para evitar a geração de precedências circulares no caso do grafo r-r conter ciclos.

Este critério é fácil de aplicar, porque se baseia apenas na informação de natureza sintáctica traduzida no grafo r-r. Além disso, garante as "boas" propriedades do processamento de regras indicadas no teorema seguinte.

Teorema 5.1: Se as regras forem executadas de acordo com o esquema de prioridades relativas do critério 5.1,

a) qualquer regra *r* não recursiva (auto-recursiva ou mutuamente recursiva com outras regras) só é executada depois das suas variáveis de entrada terem atingido valores finais (por outras palavras, as variáveis de entrada de *r* não são alteradas após o início da primeira execução de *r*);
b) qualquer regra *r* não recursiva e não conflituosa é executada no máximo uma vez em cada ponto de processamento de regras (PPR).

Demonstração:

a) Seja *r* uma regra não recursiva. Seja *Pred(r)* o conjunto de regras com precedência sobre *r*. Uma vez que *r* não é mutuamente recursiva com outras regras, *r' Î Pred(r)* se e só *r¹ 'r* e existe um caminho de *r'* para *r* no grafo r-r. Assim, pelo princípio calcular antes de usar, quando *r* é seleccionada para execução, não pode estar activada nenhuma *r' Î Pred(r)*. Uma vez que *r* não é auto-recursiva, não altera as suas próprias entradas. Uma regra *r' Î Pred(r)* pode, no entanto, ser activada mais tarde. Seja *r'* a primeira regra activada nessas circunstâncias, e suponhamos que *r'* é activada por um evento gerado por uma regra *r''*. Uma vez que *r'* é a 1ª regra activada pertencente a *Pred(r)*, *r'' Ï Pred(r)*. Suponhamos que o evento activador gerado por *r''* é a alteração de uma variável de entrada de *r'* , digamos *x*. Nesse caso, existiria um caminho de *r''* para *r* no grafo r-r por intermédio de *x* e *r'*, o que contradiz o facto de que *r'' Ï Pred(r)*. Assim, o evento activador gerado por *r''* só pode ser a alteração de uma variável de saída de *r'*, digamos *y*. Não pode existir um caminho de *y* para *r* no grafo r-v; caso contrário, existiria um caminho de *r''* para *r* no grafo r-v por intermédio de *y*, e, portanto, um caminho de *r''* para *r* no grafo r-r. Uma vez que, nem as variáveis de entrada de *r'* nem as suas variáveis de saída com um caminho para *r* podiam ter sido alteradas (desde a execução de *r*), as variáveis de saída de *r'* com um caminho para *r* não são modificadas quando *r'* é executada.

Consequentemente, a execução de *r'* não pode alterar nenhuma variável no caminho para *r*, incluindo portanto as variáveis de entrada de *r*.

b) Seja *r* uma regra não conflituosa e não recursiva. De acordo com a alínea a), as variáveis de entrada de *r* não são alteradas após o início da primeira execução de *r* num dado PPR. Por definição de regra não conflituosa, não existe nenhuma outra regra *r'*, com variáveis de saída em comum com *r*. Assim, as variáveis de saída de *r* também não são alteradas por outras regras. Uma vez que as variáveis de entrada de *r* não são alteradas após o início da execução de *r* (por *r* ou por outras regras), e as variáveis de saída de *r* não são alteradas por outras regras, a regra *r* não pode ser reactivada. Supõe-se, claro, que é seguido pelo menos o critério de activação básico indicado no capítulo anterior. Δ

Devido à propriedade a), este critério produz, em geral, a semântica esperada. Em particular, as regras de restrição (que nunca são recursivas) testam apenas valores finais das suas variáveis de entrada.

Além disso, veremos no capítulo 6 que, na ausência de regras recursivas, este critério fixa a semântica do conjunto de regras. Isto é, na ausência de regras recursivas, o processamento de regras é determinístico, no sentido de que a sua terminação e, em caso de terminação, o estado final atingido, não dependem de qual é a regra seleccionada para execução quando há várias regras candidatas a execução num dado passo do processamento. Considera-se que uma regra *r* é candidata a execução se estiver activada e não existir nenhuma outra regra activada *r'* com precedência sobre *r*.

5.2.1 Relação com ordem topológica de componentes fortemente conexos do grafo r-r

Na ausência de ciclos no grafo r-r, uma ordenação total das regras de acordo com o critério 5.1 é uma ordem topológica do grafo r-r. Na presença de ciclos, uma possível ordenação total das regras de acordo com o critério 3.1 é uma ordenação das regras por ordem topológica de componentes fortemente conexos do grafo r-r.

Um *componente fortemente conexo* (CFC) *C* de um grafo dirigido *G=(V,E)* é um sub-grafo máximo tal que, para quaisquer dois vértices distintos *u* e *v* de *C*, existem caminhos de *u* para *v* e de *v* para *u* em *G* (e, consequentemente, em *C*) [TS92]. Um CFC com um único vértice (com ou sem lacete) diz-se *trivial*. Dado um grafo *G=(V,E)*, pode-se construir um grafo condensado *G'=(V',E')* cujos vértices são os CFC's de *G*, e cujas arestas ligam CFC's que contêm vértices ligados entre si em *G*. O grafo *G'* é acíclico, pelo que ser ordenado topologicamente. Dispondo os CFC's por ordem topológica, e substituindo cada CFC pelos vértices nele contidos (dispostos por uma ordem qualquer), tem-se o que se chama uma ordenação dos vértices de *G=(V,E)* por ordem topológica de CFC's. Ver exemplos nas duas figuras seguintes.

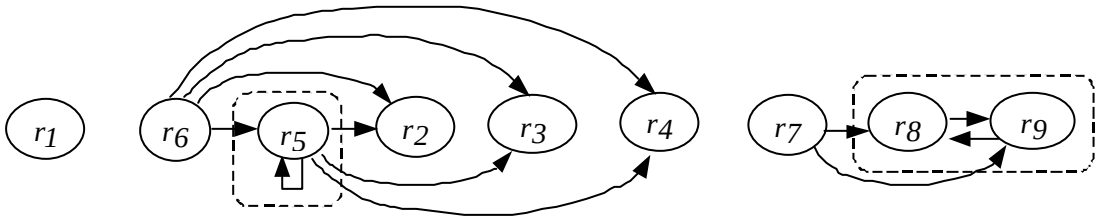


Fig. 5.1 Grafo r-r da fig. 4.3 com ciclos aglutinados em componentes fortemente conexos (CFC's) e regras dispostas por ordem topológica de CFC's e, secundariamente, por números crescentes.

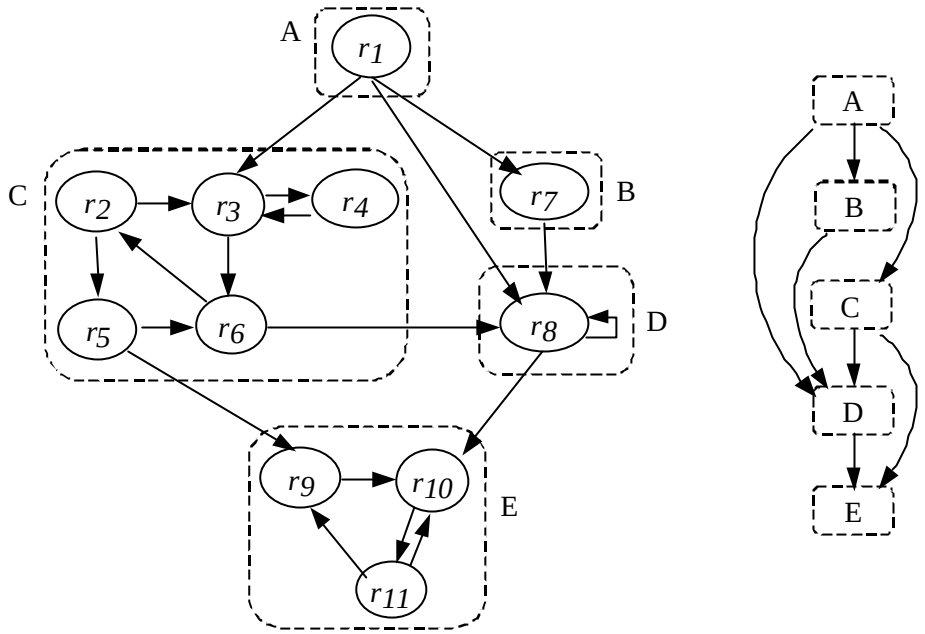


Fig. 5.2 Um grafo dirigido cíclico (à esquerda) e grafo condensado acíclico dos seus CFC's já dispostos por uma ordem topológica (à direita).

Uma vez que os CFC's de um grafo $G=(V,E)$ podem ser obtidos por ordem topológica em tempo linear no tamanho do grafo, isto é, em tempo $O(|V|+|E|)$ (usando por exemplo um algoritmo como o apresentado em [RNN77]), a ordenação dos vértices de um grafo por ordem topológica de CFC's pode também ser feita em tempo O(|V|+|E|).

A execução de regras dirigidas pelos dados por ordem topológica do grafo de dependências r-v também é proposta no projecto PARDES, mas aí não surgem ciclos. Regras dedutivas também são habitualmente executadas por ordem topológica de CFC's [RSS90].

No caso de se pretender combinar o critério 5.1 com outros critérios, como é normalmente o caso, não convém fixar uma ordem total com base neste critério apenas.

5.2.2 Ordenação de regras mutuamente recursivas

Regras mutuamente recursivas (pertencentes a CFC's não triviais do grafo r-r) não são ordenadas pelo critério 5.1. Nas secções seguintes descrevem-se diversas heurísticas que podem ser usadas isoladamente ou de forma combinada para ordenar tais regras, por forma a dispensar, na maior parte dos casos, a definição de prioridades absolutas ou relativas pelo próprio programador.

5.3 Preservação das alterações produzidas pelo utilizador

5.3.1 Preservação das alterações produzidas na mesma transacção

Conforme já foi referido, as variáveis alteradas pelo utilizador numa dada transacção podem ser de novo alteradas no processamento de regras subsequente, sem que isso seja considerado um erro. Quando isso acontece, diz-se que as alterações produzidas pelo utilizador não são preservadas pelas regras. Este comportamento é importante para suportar regras de conversão ou correcção de entradas.

No caso deste comportamento não ser desejável, isto é, no caso de se pretender que os dados introduzidos pelo utilizador numa dada transacção sejam obrigatoriamente preservados pelas regras, o processamento de regras e a transacção devem ser abortados assim que alguma regra tenta alterar o

valor de uma variável alterada pelo utilizador na mesma transacção (antes do processamento de regras).

Em qualquer dos casos, é desejável que as alterações produzidas pelo utilizador numa dada transacção sejam preservadas pelas regras (no PPR subsequente), sempre que possível. O grau em que isso acontece (e, portanto, a semântica do processamento de regras) pode depender da ordem por que as regras são executadas. Um caso extremo de não preservação das alterações produzidas pelo utilizador ocorre quando uma regra de restrição aborta o processamento de regras. Ocorrências desse tipo já são minimizadas pelo critério 5.1, pois as regras de restrição só testam valores finais das suas variáveis de entrada. Mas há outros casos, envolvendo regras de derivação mutuamente recursivas, que não são cobertos pelo referido critério.

Um critério de ordenação que, obviamente, tende a preservar as alterações produzidas pelo utilizador, é o seguinte:

Critério 5.2: As regras que não contêm variáveis de saída alteradas pelo utilizador (na mesma transacção do PPR) têm precedência sobre as restantes.

Note-se que as precedências estabelecidas por este critério são dinâmicas, porque variam de PPR para PPR. Este critério é traduzido mais facilmente por uma prioridade numérica com dois valores, estabelecida em cada transacção. Seguem-se exemplos que mostram a eficácia deste critério.

Exemplo

Sejam as seguintes regras mutuamente recursivas da figura 5.1 (transcritas da figura 3.1):

r_8: $b' = a + c$ $\qquad$ r_9: $c' = b - a$

Estas regras impõem a restrição $b = a + c$ de diferentes maneiras, isto é, com diferentes escolhas de variáveis derivadas e variáveis primitivas.

Suponhamos que o utilizador altera o valor de c e, de seguida, as regras são processadas. Se for seguido o critério de activação básico, ambas as regras são activadas. Uma vez que as regras são mutuamente recursivas, o critério 5.1 não tem qualquer efeito. Já pelo critério 5.2, a regra r_8 tem precedência sobre a regra r_9. Quando a regra r_8 é executada, o valor de b é recalculado. Seguidamente, a regra r_9 é executada, não produzindo qualquer alteração. Assim, o valor de c é preservado. Uma situação simétrica ocorre se o utilizador começar por actualizar o valor de b. Note-se, no entanto, que se obteria o mesmo resultado se as regras fossem activadas apenas por alteração das entradas, conforme se explicou no capítulo 4.

Exemplo

Suponhamos que a restrição $x = y = z$ é impostas de múltiplas maneiras pelas seguintes regras:

r_1: $y' = x$ $\qquad$ r_2: $z' = y$ $\qquad$ r_3: $x' = z$

Suponhamos que o utilizador altera o valor de x, causando a activação das regras r_1 e r_3. Uma vez que x é uma variável de saída de r_3 mas não de r_1, a regra r_1 é executada em 1º lugar, actualizando (com alteração) o valor de y. Esta alteração causa a activação da regra r_2. Uma vez x é uma variável de saída de r_3 mas não de r_2, a regra r_2 é executada, actualizando (com alteração) o valor de z. Esta alteração causaria a activação de r_3, mas esta já se encontrava activada. Finalmente, é executada a regra r_3, sem qualquer efeito, e o processamento de regras termina. Assim, o valor de x é preservado conforme pretendido. Ver ilustração na na figura 5.3.

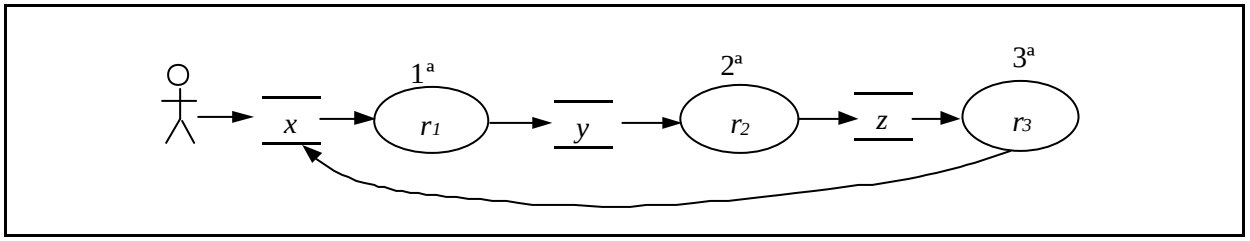


Fig. 5.3 Grafo r-v de um conjunto de regras, com a ordem de execução das regras em resposta a um comando do utilizador de acordo com o critério 5.2.

Exemplo

Suponhamos que x e y representam uma data inicial e uma data final, respectivamente, tendo de obedecer à restrição de desigualdade $x \leq y$. As seguintes regras impõem esta restrição de diferentes maneiras:

r_1: $x>y \rightarrow x'=y$ $\quad$ r_2: $x>y \rightarrow y'=x$

Se o utilizador introduzir um valor de x superior ao valor corrente de y, o valor de y será "empurrado" para cima pela regra r_2. Se o utilizador introduzir um valor de y inferior ao valor corrente de x, o valor de x será "empurrado" para baixo pela regra r_1. Note-se que, neste caso, mesmo que as regras fossem activadas só por alteração das entradas, ambas as regras seriam activadas.

5.3.2 Preservação das alterações produzidas em transacções mais recentes

Embora com muito menos peso, é também desejável que as alterações produzidas pelo utilizador em transacções passadas sejam preservadas, tanto mais quanto mais recentes forem essas transacções. Dessa forma, o utilizador pode controlar o estado do sistema por aproximações (transacções) sucessivas.

O refinamento correspondente do critério 5.2 é:

Critério 5.3: As regras com variáveis de saída mais recentemente alteradas pelo utilizador (i.e., em transacções mais recentes) perdem precedência em relação às restantes.

Este critério é de natureza dinâmica, pois as precedências variam de PPR para PPR (embora sejam fixas em cada PPR). É mais facilmente traduzido por uma prioridade numérica atribuída a cada regra, de forma inversamente proporcional à estampa temporal da transacção mais recente em que uma variável de saída da regra foi alterada pelo utilizador.

Exemplo

Suponhamos que a restrição $z=x+y$ é imposta de múltiplas maneiras pelas seguintes regras:

r_1: $z' = x+y$ $\quad$ r_2: $y' = z-x$ $\quad$ r_3: $x' = z-y$

Suponhamos que o utilizador altera o valor de x, numa transacção, e o valor de y noutra transacção a seguir. Na 1ª transacção, a regra r_1 ou a regra r_2 é executada em 1º lugar, preservando em qualquer dos casos o valor de x. Na 2ª transacção, a regra r_1 é executada em 1º lugar, preservando os valores de x e de y. O resultado seria o mesmo se o utilizador actualizasse os valores de x e de y numa única transacção. Em geral, se o utilizador fixar os valores de qualquer par de variáveis, o valor da terceira variável é calculado.

Exemplo

Para impor a seguinte restrição de igualdade não linear entre quantidade (q), preço unitário (p) e preço total (t)

$t = q \times p$

resolve-se a restrição em ordem a cada uma das variáveis, definindo-se as seguintes regras:

r_1: $p \neq 0 \rightarrow q' = t/p$ $\quad$ r_2: $q \neq 0 \rightarrow p' = t/q$ $\quad$ r_3: $t' = q \times p$

Suponhamos que estas variáveis têm valores iniciais nulos, consistentes com as regras. Se o critério 5.3 for seguido, o utilizador consegue preencher qualquer par de valores (um de cada vez), que o terceiro é calculado em função deles, preservando os valores introduzidos pelo utilizador. A única excepção é que o utilizador não consegue preencher o total em primeiro lugar, porque o total é de novo anulado pela regra r_3.

5.3.3 Combinação com o princípio calcular antes de usar

O efeito do critério 5.2 é adiar a execução das regras que poderiam contradizer os valores introduzidos pelo utilizador, na "esperança" de que, se essas regras forem executadas só a seguir a todas as outras, há menos hipóteses de contradizerem os valores introduzidos pelo utilizador (uma vez que se dá tempo a que sejam "influenciadas" pelos valores introduzidos pelo utilizador).

No caso de regras não recursivas, o critério 5.1 já adia a sua execução até ao momento em que as suas variáveis de entrada atingiram valores finais. Assim, não há qualquer vantagem em adiar a sua execução ainda mais, dando mais força ao critério 5.2 do que ao critério 5.1.

No caso de uma regra auto-recursiva *r*, mas não mutuamente recursiva com outras regras, o critério 5.1 já adia a sua execução até ao ponto em que mais nenhuma regra pode alterar os valores das variáveis de entrada de *r*. Assim, também neste caso, não há qualquer vantagem em adiar a sua execução ainda mais.

Por conseguinte, ou se restringe a aplicação do critério 5.2 a regras mutuamente recursivas com outras regras (regras que pertencem a CFC's não triviais do grafo r-r) ou, o que é quase equivalente e mais prático, aplica-se o critério 5.2 em todos os casos, mas com menos força do que o critério 5.1.

Por razões óbvias, o critério 5.3 deve ter ainda menos força do que o critério 5.2.

5.4 Ordenação de regras redundantes puramente incondicionais

O critério 5.1 não estabelece nenhuma ordenação entre as regras (mutuamente recursivas) pertencentes a um CFC não trivial do grafo r-r. Intuitivamente, interessa minimizar de alguma forma as violações ao princípio calcular antes de usar, para tornar mais eficiente o processamento de regras.

Nesta secção é descrito um critério que minimiza de uma certa forma as violações ao princípio calcular antes de usar e permite tratar eficazmente certos conjuntos de regras recursivas.

Em muitos casos (de acordo com a experiência do autor), o conjuntos de regras recursivas é tal que um ponto fixo para todas as regras se pode atingir com uma única execução de cada regra, desde que as regras sejam executadas por uma ordem apropriada. Interessam-nos aqui, em especial, os casos em que isso é possível mesmo quando se usa o critério de activação básico. Esses casos estão relacionados com a presença de regras *redundantes* que impõem as mesmas restrições de diferentes maneiras.

5.4.1 Regras redundantes que impõem as mesmas restrições de maneiras diferentes

Uma regra *r* diz-se *redundante* (em sentido estático) em relação a um conjunto *R′* de regras que não inclui *r*, se os pontos fixos de *R′* (definidos como pontos fixos para todas as regras de *R′*) são também pontos fixos de *r*, ou, equivalentemente, se a restrição imposta por *r* é implicada pela restrição imposta por *R′* (definida como a conjunção das restrições impostas pelas regras de *R′*). Formalmente, $\forall s \in S, [\forall r' \in R', r'(s)=s] \Rightarrow r(s)=s$.

Duas regras *r* e *r′* dizem-se mutuamente redundantes, se qualquer delas é redundante em relação à outra. Nesse caso, têm os mesmos pontos fixos.

No caso de regras com mais do que uma variável de saída, a redundância pode ser definida em relação a cada uma das variáveis de saída. Diz-se que uma regra *r* é *redundante numa variável* de saída *y* em relação a um conjunto *R′* de regras que não inclui *r*, se o valor de *y* não é alterado quando *r* é executada num ponto fixo de *R′*.

As regras redundante só são úteis quando impõem as mesmas restrições de maneiras diferente, isto é, com diferentes escolhas de variáveis derivadas e primitivas. Uma vez que as escolhas de variáveis derivadas e primitivas são diferentes, estas regras originam ciclos no grafo r-v.

Exemplo
No caso das regras que impõem a restrição $b=a+c$,

r_8: $b' = a+c$ r_9: $c' = b-a$

as regras r_8 e r_9 são mutuamente redundantes.

Exemplo
No caso das regras usadas para impor a restrição $t = q \acute{} p$,

r_1: $p \neq 0 \Rightarrow q' = t/p$ r_2: $q \neq 0 \Rightarrow p' = t/q$ r_3: $t' = q \acute{} p$

as regras r_1 e r_2 são redundantes em relação a r_3, mas o inverso não é verdadeiro, porque r_1 e r_2 não impõem completamente a restrição $t = q \acute{} p$.

Exemplo
No caso das regras que impõem a restrição $x=y=z$,

r_1: $y' = x$ r_2: $z' = y$ r_3: $x' = z$

qualquer regra é redundante em relação ao conjunto das restantes duas, mas não em relação a cada uma das restantes regras isoladamente.

5.4.2 Ordenações ideais

Diz-se que uma ordenação total O de um conjunto de regras R é *ideal*, se, quando as regras são executadas pela ordem O a partir de qualquer estado inicial s, nenhuma regra r altera uma variável de entrada ou saída de uma regra precedente ou uma variável de entrada da própria regra r. Esta condição garante que cada regra é executada no máximo uma vez em cada PPR mesmo quando é seguido o critério de activação básico (e, por maioria de razão, quando é seguido um critério mais refinado).

Na presença de regras recursivas e/ou conflituosas a existência de ordenações ideais está associada à existência de redundância. Numa ordenação ideal de um conjunto de regras, sempre que uma variável de saída y de uma regra r é referenciada por uma regra precedente r' ou uma variável de entrada da própria regra r, então r é redundante em y em relação ao conjunto de regras precedentes. Nesse sentido, pode-se dizer que a recursão e conflitos no conjunto de regras são causados exclusivamente por regras redundantes.

5.4.3 Ordenações ideais de conjuntos de regras puramente incondicionais

No caso de regras puramente incondicionais (isto é, regras que derivam incondicionalmente os valores de todas as variáveis de saída), é possível definir um critério para encontrar ou, pelo menos, delimitar, as ordenações ideais.

Diz-se que uma variável x é uma *entrada de um conjunto* R de regras puramente incondicionais, ordenado segundo uma ordem total O, se x é uma variável de entrada de uma regra $r \in R$ e não é uma variável de saída de nenhuma regra $r' \in R$ com $r' > r$ (isto é, uma regra r' que precede r segundo O). Vendo O como um conjunto ordenado de regras, diz-se simplesmente que x é uma entrada de O.

Note-se que o estado final atingido no fim do processamento das regras é completamente determinado pelos valores iniciais das entradas de O.

Indicam-se de seguida lemas necessários para demonstrar dois teoremas que relacionam ordenações ideias com ordenações com conjuntos mínimos de entradas.

Lema 5.1: Uma ordenação total *O* de um conjunto *R* de regras puramente incondicionais é ideal se e só os valores das entradas de *O* são preservados (i.e., nenhuma regra altera o valor de uma entrada de *O*) e não ocorrem actualizações contraditórias (i.e., nenhuma variável é actualizada com dois valores diferentes por duas regras diferentes ou em execuções diferentes da mesma regra) quando as regras são executadas pela ordem *O* a partir de qualquer estado inicial e qualquer conjunto de regras inicialmente activadas.

Demonstração:

Ideal Þ Entradas preservadas: Suponhamos que *O* é ideal. Seja *x* uma entrada de *O* e seja *r* a 1ª regra que tem *x* como entrada. Por definição de ordenação ideal, o valor de *x* não é alterado por *r* nem por nenhuma regra *r'* com *r'*>*r*. Por definição de entradas de *O*, não existe nenhuma regra antes de *r* que escreva em *x*. Portanto, o valor de *x* é preservado.

Ideal Þ Ausência de actualizações contraditórias: Suponhamos que *O* é ideal. Suponhamos que uma regra *r* actualiza o valor de uma variável *y*. Se a actualização for com alteração, *y* não pode fazer parte de nenhuma regra antes de *r* pela ordem *O*, por definição de ordenação ideal. Se a actualização for sem alteração, não há contradição com uma eventual regra que tenha actualizado *y* antes de *r*. Isto, em conjunto com o facto de que, em consequência da ordenação ser ideal, cada regra é executada no máximo uma única vez pela ordem *O*, implica que não podem ocorrer actualizações contraditórias.

Entradas preservadas e ausência de actualizações contraditórias Þ Ideal: Seja uma ordem total *O* tal que, quando as regras são executadas pela ordem *O*, as entradas de *O* são preservadas e não ocorrem actualizações contraditórias. Suponhamos que uma regra *r* altera o valor de uma variável *y* que é também uma variável de saída de uma regra precedente *r'*. Considerando o pior caso em que todas regras estão inicialmente activadas, a regra *r'* teria sido executada antes de *r*, actualizando o valor de *y* (porque se supõe que todas as regras são puramente incondicionais). Como se supõe a ausência de actualizações contraditórias, a regra *r* não pode actualizar *y* com um valor diferente do produzido por *r'*, isto é, não pode alterar *y*. Portanto, se uma regra *r* altera o valor de uma variável *y*, *y* não pode ser uma variável de saída de uma regra precedente. Pode, no entanto, ser uma variável de entrada de uma regra precedente ou da própria regra *r*. Nesse caso, *y* é uma entrada de *O* (porque é variável de entrada de uma regra antes de ser uma variável de saída doutra regra ou da mesma regra). Como se supõe que as entradas de *O* são preservadas, o valor de *y* também não pode ser alterado por *r*. Por conseguinte, nenhuma regra *r* altera uma variável de entrada ou saída de uma regra precedente ou uma variável de entrada da própria regra *r*. Esta é precisamente a condição que define uma ordem ideal. Δ

Definindo as saídas do conjunto de regras ordenado pela ordem *O* como as variáveis que são alteradas para pelo menos um estado inicial, resulta do lema anterior que os conjuntos de entradas e saídas de uma ordenação ideal são disjuntos. Como também não ocorrem actualizações contraditórias, pode-se dizer que um conjunto de regras ordenado idealmente é equivalente a uma única regra que não é auto-recursiva (porque tem conjuntos disjuntos de entradas e saídas) nem auto-conflituosa.

Lema 5.2: Uma ordenação total *O* de um conjunto *R* de regras puramente incondicionais é ideal se e só se as suas entradas não são restringidas estaticamente por *R* (i.e., a conjunção das restrições impostas pelas regras de *R* não restringe os valores possíveis das entradas de *O*).

Demonstração: Seja *X* o conjunto de entradas de *O*.

Ideal Þ entradas não restringidas: Suponhamos que *O* é ideal. Então, de acordo com o lema anterior, partindo de qualquer valor de *X* (combinação de valores das variáveis de *X*) é possível atingir um ponto fixo para todas as regras preservando esse valor. Consequentemente, *X* não é restringido estaticamente por *R*.

Entradas não restringidas Þ ideal: Suponhamos que *X* não é restringido estaticamente por *R*. Seja *s* um estado inicial arbitrário. Uma vez que *X* não é restringido estaticamente, existe um ponto fixo *t* para todas as regras com *t.X*=*s.X*. Suponhamos que, quando as regras são executadas a partir do estado *s*, uma regra *r* atribui um valor *k* a uma variável $x \in X$, com $k \neq t.x$. Por definição de entradas, a mesma atribuição ocorreria se as regras fossem executadas a partir do estado *t*. Mas então *t* não seria

um ponto fixo para todas as regras. Portanto, nenhuma regra pode alterar o valor de uma variável $x \in X$, isto é, o valor inicial de X é preservado. Suponhamos agora que duas regras r e r' actualizam a mesma variável y com dois valores diferentes. O mesmo aconteceria a partir do estado t, o que é impossível porque t é um ponto fixo. Portanto, não ocorrem actualizações contraditórias. Como os valores das entradas de O são preservados e não ocorrem actualizações contraditórias, O é ideal, de acordo com o lema anterior. Δ

Estamos agora em condições de demonstrar o seguinte:

Teorema 5.1: Para que uma ordenação total O de um conjunto R de regras puramente incondicionais seja ideal, é necessário que O tenha um conjunto mínimo de entradas (isto é, não pode existir uma ordenação O' com um subconjunto próprio das entradas de O).

Demonstração: Suponhamos que existe uma ordenação ideal O e uma ordenação O' com um subconjunto próprio das entradas de O. Sendo X e X', respectivamente, as entradas de O e O', tem-se $X' \subset X$ e $X' \neq X$. Seja então uma variável x tal que $x \hat{I} X$ e $x \ddot{I} X'$. Por definição de entradas, quando as regras são executadas pela ordem O', o valor de x é completamente determinado pelos valores iniciais das restantes variáveis de entrada de O'. Assim, o valor de x num ponto fixo é uma função dos valores das restantes variáveis de X. Uma vez que não consideramos variáveis de estado que tomam valores em domínios com um único valor, existe uma restrição estática nas variáveis de entrada de O. Portanto, O não pode ser ideal. Δ

Este teorema mostra que, se pretendemos encontrar ordenações ideais, basta considerar ordenações com conjuntos mínimos de entradas. O teorema seguinte mostra que o efeito da execução das regras por uma ordem ideal só depende do conjunto de entradas, pelo que, se pretendemos encontrar ordenações ideais não equivalentes entre si, basta considerar uma ordenação para cada conjunto mínimo de entradas.

Teorema 5.2: Se O é uma ordenação ideal de um conjunto R de regras puramente incondicionais, então qualquer ordenação O' com o mesmo conjunto de entradas é também ideal e é equivalente a O (i.e., o estado final atingido quando as regras são executadas segundo as duas ordens é o mesmo, para qualquer estado inicial).

Demonstração: Seja O uma ordenação ideal com conjunto de entradas X e seja O' outra ordenação com as mesmas entradas. De acordo com o lema 5.2, as entradas de O não são restringidas estaticamente por R. De acordo com o mesmo lema, O' é ideal. Resta provar que O e O' são equivalentes. Seja s um estado inicial arbitrário, e sejam t e t' os estados finais produzidos com uma única execução de cada regra segundo as ordens O e O', respectivamente. Suponhamos que existe uma variável y tal que $t.y \neq t'.y$. Uma vez que $t'.X = s.X$, se as regras forem executadas pela ordem O a partir de t', o valor atribuído a y seria o mesmo se as regras fossem executadas pela mesma ordem a partir de s, valor esse dado por $t.y$. Assim, o valor de y seria alterado, pelo que t' não poderia ser um ponto fixo. Isto contradiz a hipótese de que O' é ideal. Portanto, tem de ser $t' = t$. Δ

De seguida, obtêm-se condições a que um conjunto de regras deve obedecer para que todas as ordenações com conjuntos mínimos de entradas sejam ideais.

Para isso, introduz-se o seguinte conceito. Uma regra r de um conjunto R de regras diz-se *completa* num conjunto U de variáveis, se r e R impõem a mesma restrição em U. Formalmente, definindo

F_i - relação de pontos fixos da regra r_i

F - relação de pontos fixos do conjunto de regras $R=\{r_1, ..., r_n\}$

tem-se

$F = F_1 \bowtie ... \bowtie F_n$ ($\bowtie$ é o operador relacional de junção natural)

r_i é completa em $U \Leftrightarrow \pi_U F_i = \pi_U F$ (π é o operador relacional de projecção)

Note-se que uma regra é redundante em relação a qualquer regra completa que refira as mesmas ou mais variáveis.

Teorema 5.3: Seja *R* um conjunto de regras puramente incondicionais tal que, para qualquer conjunto mínimo *U* de variáveis restringidas estaticamente por *R*, existe pelo menos uma regra $r \in R$ que referencia exactamente *U*, é completa em *U* e não é auto-recursiva. Então qualquer ordenação total *O* de *R* com um conjunto mínimo de entradas é ideal.

Demonstração: Seja uma ordenação *O* de *R* com um conjunto mínimo *M* de entradas. Suponhamos que *M* é restringido estaticamente por *R*. Seja *U* um subconjunto mínimo de *M* que ainda é restringido estaticamente por *R*. De acordo com as condições do teorema acerca de *R*, existe uma regra $r \in R$ que referencia apenas *U*, é completa em *U*, não é auto-recursiva e é puramente incondicional. Sejam *X* e *Y* as variáveis de entrada e saída de *r*. Uma vez que *r* não é auto-recursiva, tem-se que $Y=U-X^1Æ$. Uma vez que *r* é puramente incondicional, define o valor final de *Y* como função do valor inicial de *X*. Seja *O'* a ordenação que se obtém a partir de *O* movendo *r* para a 1ª posição. Esta deslocação não aumenta as entradas da ordenação, porque as entradas de *r* (*X*) estão contidas nas entradas de *O* (*M*). Em vez disso, uma vez que *r* deriva incondicionalmente uma parte (*Y*) de *U* em função de outra parte (*X*), *Y* sai das entradas da ordenação. Assim o conjunto de entradas de *O'* é no máximo *M-Y*. Isto contradiz a hipótese de que *O* tem um conjunto mínimo de entradas. Assim, conclui-se que, se *O* tem um conjunto mínimo *M* de entradas, *M* não é restringido estaticamente por *R*. De acordo com o teorema anterior, isto implica que *O* é ideal. Δ

Em consequência dos teoremas anteriores, o critério proposto é:

Critério 5.4: Se o (sub)conjunto de regras a ordenar é constituído apenas por regras puramente incondicionais, seleccionar ordenações totais com um conjunto mínimo de entradas (isto é, com um conjunto mínimo de variáveis que fazem parte do conjunto de entradas de uma regra e não fazem parte do conjunto de saídas de nenhuma regra precedente).

Note-se que encontrar os conjuntos mínimos de entradas exigidos por este critério, equivale a encontrar as chaves da relação de pontos fixos (os seus atributos são as variáveis de estado e os seus tuplos são os pontos fixos) em face de conjunto de dependências funcionais estáticas entre as entradas e as saídas de cada regra (do tipo *i-vars(r)*®*o-vars(r)*), que é um problema conhecido e complexo [U88].

Seguem-se alguns exemplos ilustrativos.

Exemplo

Sejam outra vez as seguintes regras usadas para impor a restrição $x=y=z$:

r_1: $y' = x$ r_2: $z' = y$ r_3: $x' = z$

Este conjunto de regras obedece às condições do teorema anterior, conforme mostra o quadro seguinte.

Conjunto mínimo de variáveis restringidas	Restrição	Regras que impõem a restrição
x, y	$y = x$	r_1
x, z	$z = x$	r_3
y, z	$z = y$	r_2

Por isso, todas as ordenações com conjuntos mínimos de entradas, e só essas, são ideais. As ordenações com conjuntos mínimos de entradas e respectivos efeitos líquidos são:

Ordenação com conjunto mínimo de entradas	Entradas	Efeito líquido
$r_1r_2[r_3]$	*x*	$z'=y'=x$

$r_2r_3[r_1]$	y	$z'=x'=y$
$r_3r_1[r_2]$	z	$y'=x'=z$

Entre parêntesis rectos indicam-se as regras cuja execução é redundante.

Exemplo

Suponhamos que as restrições $z=x+y$ e $y=x$ são impostas de múltiplas maneiras pelas seguintes regras:

r_1: $z' = x+y$ r_2: $y' = x$ r_3: $x' = z-y$

As regras r_1 e r_3 são mutuamente redundantes. Este conjunto de regras não obedece às condições do teorema anterior, como se pode ver pelo quadro seguinte:

Conjunto mínimo de variáveis restringidas	Restrição	Regras que impõem a restrição
x, y	$y = x$	r_2
x, z	$z = 2x$	-
y, z	$z = 2y$	-

Assim, não é de estranhar que algumas ordenações com conjuntos mínimos de entradas não sejam ideais, conforme mostra o quadro seguinte:

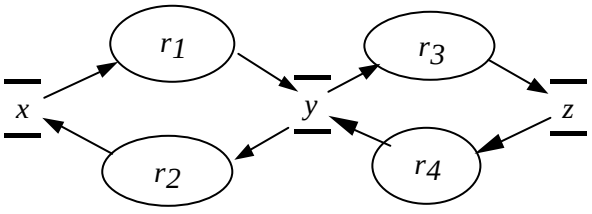
Ordenação com conjunto mínimo de entradas	Entradas	Ordenação ideal? (⇔ entradas não restringidas)
$r_2r_1[r_3]$	x	sim
$r_2r_3r_1$	z, y	não
$r_3r_2r_1$		

Exemplo

Suponhamos que as restrições $y=f(x)$ e $z=g(y)$ são impostas pelas seguintes regras:

r_1: $y' = f(x)$ r_2: $x' = f^{-1}(y)$ r_3: $z' = g(y)$ r_4: $y' = g^{-1}(z)$

a que corresponde o seguinte grafo r-v:



Note-se que as regras r_1 e r_4 são mutuamente conflituosas. O quadro seguinte mostra que este conjunto de regras não está nas condições do teorema anterior.

Conjunto mínimo de variáveis restringidas	Restrição	Regras que impõem a restrição
x, y	$y=f(x)$	r_1, r_2
y, z	$z=g(y)$	r_3, r_4
x, z	$z=g(f(x))$	-

No quadro seguinte indicam-se as ordenações com conjuntos mínimos de entradas. Para cada conjunto de entradas, em vez de se enumerarem todas as ordenações totais, indicam-se apenas as

precedências relativas a que as ordenações totais obedecem. Indicam-se as regras redundantes entre parêntesis rectos. Todas as ordenações são ideais apesar da restrição em *x* e *z* não ser imposta directamente por nenhuma regra.

Ordenação com conjunto mínimo de entradas	Entradas	Ordenação ideal? (⇔ entradas não restringidas)	Efeito líquido
$r_1 > r_3 > [r_4],\ r_1 > [r_2]$	*x*	sim	$y'=f(x)\ \dot{U}\ z'=g(f(x))$
$r_2 > [r_1],\ r_3 > [r_4]$	*y*	sim	$x'=f^{-1}(y)\ \dot{U}\ z'=g(y)$
$r_4 > r_2 > [r_1],\ r_4 > [r_3]$	*z*	sim	$y'=g^{-1}(z)\ \dot{U}\ x'=f^{-1}(g^{-1}(z))$

5.4.4 Combinação com outros critérios

Na ausência de regras recursivas e de regras conflituosas, o critério 5.4 produz o mesmo efeito que o critério 5.1. Em geral, no entanto, não substitui o critério 5.1, pelo que deve ser usado em combinação com o critério 5.1, mas com menos força.

Em geral, existem várias ordenações que satisfazem ao critério 5.4. Para escolher uma de entre essas ordenações, pode-se usar o princípio de preservar os valores introduzidos pelo utilizador. Isto é, o conjunto de entradas escolhido deve ser o que contém as variáveis que interessa preservar.

5.5 Ordens de execução justas

5.5.1 Ordens de execução justas e regras verdadeiramente recursivas

No caso de regras "verdadeiramente" recursivas, isto é, regras recursivas que efectivamente têm de ser executadas repetidas vezes até se atingir um ponto fixo para todas as regras, a ordenação das regras através de prioridades fixas pode ser injusta, porque dá às regras que têm precedência sobre outras regras mais oportunidades de execução. Ora, no caso de regras verdadeiramente recursivas não existem, à partida, razões para dar mais oportunidades de execução a umas regras do que a outras.

Em alternativa, pode-se seguir uma ordem de execução *justa* ("fair"), que dá a todas as regras o mesmo número de oportunidades de execução. Para isso, é estabelecida inicialmente uma ordenação total das regras e de seguida é dada oportunidade de execução às regras, por essa ordem, de forma rotativa.

A diferença entre ordens de execução justas e injustas traduz-se melhor através de expressões de controlo. Seja o conjunto de regras já ordenado $O=r_1r_2r_3...r_n$. Uma ordem de execução injusta é dada pela seguinte expressão de controlo:

$((((r_1^*)\ r_2)^*\ r_3)^*\ ...\ r_n)^*$

enquanto que uma ordem de execução justa é dada pela seguinte expressão de controlo:

$(r_1\ r_2\ r_3\ ...\ r_n)^* = O^*$

Nestas expressões, o asterisco significa repetir enquanto existirem regras activadas. A ocorrência de uma regra *r* nestas expressões significa que é dada à regra uma oportunidade de execução (isto é, a regra é executada se estiver activada). A primeira expressão corresponde ao caso que cada regra tem precedência sobre a regra seguinte. Assim, enquanto a regra com mais precedência (r_1) estiver activada, nenhuma outra regra pode ser executada. No segundo caso, a próxima regra a executar depende de qual foi a última regra executada. A próxima regra a ter uma oportunidade de execução é a que teve uma oportunidade de execução há mais tempo.

Ordens de execução deste tipo são normalmente seguidas na avaliação "bottom-up" de regras dedutivas [RSS90] e na resolução iterativa de sistemas de equações numéricas, mas não em sistemas de regras activas.

Critérios possíveis para estabelecer uma ordenação inicial são indicados na secção 5.6.

5.5.2 Ordens de execução justas e regras falsamente recursivas

As ordens de execução justas são também úteis para suportar eficientemente regras "falsamente" recursivas, muito frequentes em aplicações interactivas de bases de dados (segundo a experiência do autor). Por regras falsamente recursivas, entendem-se regras recursivas cujo *grafo de activação produtiva* é acíclico. O grafo de activação produtiva traduz a relação "pode activar de forma produtiva" entre regras. Diz-se que uma regra *r* activa uma regra *r'* de forma produtiva, se a execução de *r* a partir de um ponto fixo de *r'* (um estado em que a execução de *r'* não seria produtiva) pode terminar num estado que não é um ponto fixo de *r'* (um estado em que a execução de *r'* é produtiva). Por exemplo, regras recursivas com condições mutuamente exclusivas têm um grafo de activação produtiva acíclico.

Em geral, no entanto, o grafo de activação produtiva não é conhecido. Conforme se verá no capítulo 6, quando este grafo é acíclico, o número de execuções das regras, no pior caso, é de ordem $O(2^n)$ se for seguida uma ordem de execução injusta, e de ordem $O(n^2)$ se for seguida uma ordem de execução justa. Uma vez que esta situação é relativamente frequente, a ordem de execução justa será normalmente preferível.

5.6 Ordenação de regras recursivas monótonas

5.6.1 Regras recursivas monótonas

Suponhamos que as regras de um conjunto *R* de regras mutuamente recursivas obedecem à seguinte propriedade de *monotonia*: cada vez que uma regra *r* é executada, faz aproximar (ou, pelo menos, não faz afastar) os valores das suas variáveis de saída dos seus valores finais, tanto mais quanto mais próximos os valores das variáveis de entrada estiverem dos seus valores finais.

É o caso de conjuntos de regras que manipulam variáveis de estado que representam conjuntos ou relações (caso particular de conjuntos) por derivações do tipo

$$y' = y \cup f(x_1,...,x_n),$$

em que *f* representa uma função monótona não decrescente (no sentido da inclusão de conjuntos) em cada uma das variáveis de estado $x_1, ...,x_n$. Derivações deste tipo podem ser úteis em aplicações interactivas de bases de dados para carregar dados da base de dados para vistas interactivas modelizadas por variáveis de estado relacionais. Derivações em tudo semelhantes encontram-se na avaliação "bottom-up" de regras dedutivas escritas na linguagem "datalog" pura [U88b].

É o caso também de conjuntos de regras que manipulam variáveis de estado numéricas por derivações do tipo

$$y' = y + f(x_1,...,x_n),$$

em que *f* representa uma função monótona não decrescente em cada uma das variáveis de estado $x_1, ...,x_n$. Derivações deste tipo encontram-se na resolução iterativa de alguns sistemas de equações não lineares pelo método de Newton.

Estas regras são normalmente auto-recursivas nas suas variáveis de saída.

Exemplo

Seja a relação primitiva

P(x, y) - *x* é progenitor (pai ou mãe) de *y*

e a relação derivada (fecho transitivo de *P*)

A(x, y) - *x* é antepassado de *y*

Suponhamos que estas relações são representadas por variáveis de estado, ditas relacionais. Para obter *A* a partir de *P*, definem-se as seguintes regras (expressas de forma abstracta usando os operadores da álgebra relacional):

r_1: $A' = A \cup P$

r_2: $A' = A \cup r(B,A) \bowtie_{B.y = C.x} r(C,A)$ (*r* é o operador de renomeação de [R98])

Supõe-se que *A* é inicializado com o conjunto vazio.

Estas regras são monótonas no sentido acima definido. O programa correspondente em datalog [U88b] seria constituído pelas seguintes regras (com os nomes no estilo habitual):

(1) antepassado(X,Y) :- progenitor(X,Y).

(2) antepassado(X,Y) :- antepassado(X,Z) & antepassado(Z,Y).

5.6.2 Comparação de sequências de execução de pares de regras recursivas monótonas

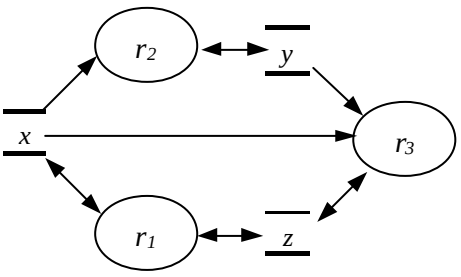
Dadas duas regras r_1 e r_2 monótonas no sentido indicado acima e auto-recursivas nas suas variáveis de saída, é possível comparar a eficiência (em termos de velocidade de aproximação do estado final) da sequência de execução r_1r_2 (executar r_1 e depois r_2, sem nenhuma outra regra pelo meio) com a sequência oposta r_2r_1 (executar r_2 e depois r_1):

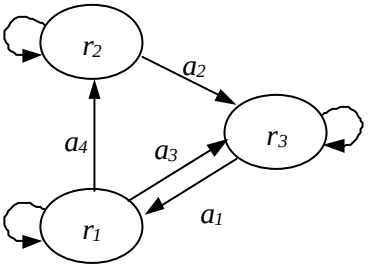
- Se as regras r_1 e r_2 não estiverem ligadas por nenhuma aresta no grafo r-r, então as duas sequências são equivalentes.
- Se as regras r_1 e r_2 estiverem ligadas por uma aresta apenas no grafo r-r, digamos de r_1 e r_2 (o outro caso é simétrico), então a sequência r_1r_2 é decididamente melhor do que a sequência r_2r_1. A execução de r_2 antes de r_1 não beneficia a execução de r_1 (não activa r_1 nem produz dados que sejam usados por r_1). Pelo contrário, a execução de r_1 antes de r_2 só beneficia a execução de r_2. Assim, os valores de todas as variáveis após a execução de r_1r_2 estão mais próximos (ou, pelo menos, não estão mais afastados) dos seus valores finais do que no caso contrário.
- Se as regras r_1 e r_2 estiverem ligadas por arestas nos dois sentidos no grafo r-r, não é possível dizer qual das sequências é melhor, na falta de mais informação. Enquadra-se aqui o caso em que as regras têm variáveis de saída comuns; uma vez que se supõe que as regras são auto-recursivas, a existência de variáveis de saída comuns implica a existência de arestas nos dois sentidos no grafo r-r.

A comparação de sequências de execução de pares de regras permite comparar diferentes ordenações do conjunto de regras, como mostra o seguinte exemplo.

Exemplo

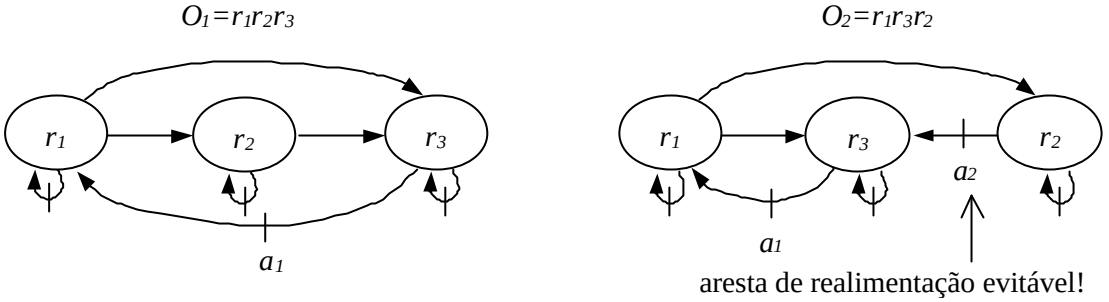
Seja um conjunto de regras recursivas monótonas a que correspondem os seguintes grafos r-v e grafo r-r (com arestas que ligam regras diferentes etiquetadas):





Sejam as ordenações $O_1=r_1r_2r_3$ e $O_2=r_1r_3r_2$. As duas ordenações diferem pela troca de posições de r_3 e r_2. Uma vez que estas duas regras estão ligadas apenas por uma aresta dirigida de r_2 para r_3 no grafo r-r, a ordenação O_2 é pior do que O_1. Por outras palavras, é possível "melhorar" O_2, trocando as posições das regras r_3 e r_2, pelas razões apontadas, obtendo-se então O_1. Em O_1, já não é possível efectuar mais trocas deste tipo.

Cada ordenação induz um conjunto de arestas de realimentação no grafo r-r, conforme mostra a figura seguinte (em que as arestas de realimentação aparecem cortadas com um traço):



Conforme veremos a seguir, O_2 é melhor do que O_1 porque tem uma aresta de realimentação que pode ser evitada.

5.6.3 Ordenações que induzem conjuntos mínimos de arestas de realimentação no grafo r-r

Indicam-se de seguida algumas propriedades de grafos que fundamentam um critério de ordenação baseado nas observações anteriores.

Lema 5.3: Dadas duas ordenações topológicas O e O' dos vértices de um grafo dirigido $G=(V,E)$, é possível passar de uma ordenação a outra (digamos de O a O') pela troca de vértices consecutivos que não são ligados por qualquer aresta de G.

Demonstração: Sem qualquer perda de generalidade, seja $O'=v_1v_2...v_n$. Se o vértice v_1 não está já na primeira posição em O, move-se v_1 para a primeira posição em O, por trocas sucessivas com os vértices que o precedem em O. Cada vez que se troca v_1 com um vértice v_i que o precede em O, os dois vértices não podem estar ligados por qualquer aresta de G: não pode existir uma aresta (v_i, v_1) porque v_1 está na 1ª posição em O' e O' é uma ordenação topológica; não pode existir uma aresta (v_1, v_i) porque nesse caso O não seria uma ordenação topológica. Tendo deslocado v_1 para a 1ª posição de O, elimina-se v_1 do grafo e procede-se da mesma forma com v_2 (a passar para a 2ª posição de O), ..., v_n. Δ

Lema 5.4: Dadas duas ordenações O e O' dos vértices de um grafo dirigido $G=(V,E)$ que induzem o mesmo conjunto F de arestas de realimentação, é possível passar de uma ordenação a outra (digamos de O a O') pela troca de vértices consecutivos que não são ligados por qualquer aresta de G.

Demonstração: Considera-se um grafo G' idêntico a G, a menos das arestas que se encontram em F, que são substituídas por arestas dirigidas em sentido contrário. Isto é, cada aresta $(u,v) \hat{I} F$ é substituída pela aresta (u,v) (se já existir uma aresta nesse sentido, não é necessário acrescentar). Em relação ao grafo G', as ordenações O e O' induzem um conjunto vazio de arestas de realimentação, isto é, O e O' são ordenações topológicas de G'. Pelo lema anterior, é possível passar de uma ordenação a outra pela troca de vértices desligados em G', e, portanto, também desligados em G. Δ

Lema 5.5: Sejam O e O' duas ordenações dos vértices de um grafo dirigido $G=(V,E)$. Sejam F e F' as arestas de realimentação induzidas por O e O', respectivamente. Suponhamos que $F \supset F'$. Então, é possível passar de O a O' pela troca de vértices consecutivos, digamos vértices u e v com v antes de u em O, que não são ligados por qualquer aresta de G ou são ligados apenas por uma aresta dirigida de u para v pertencente a F mas não a F'.

Demonstração: Considere-se um grafo *G'(V,E')* idêntico a *G*, mas sem as arestas pertencentes a *F-F'*. Para cada aresta *(u,v)* Î *F-F'* não pode existir uma aresta simétrica *(v,u)* em *G*; caso contrário, seria *(v,u)* Î *F'* e *(v,u)* Ï *F*, pelo que não poderia ser *F*⊃*F'*. Portanto os pares de vértices *u* e *v* com uma aresta *(u,v)* Î *F-F'* ficam desligados em *G'*. As duas ordenações induzem o mesmo conjunto *F'* de arestas de realimentação em *G'*. Pelo lema anterior, é possível passar de *O* a *O'* pela troca de vértices desligados em *G'*. Esses vértices estão desligados em *G* ou estão ligados apenas por uma aresta eliminada (pertencente a *F-F'*). Seja *u* e *v* o 1º par de vértices trocados do 2º tipo e seja *(u,v)* a aresta eliminada. Uma vez que *(u,v)* Î *F*, *v* tem de estar antes de *u* antes da troca. Assim que esse par de vértices é trocado, tem-se uma nova ordenação *O''* que induz em *G* um novo conjunto de arestas de realimentação dado por *F''=F-{(u,v)}* em *G*. Se *F''=F'* basta aplicar os resultado do lema anterior. Senão, recomeça-se o processo a partir de *O''* em vez de *O*. Δ

Teorema 5.4: Sejam *O* e *O'* duas ordenações de um conjunto *R* de regras recursivas monótonas processadas por uma ordem justa (*O** e *O'** respectivamente). Sejam *F* e *F'* os conjuntos de arestas de realimentação induzidas por *O* e *O'*, respectivamente, no grafo r-r. Se *F* Ì *F'* então, ao fim de qualquer número de iterações, as variáveis de estado têm valores mais próximos (ou igualmente próximos) dos seus valores finais se for seguida a ordem *O* em vez da ordem *O'*.

Demonstração: Resulta do lema anterior e das observações da secção anterior. Δ

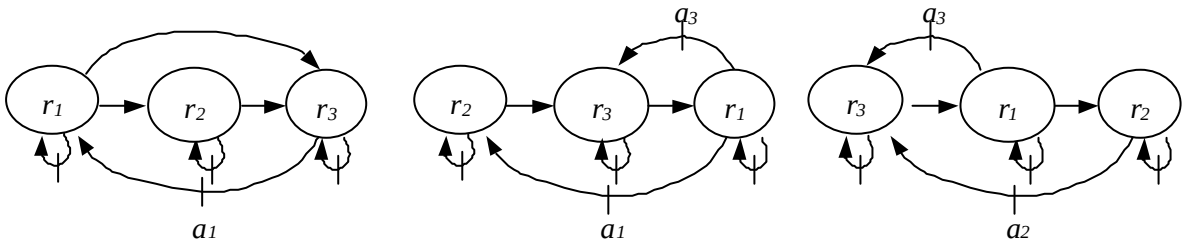
Assim, interessa ordenar as regras de acordo com o seguinte critério:

Critério 5.5: Ordenar as regras por forma a induzir um conjunto mínimo de arestas de realimentação no grafo r-r.

Note-se que há aqui um pequeno abuso de linguagem, porque normalmente o termo "conjunto de arestas de realimentação" ("feedback edge set") já subentende que o conjunto é mínimo (um conjunto de arestas de realimentação é normalmente definido como um conjunto mínimo de arestas cuja eliminação torna o grafo acíclico).

Exemplo

No seguimento do exemplo anterior, as ordenações das regras que induzem conjuntos mínimos de arestas de realimentação no grafo r-r são as seguintes:



Este critério tem diversas vantagens:

- A ordenação das regras por este critério pode ser efectuada em tempo linear no tamanho do grafo, isto é, em tempo *O(|V|+|E|)* (a ver no capítulo 8).
- Intuitivamente, uma vez que minimiza, num certo sentido, as violações ao princípio calcular antes de usar, é uma boa heurística a usar mesmo quando as regras não têm as propriedades acima consideradas ou são processadas por uma ordem injusta (em vez de justa).
- Está estreitamente relacionado com o princípio calcular antes de usar traduzido no critério 5.1. Na ausência de regras não recursivas, tem o mesmo efeito que o critério 5.1. Na presença de regras recursivas, as regras que pertencem a diferentes CFC's do grafo r-r ficam ordenadas pelo critério 5.1, e podem não ficar ordenadas pelo critério 5.5. Em contrapartida, o critério 5.5 permite ordenar

as regras que pertencem ao mesmo CFC do grafo r-r, enquanto que o critério 5.1 não o permite. Assim, os dois critérios são perfeitamente complementares (sem se oporem).

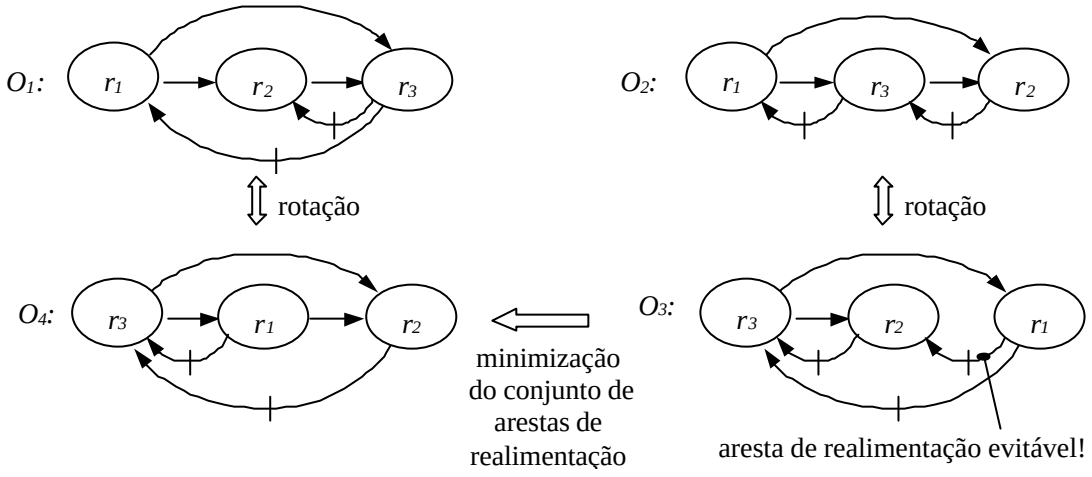
Uma vez que este critério produz, em geral, várias ordenações possíveis, deve ser combinado com outros critérios que permitam seleccionar uma de entre as várias ordenações possíveis.

5.6.4 Ordenações que induzem uma aresta de realimentação em cada ciclo do grafo r-r

No caso das regras serem processadas por uma ordem justa, é possível refinar o critério anterior, conforme sugere o seguinte exemplo.

Exemplo

Sejam as duas ordenações O_1 e O_2 que induzem conjuntos mínimos de arestas de realimentação no grafo r-r, conforme indicado na figura seguinte (possíveis lacetes são omitidos porque não fazem diferença para o caso).



Apesar de O_2 induzir um conjunto mínimo de arestas de realimentação, se as regras forem executadas por uma ordem justa, é possível melhorar a ordem de execução da seguinte forma:

$O_{2*} = (r_1r_3r_2)^* = r_1(r_3r_2r_1)^*$ que é pior do que $r_1(r_3r_1r_2)^* = r_1r_3(r_1r_2r_3)^*$

A optimização aqui efectuada, em vez de ser feita logo na 1ª iteração (uma iteração é uma passagem por todas as regras), é feita numa "janela" do tamanho de uma iteração, mas com início deslocado (ou, equivalentemente, numa rotação da ordenação inicial). No caso de O_1 tal optimização já não é possível, seja qual for a rotação que se considere. Como veremos a seguir, isso acontece porque O_1 induz apenas uma aresta de realimentação em cada ciclo do grafo r-r, contrariamente ao que acontece com O_2, que induz duas arestas de realimentação no ciclo r_2-r_3-r_1-r_2.

Lema 5.6: Se uma ordenação O dos vértices de um grafo $G=(V,E)$ fortemente conexo induz apenas uma aresta de realimentação em cada ciclo de G, então O e qualquer rotação de O induz um conjunto mínimo de arestas de realimentação em G.

Demonstração: O número de arestas de realimentação induzidas por uma ordenação dos vértices do grafo em qualquer ciclo do grafo não se altera quando se efectua uma rotação (ver anexo 1). Se o grafo é fortemente conexo e o número de arestas de realimentação por ciclo é 1, então o conjunto de arestas de realimentação é mínimo. Δ

Em face desta propriedade e do que foi observado no exemplo anterior, sugere-se o seguinte critério:

Critério 5.6: Para a execução das regras de um CFC do grafo r-r por uma ordem justa, escolher uma ordenação inicial que induz apenas uma aresta de realimentação em cada ciclo do grafo r-r.

A optimalidade das ordenações deste tipo foi demonstrada de forma especial para certos tipos de regras dedutivas em [RSS90]. Infelizmente, nem sempre existem ordenações deste tipo. O problema de determinar se existe e, em caso afirmativo, obter uma ordenação dos vértices de um grafo dirigido que induza apenas uma aresta de realimentação por ciclo foi deixado em aberto em [RSS90], e é resolvido de forma que se julga satisfatória no anexo 1 a este documento.

5.7 Outros critérios

5.7.1 Testar restrições o mais cedo possível

As regras de restrição devem ter precedência sobre as regras de derivação, porque interessa testar as restrições o mais cedo possível, para evitar efectuar cálculos que podem ser depois abortados.

De qualquer forma, as regras de restrição só devem ser executadas depois das suas variáveis de entrada terem atingido valores finais, o que é garantido pelo critério 5.1.

Assim, o critério de dar precedência às regras de restrição sobre as regras de derivação tem menos força do que o critério 5.1.

5.7.2 Prioridades definidas pelo programador

Os critérios apresentados anteriormente podem não produzir o efeito pretendido pelo programador, não só em termos de eficiência e de semântica, mas também em termos da ordem por que os efeitos laterais produzidos pelas regras ocorrem (como, por exemplo, mensagens para o utilizador).

Assim, o programador de regras deve poder indicar prioridades absolutas e/ou relativas, com mais força do que as que resultam dos critérios anteriormente apresentados.

5.7.3 Seguir a ordem de criação das regras

Mesmo aplicando os critérios descritos, pode existir mais do que uma regra candidata a execução num dado passo do processamento de regras.

Ora, convém que o processamento de regras seja completamente determinístico, para facilitar a programação e teste.

Assim, pode-se seguir o seguinte critério final (com menos força do que todos os outros): no caso de existirem várias regras candidatas a execução, seleccionar a regra mais antiga, isto é, criada há mais tempo.

5.7.4 Minimizar o conjunto de variáveis de realimentação

Na presença de regras com derivações condicionais, em vez de minimizar o conjunto de variáveis que são usadas antes de serem calculadas por *pelo menos uma* regra, como é preconizado no critério 5.4, faz mais sentido minimizar o conjunto de variáveis (ditas de *realimentação*) que são usadas antes de serem calculadas por *todas* as regras capazes de o fazer. Aqui, "usar" e "calcular" deve ser entendido no sentido de fazer parte do conjunto de entradas e saídas, respectivamente.

Como, em geral, existem regras com derivações condicionais, um critério de uso mais genérico do que o critério 5.4 (mas possivelmente com piores resultados no caso específico tratado na secção 5.4) é: seleccionar ordenações totais com um conjunto mínimo de variáveis de realimentação, isto é, um conjunto mínimo de variáveis que fazem parte do conjunto de entradas de uma regra e fazem parte do conjunto de saídas da própria regra ou de pelo menos uma regra subsequente.

5.7.5 Critérios locais

As heurísticas apresentadas para a ordenação de regras recursivas, baseadas na minimização de conjuntos de variáveis de entrada ou de conjuntos de arestas de realimentação, são de natureza global

(porque é necessário considerar todas as regras e não só as regras que se encontram activadas num dado momento), o que torna a sua implementação mais difícil.

Essa dificuldade de implementação pode não ser compensada pelos resultados alcançados, porque se trata de heurísticas cuja relevância é demonstrada apenas para certos conjuntos de regras.

Assim, pode fazer sentido seguir critérios locais mais simples, como por exemplo:

- Considerar apenas o sub-grafo r-r relativo apenas às regras activadas, ignorando a existência doutras regras (que introduzem ciclos adicionais). Uma regra activada *r* é candidata a execução se não existir outra regra *activada r'* com uma aresta dirigida de *r'* para *r* no grafo r-r. Se não existir nenhuma regra nestas condições (devido a ciclos), utiliza-se outro critério.
- Dar prioridade às regras executadas há mais tempo, por razões de "justiça".
- Dar prioridade às regras activadas mais recentemente.

6 Terminação, determinismo e velocidade de terminação do processamento de regras

Neste capítulo determinam-se condições suficientes a que um conjunto de regras deve obedecer para garantir a terminação e o determinismo do processamento de regras. Analisa-se também o impacto da ordem de execução das regras na velocidade de terminação do processamento de regras.

6.1 Terminação

As regras podem activar-se mutuamente (em *cascata*) indefinidamente, fazendo com que o processamento de regras não termine. Se o processamento de um conjunto *R* de regras termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras, o conjunto de regras diz-se *terminante*. Apresentam-se de seguida condições suficientes a que um conjunto de regras deve obedecer para garantir a terminação do processamento de regras, para qualquer estado inicial e qualquer ordem de execução das regras. São apresentadas condições mais conservadores, que exigem apenas uma análise sintáctica das regras, e condições menos conservadores, que exigem também uma análise semântica das regras.

6.1.1 Análise conservadora baseada no grafo de activação

6.1.1.1 Grafo de activação

A análise de terminação de regras activas é geralmente baseada na construção de um *grafo de activação* ("*triggering graph*"):

O grafo de activação para um conjunto *R* de regras é definido da seguinte forma [AWH92]:

- Os vértices são as regras de *R*.
- Uma aresta dirigida $r_i \rightarrow r_j$ (possivelmente com $i=j$) significa que r_i pode gerar um evento que activa r_j.

No caso de regras dirigidas pelos dados activadas de acordo com os critérios apresentados no capítulo 4, o grafo de activação obtém-se a partir das variáveis activadores e variáveis de saída de cada regra, da seguinte forma:

- Uma regra r_i pode gerar um evento que activa a própria regra só se $st\text{-}vars(r_i) \neq \{\}$.
- Uma regra r_i pode gerar um evento que activa outra regra r_j só se $o\text{-}vars(r_i) \cap et\text{-}vars(r_j) \neq \{\}$.

Quando é seguido o critério de activação básico, sem qualquer refinamento, o grafo de activação coincide com o grafo de interferências entre regras. Quando é seguido um critério de activação refinado, o grafo de activação pode ter menos arestas do que o grafo de interferências.

Exemplo

No caso do conjunto de regras da figura 3.1, uma vez que a regra r_5 é idempotente, pode-se tomar *st-vars*(r_5)={}. De resto, não é possível eliminar mais variáveis activadoras, em relação às definidas pelo critério básico. O grafo de activação resultante, apresentado na figura 6.1, é idêntico ao grafo de interferências entre regras apresentado na figura 4.5, a menos do lacete em r_5 que é removido.

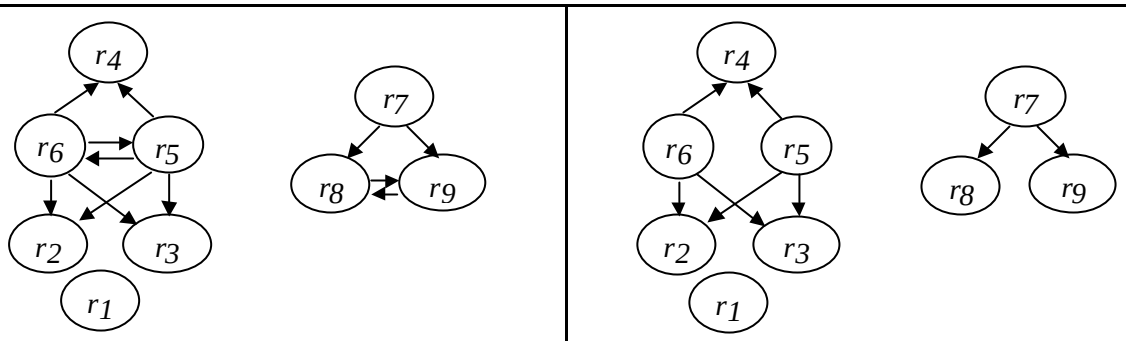


Fig. 6.1 Grafo de activação (à esquerda) e grafo de activação produtiva (à direita) correspondentes ao conjunto de regras da figura 3.1.

6.1.1.2 Condições de terminação

É sabido que se o grafo de activação para um conjunto *R* de regras é acíclico, então o processamento de regras termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras [AWH92]. É possível acrescentar o seguinte:

Teorema 6.1: Se o grafo de activação para um conjunto *R* de regras é acíclico, então o processamento desse conjunto de regras termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras [AWH92]. Além disso, o número total de execuções de regras não excede 2^n-1, em que *n* é o número de regras em *R*.

Demonstração: Se o grafo de activação é acíclico, é possível numerar as regras por uma ordem topológica do grafo de activação. Suponhamos o pior caso em que: i) o grafo de activação é um grafo acíclico completo (isto é, tem um máximo de arestas, sem criar ciclos); ii) todas as regras se encontram inicialmente activadas; iii) quando uma regra é executada activa todas as sucessoras no grafo de activação; iv) as regras são executadas por ordem inversa à ordem topológica do grafo de activação (isto é, é seleccionada para execução a regra activada com número mais alto por ordem topológica do grafo de activação). Seja a_i o número de execuções da regra com número *i* por ordem topológica do grafo de activação. No pior caso tem-se $a_1=1$, porque a regra com número 1 por ordem topológica não é reactivada por qualquer regra, e $a_i=1+a_1+...+a_{i-1}=2^{i-1}$ $(i=2, ..., n)$, porque a regra com número *i* por ordem topológica é executada uma vez antes das regras com números inferiores serem executadas, é reactivada cada vez que uma regra com um número inferior é executada, e é executada de novo antes da próxima regra com um número inferior ser executada. Consequentemente, o número total de execuções de regras é $a_1+...+a_n = 2^0+2^1+...+ 2^{n-1} = 2^n-1$ no pior caso. Δ

No caso de regras activas dirigidas pelos dados pode-se concluir o seguinte:

Teorema 6.2: Se o conjunto *R* de regras não contém regras recursivas nem regras conflituosas, o processamento de regras termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras.

Demonstração: Se *R* não contém regras recursivas nem conflituosas, o grafo de interferências entre regras não tem ciclos (e vice-versa). Uma vez que o grafo de activação tem as mesmas ou menos arestas do que o grafo de interferências, o grafo de activação é também acíclico. Pelo teorema anterior, o processamento de regras termina sempre. Δ

6.1.2 Análise refinada baseada no grafo de activação produtiva

6.1.2.1 Grafo de activação produtiva

No caso de regras activadas dirigidas pelos dados, é possível definir condições de terminação muito menos conservadoras, com base na análise do efeito da execução de cada regra na produtividade dessa e doutras regras.

Diz-se que a execução de uma regra *r* num estado *s* é *produtiva*, ou simplesmente que *r* é produtiva no estado *s*, se a execução de *r* a partir de *s* produz um novo estado diferente de *s*, o que equivale a dizer que *s* não é um ponto fixo para *r*.

Define-se um *grafo de activação produtiva* para um conjunto *R* de regras da seguinte forma:

- Os vértices são as regras de *R*.
- Uma aresta dirigida $r_i \rightarrow r_j$ entre duas regras distintas significa que r_i pode tornar a execução de r_j produtiva (ou pode activar r_j de forma produtiva) isto é, a execução de r_i a partir de um estado *s* em que a execução de r_j não é produtiva, pode produzir um estado *t* em que a execução de r_j é produtiva. Formalmente, $\exists s,t \in S: r_j(s)=s \wedge t=r_i(s) \wedge r_j(t) \neq t$.
- Um lacete $r_i \rightarrow r_i$ significa que a execução de r_i pode terminar num estado em que a execução de r_i continua a ser produtiva. Formalmente, $\exists s,t \in S: t=r(s) \wedge r(t) \neq t$. Isto equivale a dizer que r_i não é idempotente.

Desde que o critério de activação seja correcto (garanta que todas as regras produtivas estão activadas), o grafo de activação produtiva tem as mesmas ou um subconjunto das arestas do grafo de activação. Como, por sua vez, o grafo de activação tem as mesmas ou um subconjunto das arestas do grafo de interferências, conclui-se que o grafo de activação produtiva tem as mesmas ou um subconjunto das arestas do grafo de interferências.

Exemplo

No caso do conjunto de regras da figura 3.1, o grafo de activação produtiva (apresentado na figura 6.1 à direita) tem algumas arestas a menos em relação ao grafo de activação, já considerando o critério de activação mais refinado possível (apresentado na figura 6.1 à esquerda), e ao grafo de interferências (apresentado na figura 4.5).

A regra

r_5: $y' = |y|$

é idempotente, pelo que desaparece o lacete em r_5 (o que, aliás, já acontece no grafo de activação da figura 6.1).

No caso do par de regras

r_5: $y' = |y|$ e r_6: $x \geq 0 \Rightarrow y'=x$,

para averiguar se r_5 pode activar r_6 de forma produtiva, utiliza-se a definição, com as substituições adequadas:

$\exists s,t \in S: (s.x<0 \vee s.y=s.x) \wedge (t.y = |s.y| \wedge t.x = s.x) \wedge \neg(t.x<0 \vee t.y=t.x)$

o que equivale a:

$\exists s \in S: (s.x<0 \vee s.y=s.x) \wedge \neg(s.x<0 \vee |s.y|=s.x)$

Facilmente se verifica que a condição que aparece nesta expressão é impossível. Portanto, r_5 não pode activar r_6 de forma produtiva. O inverso também é verdadeiro, porque r_6 apenas atribui a *y* valores positivos, consistentes com a regra r_5.

No caso do par de regras

r_8: $b' = a+c$ e r_9: $c' = b-a$

um ponto fixo para uma das regras é também um ponto fixo para a outra regra, pelo que não se podem activar mutuamente de forma produtiva.

6.1.2.2 Condições de terminação

O lema e teorema seguintes mostram que a ausência de ciclos no grafo de activação produtiva é condição suficiente de terminação do processamento de regras.

Lema 6.1: O número de execuções de regras não produtivas consecutivas a partir de um estado *s*, durante o processamento de um conjunto *R* de regras, não excede o número de regras para as quais *s* é um ponto fixo e, portanto, não excede o número de regras em *R*.

Demonstração: O critério de activação básico e a política de efeito líquido garantem que, se uma regra *r* é executada de forma não produtiva a partir de um estado *s* (o que equivale a dizer que *s* é um ponto fixo para *r*), então não é gerado nenhum evento de alteração de dados e, por conseguinte, não é activada nenhuma regra. Assim, *r* só pode voltar a ser executada depois de ser executada outra regra *r'* de forma produtiva. No limite, podem ser executadas consecutivamente todas as regras de *R* de forma não produtiva para as quais *s* não é um ponto fixo, uma vez cada regra (se essas regras estiverem inicialmente activadas). Δ

Teorema 6.3: Se o grafo de activação produtiva para um conjunto *R* de regras é acíclico, então o processamento desse conjunto de regras termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras. Além disso, o número total de execuções de regras produtivas não excede 2^n-1, e o número total de execuções de regras (produtivas ou não produtivas) não excede $n2^n$, em que *n* é o número de regras em *R*.

Demonstração: Supõe-se que é seguida uma política de efeito líquido que garante que, quando uma regra *r* é executada a partir de um ponto fixo para *r*, não é gerado nenhum evento de alteração de dados e, consequentemente, nenhuma regra é activada. Se o grafo de activação produtiva é acíclico, é possível numerar as regras por ordem topológica do grafo de activação produtiva. Seja p_i o número de execuções produtivas da regra com número *i* por uma ordem topológica do grafo de activação produtiva. No pior caso, verifica-se o seguinte relativamente à regra com número *i* por ordem topológica: i) está inicialmente activada de forma produtiva; ii) é executada uma primeira vez de forma produtiva antes de ser reactivada de forma produtiva por outras regras; iii) é reactivada de forma produtiva sempre que uma regra com um número $j < i$ por ordem topológica é executada de forma produtiva; e iv) é executada antes de ser de novo reactivada de forma produtiva. Assim, no pior caso, $p_i=1+p_1+...+p_{i-1}$ $(i=2, ..., n)$ e $p_1=1$, pelo que $p_i=2^{i-1}$ $(i=1, ..., n)$. Consequentemente, o número total de execuções de regras produtivas é $p=p_1+...+p_n=2^n-1$ no pior caso. Pelo lema anterior, entre duas execuções produtivas, e antes da primeira execução produtiva, podem ocorrer no máximo $n-1$ execuções não produtivas. A seguir à última execução produtiva podem ocorrer no máximo *n* execuções não produtivas. Assim, o número total de execuções não produtivas não pode exceder $(n-1)p+n$, pelo que o número total de execuções (produtivas ou não produtivas) não pode exceder $p+(n-1)p+n = n(p+1) = n(2^n-1+1) = n2^n$ no pior caso. Δ

Uma vez que o grafo de activação produtiva tem as mesmas ou um subconjunto das arestas do grafo de activação, esta condição de terminação é menos conservadora do que a obtida com base no grafo de activação. No entanto, a construção do grafo de activação produtiva exige uma análise semântica das regras que, em geral, não pode ser automatizada, enquanto que a construção do grafo de activação requer apenas uma análise sintáctica, fácil de automatizar. Assim, ambas as condições são úteis.

Exemplo

O grafo de activação produtiva da figura 6.2 é acíclico, pelo que o processamento do conjunto de regras a que esse grafo se refere (da figura 3.1) termina sempre. Note-se que, em contrapartida, o grafo de activação tem ciclos.

6.1.3 Análise intermédia baseada no grafo de activação da condição

Foi proposto em [BCP95] um método refinado de análise da terminação de regras ECA genéricas, que combina a análise do grafo de activação ("triggering graph" no original) com a análise de um grafo de activação da condição ("activation graph" no original), que traduz o efeito da execução de cada regra no valor lógico da condição dessa e doutras regras.

O *grafo de activação da condição* para um conjunto *R* de regras é definido da seguinte forma [BCP95]:

- Os vértices são as regras de *R*.
- Uma aresta dirigida $r_i \rightarrow r_j$ ($i \neq j$) significa que a execução de r_i pode activar a condição de r_j, isto é, pode alterar o valor lógico da condição de r_j de falso para verdadeiro.
- Um lacete $r_i \rightarrow r_i$ significa que a execução de r_i pode não desactivar a condição de r_i, isto é, pode terminar num estado em que a condição de r_i continua a ser verdadeira.

Uma regra sem a parte de condição é tratada com se tivesse uma condição sempre verdadeira.

É então aplicado o seguinte algoritmo:

Algoritmo 6.1 (*algoritmo de redução de regras* [BCP95]):

Repetir até não ser possível efectuar mais eliminações:

Se (não existe nenhuma aresta dirigida a *r* no grafo de activação)

ou (não existe nenhuma aresta dirigida a *r* no grafo de activação da condição):

Eliminar *r* de *R*, do grafo de activação e do grafo de activação da condição

Prova-se então que:

Teorema 6.4: Se um conjunto *R* de regras é reduzido a um conjunto vazio pelo algoritmo de redução de regras, então o processamento de *R* termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras. [BCP95]

No entanto, no caso específico de regras activas dirigidas pelos dados, as condições deste teorema são mais conservadoras do que as do teorema 6.3, conforme se demonstra no teorema que se deriva a seguir.

Lema 6.2: Se uma regra *r* não tem um lacete no grafo de activação da condição, então também não tem um lacete no grafo de activação produtiva.

Demonstração: Se *r* não tem um lacete no grafo de activação da condição, a condição de *r* é sempre falsa no final da execução de *r*. Isto implica que o estado atingido no final da execução de *r* é sempre um ponto fixo para *r*. Consequentemente, *r* também não tem um lacete no grafo de activação produtiva. Δ

Lema 6.3: Uma regra *r* sem um lacete no grafo de activação da condição tem as mesmas ou menos arestas a ela dirigidas no grafo de activação produtiva do que no grafo de activação da condição.

Demonstração: Pelo lema anterior, *r* também não tem um lacete no grafo de activação produtiva. Seja $r' \rightarrow r$ uma aresta dirigida a *r* no grafo de activação produtiva, em que *r'* é distinta de *r*. Isto quer dizer que existe um estado *s* tal que *s* é um ponto fixo para *r* e *r'(s)* não é um ponto fixo para *r*. Se *s* é um ponto fixo para *r* e a condição de *r* é sempre falsa no final da execução de *r*, então a condição de *r* é falsa em *s*. Se *r'(s)* não é um ponto fixo para *r*, a condição de *r* tem de ser verdadeira em *r'(s)*. Consequentemente, a aresta $r' \rightarrow r$ também tem de existir no grafo de activação da condição. Δ

Teorema 6.5: Se um conjunto *R* de regras é reduzido a um conjunto vazio pelo algoritmo de redução de regras, então o grafo de activação produtiva de *R* é acíclico.

Demonstração: Sejam GC o grafo de activação da condição, GA o grafo de activação, e GP o grafo de activação produtiva. Suponhamos que se aplica um algoritmo de redução de regras modificado, usando o GP em vez do GA. Uma vez que o GP tem um subconjunto das arestas do GA, se *R* é reduzido a um conjunto vazio pelo algoritmo de redução, também é reduzido a um conjunto vazio pelo algoritmo de redução modificado. Para que uma regra *r* possa ser eliminada durante a aplicação do algoritmo de redução modificado, é necessário que $r \to r \in$ GP inicial (GP no início da aplicação do algoritmo) ou $r \to r \in$ GC inicial. Pelo lema 6.2, a segunda condição implica a primeira. Portanto, para que uma regra *r* possa ser eliminada durante a aplicação do algoritmo de redução modificado, é necessário que: i) $r \to r \in$ GP inicial e $r \to r \in$ GC inicial; ou ii) $r \to r \in$ GP inicial e $r \to r \notin$ GC inicial. No caso i), de acordo com o lema 6.3, as arestas dirigidas a *r* no GP inicial são um subconjunto das arestas dirigidas a *r* no GC inicial. Uma vez que os mesmos vértices são removidos do GC e do GP no decorrer do algoritmo, as arestas dirigidas a *r* no GP corrente (antes de eliminar *r*) são também um subconjunto das arestas dirigidas a *r* no GC corrente. Consequentemente, para que a regra *r* possa ser eliminada, não pode ter qualquer aresta a ela dirigida no GP corrente. No caso ii), uma vez que *r* tem uma aresta a ela dirigida no GC corrente (o próprio lacete), para que *r* possa ser eliminada, não pode ter nenhuma aresta a ela dirigida no GP corrente. Em qualquer dos casos, para que *r* possa ser eliminada, não pode ter qualquer aresta a ela dirigida no GP corrente. Consequentemente, se o GP inicial tiver ciclos, não é possível reduzir o conjunto *R* a um conjunto vazio. Uma vez que se supõe que *R* é reduzido a um conjunto vazio, conclui-se que o GP inicial não tem ciclos. Δ

Apesar de tudo, a construção do grafo de activação da condição exige uma análise semântica mais simples do que a exigida para a construção do grafo de activação produtiva, pelo que o teorema 6.4 pode ter alguma utilidade.

Note-se que os grafos de activação produtiva e de activação da condição são idênticos se cada regra for escrita numa forma condicional com a condição o mais restritiva possível - a negação da restrição imposta pela regra. Nesse caso, o grafo de activação da condição tem um subconjunto das arestas do grafo de activação, pelo que o conjunto de regras é reduzido a um conjunto vazio pelo algoritmo de redução de regras se e só se o grafo de activação da condição é acíclico. Portanto, as condições do teorema 6.3 e 6.4 equivalem-se nesse caso.

6.1.4 Detecção dinâmica de actualizações contraditórias que impedem a terminação

Duas actualizações da mesma variável de estado com valores diferentes, produzidas em execuções de regras diferentes (necessariamente conflituosas) ou em execuções diferentes da mesma regra no decurso de um PPR, dizem-se *contraditórias*. A não terminação do processamento de regras está associada à ocorrência de actualizações contraditórias que não desaparecem ao fim de várias execuções de regras.

Pelo menos no caso importante indicado no teorema seguinte, assim que ocorrem actualizações contraditórias num dado PPR, pode-se ter a certeza que o processamento de regras não termina (isto é, que essas actualizações contraditórias não desaparecem ao fim de várias execuções de regras).

Teorema 6.6: Se o conjunto de regras não contém regras recursivas e as regras estão ordenadas segundo o princípio "calcular antes de usar", e ocorrem actualizações contraditórias num PPR, o processamento de regras não termina.

Demonstração: Suponhamos que duas regras *r* e *r'* atribuem diferentes valores à mesma variável *y*. Uma vez que, de acordo com a alínea a) do teorema 5.1, os valores das variáveis de entrada dessas regras já atingiram valores finais, essas actualizações contraditórias não desaparecem em subsequentes execuções dessas regras, causando assim um comportamento oscilatório que não termina. Δ

Assim, se o conjunto de regras não contém regras recursivas e as regras estão ordenadas segundo o princípio "calcular antes de usar", o processamento de regras pode ser abortado assim que se detecta

que uma variável foi actualizada com valores diferentes durante um PPR. No caso de se detectarem apenas as actualizações com alteração, o processamento de regras pode ser interrompido quando uma variável é alterada segunda vez durante um PPR.

No caso de existirem regras recursivas ou de não ser seguido o princípio "calcular antes de usar", as actualizações contraditórias podem ser produzidas por diferentes execuções da mesma regra. Tais actualizações contraditórias podem convergir, pelo que o processamento de regras não deve ser abortado imediatamente. Na prática, pode-se impor um limite ao número de vezes que cada regra pode ser executada ou ao número de vezes que cada variável pode ser alterada num PPR.

6.2 Determinismo

Diferentes escolhas da próxima regra a executar, quando há múltiplas regras activadas candidatas a execução num dado passo do processamento de regras, podem levar a diferentes estados finais do processamento de regras. Se, para um dado conjunto R de regras, o estado final do processamento de regras é independente de qual é a regra escolhida para execução (entre as várias regras activadas candidatas a execução) em qualquer passo do processamento de regras, qualquer que seja o estado inicial, o conjunto de regras (e o processamento desse conjunto de regras) diz-se *determinístico*. Apresentam-se de seguida condições suficientes a que um conjunto de regras deve obedecer para garantir o seu determinismo. Previamente, são introduzidos instrumentos e conceitos necessários à demonstração e aplicação das condições que garantem o determinismo.

6.2.1 Grafo de execução

No trabalho de referência de [AWH92] e [AHW95], a discussão e demonstração de técnicas de análise de determinismo (e terminação) de regras activas é baseada na consideração de *grafos de execução*. Os vértices de um grafo de execução representam estados de execução e as arestas representam transições entre estados de execução causadas pela execução das regras. Cada grafo de execução tem um estado inicial distinto, que representa o início do processamento de regras, e zero ou mais estados finais, que representam a terminação do processamento de regras. Existe um grafo de execução para cada estado inicial possível. No trabalho citado, um estado de execução reúne:

- o estado da base de dados (que corresponde aqui ao estado das variáveis de estado);
- o conjunto de regras activadas;
- o estado de tabelas de transição ("transition tables"), referenciáveis na parte de condição das regras, que dão, para cada regra activada, a variação do estado da base de dados desde a última execução da regra.

No caso específico de regras activas dirigidas pelos dados, interessa adaptar o conceito de grafo de execução da seguinte forma:

- não é necessário considerar as tabelas de transição, porque se supõe que as regras não referenciam estados passados;
- não é necessário considerar o conjunto de regras activadas, porque o mecanismo de activação das regras constitui essencialmente um mecanismo de optimização no sentido de evitar execuções de regras desnecessárias (não produtivas), não afectando os estados finais que podem ser atingidos;
- em consequência do ponto anterior, não se consideram transições correspondentes a execuções de regras não produtivas (com o estado final igual ao estado inicial), por razões que se tornarão evidentes mais adiante;
- uma vez que qualquer estado $s \hat{I} S$ é um possível estado inicial do processamento de regras, considera-se um único grafo de execução, com todos os estados e transições possíveis, em vez de um estado de execução para cada estado inicial possível.

Assim, para a discussão e demonstração de técnicas de análise de determinismo de regras activas dirigidas pelos dados, considera-se um *grafo de execução* definido da seguinte forma:

- os vértices são os estados do espaço de estados (isto é, o conjunto de vértices é o espaço de estados *S*);
- as arestas representam as transições entre estados diferentes causadas pela execução de regras produtivas, sendo cada transição etiquetada com o nome da regra que causa a transição.

O grafo de execução pode ter arestas paralelas, etiquetadas com nomes de regras diferentes. Não tem lacetes, porque só se representam transições entre estados diferentes.

Formalmente, designando por *T* o conjunto de transições representadas por arestas do grafo de execução, tem-se que, no caso de não estarem definidas prioridades entre as regras:

$$T = \{(s,s',r) \in S \times S \times R: r(s) = s' \wedge s \neq s'\}.$$

Cada tuplo (s,s',r) de *T* corresponde a uma aresta etiquetada $s \xrightarrow{r} s'$ no grafo de execução.

No caso de estar definido um conjunto *P* de prioridades entre as regras, não é possível executar uma regra produtiva *r* num estado *s* se existir outra regra produtiva *r'* no mesmo estado com precedência sobre *r* (i.e. com $r' > r \in P$). Assim, tem que se acrescentar:

$$T = \{(s,s',r) \in S \times S \times R: r(s) = s' \wedge s \neq s' \wedge \neg[\exists r' \in R: r'(s) \neq s \wedge r' > r \in P]\}.$$

A relação binária *T'* que se obtém projectando *T* nos seus dois primeiros componentes (*s* e *s'*), constitui uma *relação de redução* ("*reduction relation*") no sentido de [H80]. A relação binária *T'* é útil para aplicar mais directamente os conceitos e técnicas clássicas de análise de confluência apresentados em [H80]. Considerar a relação *T'* corresponde, no grafo de execução, a substituir conjuntos de arestas paralelas por uma única aresta (possivelmente etiquetada com os nomes de todas as regras que podem provocar essa transição).

Todos os vértices do grafo de execução podem ser estados iniciais do processamento de regras. Os vértices dos quais não parte nenhuma aresta representam estados finais do processamento de regras (pontos fixos para todas as regras). Correspondem às *formas normais* de [H80].

Os caminhos do grafo de execução correspondem às sequências de execução de regras, conforme se esclarece e demonstra no lema seguinte.

Lema 6.4: Para qualquer estado inicial s_0, os caminhos no grafo de execução que partem de s_0 (mais precisamente, as etiquetas das arestas que constituem esses caminhos) representam as possíveis sequências de execução de regras a partir de s_0, uma vez retiradas dessas sequências as execuções de regras não produtivas. No caso de estarem definidas prioridades entre as regras, isto continua a ser verdade desde que as prioridades sejam transitivas.

Demonstração: Consideremos primeiro o caso em que não estão definidas prioridades entre as regras. Suponhamos que, na construção de um caminho no grafo de execução, se escolhe uma aresta a sair de um estado *s* com etiqueta *r*. Tal aresta existe se $r(s) \neq s$. O mecanismo de activação garante que, para qualquer estado *s* e regra *r*, se $r(s) \neq s$ então *r* está activada no estado *s*. Portanto, *r* pode ser a próxima regra a ser executada de forma produtiva. Reciprocamente, suponhamos que uma regra *r* é executada de forma produtiva no estado *s*. Devido à forma como o grafo de execução é definido, existe uma aresta a sair do estado *s* com etiqueta *r* no grafo de execução. Portanto, essa aresta pode ser a próxima aresta escolhida.

Consideremos agora o caso em que estão definidas prioridades entre as regras. Suponhamos que, na construção de um caminho no grafo de execução, se escolhe uma aresta a sair de um estado *s* com etiqueta *r*. Tal aresta existe se *r* é produtiva em *s* e não existe nenhuma outra regra *r'* produtiva em *s* com precedência sobre *r*. Temos que provar que, mesmo que existam outras regras activadas não produtivas no estado *s*, a regra *r* pode ser sempre a 1ª regra a ser executada de forma produtiva a partir de *s*. Para começar, podem-se executar todas as regras activadas não produtivas que não são precedidas (directa ou indirectamente) por regras activadas produtivas. Suponhamos que resta uma regra *r''* activada não produtiva com precedência sobre *r*. Isso só pode acontecer se existir uma regra

activada produtiva *r'* com precedência sobre *r''*. Se as precedências forem transitivas, então *r'* também tem precedência sobre *r*, o que contradiz a hipótese inicial acerca de *r* (de que não existe nenhuma outra regra *r'* produtiva em *s* com precedência sobre *r*). Assim, só pode restar uma regra *r''* nas condições indicadas se as precedências não forem transitivas. Δ

Note-se que, se fossem incluídos lacetes no grafo de execução, correspondentes a execuções de regras não produtivas, nem todos os caminhos no grafo de execução corresponderiam a sequências de execução válidas. Por exemplo, um caminho que atravessasse duas vezes seguidas o mesmo lacete não seria válido, porque o mecanismo de activação impede que a mesma regra seja executada duas vezes consecutivas de forma não produtiva. Para incluir execuções de regras não produtivas no grafo de execução, seria necessário incluir também o conjunto de regras activadas em cada estado de execução.

É mais fácil garantir o determinismo quando está garantida a terminação do processamento de regras. A propriedade de terminação do processamento de regras corresponde à ausência de caminhos infinitos do grafo de execução, conforme se demonstra no lema seguinte.

Lema 6.5: O processamento de um conjunto *R* de regras termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras, se e só se o grafo de execução correspondente a *R* não tem caminhos infinitos.

Demonstração: Resulta dos lemas 6.4 e 6.1. Pelo lema 6.1, se o número de execuções de regras produtivas numa sequência de execução é *p*, o número de execuções de regras não produtivas é no máximo *(n-1)p+n*, em que *n* é o número de regras. Δ

Note-se que os caminhos infinitos podem ser devidos a ciclos ou a um espaço de estados infinito. A relação binária (*T'*) correspondente a um grafo dirigido sem caminhos infinitos (o grafo de execução) também é chamada *não etérea* ("*noetherian*") [H80].

6.2.2 Noções de determinismo e confluência

O grafo de execução permite clarificar e relacionar noções importantes de determinismo e confluência.

Um conjunto *R* de regras diz-se *determinístico em sentido fraco* (ou simplesmente *determinístico*) se o estado final atingido em caso de terminação do processamento de regras não depende de qual é a regra escolhida para execução (entre as várias regras activadas candidatas a execução) em qualquer passo do processamento de regras, qualquer que seja o estado de partida do processamento de regras. Em termos do grafo de execução, isto equivale a dizer que existe no máximo um estado final atingível (por um caminho de comprimento 0 ou mais) a partir de qualquer estado *s*. No trabalho de referência de [AWH92], um conjunto de regras que obedece a esta propriedade é chamado confluente. No entanto, prefere-se usar o termo "confluente" num sentido mais tradicional (conforme H80]) e mais literal, a definir adiante. Os estados finais correspondem às formas normais de [H80], pelo que esta noção de determinismo equivale à existência de no máximo uma forma normal para cada elemento $s \hat{I} S$ pela relação binária de transições de estado *(T')*.

Um conjunto *R* de regras diz-se *determinístico em sentido forte* se a própria terminação ou não terminação do processamento de regras, para além do estado final atingido em caso de terminação, não depende de qual é a regra escolhida para execução (entre as várias regras activadas candidatas a execução) em qualquer passo do processamento de regras, qualquer que seja o estado de partida do processamento de regras. Em termos do grafo de execução, isto equivale a dizer que, para qualquer estado *s*, ou não existe nenhum estado final atingível a partir de *s* (caso em que o processamento de regras nunca termina) ou existe exactamente um estado final e nenhum caminho infinito atingível a partir de *s* (caso em que o processamento de regras termina sempre no mesmo estado final). Este tipo de determinismo é mais interessante, porque garante que a ordem por que as regras são escolhidas para execução não tem impacto na semântica do conjunto de regras. Obviamente, as duas noções de

determinismo equivalem-se no caso do conjunto de regras ser terminante, que é o caso em que o determinismo é mais facilmente analisado.

Um conjunto *R* de regras diz-se *confluente* (ou *globalmente confluente*) se, dados quaisquer dois estados s_1 e s_2 atingíveis a partir do mesmo estado *s* no grafo de execução de *R*, existe um estado *s'* atingível tanto a partir de s_1 como a partir de s_2. Esta noção de confluência equivale à noção de confluência de [H80] aplicada à relação binária de transições de estado (*T'*), e à noção de confluência de caminhos ("path confluence") de [AWH92].

Um conjunto *R* de regras diz-se *localmente confluente* se, dadas quaisquer duas arestas $s \rightarrow s_1$ e $s \rightarrow s_2$ (partindo do mesmo estado *s*) no grafo de execução de *R*, existe um estado *s'* atingível tanto a partir de s_1 como a partir de s_2. Esta noção de confluência local provém de [H80] (aplicada à relação *T'*), e equivale à noção de confluência de arestas ("edge confluence") de [AWH92].

Qualquer das noções de confluência ou determinismo pode-se aplicar equivalentemente: ao conjunto *R* de regras (possivelmente acompanhado de um conjunto *P* de prioridades), ao processamento do conjunto *R* de regras, ao grafo de execução de *R*, ou à relação binária (*T'*) de transições de estado causadas pela regras de *R*.

Verificam-se, obviamente, as seguintes relações entre as noções apresentadas:

i) determinismo em sentido forte ⇒ determinismo em sentido fraco
ii) confluência ⇒ determinismo em sentido fraco
iii) confluência ⇒ confluência local

Se o conjunto *R* de regras é terminante (o que equivale a dizer que não existem caminhos infinitos no grafo de execução), verificam-se as implicações em sentido contrário. A 1ª é óbvia. A 2ª é referida a seguir ao lema 2.2 de [H80]. A 3ª corresponde ao lema 2.4 de [H80]. Assim, se *R* é terminante, as quatro noções apresentadas (de determinismo e confluência) equivalem-se. A figura seguinte apresenta alguns exemplos.

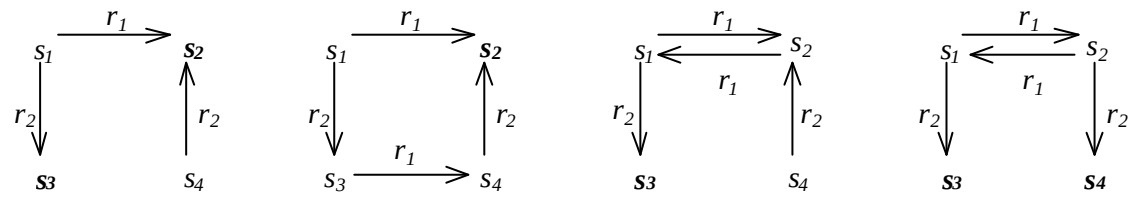


Fig. 6.2 Exemplos de grafos de execução com diferentes propriedades de terminação, confluência e determinismo. Os estados finais são indicados em letra mais forte.

6.2.3 Grafo de execução parametrizado

O grafo de execução é útil para a discussão e demonstração de técnicas de análise de determinismo (e terminação) de regras activas, mas não para averiguar a terminação e determinismo de conjuntos de regras particulares, devido ao tamanho infinito ou quase infinito do espaço de estados.

Para averiguar o determinismo de conjuntos com poucas regras definidas por expressões simples, um instrumento mais adequado é um *grafo de execução parametrizado*, com um único estado inicial mais todos os estados que podem ser atingidos a partir dele, sendo todos os estados parametrizados em função dos valores iniciais das variáveis de estado (i.e. descritos por expressões que dão o valor de

cada variável de estado em função dos valores iniciais de todas as variáveis de estado). Estados parametrizados com expressões equivalentes devem ser agrupados num único estado parametrizado.

A partir do grafo de execução parametrizado pode-se construir um *grafo de execução instanciado* para cada estado inicial concreto, substituindo os parâmetros pelos valores iniciais concretos das variáveis de estado, e condensando vértices com os mesmos valores das variáveis de estado num único vértice. O grafo de execução instanciado coincide com a parte do grafo de execução que contém o estado inicial considerado e todos os estados que podem ser atingidos a partir dele.

As propriedades de terminação e determinismo podem ser analisadas com base no grafo de execução parametrizado, de acordo com os lemas seguintes.

Lema 6.6: Se o grafo de execução parametrizado correspondente a um conjunto *R* de regras não tem caminhos infinitos, então o processamento de *R* termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras.

Demonstração: Os caminhos do grafo de execução parametrizado (mais precisamente, as etiquetas das arestas que constituem esses caminhos), intercalados possivelmente de algumas execuções de regras não produtivas (até um máximo de $|R|$ entre cada duas arestas de acordo com o lema 6.1), representam as possíveis sequências de execução de regras. Assim, se não há caminhos infinitos no grafo de execução parametrizado, não podem existir sequências de execução infinitas. Note-se que, contrariamente ao que acontece no grafo de execução, algumas arestas do grafo de execução parametrizado podem corresponder a execuções de regras não produtivas para alguns valores dos parâmetros (valores iniciais das variáveis de estado). Δ

Lema 6.7: Se o grafo de execução parametrizado correspondente a um conjunto *R* de regras tem ciclos, então o processamento de *R* não termina, para pelo menos um estado inicial e uma ordem de execução das regras.

Demonstração: Sejam e_1 e e_2 dois vértices consecutivos de um ciclo *C* do grafo de execução parametrizado. Uma vez que se supõe que as expressões de e_1 e e_2 não são equivalentes, existe pelo menos um estado inicial concreto *s* tal que, no grafo de execução instanciado para *s*, os vértices e_1 e e_2 não são condensados num único vértice. Mesmo que todos os outros pares de vértices de *C* se reduzam ao mesmo vértice no grafo de execução instanciado para *s*, continua a existir um ciclo contendo e_1 e e_2. Por conseguinte, o processamento de regras não termina para esse estado inicial e pelo menos uma ordem de execução das regras. Δ

Lema 6.8: Um conjunto *R* de regras é determinístico em sentido fraco se e só se o respectivo grafo de execução parametrizado tem no máximo um estado final.

Demonstração: Suponhamos que o grafo de execução parametrizado tem dois estados parametrizados finais e_1 e e_2. Uma vez que se supõe que as expressões de e_1 e e_2 não são equivalentes, existe pelo menos um estado inicial concreto *s* tal que, no grafo de execução instanciado para *s*, os vértices e_1 e e_2 não são condensados num único vértice. Por outro lado, os estados parametrizados finais do grafo de execução parametrizado correspondem a estados finais em qualquer grafo de execução instanciado. Portanto, no grafo de execução instanciado para *s*, existem dois estados finais diferentes que podem ser atingidos a partir de *s*, pelo que o conjunto *R* não é determinístico em sentido fraco. O inverso é óbvio, porque estados finais diferentes num grafo de execução instanciado não podem corresponder ao mesmo estado parametrizado. Δ

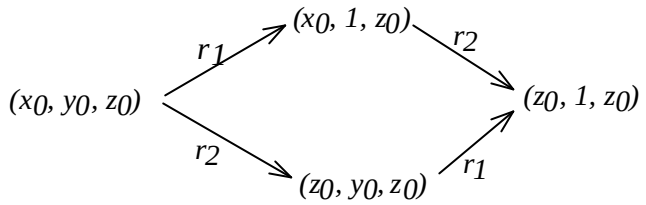
Exemplo

Sejam as regras:

r_1: $y'=1$

r_2: $x'=z$

Considerando o vector de estado $s=(x, y, z)$, o grafo de execução parametrizado correspondente é:



Há um único estado final parametrizado e não há caminhos infinitos, pelo que este conjunto de regras é terminante e determinístico.

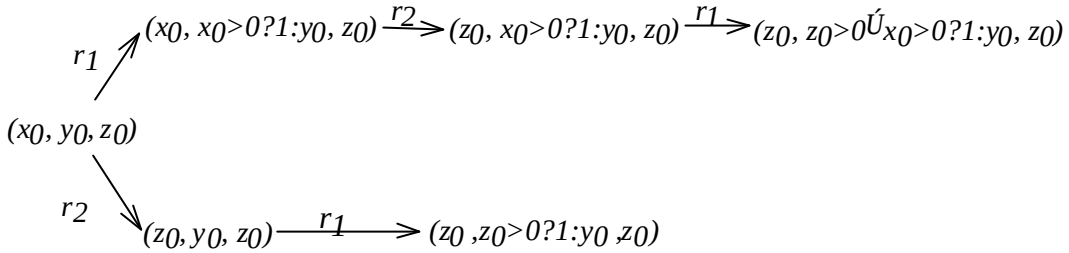
Exemplo

Sejam as regras:

r_1: $x>0 \rightarrow y'=1$

r_2: $x'=z$

Considerando o vector de estado $s=(x, y, z)$, o grafo de execução parametrizado correspondente é:

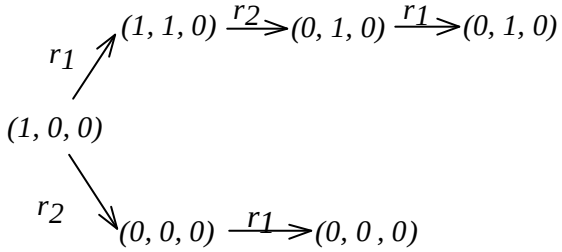


Nas expressões, o operador ternário "?:" do C é usado com significado semelhante a "if then else".

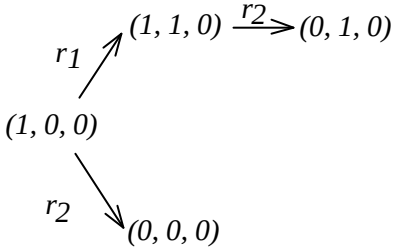
Há dois estados finais parametrizados, pelo que este conjunto de regras não é determinístico.

Por exemplo, o grafo de execução instanciado para o estado inicial $(1, 0, 0)$ é:

antes de juntar estados idênticos:



depois de juntar estados idênticos:



6.2.4 Regras comutativas

Para analisar a confluência e determinismo de um conjunto de regras é importante analisar a comutatividade das regras duas a duas.

Duas regras r_i e r_j dizem-se *comutativas* se, para qualquer estado s, executar a regra r_i e depois a regra r_j a partir do estado s produz o mesmo estado que executar a regra r_j e depois a regra r_i. Formalmente, $\forall s \in S,\ r_i(r_j(s)) = r_j(r_i(s))$, ou $r_i \circ r_j = r_j \circ r_i$, em que "$\circ$" denota a composição de funções. Também se diz que r_i e r_j *comutam.*

Para um dado conjunto de regras R, os pares de regras não comutativas podem ser representados num *grafo de não comutatividade* com conjunto de vértices R e arestas não dirigidas a ligar pares de regras não comutativas.

O grafo de não comutatividade não tem lacetes, porque qualquer regra comuta consigo própria. Indicam-se de seguida outras propriedades úteis na construção do grafo de não comutatividade.

Teorema 6.7: Para que duas regras r_i e r_j comutem é suficiente, mas não necessário, que não interfiram uma com a outra.

Demonstração: *Suficiência*: Sejam r_i e r_j duas regras não interferentes entre si. Isto quer dizer que *o-vars*(r_i) Ç *vars*(r_j) = Æ e *o-vars*(r_j) Ç *vars*(r_i) = Æ. Seja y uma variável de saída de r_i. A condição na qual r_i actualiza y, assim como o valor atribuído, só dependem do valor inicial de *i-vars*(r_i). Uma vez que *o-vars*(r_j) Ç *i-vars*(r_i) = Æ, o valor inicial de *i-vars*(r_i) não é afectado pela execução de r_j, pelo que o valor atribuído a y por r_i é o mesmo quer a regra r_j seja executada antes ou depois de r_i. Uma vez que *o-vars*(r_j) Ç *o-vars*(r_i) = Æ, só r_i actualiza y. Assim, o valor final de y é o mesmo quer seja executado r_i e depois r_j, ou r_j e depois r_i. Repetindo o mesmo raciocínio para as outras variáveis de saída de r_i e r_j, conclui-se que o valor final de todas as variáveis de estado é o mesmo quer seja executado r_i e depois r_j, ou r_j e depois r_i.

Não necessidade: Basta ver o caso de duas regras idênticas, como:

r_i: $y'=x$ r_j: $y'=x$

Comutam apesar de interferirem uma com a outra (porque actualizam a mesma variável y). Δ

Teorema 6.8: Para que duas regras r_i e r_j comutem é necessário, mas não suficiente, que não se activem mutuamente de forma produtiva.

Demonstração: *Necessidade:* Suponhamos que uma das regras, digamos r_i, pode activar a outra (r_i) de forma produtiva. Isto quer dizer que $\$s \hat{I} S$: $r_j(s)=s$ Ù $r_j(r_i(s)) \neq r_i(s)$, ou, equivalentemente, $\$s \hat{I} S$: $r_j(s)=s$ Ù $r_j(r_i(s)) \neq r_i(r_j(s))$, o que implica que $\$s \hat{I} S$: $r_j(r_i(s)) \neq r_i(r_j(s))$, isto é, r_i e r_j não comutam.

Não suficiência: Basta ver o caso das seguintes regras:

r_i: $y'=x$ r_j: $x'=y$

Estas duas regras têm os mesmos pontos fixos. Portanto, nenhuma delas pode activar a outra de forma produtiva. No entanto, não comutam. Δ

Assim, para se obter o grafo de não comutatividade, pode-se começar por considerar um grafo de não comutatividade conservador (com arestas em excesso) que se obtém do grafo de interferências substituindo arestas dirigidas por arestas não dirigidas e removendo arestas paralelas e lacetes.

Exemplo

No caso do conjunto de regras da figura 3.1, depois de construir um grafo de não comutatividade conservador a partir do grafo de interferências entre regras, chega-se à conclusão que apenas é possível eliminar a aresta que liga as regras r_5 e r_6.

De facto, sendo

r_5: $y' = |y|$ r_6: $x \geq 0 \Rightarrow y'=x$

o efeito líquido da execução de r_5 seguido de r_6 é:

$r_5 \circ r_6$: $x \geq 0 \Rightarrow y'=x$ Ù $x<0 \Rightarrow y'=|y|$

enquanto que o efeito líquido da execução de r_6 seguido de r_5 é:

$r_6 \circ r_5$: $x \geq 0 \Rightarrow y'=|x|$ Ù $x<0 \Rightarrow y'=|y|$

que é equivalente ao anterior. Portanto r_5 e r_6 comutam.

Obtém-se assim o grafo de não comutatividade indicado na figura seguinte.

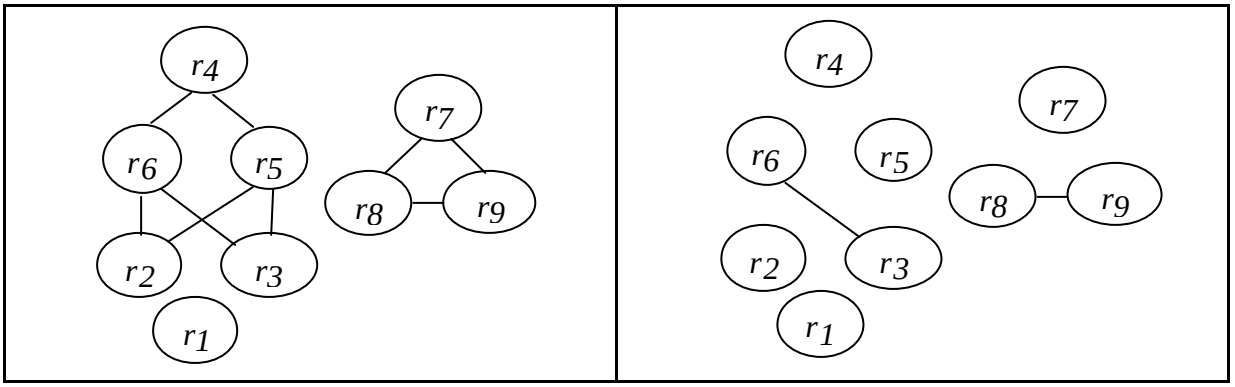


Fig. 6.3 Grafo de não comutatividade (esquerda) e grafo de não confluência (direita) correspondente ao conjunto de regras da figura 3.1.

6.2.5 Pares de regras confluentes

Para analisar a confluência e determinismo de um conjunto de regras é importante analisar a confluência das regras duas a duas, em particular das regras que não são comutativas (pois as regras comutativas são trivialmente confluentes).

Duas regras r_i e r_j de um conjunto R de regras dizem-se confluentes se o conjunto de regras reduzido a essas duas regras, sem prioridades, é confluente.

Indicam-se de seguida algumas propriedades úteis para averiguar se duas regras r_i e r_j são confluentes.

Teorema 6.9: Se duas regras r_i e r_j são comutativas, então são confluentes.

Demonstração: Sejam dois estados s_1 e s_2 atingíveis a partir do mesmo estado s. Sejam E_1 e E_2 as sequências de execução de regras correspondentes aos caminhos de s para s_1 e de s para s_2, respectivamente. Se as regras são comutativas, a sequência de execução E_1E_2 (E_1 seguida de E_2) é equivalente (no sentido que produz o mesmo estado final, para o mesmo estado inicial) à sequência de execução E_2E_1, porque as duas sequências têm o mesmo número de ocorrências de cada regra, e todas as ocorrências (da mesma regra ou de regras diferentes) são comutativas. Assim, executando a sequência E_2 a partir de s_1 atinge-se o mesmo estado que executando a sequência E_1 a partir de s_2. Portanto, as duas regras são confluentes. Δ

Teorema 6.10: Se duas regras r_i e r_j são idempotentes e não se activam mutuamente de forma produtiva, então são confluentes se e só se são comutativas.

Demonstração: Dado o teorema anterior, basta provar que, se r_i e r_j não são comutativas, então não são confluentes. Suponhamos que não são comutativas, isto é, que $r_j \circ r_i \neq r_i \circ r_j$. Uma vez que se supõe que as regras são idempotentes e não se activam mutuamente, o estado que se atinge ao executar quer $r_j \circ r_i$ quer $r_i \circ r_j$ é um ponto fixo para todas as regras (um estado final do processamento de regras). Assim, existe pelo menos um estado inicial s a partir do qual podem ser atingidos dois estados finais diferentes, pelo que o conjunto de regras não é determinístico em sentido fraco e, portanto, não é confluente. Δ

Teorema 6.11: Se duas regras r_i e r_j são idempotentes e apenas uma das regras, digamos r_i, pode activar a outra (r_j) de forma produtiva, então são confluentes se e só se $r_j \circ r_i \circ r_j = r_i \circ r_j$.

Demonstração: Se r_i e r_j obedecem às condições indicadas, as possíveis sequências de execução de regras produtivas são $r_jr_ir_j$, r_ir_j, r_jr_i, r_i, r_j e a sequência vazia Assim, não há caminhos infinitos no grafo de execução, pelo que a confluência é equivalente à confluência local.

Suponhamos que $r_j \circ r_i \circ r_j = r_i \circ r_j$ e que existem duas arestas divergentes $s \xrightarrow{r_i} s_i$ e $s \xrightarrow{r_j} s_j$ no grafo de execução. Então, executando r_j a partir de s_i (o que corresponde a executar $r_i \circ r_j$ a partir de s), obtém-se o mesmo estado que executando $r_i \circ r_j$ (r_i seguido de r_j) a partir de s_j (o que corresponde a executar $r_j \circ r_i \circ r_j$ a partir de s). Portanto, as duas regras são localmente confluentes.

Suponhamos que $r_j \circ r_i \circ r_j \neq r_i \circ r_j$. Uma vez que se supõe que r_i e r_j são idempotentes e não se activam mutuamente de forma produtiva, o estado que se atinge ao executar quer $r_j \circ r_i \circ r_j$ quer $r_i \circ r_j$ é um ponto fixo para todas as regras (um estado final do processamento de regras). Assim, existe pelo menos um estado inicial s a partir do qual podem ser atingidos dois estados finais diferentes, pelo que o conjunto de regras não é determinístico em sentido fraco e, portanto, não é confluente. Δ

Teorema 6.12: Suponhamos que duas regras r_i e r_j obedecem às seguintes condições: i) não são conflituosas; ii) r_i actualiza variáveis lidas por r_j, mas o inverso não acontece; iii) a execução de r_i nunca passa de verdadeiro a falso a condição na qual uma variável é actualizada por r_j; iv) r_i é idempotente; v) r_j não é auto-recursiva. Então, r_i e r_j são confluentes.

Demonstração: As duas regras estão nas condições do teorema anterior, pelo que basta provar que $r_j \circ r_i \circ r_j = r_i \circ r_j$. Suponhamos que as regras são executadas pela sequência $r_j r_i r_j$. Se a condição na qual uma variável y é actualizada por r_j é inicialmente verdadeira, uma vez que se supõe que a condição não passa a falso pela execução de r_i, a actualização de y produzida na primeira execução de r_j será substituída pela actualização de y produzida na segunda execução de r_j. Além disso, as actualizações produzidas por r_i ou pela segunda execução de r_j não dependem das actualizações produzidas na primeira execução de r_j. Assim, a primeira execução de r_j é redundante em relação a y. Se a condição na qual uma variável z é actualizada por r_j é inicialmente falsa, a primeira execução de r_j é também redundante nessa variável z. Em todos os casos (relativamente a todas as variáveis de saída de r_j), a 1ª execução de r_j é redundante, isto é, $r_j \circ r_i \circ r_j = r_i \circ r_j$. Δ

Note-se que a condição iv) do teorema anterior verifica-se trivialmente se r_i não é auto-recursiva. A condição iii) verifica-se trivialmente se todas as variáveis actualizadas por r_j o são incondicionalmente (caso em que a condição é sempre verdadeira). Intuitivamente, o problema das regras que actualizam variáveis condicionalmente é que podem "reagir" a situações transitórias que deixam de se verificar no final. A condição iii) também se verifica trivialmente se r_i não actualiza variáveis referenciadas na condição ou condições pelas quais variáveis são actualizadas por r_j.

Exemplo

Seja o conjunto de regras da figura 3.1. Pelo teorema 6.9, os pares de regras comutativas são também confluentes. Assim, basta analisar a confluência dos pares de regras não comutativas (assinalados na figura 6.3).

Os seguintes pares de regras não comutativas estão nas condições do teorema 6.12 (com a regra correspondente a r_i do lado esquerdo), pelo que são confluentes:

r_5: $y' = |y|$ e r_2: $z' = x+y$

r_6: $x \geq 0 \Rightarrow y' = x$ e r_2: $z' = x+y$

r_5: $y' = |y|$ e r_3: $y = 1 \Rightarrow w' = 1$

r_5: $y' = |y|$ e r_4: $u' = max(x,y) \wedge v' = min(x,y)$

r_6: $x \geq 0 \Rightarrow y' = x$ e r_4: $u' = max(x,y) \wedge v' = min(x,y)$

r_7: $a' = 1$ e r_8: $b' = a+c$

r_7: $a' = 1$ e r_9: $c' = b-a$

As regras:

r_8: $b' = a+c$ e r_9: $c' = b-a$

não obedecem à condição ii) do teorema 6.12. Como estão nas condições do teorema 6.10 e não são comutativas, também não são confluentes (o que aliás é imediatamente óbvio porque impõem a mesma restrição de diferentes maneiras).

As regras:

r_6: $x \geq 0 \Rightarrow y' = x$ e r_3: $y = 1 \Rightarrow w' = 1$

não obedecem à condição iii) do teorema 6.12, porque a execução de r_6 pode passar de verdadeiro a falso a condição ($y=1$) na qual a variável w é actualizada por r_6. Como estão nas condições do teorema 6.11 (são idempotentes e só r_6 pode activar r_3 de forma produtiva) basta comparar os efeitos líquidos de $r_6 \circ r_3$ e $r_3 \circ r_6 \circ r_3$, que são:

$r_6 \circ r_3$: $(x \geq 0 \Rightarrow y'=x) \wedge (x \geq 0 \wedge x = 1 \vee x<0 \wedge y = 1 \Rightarrow w'=1)$

$r_3 \circ r_6 \circ r_3$: $(x \geq 0 \Rightarrow y'=x) \wedge (y=1 \vee x \geq 0 \wedge x = 1 \vee x<0 \wedge y = 1 \Rightarrow w'=1)$

Como as expressões não são equivalentes, conclui-se que estas duas regras também não são confluentes.

Conclui-se, portanto, que há apenas dois pares de regras não confluentes.

Note-se que todos os pares de regras comutativas estão nas condições do teorema 6.12, à excepção das regras r_5 e r_6, pelo que, no total, há apenas 3 pares de regras (de um total de 72 pares possíveis) que não obedecem às condições do teorema 6.12.

6.2.6 Condições suficientes de determinismo

Estamos agora em condições de estabelecer condições suficientes de determinismo de conjuntos de regras não ordenadas (sem prioridades ou qualquer outro tipo de restrição à ordem de execução das regras). Obviamente, a introdução de restrições à ordem de execução das regras apenas pode contribuir para melhorar o determinismo, e nunca piorar, porque, ao se restringirem os caminhos de execução possíveis, pode acontecer que menos estados finais sejam atingíveis a partir de um dado estado inicial, e nunca o contrário. Assim, as condições suficientes que se apresentam a seguir são igualmente válidas para regras ordenadas por qualquer critério.

Teorema 6.13: Se as regras de um conjunto R de regras são confluentes duas a duas e R é terminante, então R é determinístico.

Demonstração: Se R é terminante, o determinismo é equivalente à confluência local. Suponhamos que há duas transições divergentes $s \rightarrow s_i$ e $s \rightarrow s_j$ no grafo de execução para R com $s_i \neq s_j$. Obviamente, as transições $s \rightarrow s_i$ e $s \rightarrow s_j$ têm de ser efectuadas por regras distintas, digamos r_i e r_j. Uma vez que se supõe que todas as regras são confluentes duas a duas, existe um estado s' que pode ser atingido tanto a partir de s_i como a partir de s_j executando r_i e r_j apenas. Portanto, R é confluente localmente e, consequentemente, determinístico. Δ

Corolário 6.1: Se um conjunto R de regras não contém regras recursivas (auto-recursivas ou mutuamente recursivas), nem regras conflituosas, nem regras capazes de passar de verdadeiro a falso a condição na qual uma variável é actualizada por outra regra, então R é terminante e determinístico.

Demonstração: Nas condições indicadas, todos os pares de regras estão nas condições do teorema 6.12, pelo que as regras de R são confluentes duas a duas. Nas condições indicadas, R é terminante, de acordo com o teorema 6.2. De acordo com o teorema 6.13, R é determinístico. Δ

Corolário 6.2: Se um conjunto R de regras não contém regras recursivas (auto-recursivas ou mutuamente recursivas), nem regras conflituosas, e todas as regras são puramente incondicionais (no sentido de que todas as actualizações provocadas por cada regra são incondicionais), então R é terminante e determinístico.

Demonstração: Caso particular do corolário 6.1. Δ

Por exemplo, as folhas de cálculo tradicionais podem-se enquadrar neste caso.

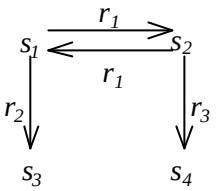
Exemplo

Já concluímos anteriormente que o conjunto de regras da figura 3.1 é terminante, e que há dois pares de regras que não são confluentes, pelo que não se pode garantir a confluência do conjunto de regras. De facto, é fácil de verificar que o conjunto de regras não é confluente, porque a não confluência criada pelos dois pares de regras não confluentes (nas variáveis b, c e w) não é "anulada" pelas outras regras.

No teorema 6.13, não é suficiente que as regras sejam confluentes duas a duas, ou até confluentes e terminantes duas a duas, como os exemplos seguintes demonstram.

Exemplo

Seja um conjunto de regras R com o seguinte grafo de execução:



Estas regras são confluentes duas a duas, mas o conjunto das 3 regras não é confluente nem determinístico.

Exemplo

Sejam as seguintes regras:

r_1: $y' = x$ r_2: $z' = y$ r_3: $x' = z$

Estas regras são confluentes e terminantes duas a duas, mas as três regras não são nem terminantes nem confluentes.

6.2.7 Imposição do determinismo através de prioridades

No caso de conjuntos de regras com prioridades, o teorema 6.13 não entra em conta com as prioridades, as quais podem forçar o determinismo.

No trabalho de referência [AWH92], é desenvolvido um requisito para garantir o determinismo de um conjunto de regras activas com prioridades transitivas, que se transcreve a seguir com pequenos refinamentos para o caso específico de regras activas dirigidas pelos dados.

Requisito 6.1 (*requisito de confluência de conjunto R de regras com conjunto P de prioridades transitivas*):

Considere-se qualquer par de regras não ordenadas r_i e r_j de R.

Caso 1: Se nenhuma das regras, r_i ou r_j, é capaz de activar de forma produtiva uma regra com precedência sobre qualquer das duas regras, então r_i e r_j têm de ser confluentes.

Caso 2: Senão, sejam $R_1 \subseteq R$ e $R_2 \subseteq R$ os conjuntos de regras construídos pelo seguinte algoritmo:

$R_1 \neg \{r_i\}$

$R_2 \neg \{r_j\}$

repetir até não haver alterações:

$R_1 \neg R_1 \grave{E} \{r \hat{I} R \mid r$ pode ser activada de forma produtiva por alguma regra $r_1 \hat{I} R_1$ e $r > r_2 \hat{I} P$ para algum $r_2 \hat{I} R_2$ e $r \neq r_j\}$

$R_2 \neg R_2 \grave{E} \{r \hat{I} R \mid r$ pode ser activada de forma produtiva por alguma regra $r_2 \hat{I} R_2$ e $r > r_1 \hat{I} P$ para algum $r_1 \hat{I} R_1$ e $r \neq r_i\}$

Então, para qualquer par de regras $r_1 \hat{I} R_1$ e $r_2 \hat{I} R_2$, r_1 e r_2 têm de comutar.

Os refinamentos em relação ao requisito original de [AWH92] são:

- Em [AWH92] aparece (dito por outras palavras) "r pode ser activada por ... " onde aqui aparece "r pode ser activada de forma produtiva por ...". O refinamento introduzido aqui está relacionado com o facto de, no grafo de execução aqui considerado, não se incluírem execuções de regras não produtivas. Os conjuntos R_1 e R_2 que se obtêm com este refinamento podem ter menos regras, pelo que o requisito de confluência é mais facilmente satisfeito.
- Em [AWH92] a comutatividade é em relação a estados de execução que incluem informação adicional para além do estado das variáveis de estado (conforme explicado na secção 6.2.1), enquanto que aqui apenas se considera o estado das variáveis de estado. A comutatividade considerada aqui é, por conseguinte, mais facilmente satisfeita.
- Em [AWH92] não aparece o caso 1. As regras nas condições do caso 1 são tratadas da mesma forma que no caso 2, obtendo-se $R_1=\{r_1\}$ e $R_2=\{r_2\}$, pelo que se exige que r_1 e r_2 sejam

comutativas. Em contrapartida, aqui apenas se exige que r_1 e r_2 sejam confluentes. Sem este refinamento, o requisito de confluência aplicado a um conjunto de regras sem prioridades resume-se a exigir que as regras sejam comutativas duas a duas, o que é extremamente conservador. Com este refinamento, apenas se exige que as regras sejam confluentes duas a duas, o que vai de encontro ao teorema 6.13. Provavelmente, este refinamento pode-se aplicar a regras activas genéricas, e não só a regras activas dirigidas pelos dados.

Com base no requisito de confluência pode então estabelecer-se o seguinte:

Teorema 6.14: Se o requisito de confluência se verifica em R e R é terminante, então R é determinístico.

Demonstração: O requisito de confluência garante a confluência local. Isto é, se existem duas arestas divergentes $s \xrightarrow{r_i} s_i$ e $s \xrightarrow{r_j} s_j$ no grafo de execução para R, então existe um estado s' que pode ser atingido tanto a partir de s_i como a partir de s_j. No caso 1, é óbvio que, se as regras r_i e r_j são confluentes e não activam regras com precedência sobre qualquer das duas, existe um estado s' que pode ser atingido tanto a partir s_i como a partir de s_j, executando apenas r_i e r_j. No caso 2, o caminho de s_i para s' é baseado na execução de $R_1\backslash\{r_i\}$ seguida de r_j e de $R_2\backslash\{r_j\}$, enquanto que o caminho de s_j para s' é baseado na execução de $R_2\backslash\{r_j\}$ seguida de r_i e de $R_1\backslash\{r_i\}$, conforme é explicado em [AWH92]. Os refinamentos considerados aqui não afectam o raciocínio aí apresentado. A confluência local em conjunto com a terminação, implicam a confluência e o determinismo de R. Δ

Exemplo

O conjunto de regras da figura 3.1, que já concluímos ser terminante mas não confluente, pode ser tornado confluente com a adição de algumas prioridades. Para cumprir o requisito de confluência é obrigatório, em primeiro lugar, ordenar as regras não confluentes. Considere-se, para esse efeito, o seguinte conjunto de prioridades (em que a primeira está de acordo com o princípio "calcular antes de usar"):

$P_1=\{r_6 > r_3,\ r_8 > r_9\}$

À excepção das regras r_7 e r_9, todos os pares de regras não ordenadas se enquadram nas condições do caso 1 do requisito de confluência (em que nenhuma das regras activa de forma produtiva uma regra com precedência sobre a outra) e, consequentemente, obedecem ao requisito de confluência (porque todos os pares de regras não ordenadas são agora confluentes).

As regras r_7 e r_9 não se enquadram nas condições do caso 1, porque a regra r_7 pode activar de forma produtiva a regra r_8, a qual tem precedência sobre r_9. Uma vez que r_7 e r_9 não são comutativas, não é necessário sequer construir R_1 e R_2 para concluir que o requisito de confluência é violado devido a estas duas regras.

Ordenando as regras r_7 e r_9 de acordo com o princípio "calcular antes de usar", obtém-se o novo conjunto de prioridades:

$P_2=\{r_6 > r_3,\ r_8 > r_9,\ r_7 > r_9\}$

Agora, todos os pares de regras não ordenadas se enquadram nas condições do caso 1 do requisito de confluência e, consequentemente, obedecem ao requisito de confluência (porque todos os pares de regras não ordenadas são confluentes desde a consideração de P_1). Portanto, o conjunto de regras da figura 3.1 com o conjunto de prioridades P_2 é determinístico. Na realidade, pode-se verificar que seria determinístico mesmo com o conjunto de prioridades P_1.

6.2.8 Imposição do determinismo através do princípio calcular antes de usar

Na ausência de regras recursivas, as prioridades estabelecidas de acordo com o princípio "calcular antes de usar", são suficientes para forçar o determinismo, conforme demonstra o teorema seguinte.

Teorema 6.15: Se forem estabelecidas prioridades entre as regras de acordo com o "princípio calcular antes de usar" (critério 5.1) e não existirem regras recursivas, o processamento de regras é determinístico em sentido forte.

Demonstração: Suponhamos primeiro o caso em que não há regras conflituosas. De acordo com o teorema 6.2, o processamento de regras termina sempre. Basta portanto verificar se o requisito de confluência é satisfeito. Considere-se qualquer par de regras não ordenadas r_i e r_j. Se não estão ordenadas pelo critério 5.1, e não há regras conflituosas nem mutuamente recursivas, as regras r_i e r_j não interferem uma com a outra. Consequentemente, são comutativas (pelo teorema 6.7). Suponhamos que uma das regras, digamos r_i, pode activar uma terceira regra r_k com precedência sobre r_i ou r_j. Isso só pode acontecer se r_i actualiza uma variável lida por r_k, e r_k actualiza uma variável y tal que existe um caminho de y para r_i ou r_j no grafo r-r. Mas, r_k não pode actualizar uma variável y tal que existe um caminho de y para r_i no grafo r-r, porque se supõe que não há regras mutuamente recursivas, e r_k não pode actualizar uma variável y tal que existe um caminho de y para r_j no grafo r-r, porque nesse caso r_i teria precedência sobre r_j (uma vez que não há regras mutuamente recursivas). Portanto, nenhum das duas regras pode activar (de forma produtiva ou não) uma regra com precedência sobre qualquer das duas. Por conseguinte, r_i e r_j estão nas condições do caso 1 do requisito de confluência. Uma vez que são comutativas, também são confluentes (pelo teorema 6.9). Portanto o requisito de confluência é satisfeito. De acordo com o teorema 6.14, o processamento de regras é determinístico (em sentido forte ou fraco, porque os dois sentidos são equivalentes quando o conjunto de regras é terminante).

Suponhamos agora que há regras conflituosas. Neste caso segue-se um raciocínio diferente. Uma vez que não há regras recursivas, o grafo v-v é acíclico, pelo que é possível numerar as variáveis de estado por uma ordem topológica do grafo v-v. Sejam v_1, v_2, ..., v_m as variáveis de estado já numeradas dessa forma. Considere-se um estado inicial qualquer. Seja v_i a variável com número mais baixo que é calculada por uma ou mais regras. As variáveis de entrada das regras que actualizam v_i não são actualizadas por qualquer regra, pelo que os valores atribuídos a v_i por essas regras, assim como as condições nas quais essas actualizações ocorrem, são independentes da ordem de execução das regras. Se essas regras actualizam v_i com valores diferentes, o processamento de regras não termina, seja qual for a sequência de execução das regras. Se essas regras actualizam v_i com o mesmo valor (ou não actualizam com qualquer valor), o valor final de v_i é independente da sequência de execução das regras. Passe-se então à variável com número seguinte que é calculada por uma ou mais regras, digamos v_j. De acordo com o critério 5.1, no caso dessas regras lerem v_i, só são executadas depois de v_i ter atingido um valor final que, como vimos, é independente da sequência de execução das regras. Em geral, as regras que actualizam v_j, só são executadas depois das suas variáveis de entrada terem atingido valores finais, os quais são independentes da sequência de execução das regras (desde que compatível com o critério 5.1). Portanto, a situação é semelhante à das regras que actualizam v_i. Se essas regras actualizam v_j com o mesmo valor (ou não actualizam com qualquer valor), passa-se à variável seguinte. Senão, o processamento de regras não termina, seja qual for a sequência de execução das regras compatível com o critério 5.1. Repetindo o raciocínio para todas as variáveis, chega-se à conclusão que, para qualquer estado inicial, ou o processamento nunca termina ou termina sempre no mesmo estado final (para qualquer sequência de execução das regras compatível com o critério 5.1). Portanto, o conjunto de regras é determinístico em sentido forte. Δ

6.2.9 Imposição do determinismo através de ordens de execução justas

Outra forma de forçar o determinismo do processamento de um conjunto R de regras que não obedece às condições do teorema 6.13, é através da execução das regras por uma ordem justa (ver secção 5.5) compatível com um conjunto P de prioridades, isto é, por uma ordem de execução traduzida pela expressão de controlo O^*, em que O é uma ordenação total de R compatível com P.

Por exemplo, dado o conjunto de regras $R=\{r_1, r_2, r_3, r_4\}$ e o conjunto de prioridades $P=\{r_1 > r_2, r_1 > r_3, r_3 > r_4\}$, as ordenações totais de R compatíveis com P são $O_1=r_1r_2r_3r_4$, $O_2=r_1r_3r_2r_4$ e $O_3=r_1r_3r_4r_2$. As ordens de execução justas compatíveis com P são O_1^*, O_2^* e O_3^*. Relembre-se que

a ocorrência de uma regra *r* nas expressões de controlo não significa necessariamente que a regra é executada, mas apenas que tem uma oportunidade de execução (i.e., é executada se estiver activada). De qualquer forma, o resultado é o mesmo se todas as regras forem executadas quando têm oportunidade de execução, porque o critério de activação garante que as regras que não estão activadas não são produtivas.

De seguida, obtêm-se condições suficientes a que *R* e *P* devem obedecer para garantir o determinismo do processamento de regras por uma ordem de execução justa. Note-se que o grafo de execução não é adequado para analisar o determinismo de regras executadas por ordens justas, porque as regras que podem ser executadas a partir de um estado *s* não dependem apenas de *s* e *P*.

Lema 6.9: Duas ordenações totais *O* e *O'* de um conjunto *R* de regras, vistas como sequências de execução de regras, são equivalentes (no sentido de produzirem o mesmo estado final para qualquer estado final), se quaisquer duas regras não comutativas têm as mesmas posições relativas em *O* e *O'* (isto é, se r_i e r_j são regras não comutativas e r_i precede r_j em *O*, então r_i também precede r_j em *O'*).

Demonstração: Suponhamos que *O* e *O'* são duas ordenações de *R* tais que quaisquer duas regras não comutativas de *R* têm as mesmas posições relativas em *O* e *O'*. Indica-se em primeiro lugar um método para transformar uma das ordenações (digamos *O*) na outra (*O'*) pela troca de pares de regras consecutivas comutativas. Sem qualquer perda de generalidade, seja $O'=r_1r_2...r_n$. Se r_1 não está já na 1ª posição em *O*, move-se r_1 para a 1ª posição de *O* trocando de posição sucessivamente com a regra imediatamente precedente em *O*. Em cada troca de r_1 com uma regra r_j, uma vez que as posições relativas de r_1 e r_j são diferentes em *O* e *O'*, as duas regras têm de ser comutativas. As posições relativas de todos os pares de regras que não contêm r_1 não são afectadas. Então, prossegue-se da mesma maneira com as restantes regras de *O*.

Uma vez que *O* pode ser obtido de *O'* (e vice-versa) pela troca de pares de regras consecutivas comutativas, as duas ordenações (vistas como sequências de execução de regras) produzem o mesmo estado final para qualquer estado inicial Δ

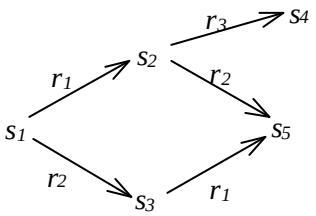
Teorema 6.16: Um conjunto *R* de regras com um conjunto *P* de prioridades, processado por qualquer ordem de execução justa compatível com *P*, é determinístico em sentido forte se quaisquer duas regras não comutativas estão ordenadas (em *P*).

Demonstração: Sejam *O* e *O'* quaisquer duas ordenações totais de *R* compatíveis com *P*. De acordo com as condições do teorema, se duas regras r_i e r_j não são comutativas, então $r_i > r_j$ $\hat{I}$ *P* ou $r_i < r_j$ $\hat{I}$ *P*. Portanto, se *O* e *O'* são compatíveis com *P*, r_i e r_j têm as mesmas posições relativas em *O* e *O'*. Pelo lema anterior, *O* e *O'* são equivalentes, pelo que o processamento de regras *O** é equivalente a *O'**. Δ

É de notar que, para forçar o determinismo, não é suficiente ordenar os pares de regras não confluentes. É de notar também que, no caso das regras não serem executadas por uma ordem justa (caso considerado na secção 6.2.7), algumas regras comutativas podem ter de ser também ordenadas para satisfazer o requisito de confluência (requisito 6.1). Estas observações são ilustradas pelo exemplo seguinte.

Exemplo

Seja um conjunto *R* de regras com o seguinte grafo de execução:



O conjunto de regras não ordenado não é determinístico, porque existem dois estados finais (s_4 e s_5) que podem ser atingidos a partir do mesmo estado inicial (s_1 ou s_2). As regras r_1 e r_2 são comutativas

e, consequentemente, confluentes. As regras r_1 e r_3 são confluentes mas não são comutativas, porque $r_1(r_3(s_1)) = s_2 \neq r_3(r_1(s_1)) = s_4$ (recorde-se que as auto-transições não são representados no grafo). As regras r_2 e r_3 não são confluentes e, consequentemente, também não são comutativas.

Suponhamos que, para tentar forçar o determinismo, as duas regras não confluentes são ordenadas por $P_1=\{r_3 > r_2\}$. As ordens totais compatíveis com P são $O_1=r_1r_3r_2$, $O_2=r_3r_1r_2$ e $O_3=r_3r_2r_1$. Com o estado inicial s_1, O_1* termina no estado s_4, mas O_2* e O_3* terminam no estado s_5. Assim, o conjunto de regras ainda não é determinístico quando as regras são executadas por ordens justas compatíveis com P.

Suponhamos que as outras regras não comutativas - r_1 e r_3 - são também ordenadas por $P_1=\{r_3 > r_2, r_3 > r_1\}$. As ordens totais compatíveis com P_2 são O_2 e O_3. Com os estados iniciais s_1 ou s_2, tanto O_2* como O_3* terminam no estado s_5. Portanto, o processamento de regras é determinístico.

Se as regras forem processadas de acordo com P_2, mas por uma ordem injusta, o processamento de regras ainda não é determinístico. De facto, com o estado inicial s_1, $(((r_3)*r_2)*r_1)$* termina no estado s_5, enquanto que $(((r_3)*r_1)*r_2)$* termina no estado s_4.

6.3 Velocidade de terminação

Para além do impacto que pode ter no estado final atingido (quando o conjunto de regras não é determinístico), a ordem por que as regras são executadas tem quase sempre impacto na eficiência temporal (velocidade de terminação) do processamento de regras. Na ausência de informação adicional a eficiência pode ser medida pelo número de vezes que cada regra é executada ou pelo número total de execuções de regras.

No capítulo 5 foram indicados diversos critérios de ordenação, e foi analisado o impacto de alguns desses critérios na eficiência temporal do processamento de regras. O objectivo desta secção não é o de desenvolver métodos genéricos para analisar a eficiência do processamento de regras, mas antes apresentar alguns resultados que complementam os do capítulo 5. Na secção 6.3.1 demonstra-se a vantagem (referida mas não demonstrada na secção 5.5.2) das ordens de execução justas para o processamento de regras falsamente recursivas (com um grafo de activação produtiva acíclico). Na secção 6.3.2 indica-se uma condição suficiente para garantir que duas ordens de execução justas têm velocidades de terminação semelhantes, complementando os resultados da secção 5.5.6.

6.3.1 Vantagem das ordens de execução justas para o processamento de regras falsamente recursivas

Muito frequentemente, as regras recursivas são "falsamente" recursivas, no sentido de que o respectivo grafo de activação produtiva é acíclico. Assim, basta uma única execução de cada regra por uma ordem apropriada (uma ordem topológica do grafo de activação produtiva acíclico) para se atingir um ponto fixo para todas as regras. Na prática, verifica-se que: i) mesmo depois de se atingir um ponto fixo para todas as regras, pode ser necessário executar mais uma vez cada regra, devido às limitações próprias do critério de activação; ii) em geral, o grafo de activação produtiva não é conhecido (porque exige uma análise semântica das regras impossível de automatizar no caso geral), pelo que não se sabe a priori qual é a ordem de execução apropriada. As ordens de execução justas são vantajosas para tratar mais eficientemente regras recursivas com um grafo de activação acíclico mas desconhecido, como mostra o teorema seguinte.

Teorema 6.17: Suponhamos que um conjunto R de regras, com um grafo de activação produtiva acíclico, é processado por uma ordem de execução justa qualquer. Então, o número de execuções de regras não excede $n \cdot (n+1)$, em que n é o número de regras.

Demonstração: Intuitivamente, o pior caso acontece quando: i) o grafo de activação produtiva é um grafo acíclico completo; ii) o critério de activação é tão conservador que todas as regras são activadas quando é alterado o valor de qualquer variável de estado; iii) as regras são ordenadas por ordem inversa à única ordem topológica do grafo de activação produtiva, i.e., a ordem topológica do

grado de activação é $r_n \ldots r_2 r_1$ e a ordem de execução das regras é $r_1 r_2 \ldots r_n$; iv) todas as regras estão inicialmente activadas. No final da 1ª iteração (uma passagem completa por todas as regras), foi atingido um ponto fixo para a regra r_n. Assim, em iterações subsequentes, mesmo que r_n seja executada, não activa outras regras. Ao fim de $n+1$ iterações, no máximo, é atingido um ponto fixo para a regra r_1. Na iteração $n+1$, nenhuma regra activa outras regras. O número máximo de iterações é portanto $n+1$. Se todas as regras forem executadas em todas as iterações, o número de execuções de regras é $n(n+1)$. Δ

Este número de execuções quadrático compara muito favoravelmente com o número de execuções exponencial obtido no teorema 6.3 para o pior caso de uma ordem de execução do tipo $(\ldots((r_1)^* r_2)^* \ldots r_n)^*$.

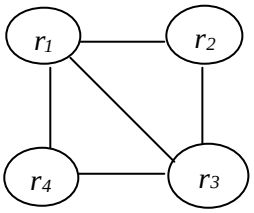
6.3.2 Ordens de execução justas com velocidades de terminação semelhantes

Dadas duas ordens de execução justas O_1^* e O_2^* das regras de um conjunto R de regras recursivas, em que O_1 e O_2 são duas ordenações totais de R, não é possível, com base apenas na informação de natureza sintáctica traduzida no grafo r-v, dizer que uma delas é mais eficiente do que outra. No entanto, é possível assegurar que certas ordens de execução têm eficiência semelhante, com base no conhecimento de que certas regras são comutativas. Note-se que, com base apenas no conhecimento do grafo de interferências (que é obtido a partir do grafo r-v), já é possível assegurar que certas regras são comutativas, de acordo com o teorema 6.7.

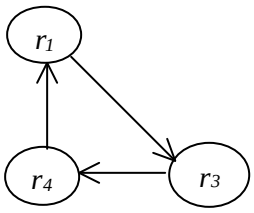
A propriedade importante de cada ordenação total O de R a considerar para este efeito é o número de arestas de realimentação induzidas por O em cada ciclo do grafo de não comutatividade, após se escolher uma orientação arbitrária para cada ciclo. As arestas de realimentação induzidas por O são as arestas $r_i \to r_j$ orientadas em sentido contrário a O, isto é, tal que r_i não precede r_j em O.

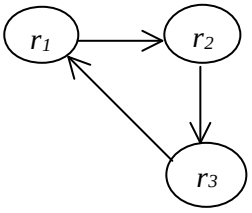
Exemplo

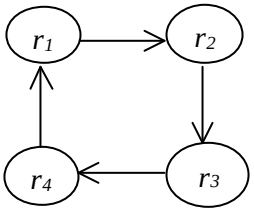
Sejam um conjunto de regras $R=\{r_1, r_2, r_3, r_4\}$ com o seguinte grafo de não comutatividade:



Este grafo tem os três ciclos seguintes, já orientados no sentido dos ponteiros do relógio:







A tabela seguinte mostra as arestas de realimentação induzidas em cada um destes ciclos por três ordenações diferentes de R.

arestas de realimentação induzidas por cada ordenação em cada ciclo orientado		ordenações		
		$O_1=r_1r_3r_2r_4$	$O_2=r_3r_4r_2r_1$	$O_3=r_1r_2r_3r_4$
ciclos:	$r_1 \to r_3 \to r_4 \to r_1$	$r_4 \to r_1$	$r_1 \to r_3$	$r_4 \to r_1$

	$r_1 \circledR r_2 \circledR r_3 \circledR r_1$	$r_2 \circledR r_3,\ r_3 \circledR r_1$	$r_1 \circledR r_2,\ r_2 \circledR r_3$	$r_3 \circledR r_1$
	$r_1 \circledR r_2 \circledR r_3 \circledR r_4 \circledR r_1$	$r_2 \circledR r_3,\ r_4 \circledR r_1$	$r_1 \circledR r_2,\ r_2 \circledR r_3$	$r_4 \circledR r_1$

As ordenações O_1 e O_2 induzem o mesmo nº de arestas de realimentação em cada ciclo. Note-se que isso aconteceria mesmo que os ciclos fossem orientados de maneira diferente.

Com base neste conceito, é possível mostrar o seguinte:

Lema 6.10: Sejam O_1 e O_2 duas ordenações totais (vistas como sequências de execução de regras) de um conjunto de regras R que induzem o mesmo número de arestas de realimentação em cada ciclo do grafo de não comutatividade de R. Seja D a distância máxima entre quaisquer duas regras conexas no grafo de não comutatividade de R, ou 0 no caso de todas as regras comutarem duas a duas. [1] Então, existem sequências de regras A e B, e um inteiro h, com $0 \pounds h \pounds D$, tais que, para qualquer número natural N, $O_2^{N+h} = AO_1^N B$.

Demonstração: Resulta do teorema 6 no anexo 2 a este documento. A demonstração é baseada no facto de que O_1 e O_2 induzem o mesmo número de arestas de realimentação em cada ciclo do grafo de não comutatividade se e só se uma delas (digamos O_2) pode ser obtida da outra (O_1) aplicando repetidamente as seguintes operações: i) rotação, ii) troca de regras consecutivas comutativas. Δ

Este lema tem, entre outras, a seguinte consequência óbvia:

Teorema 6.18: Sejam O_1 e O_2 duas ordenações totais de um conjunto de regras R que induzem o mesmo número de arestas de realimentação em cada ciclo do grafo de não comutatividade de R. Seja D a distância máxima entre quaisquer duas regras conexas no grafo de não comutatividade de R, ou 0 no caso de todas as regras comutarem duas a duas. Então, se o processamento de O_1* termina em N_1 iterações no máximo, para qualquer estado inicial, o processamento de O_2* termina em $N_2 = N_1 + D + 1$ iterações no máximo, para qualquer estado inicial.

Demonstração: Pelo lema anterior, $O_2^{N+h} = AO_1^N B$. Se o processamento de O_1* termina em N iterações no máximo, AO_1^N é um ponto fixo para todas as regras. Portanto, $AO_1^N B = O_2^{N+h}$ também é. Como pode ser necessária mais uma iteração depois de se ter atingido um ponto fixo para todas as regras, o processamento de O_2* termina ao fim de $N+h+1$ iterações no máximo. Como $h \pounds D$, o processamento de O_2* termina ao fim de $N+D+1$ iterações no máximo. Δ

Neste sentido, pode-se dizer que O_1* e O_2* têm velocidades de terminação semelhantes.

[1] A distância entre dois vértices conexos (ligados por algum caminho) de um grafo é a distância, em número de arestas, do caminho mais curto que une os dois vértices [TS92].

7 Refinamentos para o tratamento de dados complexos

Neste capítulo discutem-se refinamentos à abordagem introduzida no capítulo 3 e prosseguida nos capítulos seguintes, necessários para a manutenção eficaz de dados derivados e restrições de integridade em formulários de ecrã (e outros interfaces para o utilizador presentes em aplicações interactivas de bases de dados) com uma estrutura de dados complexa através de regras activas dirigidas pelos dados com semântica de ponto fixo. Esses refinamentos consistem principalmente na adaptação à abordagem preconizada no capítulo 3 de soluções encontradas noutros sistemas de regras activas.

7.1 Introdução

O modelo de regras introduzido no capítulo 3 baseia-se num modelo de dados simples e genérico, em que os dados são representados abstractamente por variáveis de estado atómicas ou tratadas como tal para efeito do processamento e análise de regras. Esse modelo de regras é adequado, sem mais refinamentos, para a manutenção de restrições de integridade e dados derivados em formulários de ecrã uni-registo, contendo uma sequência simples de campos atómicos, e outros interfaces para o utilizador igualmente simples.

No entanto, os formulários de ecrã e outros interfaces para o utilizador que aparecem em aplicações interactivas de base de dados, têm frequentemente uma estrutura mais complexa, geralmente hierárquica. São frequentes formulários e relatórios com uma estrutura de dados tabular (contendo um conjunto de registos), e formulários com sub-formulários, também designados formulários mestre-detalhe ("master-detail"), conforme vimos no capítulo 2.

Um exemplo típico de um formulário com um sub-formulário tabular, que será utilizado ao longo deste capítulo, é apresentado na figura 7.1. A sua estrutura de dados hierárquica é colocada em evidência na figura 7.2. Na figura 7.3 indicam-se algumas restrições de integridade a manter nesse formulário através de regras de restrição ou derivação, representativas do tipo de restrições de integridade que se encontram em aplicações reais.

O modelo de regras introduzido no capítulo 3 continua aplicável, porque as variáveis de estado podem representar dados arbitrariamente complexos. No entanto, os resultados que se obtêm podem ser claramente melhorados se forem considerados os refinamentos tratados nas secções seguintes. Esses refinamentos dividem-se em dois grupos, que têm a ver com a orientação a conjuntos e a orientação a instâncias. Numa abordagem orientada a conjuntos, um restrição é imposta (por uma regra de restrição ou de derivação que manipula conjuntos de dados) para todos os itens de um conjunto de dados, antes de passar à restrição seguinte. Numa abordagem orientada a instâncias, são impostas todas as restrições para um item de dados (instância de um conjunto), antes de passar a outro item de dados. Regras orientadas a conjuntos são normalmente executadas de forma diferida, enquanto que regras orientadas a instâncias são normalmente executadas de forma imediata.

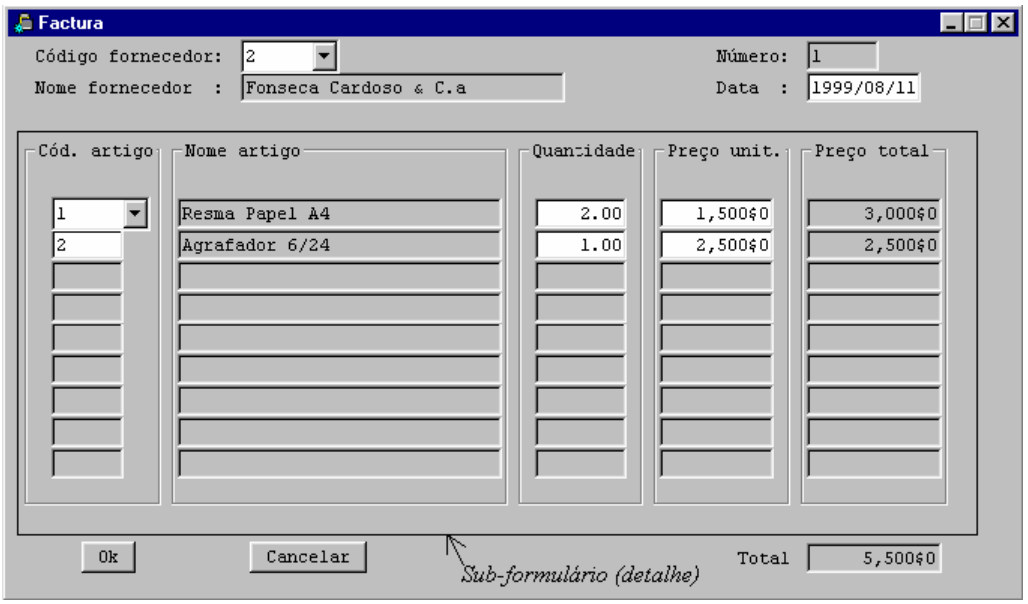


Fig. 7.1 Formulário de ecrã para introdução de facturas com sub-formulário tabular.

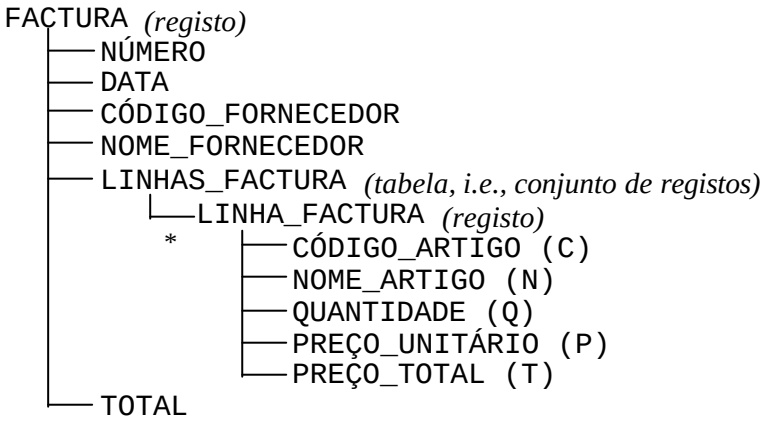


Fig. 7.2 Estrutura de dados hierárquica do formulário da figura 7.1.

R1 - O preço unitário em cada linha da factura não pode ser negativo

R2 - O preço total em cada linha da factura é o produto da quantidade pelo preço unitário

R3 - O total da factura é o somatório do preço total para todas as linhas da factura

R4 - Não podem existir duas linhas de factura com o mesmo artigo

R5 - O código do artigo tem de existir na tabela de artigos (*)

R6 - O nome do artigo é obtido a partir do código do artigo por consulta ("lookup") à tabela de artigos (*)

R7 - O código do fornecedor tem de existir na tabela de fornecedores (*)

R8 - O nome do fornecedor é obtido a partir do código do fornecedor por consulta ("lookup") à tabela de fornecedores (*)

R9 - A data da factura não pode ser posterior à data actual

(*) Supõe-se que as tabelas de artigos e de fornecedores se mantêm inalteradas durante o preenchimento do formulário

Fig. 7.3 Restrições de integridade no formulário da figura 7.1.

7.2 Regras orientadas a conjuntos

A forma mais directa de aplicar a abordagem indicada no capítulo 3 a dados complexos consiste em utilizar regras orientadas a conjuntos (que manipulam conjuntos de dados de cada vez) e variáveis de estado que representam conjuntos de dados. No caso da formulário da figura 7.1, isso significa representar todo o conjunto de linhas da factura por uma única variável de estado (e cada campo do cabeçalho e rodapé da factura por uma variável de estado diferente). Descrevem-se de seguida alguns refinamentos necessário para tornar essa abordagem eficiente.

7.2.1 Refinamento do critério e mecanismo de activação

O critério de activação básico apresentado no capítulo 4 baseia-se no conhecimento estático das variáveis de estado lidas e actualizadas por cada regra e na monitorização dinâmica dos eventos de alteração do estado dessas variáveis de estado (causados pelas operações de actualização). Considera-se um único tipo de operação de actualização (a atribuição de um novo valor que substitui o antigo) e, consequentemente, um único tipo de evento de alteração (a atribuição de um valor diferente do antigo). No caso de variáveis de estado que representam dados complexos, essa simplificação é insatisfatória. Com base no conhecimento estático das formas (ou operações) especializadas por que as variáveis de estado são consultadas e actualizados por cada regra, e na monitorização dinâmica de eventos especializados de alteração do estado dessas variáveis de estado, é possível definir critérios de activação mais precisos e satisfatórios. Isso é ilustrado a seguir através de exemplos para alguns casos importantes. Os critérios de ordenação (definidos no capítulo 5) e as técnicas de análise de terminação e determinismo (definidas no capítulo 6) continuam aplicáveis. Para a análise de determinismo, são úteis as condições suficientes de comutatividade de regras que manipulam conjuntos indicadas em [AWH92] e [AHW95].

7.2.1.1 Caso de regras que manipulam conjuntos

No caso de conjuntos, interessa dividir as operações de actualização (e os eventos de alteração de dados correspondentes) nos seguintes tipos:

- operações que apenas inserem elementos no conjunto;
- operações que apenas eliminam elementos do conjunto;
- operações que podem inserir e eliminar elementos.

Exemplo

Seja o conjunto de regras da figura 7.4, em que todas as variáveis de estado representam conjuntos. Note-se que expressões do tipo "$x' = x \cup \ldots$" e "$x' = x - \ldots$" corresponderiam, numa linguagem prática procedimental, à invocação de operações de inserção e de eliminação.

A restrição imposta por r_1 é $x = x \cup y$, que é equivalente a $y \subseteq x$. Esta restrição não é violada quando se inserem elementos em x ou quando se eliminam elementos de y. Portanto, a regra r_1 não deve ser activada por eventos de alteração de dados desses tipos. A situação é semelhante no caso da regra r_2.

A restrição imposta por r_3 é $y = y - w$, que é equivalente a $y \cap w = \{\}$. Esta restrição não é violada quando se eliminam elementos de y ou de w. Portanto, a regra r_3 não deve ser activada por eventos de alteração de dados desses tipos.

A restrição imposta pela regra r_4 é $z = a \cap b$, a qual pode ser violada por qualquer tipo de alteração de z, a ou b.

Estas conclusões encontram-se traduzidas no grafo r-v anotado da figura 7.4. As anotações traduzem a forma como as variáveis de estado são consultadas e actualizadas por cada regra.

A partir do grafo r-v anotado da figura 7.4, e atendendo ao facto de que todas as regras são idempotentes, obtém-se o grafo de activação (refinado) da figura 7.5-b. Note-se que, se fosse seguido o critério de activação básico, se obteria o grafo de activação da figura 7.5-a. Uma vez que o grafo de activação (refinado) é acíclico, o processamento deste conjunto de regras termina sempre, para qualquer estado inicial e qualquer ordem de execução das regras (de acordo com o teorema 6.1). Uma vez que existem dois pares de regras não confluentes (indicados na figura 7.5-c) não há garantia de que o conjunto de regras seja determinístico (de facto, é fácil de ver que não é determinístico).

Suponhamos que são estabelecidas prioridades entre as regras de acordo com o princípio "calcular antes de usar" (critério 5.1), ou seja, que é estabelecido o seguinte conjunto de prioridades:

$P = \{r_4 > r_2, r_4 > r_1, r_3 > r_1, r_3 > r_2\}$

Facilmente se verifica que o conjunto de regras com este conjunto de prioridades obedece ao requisito de confluência (requisito 6.1), pelo que é determinístico (de acordo com o teorema 6.14).

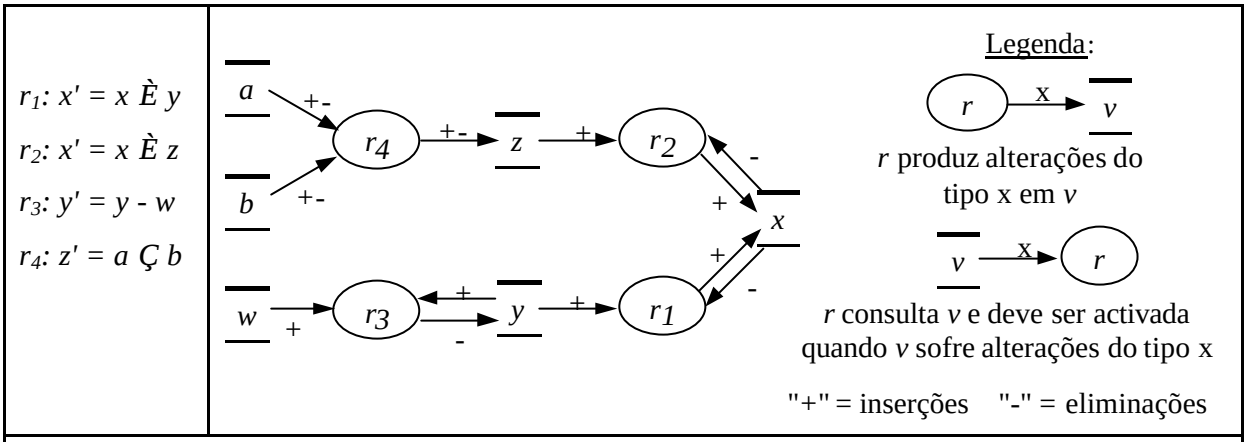


Fig. 7.4 Conjunto de regras que manipulam conjuntos e respectivo grafo r-v anotado.

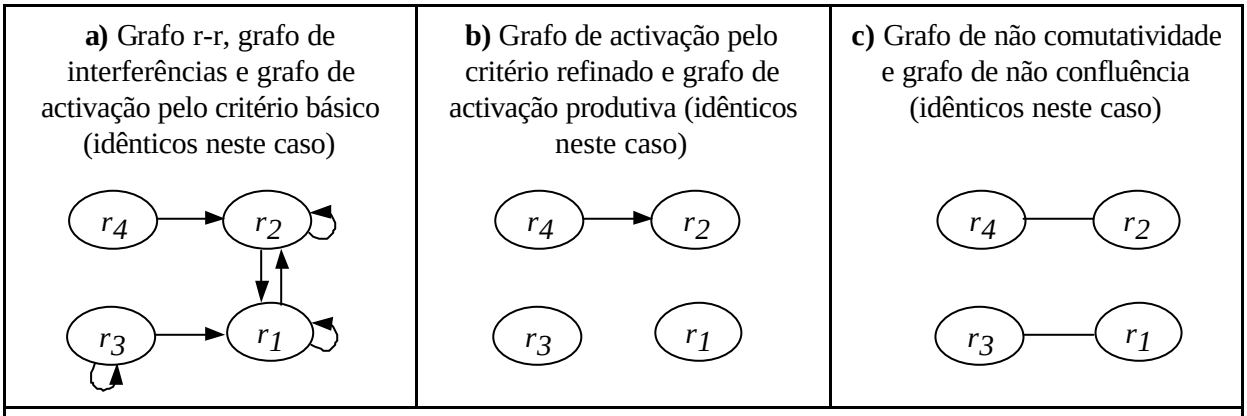


Fig. 7.5 Grafos correspondentes ao conjunto de regras da Fig. 7.4.

7.2.1.2 Caso de regras que manipulam relações

Uma relação (ou tabela) é um conjunto de tuplos (n-uplos) do mesmo tipo. Como tal, as optimizações consideradas na secção anterior permanecem válidas. Apesar da actualização de um tuplo poder ser tratada como um par de uma eliminação seguida de uma inserção, são possíveis mais optimizações se as actualizações forem tratadas de forma especial, nomeadamente quando envolvem apenas alguns atributos (ou colunas).

Assim, interessa considerar as seguintes operações especializadas de actualização de uma relação (e os eventos de alteração de dados correspondentes):

- inserção (+) de um conjunto de tuplos;
- eliminação (-) de um conjunto de tuplos;
- actualização de atributos a_i, a_j, ... num conjunto de tuplos.

Exemplo

Seja o formulário da figura 7.1 com a estrutura de dados indicada na figura 7.2. As variáveis de estado a considerar correspondem aos itens em que se decompõe a factura, ou seja: NÚMERO, DATA, CÓDIGO_FORNECEDOR, NOME_FORNECEDOR, LINHAS_FACTURA e TOTAL. A variável LINHAS_FACTURA representa uma tabela (ou relação).

Para impor a restrição R1 da figura 7.3 pode escrever-se a seguinte regra de restrição em notação abstracta (usando cálculo relacional de n-uplos):

r_1: *($ L Î LINHAS_FACTURA: L.PREÇO_UNITÁRIO < 0) Þ e' = true*

ou (divergindo um pouco da notação definida no capítulo 3):

r_1: *" L Î LINHAS_FACTURA, (L.PREÇO_UNITÁRIO < 0 Þ e' = true)*

Pelo critério de activação básico, esta regra seria activada por qualquer alteração de LINHAS_FACTURA (a única variável de estado consultada pela regra), o que é claramente excessivo. Na realidade, basta reactivar a regra quando é inserida uma linha ou é alterado o PREÇO_UNITÁRIO numa linha.

Para impor a restrição R2 pode escrever-se a seguinte regra de derivação em notação abstracta:

r_2: *" L Î LINHAS_FACTURA,*
*L.PREÇO_TOTAL ' = L.PREÇO_UNITÁRIO * L.QUANTIDADE*

Pelo critério de activação básico, esta regra seria activada por qualquer alteração de LINHAS_FACTURA, o que é claramente excessivo. Na realidade, basta reactivar a regra quando é inserida uma linha, ou é alterado o PREÇO_UNITÁRIO, a QUANTIDADE ou o PREÇO_TOTAL (pelo utilizador ou por outras regras) numa linha.

Para impor a restrição R3 pode escrever-se a seguinte regra em notação abstracta:

r_3: $TOTAL' = \sum_{L \in LINHAS\_FACTURA} L.PREÇO\_TOTAL$

Pelo critério de activação básico, esta regra seria activada por qualquer alteração de LINHAS_FACTURA, e pela alteração do campo TOTAL (pelo utilizador ou por outras regras), o que é claramente excessivo. Na realidade, basta reactivar a regra quando é inserida ou eliminada uma linha, ou é alterado o PREÇO_TOTAL numa linha (ou quando o TOTAL é alterado por outros).

Estas conclusões encontram-se traduzidas no grafo r-v anotado da figura 7.6, que também contempla as regras necessárias para impor as restantes restrições da figura 7.3. A partir do grafo r-v anotado da figura 7.6 pode inferir-se o grafo de activação da figura 7.7-b. Note-se que, se fosse seguido o critério de activação básico, se obteria o grafo de activação da figura 7.7-a. De acordo com os grafos da figura 7.7-b e 7.7-c, conclui-se que o conjunto de regras é determinístico e terminante.

Se as regras forem ordenadas pelo princípio "calcular antes de usar", introduzem-se prioridades indesejáveis (em termos apenas de eficiência): as regras de restrição r_4, r_5 e r_1 perdem precedência em relação às regras de derivação r_6 e r_2. Isso é evitado com a abordagem da secção 7.2.2.

7.2.1.3 Caso de regras que manipulam objectos de classes

No caso de ser seguido um modelo de dados orientado a objectos, a extensão de cada classe (conjuntos de objectos da classe existentes num dado momento) deve ser representada por uma variável de estado. As operações de alteração de dados a considerar são as operações de criação e eliminação de objectos e as operações de modificação do estado dos objectos (normalmente várias em cada classe). Podem considerar-se as mesmas optimizações da secção anterior, porque os atributos dos objectos de um classe (usando a terminologia habitual, como em [UML97]) correspondem aos atributos de uma relação.

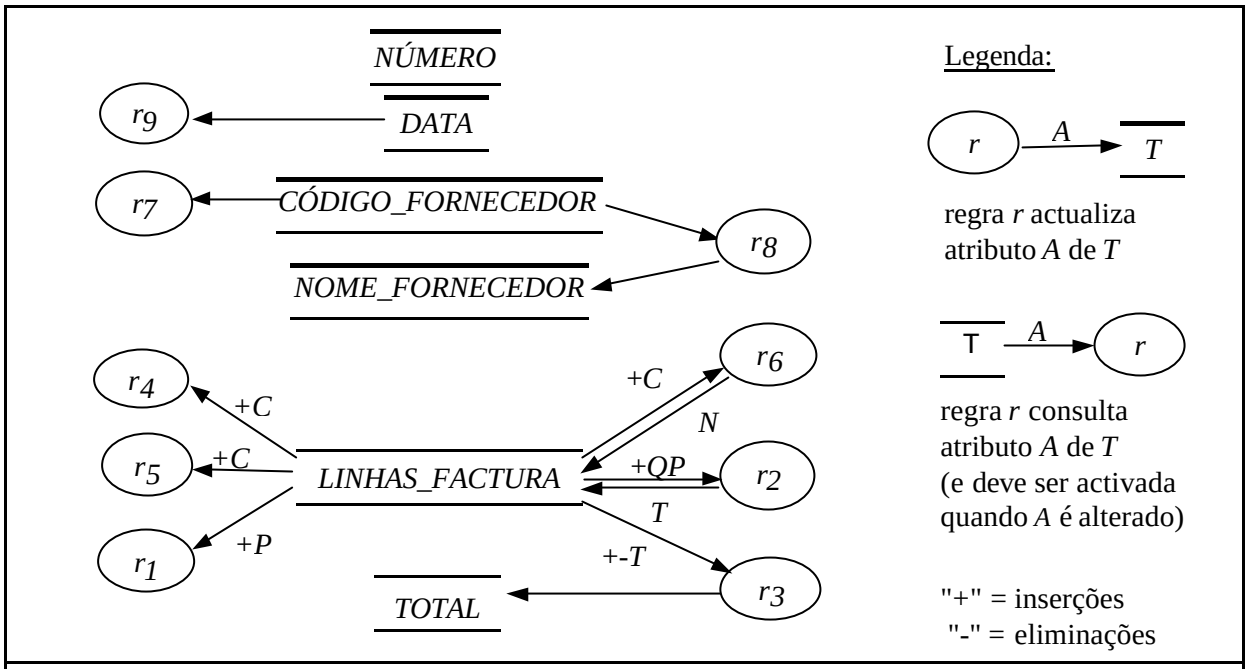


Fig. 7.6 Grafo r-v anotado relativo às regras que impõem as restrições da figura 7.3. São usadas as abreviaturas da figura 7.2. A variável *e* não é representada.

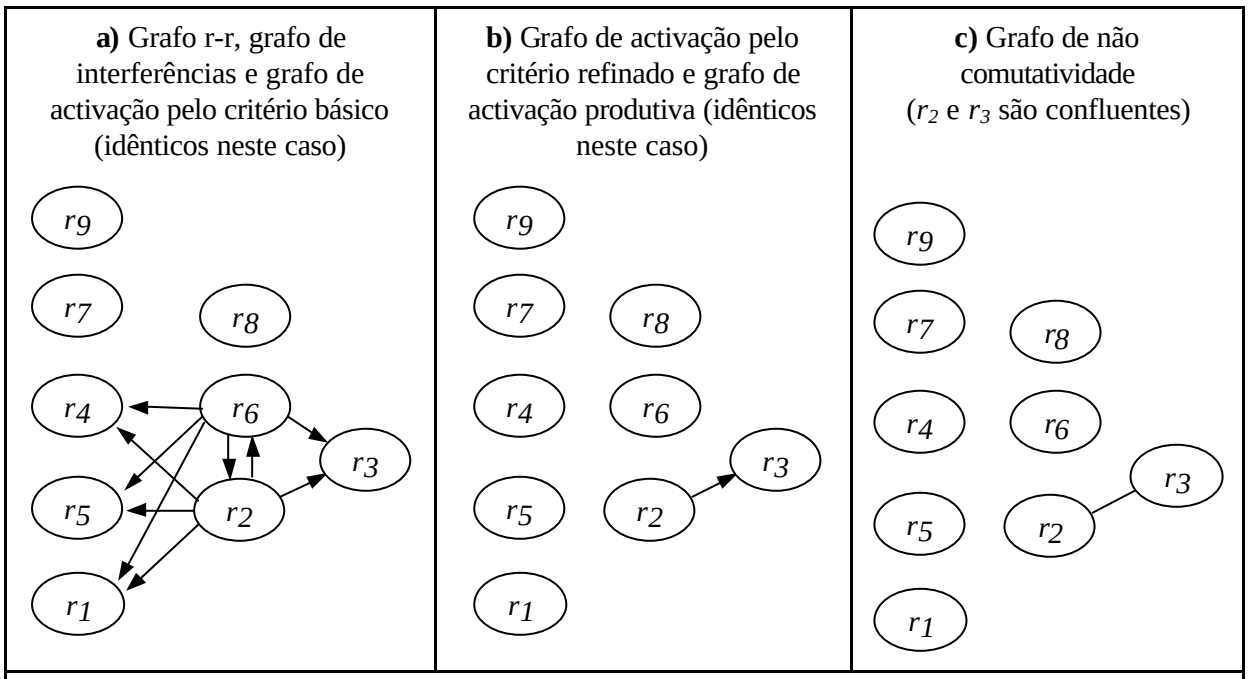


Fig. 7.7 Grafos correspondentes ao conjunto de regras da figura 7.6.

7.2.2 Decomposição de variáveis de estado

Uma relação R com atributos $A_1, A_2, ..., A_n$ e identificador interno dos tuplos I, pode ser vista como um conjunto de n relações binárias $R_1, R_2, ..., R_n$, em que R_i é uma relação binária em I e A_i que dá o valor do atributo A_i em função (definida em extensão) do identificador interno I. Cada uma destas relações binárias pode ser representada por uma variável de estado com o mesmo nome do atributo (ou com nome $R.A_i$ para evitar ambiguidades). Tem-se assim uma variável de estado para cada atributo (coluna) de R. Opcionalmente, pode considerar-se uma variável de estado adicional, com o mesmo

nome da relação (ou com nome *R.I* para evitar ambiguidades), para representar o conjunto de identificadores de tuplos. Esta abordagem é igualmente válida em relação a classes de objectos com atributos.

Esta abordagem permite aumentar a precisão do grafo r-v e, consequentemente, aumentar a eficácia dos critérios de activação e ordenação e das técnicas de análise de determinismo e terminação definidos nos capítulos anteriores.

Note-se que a inserção ou eliminação de um tuplo numa relação *R* corresponde à inserção ou eliminação de um tuplo em todas as relações binárias correspondentes aos atributos de *R* (a menos que não sejam representados tuplos com valores nulos nos atributos). Em contrapartida, na consulta e actualização de tuplos de *R* podem estar envolvidas apenas algumas dessas relações binárias.

Exemplo

Relativamente ao formulário da figura 7.1, pode desdobra-se a variável de estado `LINHAS_FACTURA` (tabular) nas seguintes variáveis de estado, correspondentes às colunas da tabela: `CÓDIGO_ARTIGO`, `NOME_ARTIGO`, `QUANTIDADE`, `PREÇO_UNITÁRIO` e `PREÇO_TOTAL`.

Com esta escolha de variáveis de estado, obtém-se o grafo r-v anotado da figura 7.8. O grafo r-r (e de interferências) que se obtém a partir deste grafo r-v coincide com o grafo da figura 7.7-b, que é mais preciso do que o obtido sem esta escolha de variáveis de estado (ver figura7.7-a). A ordenação das regras pelo princípio "calcular antes de usar" também é mais precisa. A análise de comutatividade e confluência é claramente facilitada.

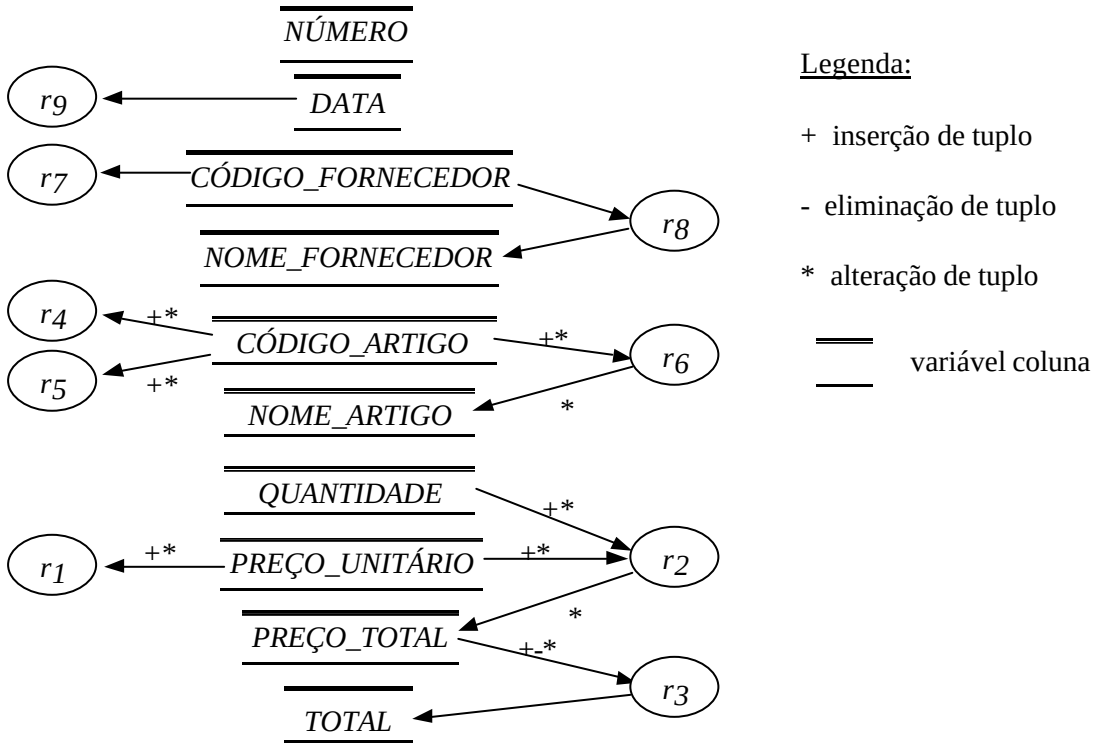


Fig. 7.8 Grafo r-v anotado com algumas variáveis de estado correspondentes a colunas.

7.2.3 Regras incrementais

Em alguns casos, o efeito de uma regra *r* orientada a conjuntos pode ser obtido de forma mais eficiente, se for conhecido e poder ser referenciado na expressão de *r* um estado passado especial

(das variáveis de estado), com poucas diferenças em relação ao estado corrente (das variáveis de estado). Regras desse tipo dizem-se incrementais ou diferenciais.

Podem interessar estados passados nos seguintes instantes de tempo especiais:

- no início da transacção;

  Este estado tem interesse porque é consistente com todas as regras existentes no início da transacção e já tem de ser conhecido para efeito de "rollback".

- no fim da última execução de *r* ou no início da transacção, conforme o que for mais recente;

  O estado no fim da última execução de *r* também é consistente com *r*, desde que *r* seja idempotente. O acesso a este estado é útil para optimizar a execução das regras que são executadas várias vezes na mesma transacção. No entanto, exige o armazenamento de mais informação.

- no início da última execução de *r* ou no início da transacção, conforme o que for mais recente.

  O acesso a este estado é útil para optimizar a execução das regras que se auto-activam (porque não são idempotentes).

No modelo de regras do capítulo 3, exige-se que o estado *s'* atingido no final da execução de uma regra *r* só dependa do estado *s* existente no início da execução de *r*, e não de estados passados. A escrita de regras na forma incremental (com referência a estados passados especiais) não implica necessariamente a violação desta exigência. O que está em causa aqui é obter o mesmo efeito de uma regra não incremental *r*, do tipo *s'=r(s)*, através de uma regra incremental (ou diferencial) *r'* do tipo *s'=r'(s,old(s))=r(s)*, sem violar a exigência acima referida.

Normalmente, não interessa referenciar directamente o estado passado, mas sim as diferenças entre o estado passado e o estado corrente. Essas diferenças são normalmente dadas por *variáveis delta* (uma ou mais para cada variável de estado), cuja definição depende da estrutura interna das variáveis de estado, conforme se verá a seguir. As variáveis delta dão o efeito líquido de sequências de alterações.

7.2.3.1 Regras incrementais que manipulam conjuntos

Para cada variável de estado *x* que representa um conjunto, o incremento entre um estado passado denotado por *old(x)* e o estado corrente denotado por *x* pode ser dado por duas variáveis delta:

$D^+x = x - old(x)$	-	elementos inseridos em *x*, em termos líquidos (i.e., contém os elementos inseridos que não foram posteriormente eliminados)
$D^-x = old(x) - x$	-	elementos eliminados de *x*, em termos líquidos (i.e., contém os elementos eliminados que não foram posteriormente reinseridos)

Exemplo

Seja a seguinte regra de derivação não incremental que atribui a uma variável *y* o somatório dos valores de um conjunto *x*:

r_1: $y' = \sum_{i \in x} i$

A regra incremental correspondente é:

r_{1i}: $y' = old(y) + \sum_{i \in D^+x} i - \sum_{i \in D^-x} i$

Da primeira vez que a regra é executada (na transacção em que a regra é criada), o estado no início da transacção pode não ser consistente com r_1, a não ser que a regra seja criada quando as variáveis de estado têm um estado especial, como por exemplo $y=0$ e $x=\{\}$. Assim, a forma não incremental pode ser necessária para a 1ª execução. Após muitas alterações, a forma incremental também pode ser menos eficiente do que a forma não incremental.

Se *y* só é actualizado por esta regra e o estado passado referenciado é o estado no fim da última execução da regra ou no início da transacção, conforme o que for mais recente, pode-se substituir *old(y)* por *y*, ficando:

r_{1i}: $y' = y + \sum_{i \in D^+(x)} i - \sum_{i \in D^-(x)} i$

Em todo o caso, as variáveis de entrada e saída da regra devem ser estabelecidas com base na regra não incremental equivalente (r_1), pelo que *y* não deve ser considerado uma variável de entrada da regra.

Exemplo

Seja a seguinte regra de restrição não incremental que verifica uma restrição de integridade em todos os elementos de um conjunto:

r_1: $(\exists\, i \in x: i < 0) \Rightarrow e' = true$

ou, equivalentemente (com notação diferente da introduzida no capítulo 3):

r_1: $\forall i \in x, (i < 0 \Rightarrow e' = true)$

A regra incremental correspondente é:

r_{1i}: $(\exists\, i \in D^+x: i < 0) \Rightarrow e' = true$

ou, equivalentemente:

r_{1i}: $\forall i \in D^+x, (i < 0 \Rightarrow e' = true)$

Exemplo

A tabela seguinte mostra as regras incrementais correspondentes às regras não incrementais (e idempotentes) da figura 7.4.

Regra não incremental	Regra incremental
r_1: $x' = x \cup y$	r_1: $x' = x \cup D^+y$
r_2: $x' = x \cup z$	r_2: $x' = x \cup D^+z$
r_3: $y' = y - w$	r_3: $y' = y - D^+w$
r_4: $z' = a \cap b$	r_4: $z' = old(z) - D^-a - D^-b \cup (D^+a \cap b) \cup (D^+b \cap a)$

7.2.3.2 Regras incrementais que manipulam relações

Uma vez que as relações são conjuntos, podem-se usar as variáveis delta da secção anterior. Isso corresponde a tratar a alteração de um tuplo de uma relação como uma eliminação seguida de uma inserção. No entanto, obtém-se uma precisão superior se forem consideradas variáveis delta correspondentes às 3 operações habituais de manipulação de relações: inserir (*insert*), eliminar (*delete*) e actualizar (*update*) tuplos da relação. Para cada variável de estado *x* que representa uma relação, o incremento entre um estado passado denotado por *old(x)* e o estado corrente denotado por *x* pode ser dado pelas seguintes variáveis (relações) delta:

ins(x)	-	tuplos inseridos, em termos líquidos (i.e., contém os tuplos inseridos, ou inseridos e depois alterados, no seu novo estado; não contém os tuplos inseridos e depois eliminados)
del(x)	-	tuplos eliminados, em termos líquidos (i.e., contém os tuplos eliminados, ou alterados e depois eliminados, no seu estado antigo)
old_upd(x)	-	tuplos alterados, em termos líquidos, no seu estado antigo (em *old(x)*)
new_upd(x)	-	tuplos alterados, em termos líquidos, no seu novo estado (em *x*)

Em alternativa às relações *old_upd(x)* e *new_upd(x)* também se pode considerar uma única relação *upd(x)*, com colunas $old(a_i)$ e $new(a_i)$ (ou simplesmente a_i), com o estado antigo e o novo estado de cada atributo (coluna) a_i de *x*. Se os atributos de *x* forem $a_1, .., a_n$, tem-se que:

$old\_upd(x) = p_{\,old(a_1),\ \ldots,\ old(a_n)}\ upd(x)$

$new\_upd(x) = p_{\,a_1,\ldots,\ a_n}\ upd(x)$

Note-se que só é possível obter *old_upd(x)* e *new_upd(x)* a partir de *x* e *old(x)*, se os tuplos tiverem algum tipo de identificador interno único e imutável (pelo menos durante uma transacção), que permita dizer que dois tuplos de *x* e *old(x)* são o mesmo objecto em estados diferentes. É o que acontece nos sistemas Starbust [WF90][W96], Ariel [H92][H96] e Chimera [CFPB96], que usam variáveis delta (também chamadas *tabelas de transição*) deste tipo.

Uma vez que os identificadores internos não são relevantes para o utilizador, as variáveis delta definidas desta forma têm alguns inconvenientes. Suponhamos que os identificadores internos são ignorados (ou eliminados) nas variáveis delta. Desde que não possam existir dois tuplos com os mesmos valores (o que acontece se todas as relações tiverem pelo menos uma chave), garante-se que:

$ins(x) \cap new\_upd(x) = \emptyset$ e $del(x) \cap old\_upd(x) = \emptyset$

No entanto, não se garante que:

$[del(x) \cup old\_upd(x)] \cap [ins(x) \cup new\_upd(x)] = \emptyset$

Isto acontece porque pode ser inserido ou alterado um tuplo com novos valores iguais aos valores antigos doutro tuplo eliminado ou alterado.

Assim, tem-se, de forma pouco natural:

$D^+x = x - old(x) = [ins(x) \cup new\_upd(x)] - [del(x) \cup old\_upd(x)]$

$D^-x = old(x) - x = [del(x) \cup old\_upd(x)] - [ins(x) \cup new\_upd(x)]$

Já no caso do sistema A-RDL [SK96], as variáveis delta são definidas de forma a garantir que:

$[ins(x) \cup new\_upd(x)] \cap [del(x) \cup old\_upd(x)] = \emptyset$

pelo que, nesse caso:

$D^+x = x - old(x) = ins(x) \cup new\_upd(x)$

$D^-x = old(x) - x = del(x) \cup old\_upd(x)$

Exemplo

Para a imposição incremental da restrição R3 da figura 7.3 (cálculo do total da factura) pode escrever-se a seguinte regra incremental:

r_{3i}: $TOTAL' = old(TOTAL) +$
$+ \sum_{L \in ins(LINHAS\_FACTURA)} L.PREÇO\_TOTAL$
$- \sum_{L \in del(LINHAS\_FACTURA)} L.PREÇO\_TOTAL$
$+ \sum_{L \in upd(LINHAS\_FACTURA)} (L..PREÇO\_TOTAL - old(L.PREÇO\_TOTAL))$

Para a imposição incremental da restrição R2 (cálculo do preço total de cada linha) pode escrever-se a seguinte regra incremental:

r_{2i}: $\forall L \in ins(LINHAS\_FACTURA) \cup upd(LINHAS\_FACTURA),$
$L.PREÇO\_TOTAL' = L.PREÇO\_UNITÁRIO * L.QUANTIDADE$

Para a imposição incremental da restrição R4 (ausência de duas linhas com o mesmo código de artigo) pode escrever-se a seguinte regra incremental:

r_{4i}: $\forall L_1 \in ins(LINHAS\_FACTURA) \cup upd(LINHAS\_FACTURA),$
$(\exists L_2 \in LINHAS\_FACTURA: L_2 \neq L_1 \wedge L_2.CÓDIGO\_ARTIGO = L_1.CÓDIGO\_ARTIGO)$
$\Rightarrow e' = true$

Exemplo

Suponhamos que o preço total de cada linha da factura é acrescido de IVA à taxa indicada num campo TAXA_IVA do cabeçalho da factura, da seguinte forma:

r'2: "L Î LINHAS_FACTURA,
*L.PREÇO_TOTAL ' = L.PREÇO_UNITÁRIO * L.QUANTIDADE * (1 + TAXA_IVA)*

Neste caso, a alteração de TAXA_IVA obriga a recalcular o preço total para todas as linhas (de forma não incremental). Se for possível, no corpo da regra, testar se TAXA_IVA foi alterado, é possível tratar todos os casos numa única regra, como em:

r'2i: ∀L Î LINHAS_FACTURA: updated(TAXA_IVA) Ú
L Î ins(LINHAS_FACTURA) È upd(LINHAS_FACTURA),
*L.PREÇO_TOTAL ' = L.PREÇO_UNITÁRIO * L.QUANTIDADE * (1 + TAXA_IVA)*

Em alternativa, podem-se ter duas regras, uma incremental e outra não incremental, activadas por eventos diferentes, sendo a regra incremental desactivada sempre que a regra incremental é activada.

Exemplo

Seja uma relação *A* com atributos *x* e *y*, e seja a seguinte regra não incremental para obter o fecho transitivo de *A*:

r_1: $A' = A \; È \; \rho_{A.x,\, B.y}\,[A \bowtie_{A.y=B.x} r(B,A)]$ (*r* é o operador de renomeação de [R98])

A regra incremental correspondente é:

r_{1i}: $A' = A \; È \; \rho_{(D^+A).x=A.y}\,[D^+A \bowtie_{(D^+A).y=A.x} A] \; È \; \rho_{A.x=(D^+A).y}\,[A \bowtie_{A.y=(D^+A).x} D^+A]$

Neste caso, o estado antigo que interessa referenciar é o estado no início da última execução da regra, porque a regra r_1 não é idempotente.

7.2.3.3 Discussão

A geração automática de regras incrementais a partir de regras não incrementais só é possível em casos relativamente simples. Podem ser adaptados para esse fim os métodos descritos em [CW90], [CW91] e [CW94] para a geração automática de regras ECA incrementais para a manutenção de restrições de integridade e dados derivados a partir de especificações de alto nível, assim como os métodos descritos em [BW95] para a avaliação incremental de condições sobre conjuntos, e o cálculo de diferenciação parcial ("partial differencing calculus") descrito em [RS98].

Em geral, será o programador a escrever as regras directamente na forma incremental, ou simultaneamente na forma incremental e na forma não incremental (porque a forma incremental pode não ser aplicável na 1ª execução da regra ou quando ocorrem certos eventos). Em certos casos, os eventos activadores podem ser inferidos facilmente a partir das variáveis delta referenciadas em cada regra.

A grande desvantagem das regras incrementais prende-se com a manutenção das variáveis delta (ou dos estados passados a partir dos quais elas se podem obter), nomeadamente quando se usam os estados intermédios no início ou no fim da última execução de cada regra, para além do estado no início da transacção. As regras orientadas a instâncias tratadas a seguir constituem uma alternativa interessante porque permitem dispensar as variáveis delta, à custa de uma semântica mais complexa.

7.3 Regras orientadas a instâncias

As regras orientadas a instâncias aplicam-se a uma instância de cada vez (de um conjunto, classe ou relação), podendo ser executadas imediatamente, assim que ocorre um evento activador relacionado com uma instância. Muitas restrições de integridade e dados derivados em dados complexos podem ser mantidos eficientemente através de regras orientadas a instâncias, sem necessidade de recorrer a variáveis delta.

Regras orientadas a instâncias encontram-se mais habitualmente em sistemas de regras activas para SGBD orientados a objectos, de que se destacam o sistema Ode [GJ91] [GJ96], sobre C++ com persistência, e o sistema NAOS [CC95][C98], para o SGBDOO comercial O_2. Estes dois sistemas suportam tanto regras orientadas a instâncias como a conjuntos. Note-se que, no âmbito dos SGBD relacionais, a norma SQL3 e os SGBD comerciais mais importantes também prevêem gatilhos orientados a instâncias ("for each row") e a conjuntos ("for each statement"), embora os gatilhos orientados a conjuntos sejam processados imediatamente após a execução de cada instrução SQL e não no fim da transacção (conforme vimos nas secções 2.3.2 e 2.3.3).

O modelo de execução de regras orientadas a instâncias é significativamente diferente do modelo de execução de regras orientadas a conjuntos. No caso de regras orientadas a conjuntos, basta um único PPR no final de cada transacção (ou em "checkpoints" intermédios), onde as regras são executadas de forma diferida. No caso de regras orientadas a instâncias, ocorrem múltiplos PPR's encaixados ("nested"), onde as regras são executadas de forma imediata.

Nas secções seguintes discutem-se várias abordagens para a integração de regras orientadas a instâncias (em combinação com regras orientadas a conjuntos) no modelo de regras dirigidas pelos dados introduzido no capítulo 3. Considera-se um modelo de dados orientado a objectos, que se julga mais adequado para capturar a complexidade dos interfaces para o utilizador que se encontram em aplicações interactivas de bases de dados (mesmo que estas sejam relacionais). Considera-se também um modelo de transacções encaixadas ("nested") [GR93], que permite tirar melhor partido das regras orientadas a instâncias do que o modelo de transacções planas ("flat"). A principal dificuldade a considerar tem a ver com a passagem de um modelo de execução com PPR's não encaixados para um modelo de execução com PPR's encaixados, onde as interferências entre as regras são mais complexas.

7.3.1 Imposição de restrições intra-objecto através de regras intra-objecto puras

Descreve-se de seguida sumariamente a forma como o modelo de regras introduzido no capítulo 3 pode ser adaptado ao contexto de um modelo de dados orientado a objectos, com regras orientadas a instâncias e a conjuntos, sem violar os pressupostos definidos no capítulo 3.

7.3.1.1 Modelo de dados e modelo de definição das regras

Em geral, os dados de um sistema estão organizados hierarquicamente, com objectos de dados mais pequenos e mais simples fazendo parte (como sub-objectos ou objectos componentes) de objectos de dados maiores e mais complexos. Nas folhas da hierarquia estão objectos atómicos, não decomponíveis. Na raiz da hierarquia está o sistema completo visto como um objecto que agrega todos os outros objectos do sistema. Para o efeito que nos interessa aqui, os atributos de um objecto e as referências que um objecto mantém para outros objectos são tratados como objectos componentes.

Qualquer objecto da hierarquia de objectos pode ser visto como um sistema em pequena escala, com *regras intra-objecto* e *variáveis de estado intra-objecto*. Não são violados os pressupostos estabelecidos no capítulo 3, desde que se verifiquem certos pressupostos *de encapsulamento* (interessantes de um ponto de vista de Engenharia de Software).

Assim, as variáveis de estado intra-objecto de um objecto composto devem representar objectos componentes que não interagem directamente entre si e que só podem ser acedidos pelos métodos (operações) do objecto composto. Objectos componentes deste tipo correspondem em C++ [S97] a membros dados ("data members") privados. No caso de objectos compostos do tipo *registo*, que se dividem num número fixo de objectos componentes de natureza diferente (de tipos diferentes ou em papéis diferentes), cada objecto componente pode ser representada por uma variável de estado. No caso de objectos compostos do tipo *conjunto*, que se dividem num número variável de objectos componentes do mesmo tipo, todo o conjunto tem de ser representado por uma única variável de estado. Objectos atómicos são também representados por uma única variável de estado.

Estes objectos componentes que não interagem directamente entre si são integrados através dos métodos (operações) e regras associadas ao objecto composto, que representam, respectivamente, o comportamento invocável e reactivo do objecto composto. Regras dirigidas pelos dados associadas a um objecto composto (regras intra-objecto ao nível do objecto composto) podem ser usadas para manter dados derivados e restrições de integridade entre os seus componentes.

Para efeito da definição das regras intra-objecto, interessa considerar classes de objectos, e definir as regras intra-objecto uma única vez para todos os objectos da mesma classe, possivelmente junto com a definição da classe. Obviamente, as regras de uma classe devem ser herdadas pelas suas sub-classes.

7.3.1.2 Modelo de execução das regras

As regras dirigidas pelos dados associadas a um dado objecto, destinadas à manutenção de restrições e dados derivados, devem ser executadas em PPR's no fim dos métodos de criação e de alteração do estado do objecto invocados do seu exterior (correspondem a construtores e funções públicas sem qualificador `const` em C++). Em cada PPR, as regras são executadas sequencialmente. Quando o objecto é criado, todas as regras estão activadas (é como se as regras fossem criadas nessa altura). Se o processamento de regras for abortado no PPR associado à operação de criação do objecto, o objecto deve ser subsequentemente destruído. As regras são reactivadas pelos eventos de alteração das variáveis de estado intra-objecto que ocorrem no decurso da execução desses métodos. Uma operação ou regra de um objecto pode invocar outra operação sobre o mesmo objecto. Para garantir que as regras são atómicas umas em relação às outras, deve existir um PPR apenas na operação mais externa.

Uma regra associada a um objecto composto o_1, executada num PPR p_1 de uma operação (método) m_1 de o_1, pode invocar uma operação m_2 sobre um objecto componente o_2, com um PPR p_2 na qual é executada uma regra r_2 associada a o_2. Diz-se que o PPR p_2 ocorre de forma encaixada em p_1 e que a regra r_2 é executada de forma encaixada em r_1. A ocorrência de PPR's encaixados ("nested") e a consequente execução das regras de forma encaixada constitui uma diferença significativa em relação à situação em que as regras são executadas sequencialmente num único PPR.

Para suportar regras de restrição, supõe-se que as modificações do estado do sistema (raiz da hierarquia de objectos) são protegidas por transacções. Essas transacções constituem as operações (métodos) de alteração do estado do objecto-raiz. As operações de manipulação dos outros objectos da hierarquia de objectos podem ser ou não protegidas por sub-transacções (com "rollback" local), isto é, podem ser ou não transaccionais, de acordo com o modelo de transacções encaixadas ("nested"). O facto de as regras serem executadas nas mesmas operações em que ocorrem os eventos activadores, garante que, no caso de uma operação transaccional ser abortada, tanto os eventos como os efeitos das regras activadas por esses eventos são desfeitos.

7.3.1.3 Exemplos

Nos exemplos seguintes, utiliza-se a seguinte notação para definir regras intra-objecto (regra como função-membro de classe em C++ com qualificador `rule`):

```
rule class_name::rule_name(){ statements }
```

e a seguinte notação para explicitar os eventos activadores:

```
rule class_name::rule_name() on events { statements }
```

Os eventos que interessa considerar são a criação do objecto (`create`) e a modificação do estado de uma variável de estado intra-objecto (`modify(x)`) ou de qualquer parte do objecto (`modify`). Outros eventos possíveis seriam a eliminação do objecto (`destroy`), e a inserção (`insert`), actualização (`update`) ou eliminação (`delete`) de elementos num objecto do tipo conjunto. Salvo indicação em contrário com o prefixo "`self`", excluem-se os eventos gerados durante a execução da regra (gerados pela própria regra ou por regras executadas de forma encaixada). Sempre que é necessário referir o objecto alvo no corpo de uma regra, usa-se a palavra chave `this`, como é habitual nas funções-membro em C++.

Nos exemplos que se seguem, o corpo das regras (instruções entre chavetas) é escrito em pseudo-código baseado em C++ e SQL.

Exemplo

Seja de novo o formulário da figura 7.1. Neste caso, interessa associar regras a objectos de dois tipos: `FACTURA` e `LINHA_FACTURA`. As variáveis de estado de um objecto do tipo `FACTURA` são: `NÚMERO`, `DATA`, `FORNECEDOR`, `LINHAS_FACTURA` e `TOTAL`. As variáveis de estado de um objecto do tipo `LINHA_FACTURA` são: `ARTIGO`, `QUANTIDADE`, `PREÇO_UNITÁRIO` e `PREÇO_TOTAL`. Para obedecer aos pressupostos de encapsulamento, as operações de criação, alteração ou eliminação de linhas de factura devem ser englobadas em operações de alteração ou criação da factura.

As regras de integridade R1, R2, R5 e R6 (ver figura 7.3) são claramente internas às linhas da factura, pelo que se podem associar aos objectos do tipo `LINHA_FACTURA`.

A regra `r1` pode escrever-se (já com os eventos explicitados, inferidos a partir do corpo da regra):

```
rule LINHA_FACTURA::r1()
on create, modify(PREÇO_UNITÁRIO)
{ if (PREÇO_UNITÁRIO < 0) abort(); }
```

A regra `r2` pode escrever-se:

```
rule LINHA_FACTURA::r2()
on create, modify(QUANTIDADE), modify(PREÇO_UNITÁRIO),
   modify(PREÇO_TOTAL)
{ PREÇO_TOTAL = QUANTIDADE * PREÇO_UNITÁRIO; }
```

A regra `r3` continua a escrever-se de forma semelhante, mas agora associada explicitamente a objectos do tipo `FACTURA`:

```
rule FACTURA::r3()
on create, modify(TOTAL), modify(LINHAS_FACTURA)
{ TOTAL = select sum(PREÇO_TOTAL) from LINHAS_FACTURA; }
```

Trata-se de uma regra orientada a conjuntos (porque se aplica a um conjunto de instâncias da classe `LINHA_FACTURA`) e ao mesmo tempo orientada a instâncias (porque se aplica a uma instância da classe `FACTURA`). Podem aplicar-se as optimizações referidas anteriormente para regras orientadas a conjuntos (refinamento dos eventos activadores e escrita da regra de forma incremental).

Uma vez que as alterações de linhas de factura são englobadas em operações de criação ou alteração da própria factura, é primeiro calculado o preço total de cada linha da factura, e só depois é que é calculado o total da factura.

7.3.1.4 Dificuldades devidas à presença de regras encapsuladas nos sub-objectos actualizados por uma regra

Um dos pressupostos considerados no capítulo 3 é de que os estados consistentes com uma regra são os seus pontos fixos, em termos líquidos. Ora, os pontos fixos, em termos líquidos, de uma regra associada a um objecto composto podem depender das regras encapsuladas nos objectos componentes, complicando assim a semântica das regras. Note-se que isso não aconteceria se os estados consistentes com uma regra fossem definidos por uma condição sem efeitos laterais (uma condição cuja avaliação não causa alterações que possam desencadear a execução de regras), como acontece no sistema Ode (conforme vimos na secção 2.3.6).

Outro dos pressupostos considerados no capítulo 3 é o de que as regras são individualmente determinísticas. Ora, se o conjunto de regras encapsuladas num objecto componente não é determínistico, as operações e as regras associadas ao objecto composto deixam de ser determinísticas, o que viola aquele pressuposto. Para evitar isso, é importante forçar o determinismo, nem que seja através de uma ordenação total das regras escolhida pelo sistema.

Alguns refinamentos do critério de activação anteriormente considerados para regras orientadas a conjuntos (associadas a objectos do tipo conjunto) podem deixar de ser válidos quando os objectos componentes têm regras neles encapsuladas, conforme ilustram os exemplos a seguir.

Exemplo

Seja um tipo de tabela `t` constituída por registos do tipo `r`, com campos `x` e `y`. Sejam as seguintes regras:

```
rule t::r1() { update this set x = 1; }
rule r::r2() { if (x > y) x = y; }
```

As regras `r1` e `r2` são potencialmente contraditórias, embora de uma forma especial, pois encontram-se em níveis diferentes da hierarquia de objectos. O efeito líquido completo da regra `r1`, entrando em conta com a possível execução encaixada de `r2`, pode ser expresso por:

```
{ update this set x = min(1, y); }
```

Pelo critério de activação básico, a regra `r1` seria activada (de forma segura) por qualquer alteração da tabela-alvo da regra, o que pode ser explicitado por:

```
rule t::r1() on create, modify
{ update this set x = 1; }
```

Se não existisse a regra `r2`, bastaria considerar os seguintes eventos activadores para `r1`:

```
rule t::r1()
on create,   /* criação da tabela */
   insert,   /* inserção de linha na tabela */
   update(x) /* actualização de x numa linha da tabela */
{ update this set x = 1; }
```

Mas, atendendo à existência da regra `r2`, um critério de activação seguro tem de atender ao efeito líquido completo, pelo que seria necessário acrescentar "`update(y)`" à lista de eventos anterior. Note-se que se excluem os eventos gerados por `r1` ou por regras executadas de forma encaixada (neste caso `r2`) durante a execução de `r1`.

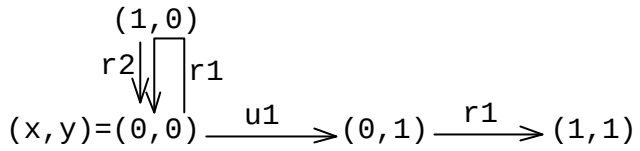
Considere-se um registo com estado inicial `(x,y)=(0,0)`. Este estado é consistente com a regra `r2`, e também é consistente com a regra `r1` porque o efeito líquido da aplicação de `r1` a partir desse estado é nulo (o estado final coincide com o estado inicial), apesar da execução de `r1` causar uma alteração de estado que é contrariada por `r2`.

Suponhamos agora que é submetido o seguinte comando do utilizador:

`u1: update t set y=1` (em que `t` é uma tabela do tipo `t`)

No caso do registo com estado inicial `(x,y)=(0,0)`, o comando do utilizador gera uma operação de actualização intra-registo, em que o estado passa para `(0,1)`; a regra `r2` é executada imediatamente, mas sem qualquer efeito. No final da actualização de toda a tabela pelo comando do

utilizador, é executada a regra `r1`. No caso do registo referido, a regra `r1` gera uma operação de actualização intra-registo em que o estado passa de `(0,1)` para `(1,1)`; a regra `r2` é executada imediatamente, mas sem qualquer efeito. Esta situação é esquematizada no seguinte grafo de execução instanciado:



Assim, conclui-se que a alteração de `y` efectivamente torna a execução da regra `r1` produtiva, apesar de `y` não ser referenciado em `r1`.

Em geral, o problema ilustrado no exemplo anterior prende-se com as regras que efectuam actualizações de estado contraditórias, por si só ou por intermédio de regras nelas encaixadas. Mais precisamente, prende-se com as regras que, mesmo em estados consistentes em sentido líquido, geram mudanças de estado que são anuladas por regras encapsuladas nos sub-objectos, ou geram por si só mudanças de estado que se cancelam mutuamente, havendo regras associadas aos sub-objectos que são executadas em estados intermédios. As actualizações contraditórias não podem simplesmente ser proibidas. Um caso evidente é o de "rollback" local, ilustrado no exemplo seguinte.

Exemplo

Sejam as seguintes regras:

```
rule t::r1() { update this set x = 1; }
rule r::r2() { if (x > y) abort(); }
```

Suponhamos que a actualização de cada registo (por `r1`) é efectuada numa sub-transacção, de forma que a invocação de `abort` em `r2` apenas cancela a actualização de um registo, e não impede a actualização de outros registos. A regra `r2` impõe a mesma restrição ($x \leq y$) que no exemplo anterior, mas de maneira diferente. No caso do estado inicial e do comando do utilizador do exemplo anterior, a situação é em tudo semelhante.

O exemplo seguinte ilustra o caso das regras que geram por si só mudanças de estado que se cancelam mutuamente, havendo regras associadas aos sub-objectos que são executadas em estados intermédios.

Exemplo

Seja um tipo de tabela `t` constituída por registos do tipo `r`, com campos `x`, `y` e `z`. Sejam as seguintes regras:

```
rule t::r1() { update this set x = 1; update this set x = 0; }
rule r::r2() { if (x == 1) z = y; }
```

A regra `r1` gera duas actualizações contraditórias, admissíveis se a noção de consistência for entendida em sentido líquido. O efeito líquido completo da regra `r1`, entrando em conta com a possível execução encaixada de `r2`, pode ser expresso por:

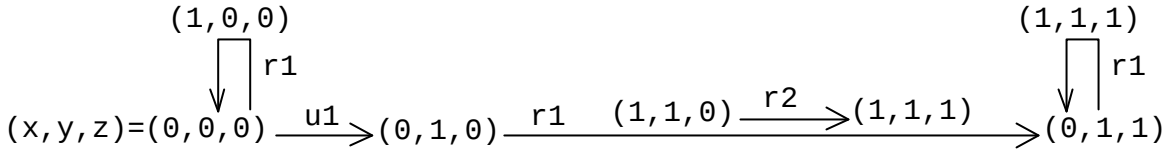
```
{ update this set y = 1, x = 0; }
```

Considere-se um registo com estado inicial `(x,y,z)=(0,0,0)`. Este estado é consistente com a regra `r2`, e também é consistente com a regra `r1` porque o efeito líquido da aplicação de `r1` a partir desse estado é nulo, apesar da execução de `r1` causar alterações de estado contraditórias.

Suponhamos agora que é submetido o seguinte comando do utilizador:

`u1: update t set y=1` (em que `t` é uma tabela do tipo `t`)

O estado do registo com estado inicial `(x,y,z)=(0,0,0)` evolui da forma esquematizada no seguinte grafo de execução instanciado, em resposta a este comando do utilizador:



Assim, conclui-se que a alteração de `y` efectivamente torna a execução da regra `r1` produtiva, apesar de `y` não ser referenciado em `r1`.

Normalmente, as regras são escritas de tal forma que não ocorrem actualizações contraditórias, pelo que não é necessário entrar em conta com as regras encapsuladas nos sub-objectos para efeito da determinação dos eventos activadores, como ilustra o exemplo seguinte.

Exemplo

Seja um tipo de tabela `t` constituída por registos do tipo `r`, com campos `x`, `y` e `z`. Sejam as seguintes regras:

```
rule t::r1() { update this set x = 1; }
rule r::r2() { z = x + y; }
```

O efeito líquido completo da regra `r1`, entrando em conta com a possível execução encaixada de `r2`, pode ser expresso por:

```
{ update this set x = 1, z = x +y; }
```

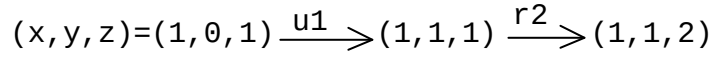
No entanto, como não há actualizações de estado contraditórias no decurso da execução de `r1`, não é necessário considerar o efeito líquido aumentado para definir os eventos activadores, que ficam simplesmente:

```
rule t::r1()
on create, insert, update(x)
{ update this set x = 1; }
```

Suponhamos que é submetido o seguinte comando do utilizador:

`u1: update t set y=1` (em que `t` é uma tabela do tipo `t`)

A figura seguinte mostra o grafo de execução respectivo, relativo a um único registo com um dado estado inicial:



A regra `r2` é imediatamente activada a seguir à alteração de y em cada registo, "consumindo" essa alteração. A regra `r1` não é activada, apesar de referenciar `y` na forma aumentada.

Mesmo quando não ocorrem actualizações contraditórias, uma regra pode ser (auto)activada por alterações causadas por regras encapsuladas nos sub-objectos, conforme ilustra o exemplo seguinte.

Exemplo

Seja um tipo de tabela `t` constituída por registos do tipo `r`, com campos `x`, `y` e `z`. Sejam as seguintes regras:

```
rule t::r1() { update this set y = exp(-x); }
rule r::r2() { x = exp(-y); }
```

O efeito líquido completo da regra `r1`, entrando em conta com a possível execução encaixada de `r2`, pode ser expresso por:

```
{ update this set y = exp(-x), x = exp(-exp(-x)); }
```

Assim, na forma aumentada, a regra `r1` é recursiva em `x`, devendo ser activada quando o valor de `x` é alterado pela regra `r2` durante a execução de `r1`, o que pode ser traduzido da seguinte forma:

```
rule t::r1()
on create, insert, update(x), self update(x)
{ update this set y = exp(-x); }
```

7.3.2 Imposição de restrições inter-objecto através de regras intra-objecto parciais

Para cumprir os pressupostos de encapsulamento, uma relação de integridade (restrição ou derivação) entre vários objectos (*inter-objecto*) tem de ser imposta por uma regra associada a um objecto de nível de granularidade suficientemente alto (no limite o objecto-raiz) que agregue os objectos envolvidos (i.e., um objecto em relação ao qual a restrição é intra-objecto).

Uma solução alternativa, mais eficiente em muitos casos (nomeadamente quando está envolvido um número grande de objectos), é através de regras "parciais" associadas a cada um dos objectos envolvidos, violando os pressupostos de encapsulamento definidos na secção anterior.

Uma regra intra-objecto parcial r_1 associada a um objecto o_1, destinada a impor uma restrição R_1, deve ser executada em todas as operações de alteração, criação ou eliminação de o_1 que possam violar R_1. Para impor R_1, a regra tem, em geral, que aceder (para consulta ou alteração) aos outros objectos envolvidos. Regras intra-objecto deste tipo são preconizadas no sistema Ode (ver a secção 2.3.6.4).

Não existem dificuldades a considerar em relação ao modelo de regras definido no capítulo 3, quando se verificam as seguintes condições: as regras de um objecto acedem aos outros objectos envolvidos apenas para consulta (e não para alteração); e só está alterado um objecto de cada vez. Nestas condições, os outros objectos acedidos podem ser tratados como constantes.

Quando uma regra de derivação associada a um objecto altera o estado doutros objectos (exteriores ao objecto alvo da regra), essas alterações podem ser vistas como efeitos laterais que não podem ser ignorados na definição de estados consistentes com a regra, o que contraria um dos pressupostos considerados no capítulo 3. Assim, o critério de activação é diferente. Normalmente, basta activar uma regra desse tipo quando as variáveis de estado intra-objecto lidas pela regra são alteradas, mas nem sempre é assim, pelo que os eventos activadores têm, em geral, de ser fornecidos pelo programador (a não ser que as regras intra-objecto parciais sejam geradas automaticamente).

Exemplo

Seja o formulário da figura 7.1. Indicam-se de seguida regras intra-objecto para a manutenção incremental de algumas restrições, com acessos a objectos exteriores indicadas a sublinhado.

Para a manutenção incremental (sempre que possível) do `TOTAL` em `FACTURA` definem-se as seguintes regras (uma regra para cada tipo de objecto ou tipo de evento envolvido):

```
rule FACTURA::r3a()
on create, modify(TOTAL)
{ TOTAL = (select sum(PREÇO_TOTAL) from LINHAS_FACTURA); }

rule LINHA_FACTURA::r3d()
on create
{ TOTAL = TOTAL + PREÇO_TOTAL; }

rule LINHA_FACTURA::r3c()
on destroy
```

```
{ TOTAL = TOTAL – old(PREÇO_TOTAL); }

rule LINHA_FACTURA::r3b()
on modify(PREÇO_TOTAL) // excepto dentro de create ou destroy!
{ TOTAL = TOTAL + PREÇO_TOTAL – old(PREÇO_TOTAL); }
```

Para a manutenção incremental (sempre que possível) de preço total de cada linha incluindo IVA à taxa indicada no campo TAXA_IVA do cabeçalho da factura, definem-se as seguintes regras:

```
rule FACTURA::r2a()
on create, modify(TAXA_IVA)
{ update LINHAS_FACTURA
  set PREÇO_TOTAL = QUANTIDADE * PREÇO_UNITÁRIO * (1+TAXA_IVA);
}

rule LINHA_FACTURA::r2b()
on create, modify(QUANTIDADE), modify(PREÇO_UNITÁRIO),
   modify(PREÇO_TOTAL)
{ PREÇO_TOTAL = QUANTIDADE * PREÇO_UNITÁRIO * (1 + TAXA_IVA); }
```

Para a manutenção incremental da restrição R4 (unicidade de CÓDIGO_ARTIGO em LINHAS_FACTURA), define-se a seguinte regra:

```
rule LINHA_FACTURA::r4()
on create, modify(CÓDIGO_ARTIGO)
{ if (exists(select * from LINHAS_FACTURA L
             where L <> this
             and L.CÓDIGO_ARTIGO = this.CÓDIGO_ARTIGO))
  then abort();
}
```

7.3.3 Imposição de restrições inter-objecto que interessam a parte dos objectos envolvidos

Em certos casos, uma restrição de integridade R_1 interessa só a um objecto o_1, mas depende do estado doutro objecto o_2 de que o_1 é cliente. Nesses casos, não faz sentido sobrecarregar o objecto o_2 com a manutenção da restrição R_1. O objecto o_1 sozinho deve manter a restrição R_1, para o que tem de monitorar os eventos de alteração de o_2. A restrição R_1 pode ser mantida por uma regra localizada no objecto o_1, que é activada por alterações de estado ocorridas em o_1 ou o_2.

Exemplo

As regras de integridade R5 a R8 da figura 7.3 interessam só ao formulário de preenchimento de facturas e não às tabelas de artigos e fornecedores nela referenciadas. Supondo que os dados dessas tabelas (ou pelo menos os dados dessas tabelas referenciados no formulário) não são alterados durante o preenchimento do formulário, essas tabelas podem ser tratadas como constantes para efeito do processamento de regras. Isso pode ser garantido, por exemplo através de mecanismos de bloqueio ("lock"). Essa solução pode ser penalizadora em termos de concorrência. Uma solução alternativa consiste em permitir alterações nas tabelas referenciadas (artigos e fornecedores), sendo o formulário notificado quando isso acontece, para proceder (por alterações internas ao formulário, em transacções independentes das transacções nas tabelas de artigos e fornecedores) aos ajustes necessários para que as regras R5 a R8 sejam satisfeitas. Não faria sentido sobrecarregar o processamento de transacções nas tabelas de artigos e fornecedores com acções de reposição de integridade no formulário. Tecnicamente, o modo de acoplamento entre o evento activador e a execução da regra é "detached" (ver secção 2.1.4). Quando o formulário acede às tabelas de artigos, em vez de solicitar "locks",

inscreve-se em (subscreve) uma lista de objectos a ser notificados quando a tabela ou registos dessas tabelas são alterados.

Uma situação semelhante acontece com formulários de consulta da bases de dados, que são notificados para se refrescarem, total ou parcialmente (incrementalmente), quando os dados da base de dados neles referenciados são alterados.

Regras deste tipo podem ser enquadradas no modelo de regras definido no capítulo 3. No exemplo acima, as tabelas referenciadas podem ser modelizadas por variáveis de estado "virtuais" do formulário, cuja alteração é sinalizada do exterior (iniciando, possivelmente, transacções no formulário). Uma vez que a sinalização dessas alterações pode ocorrer durante o processamento de regras, pode ser violado o princípio de que as únicas alterações do estado das variáveis de estado que ocorrem durante o processamento das regras são produzidas pelas regras. De qualquer forma, seria fácil reactivar as regras aquando da alteração concorrente das variáveis de estado "virtuais".

8 Implementação de um motor de regras activas dirigidas pelos dados

Neste capítulo descreve-se um motor de regras activas dirigidas pelos dados que foi implementado para servir de componente nuclear de uma ferramenta CASE específica (SAGA), mas de forma a poder ser usado noutras aplicações. Esse motor de regras suporta regras com eventos e prioridades implícitos para a manutenção de restrições de integridade e dados derivados, de acordo com o modelo de regras introduzido no capítulo 3 e extensões discutidas no capítulo 7 (sem suporte para objectos compostos), mas também suporta regras com eventos e prioridades explícitos. O motor de regras oferece um interface em C através do qual as aplicações declaram regras e sinalizam eventos. Em resposta a esses eventos, o motor de regras desencadeia a execução das regras, que é coordenada com a execução de transacções. Para servir para diferentes aplicações, o motor de regras trata apenas da execução das regras, que são declaradas ao motor de regras já compiladas. Ainda assim, o motor de regras é acompanhado de um compilador, analisador e avaliador de expressões escritas numa linguagem extensível que pode ser útil para suportar regras definidas dinamicamente pelo utilizador, conforme se verá no capítulo 9. A descrição é baseada em UML ("Unified Modeling Language") na versão descrita em [UML97].

8.1 Arquitectura lógica

O motor de regras está organizado em três módulos ("packages"), indicados no diagrama de módulos da figura seguinte, juntamente com as dependências existentes entre eles.

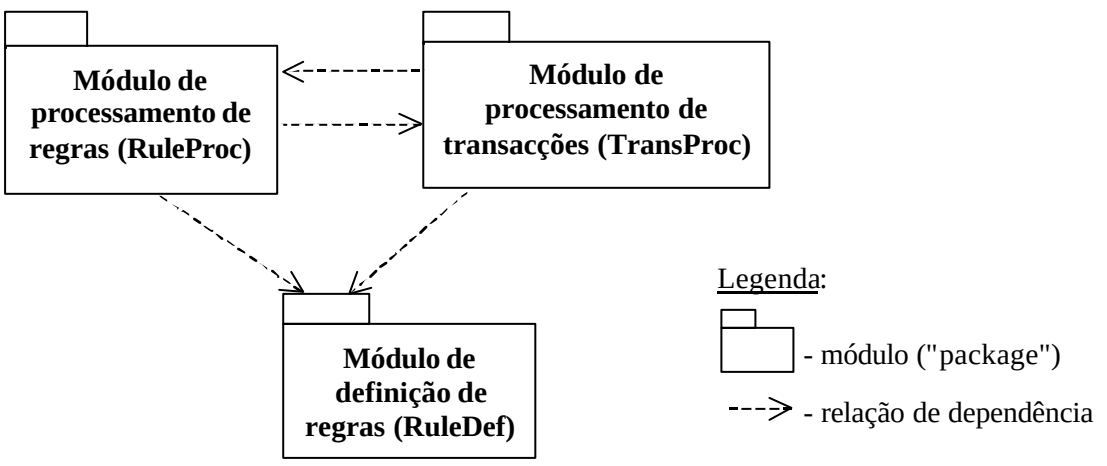


Fig. 8.1 Diagrama de módulos ("packages") do motor de regras.

Estes módulos são descritos nas secções seguintes.

8.2 Módulo de definição de regras *(RuleDef)*

Este módulo mantém meta-informação sobre as regras e os dados controlados por essas regras, de acordo com o diagrama da figura seguinte. Consideram-se as regras e os dados da aplicação organizados em classes da aplicação por forma a suportar (de forma limitada) as regras intra-objecto descritas no capítulo 7. Para evitar confusão com as classes da aplicação, as classes deste módulo são designadas meta-classes.

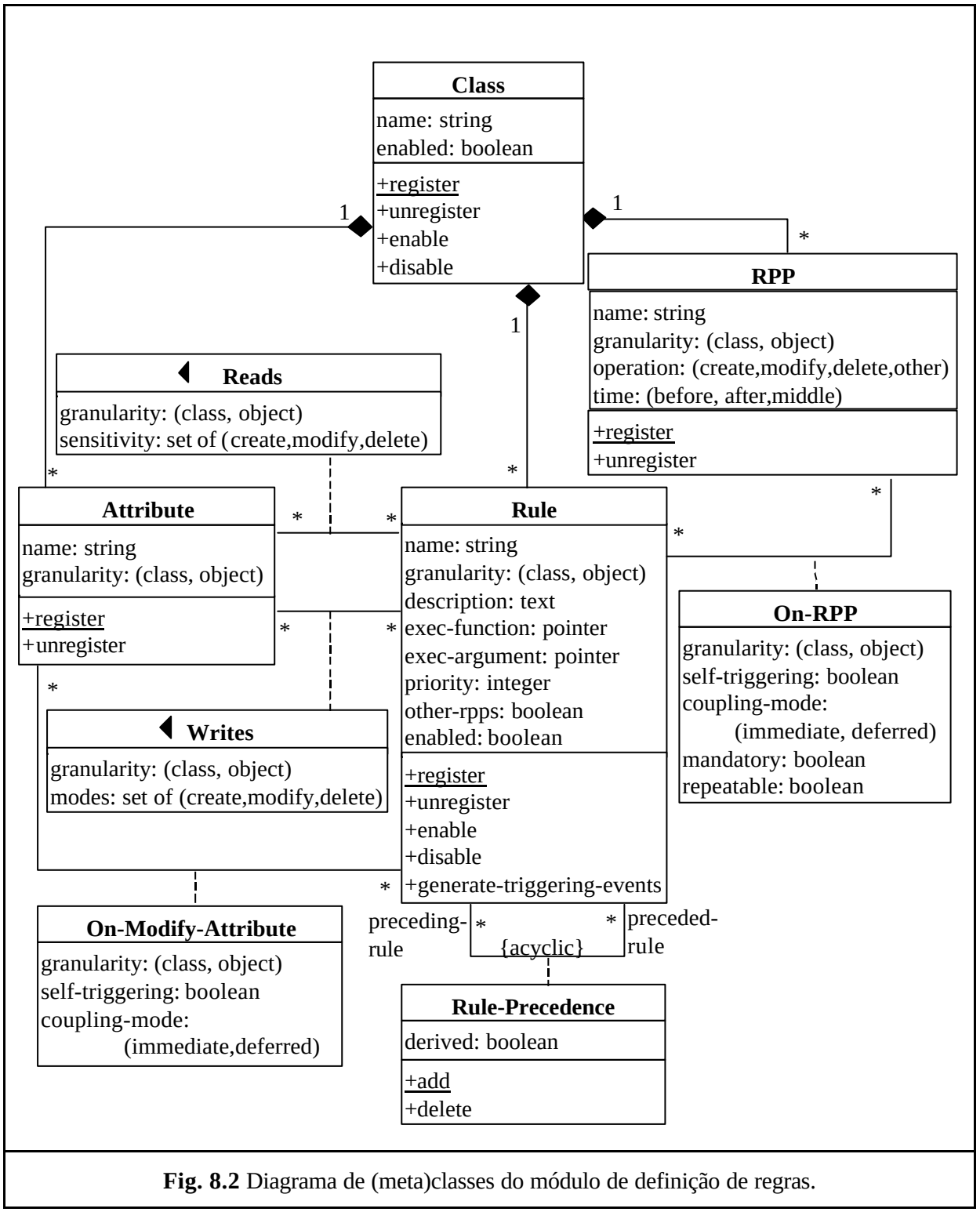


Fig. 8.2 Diagrama de (meta)classes do módulo de definição de regras.

Segue-se uma descrição das meta-classes e associações deste diagrama.

8.2.1 Meta-classe *Class*

Aqui são registadas dinamicamente, através das operações *register* e *unregister*, as classes da aplicação que têm regras a elas associadas. Note-se que podem ser classes implementadas como tal em C++ ou noutra linguagem, mas também podem ser estruturas de dados que são vistas como classes.

A operação *disable* elimina logicamente o registo de uma classe (incluindo todos os seus atributos e regras), sem o eliminar fisicamente de memória (como aconteceria com *unregister*). A operação *enable* tem a função oposta de *disable*. Estas operações têm interesse em ambientes muito dinâmicos. Além disso, quando uma classe é registada, fica inicialmente no estado de *disabled*, para permitir a definição dos seus atributos, regras e PPR's, antes de passar ao estado de *enabled*.

8.2.2 Meta-classe *Attribute*

Aqui são registadas dinamicamente os atributos das classes da aplicação registadas na meta-classe *Class*, com o significado de variáveis de estado para efeito da execução das regras.

Os atributos podem ser de duas granularidades diferentes:

- *object* - São locais aos objectos da classe. Também são designados atributos intra-objecto. Têm um valor (estado) possivelmente diferente de objecto para objecto da mesma classe. Correspondem a membros-dados ("data members") sem prefixo "static" em C++.
- *class* - São globais à classe. Também são designados atributos globais. Têm um valor (estado) único para todos os objectos da classe, o qual está definido mesmo que não exista nenhum objecto. Correspondem a membros-dados ("data members") com prefixo "static" em C++ e aos atributos sublinhados nos diagramas de classes em UML.

Do ponto de vista de cada objecto, os atributos locais ao objecto têm o significado de variáveis de estado locais ao objecto. Do ponto de vista do sistema completo, todos os atributos (intra-objecto ou globais) de todas as classes têm o significado de variáveis de estado do sistema; os atributos intra-objecto são vistos como variáveis de estado orientadas a conjuntos, isto é, o que interessa é o conjunto de valores do atributo para todos os objectos da classe (como função, definida em extensão, dos identificadores dos objectos). Segue-se, portanto, a abordagem descrita na secção 7.2.2.

Podem existir classes sem objectos, apenas com atributos e regras globais. Uma classe desse tipo funciona como um módulo.

8.2.3 Meta-classe *RPP*

"RPP" é uma abreviatura de "Rule Processing Point". Aqui são registados dinamicamente os pontos de processamento de regras (PPR's ou RPP's) que existem explicitamente nas operações das classes da aplicação registadas na meta-classe *Class*, isto é, os pontos nessas operações onde é invocado explicitamente o processamento de regras. Estes não são os únicos pontos onde é invocado o processamento de regras pois, como veremos adiante, o processamento de regras também é invocado no final de cada transacção.

Os PPR's têm um nome (*name*) para permitir a associação explícita de regras definidas numa linguagem textual a PPR's (ver associação *On-RPP*). Também podem ser vistos como eventos definidos pela aplicação com processamento imediato de regras.

Tipicamente, cada operação de alteração de dados com nome *op* tem um PPR no início e no fim, designados *before_op* e *after_op*. Em cada classe com atributos intra-objecto, devem existir, no mínimo, operações para criar, eliminar e actualizar o estado de objectos da classe.

Tal como os atributos, os PPR's podem ser de duas granularidades diferentes:

- *object* - Ocorrem para um objecto específico da classe, normalmente em operações intra-objecto. Também são designados PPR's intra-objecto.

- *class* - Ocorrem numa operação da classe, mas não para um objecto específico da classe. Também são designados PPR's globais.

A propriedade *operation* indica o tipo genérico (não o nome concreto) da operação primitiva a que o PPR está associado:

- *create* - criação de objecto ou classe (com *enable*), conforme a granularidade;
- *delete* - eliminação de objecto ou classe (com *disable*), conforme a granularidade;
- *modify* - modificação do estado do objecto (através da modificação do estado de atributos intra-objecto) ou da classe (através da modificação do estado de atributos globais ou da criação, eliminação ou modificação do estado de conjuntos de objectos da classe), conforme a granularidade;
- *other* - outras situações (pode ser uma operação de consulta, pode ser uma operação que combina várias operações primitivas, etc.).

A propriedade *time* indica o momento em que o PPR ocorre na operação:

- *before* - no início da operação;
- *after* - no fim da operação;
- *middle* - num ponto intermédio da operação.

Estas duas propriedades interessam para a geração automática de eventos activadores.

8.2.4 Meta-classe *Rule*

Aqui são registadas dinamicamente as regras que existem nas classes da aplicação registadas na meta-classe *Class*.

Tal como os atributos e os PPR's, as regras podem ser de duas granularidades diferentes:

- *object* - Aplicam-se a um objecto específico da classe. Também são designadas regras intra-objecto. O identificador do objecto é um argumento implícito da regra (fornecido quando a regra é activada, conforme se verá adiante), semelhante ao apontador "this" em C++. Não se impõe nenhuma restrição de encapsulamento, pelo que uma regra intra-objecto pode aceder a outros objectos e a atributos globais.
- *class* - Têm existência ligada à existência da classe, mas não se aplicam a um objecto específico da classe. Também são designadas regras globais.

A parte de acção da regra (a regra em si) é indicada por um apontador para uma função (*exec-function*) e um argumento (*exec-arg*). Executar a regra corresponde a invocar essa função com esse argumento. No caso de regras intra-objecto, é passado um argumento adicional com o identificador do objecto. Isto permite que as regras sejam escritas como funções em C ou C++, permite que a mesma função corresponda a muitas regras (com argumento diferentes), e permite que as regras sejam definidas dinamicamente por expressões (nesse caso, a função a invocar é uma função de avaliação de expressões e o argumento é um apontador para a expressão).

As regras podem ter uma prioridade numérica (*priority*), que é usada, com pouco peso, para ordenar a execução das regras, conforme se verá adiante. Diferentes regras podem ter a mesma prioridade numérica. Conforme foi referido no capítulo 5, a prioridade numérica pode servir para fazer com que, na falta de razões em contrário (daí o pouco peso), as regras de restrição (que invocam *abort*) sejam executadas antes das regras de derivação. A prioridade numérica também pode ser atribuída de acordo com a ordem por que as regras são definidas, para fazer com que, na falta de mais informação (daí o pouco peso), as regras sejam executadas pela ordem por que são definidas.

A opção *other-rpps* indica se a execução da regra está limitada aos PPR's indicados explicitamente pela associação *On-RPP*, ou se pode também ser executada noutros PPR's.

A operação *disable* elimina logicamente uma regra, sem a eliminar fisicamente de memória (como aconteceria com *unregister*). A operação *enable* tem a função oposta de *disable*. Estas operações permitem inibir selectivamente algumas regras de uma classe.

A operação *generate-triggering-events* é explicada mais adiante.

8.2.5 Associação *Rule-Precedence*

Esta associação, ou melhor classe-associação ("association-class"), permite especificar prioridades (ou precedências) relativas entre pares de regras, que são usadas, com peso máximo, para ordenar a execução das regras, conforme se verá adiante.

As prioridades relativas não podem ter ciclos.

As prioridades relativas podem ser transitivas ou não transitivas, de acordo com um parâmetro de configuração. No caso de serem transitivas, sempre que é adicionada uma prioridade relativa pela aplicação (que fica com *derived=false*), são automaticamente adicionadas (com *derived=true*) as prioridades relativas que resultam transitivamente da prioridade relativa adicionada e do conjunto de prioridades existentes. Quando uma prioridade relativa é eliminada, procede-se de forma simétrica. Na implementação corrente do motor de regras, este automatismo ainda não está implementado.

8.2.6 Associações *Reads* e *Writes*

As associações *Reads* e *Writes* relacionam cada regra com os atributos lidos e actualizados, respectivamente, pela regra. Estas associações definem o grafo r-v mencionado no capítulo 4 com os refinamentos do capítulo 7.

Qualquer destas associações tem uma propriedade que indica a granularidade do acesso ao atributo, que pode ser:

- *object* - O atributo é acedido no mesmo objecto para que a regra é instanciada. Também se diz que o acesso é intra-objecto. Só pode acontecer no caso do atributo e a regra serem intra-objecto e da mesma classe.
- *class* - Outros casos. Também se diz que o acesso é global.

No caso do acesso global para escrita a um atributo intra-objecto (visto como uma variável de estado que representa o conjunto de valores do atributo para todos os objectos da classe em função dos identificadores dos objectos), podem-se especificar os modos de actualização (conforme sugerido na secção 7.2.2), que podem ser:

- *create* - Por criação de objectos da classe. Note-se que a criação de objectos de uma classe, altera o conjunto de valores de qualquer atributo intra-objecto da classe.
- *delete* - Por eliminação de objectos da classe. Note-se que a eliminação de objectos de uma classe altera o conjunto de valores de qualquer atributo intra-objecto da classe.
- *modify* - Por modificação do valor do atributo em objectos da classe.

No caso do acesso global para leitura (consulta) a um atributo intra-objecto (visto como uma variável de estado que representa o conjunto de valores do atributo para todos os objectos da classe em função dos identificadores dos objectos), podem-se especificar os modos de actualização a que a regra é sensível (conforme sugerido na secção 7.2.2).

Note-se que, quando se criam ou eliminam objectos de uma classe, o conjunto final de valores de cada atributo intra-objecto depende normalmente do conjunto inicial de valores do mesmo atributo, caso em que o atributo também deve ser considerado acedido globalmente para leitura (de forma sensível à eliminação e criação, respectivamente).

Se existir um atributo correspondente ao identificador interno dos objectos de uma classe, as regras que percorrem os objectos da classe (para fazer alguma coisa com cada objecto) lêem implicitamente esse atributo (e possivelmente outros atributos explicitamente), devendo essa leitura ser indicada pela associação *Reads*. As regras que criam ou eliminam objectos duma classe actualizam implicitamente esse atributo, devendo essa actualização ser indicada pela associação *Writes*.

A informação dada pelas associações *Reads* e *Writes* é usada para a geração automática dos eventos activadores de cada regra e para a ordenação dinâmica das regras.

8.2.7 Associações *On-RPP* e *On-Modify-Attribute*

Estas associações especificam os eventos que activam as regras e os PPR's em que podem ser executadas.

A associação *On-Modify-Attribute* especifica, para cada atributo, as regras que são activadas quando o valor do atributo é alterado. No caso de um atributo intra-objecto, supõe-se que a alteração é sinalizada cada vez que o valor do atributo é alterado num objecto, mas não quando um objecto é criado com determinados valores iniciais ou quando um objecto é eliminado (a criação e a eliminação são sinalizadas sem referência aos atributos). No caso de um atributo global, supõe-se que a alteração é sinalizada cada vez que o valor do atributo é alterado, mas não quando a respectiva classe é criada com determinados valores iniciais ou é eliminada (a criação e a eliminação são sinalizadas sem referência aos atributos).

A associação *On-RPP*, especifica para cada PPR, as regras que são activadas pela ocorrência do PPR (caso de *mandatory=true*) ou que podem ser executadas nesse PPR (caso de *mandatory=false*). O caso *mandatory=false* interessa para restringir os PPR's em que uma regra com *other-rpps=false* pode ser executada, sendo a activação em si causada por outros eventos.

Na associação *On-RPP*, quando *repeatable=false* a regra em causa só pode ser executada no máximo uma vez em cada ocorrência do PPR. Esta opção interessa, por exemplo, para regras que são activadas pela ocorrência do PPR (caso de *mandatory=true*) mas não são reactivadas pela modificação de atributos indicados pela associação *On-Modify-Attribute* (que interessam apenas para outros PPR's).

A granularidade da activação de uma regra devida à ocorrência de um evento pode ser:

- *object* - A regra é activada (instanciada) para o mesmo objecto do evento. Também se diz que a activação é intra-objecto. Só pode acontecer no caso do evento e a regra serem intra-objecto e da mesma classe. No caso da ocorrência de um PPR intra-objecto, o objecto do evento é o objecto a que se refere a ocorrência do PPR. No caso da alteração do valor de um atributo intra-objecto, o objecto do evento é o objecto em que o valor do atributo foi alterado.
- *class* - Outros casos. Também se diz que a activação é global. Se a regra for intra-objecto, é activada automaticamente para todos os objectos da respectiva classe.

É também especificado o modo de acoplamento entre a ocorrência do evento activador e a execução da regra, que pode ser:

- *immediate* - A regra é executada no PPR em curso ou mais próximo da transacção corrente. Se várias regras forem activadas, são executadas sequencialmente. No caso da associação *On-RPP*, o PPR em que a regra executada é o mesmo que activa a regra.
- *deferred* - A regra é executada no PPR em curso ou mais próximo da transacção de nível de topo.

No caso da granularidade ser *object* o modo de acoplamento tem de ser *immediate*, por razões de eficiência, e para evitar ter de armazenar conjuntos de identificadores de objectos.

No caso de *other-rpps=false*, se o PPR em que a regra seria executada não está associado explicitamente à regra pela associação *On-RPP*, a activação não tem efeito.

No caso de *self-triggering=false*, a ocorrência do evento quando a regra está a ser executada (mais precisamente, quando alguma instância da regra está a ser executada) é ignorada, isto é, não causa a activação da regra.

Os vários tipos de eventos activadores e modos de activação acima referidos permitem suportar diferentes tipos de regras:

- regras de restrição ou de derivação destinadas à imposição de restrições de integridade estáticas - normalmente activadas pela criação de um objecto ou classe e reactivadas pela alteração de atributos referenciados na regra (para leitura ou escrita);

- regras de restrição ou de derivação destinadas à imposição de restrições de integridade locais a um PPR (sem o que a respectiva operação não pode prosseguir) - normalmente activadas pela ocorrência do PPR e reactivadas pela alteração de atributos referenciados na regra (para leitura ou escrita);
- regras de "propagação", que são executadas (possivelmente com efeitos laterais) sempre que certos dados (atributos) consultados pela regra são alterados;
- regras intra-objecto destinadas a impor restrições intra-objecto ou restrições inter-objecto;
- regras que servem simplesmente para aumentar ou restringir operações de uma forma livre e arbitrária, sendo executadas uma única vez em PPR's dessas operações.

8.2.8 Geração automática dos eventos activadores

A operação *generate-triggering-events* da classe *Rule* gera automaticamente os eventos activadores de uma regra a partir dos atributos cujo estado é consultado ou alterado por cada regra (dados pelas associações *Reads* e *Writes*), caso isso seja desejado. Esta operação implementa o critério de activação básico descrito no capítulo 4, excluindo a reactivação de uma regra devido à alteração de variáveis de saída da regra, com alguns dos refinamentos discutidos no capítulo 7. Mais precisamente,

- Para cada ocorrência da associação *Reads*:
  - se *granularity=object*
    - cria uma ocorrência em *On-Modify-Attribute* com *granularity=object*
    - para cada ocorrência de *RPP* da mesma classe do atributo (e da regra) com *time=after*, *type=create* e *granularity=object*, cria uma ocorrência em *On-RPP* com *granularity=object*
  - se *granularity=class*
    - se o atributo é global
      - cria uma ocorrência em *On-Modify-Attribute* com *granularity=class*
      - para cada ocorrência de *RPP* da mesma classe do atributo com *time=after*, *type=create* e *granularity=class*, cria uma ocorrência em *On-RPP* com *granularity=class*
    - se o atributo é intra-objecto
      - se *modify Î sensitivity*, cria uma ocorrência em *On-Modify-Attribute* com *granularity=class*
      - se *create Î sensitivity*, para cada ocorrência de *RPP* da mesma classe do atributo com *time=after*, *type=create* e *granularity=object*, cria uma ocorrência em *On-RPP* com *granularity=class*
      - se *delete Î sensitivity*, para cada ocorrência de *RPP* da mesma classe do atributo com *time=after*, *type=delete* e *granularity=object*, cria uma ocorrência em *On-RPP* com *granularity=class*
- Para cada associação da associação *Writes*,
  - se *granularity=object*
    - para cada ocorrência de *RPP* da mesma classe do atributo (e da regra) com *time=after*, *type=create* e *granularity=object* cria uma ocorrência em *On-RPP* com *granularity=object*
  - se *granularity=class*
    - se o atributo é global
      - para cada ocorrência de *RPP* da mesma classe do atributo com *time=after*, *type=create* e *granularity=class* cria uma ocorrência em *On-RPP* com *granularity=class*
    - se o atributo é intra-objecto

- para cada ocorrência de *RPP* da mesma classe do atributo com *time=after*, *type=create* e *granularity=object* cria uma ocorrência em *On-RPP* com *granularity=class*

Em todos os casos, considera-se *self-triggering=false*. O modo de acoplamento é imediato quando *granularity=object*, e diferido no caso contrário. Em *On-RPP* é sempre *mandatory=true* e *repeatable=true*. No caso de colisão entre várias ocorrências em *On-RPP* ou *On-Modify-Attribute*, prevalece a granularidade mais alta (*class* sobre *object*). Supõe-se existir em cada classe pelo menos um PPR de cada um dos tipos indicados.

8.3 Módulo de processamento de transacções (*TransProc*)

Este módulo coordena a imposição das propriedades de atomicidade (A) e preservação de consistência (C) de transacções na memória de trabalho da aplicação, em colaboração com a aplicação.

O termo "transacção" é usado de forma um pouco abusiva, porque não estão em causa as outras propriedades de isolamento (I) e durabilidade (D) normalmente associadas às transacções [GR93]. Essas propriedades interessam quando os dados são partilhados e persistentes, respectivamente. Caso seja necessário, essas propriedades podem, no entanto, ser mantidas pela aplicação.

A propriedade de atomicidade é mantida à custa de um registo ("log") de alterações, que permite a reposição do estado existente no início de uma transacção ("rollback") no caso da transacção ser abortada. No entanto, este módulo apenas disponibiliza os mecanismos genéricos para esse efeito. Compete à aplicação solicitar o registo dos itens alterados no "log" de alterações. Uma aplicação pode, inclusivé, não suportar "rollback" (basta não colocar nada no "log"), ou suportar "rollback" parcialmente.

A propriedade de preservação de consistência é mantida através do processamento de regras, que é invocado automaticamente no fim de cada transacção, podendo ser invocado explicitamente pela aplicação em pontos intermédios das transacções. Note-se, no entanto, que as regras não se destinam exclusivamente a manter a consistência dos dados, pelo que está em causa algo mais do que a propriedade de preservação de consistência.

É seguido um modelo de transacções encaixadas ("nested transactions") conforme descrito em [GR93]. Isto é, podem ocorrer transacções (chamadas *sub-transacções*) dentro de transacções. Uma transacção que não é sub-transacção doutra é chamada uma *transacção de nível de topo* ("top-level transaction"). Chama-se *transacção encaixada* ("nested transaction") à árvore completa com uma transacção de nível de topo e todas as sub-transacções. As folhas da árvore são chamadas transacções planas ("flat"). O modelo de transacções encaixadas prevê a execução concorrente de sub-transacções com o mesmo pai ("parent transaction"), mas essa possibilidade não é considerada aqui. Assim, as transacções encaixadas que se consideram aqui são essencialmente equivalentes a transacções planas ("flat transactions") com "savepoints" intermédios [GR93].

O modelo de transacções encaixadas permite suportar *operações transaccionais* (operações globais ou intra-objecto das classes da aplicação, "protegidas" por transacções) que se invocam mutuamente. No caso de falha de uma operação transaccional, são desfeitas ("rolled back") apenas as alterações ocorridas no decurso dessa operação, competindo à operação invocadora tomar o curso mais apropriado (abortar por sua vez, ignorar, seguir um caminho alternativo, etc.).

Podem definir-se sub-transacções sem "rollback" local. O abortamento de uma sub-transacção deste tipo propaga-se para a transacção invocadora. Mais uma vez, é cometido um pequeno abuso de linguagem. Sub-transacções deste tipo interessam para definir apenas um contexto (horizonte) para efeito de processamento de regras e não para efeito de "rollback".

O diagrama de classes deste módulo é indicado na figura seguinte. As classes que aparecem no diagrama são descritas nas secções seguintes.

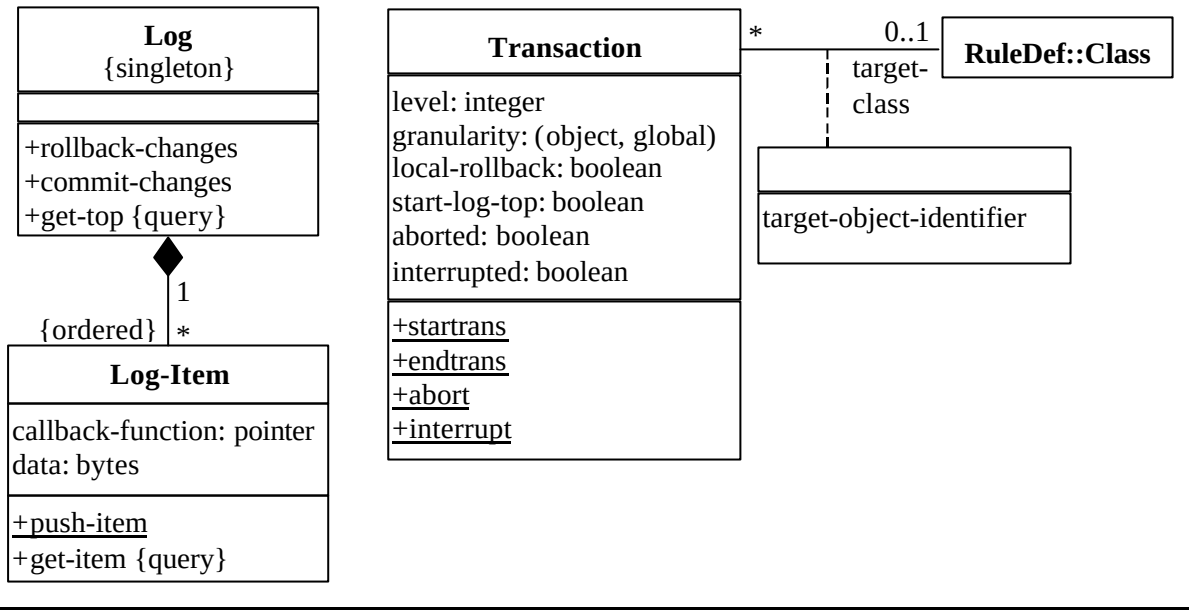


Fig. 8.3 Diagrama de classes do módulo de processamento de transacções.

8.3.1 Classe *Transaction*

Nesta classe é registada a pilha de transacções em curso num dado momento na memória de trabalho da aplicação, isto é, a transacção de nível de topo e as sub-transacções iniciadas mas ainda não terminadas. Cada instância desta classe é um elemento (transacção) da pilha de transacções. O conjunto de instâncias é que constitui a pilha de transacções.

Cada transacção tem um nível, que começa em 1 na transacção de nível de topo e vai crescendo ao longo da pilha. Note-se que a transacção de nível de topo está na base e não no topo da pilha. A transacção de nível mais alto num dado momento é designada *transacção corrente*.

Conforme foi explicado acima, permitem-se transacções dos seguintes dois tipos:

- *com "rollback" local* - Se a transacção for abortada, são desfeitas apenas as alterações ocorridas desde o início dessa transacção. A transacção invocadora, no caso de existir, não é abortada.
- *sem "rollback" local* - Se a transacção (*t*) for abortada, a transacção invocadora também é abortada. Assim, as alterações ocorridas durante a transacção invocadora (mesmo antes do início de *t*) também são desfeitas.

Obviamente, no caso de uma transacção de nível de topo, os dois tipos são equivalentes.

As transacções podem ser de uma das seguintes granularidades:

- *objecto* - Incide sobre um objecto específico (chamado *objecto-alvo* da transacção), cuja classe e identificador são dados pelos atributos *target-class* e *target-object-identifier*. Uma transacção deste tipo também é chamada intra-objecto. Normalmente, corresponde à invocação de uma operação intra-objecto transaccional.
- *global* - Não incide sobre um objecto específico. Uma transacção deste tipo também é chamada global.

O atributo *start-log-top* regista o topo da pilha que implementa o "log" de alterações aquando do início da transacção, para efeitos de "rollback" e para efeito de controlo de alterações (a ver na secção seguinte). Uma transacção sem "rollback" local herda o valor de *start-log-top* da transacção invocadora.

O atributo *aborted* indica se a transacção foi abortada. Este atributo interessa porque pode mediar algum tempo desde que uma transacção é abortada até terminar efectivamente (momento em que desaparece da pilha de transacções).

O atributo *interrupted* indica se a transacção foi interrompida (ver adiante). Este atributo interessa porque pode mediar algum tempo desde que uma transacção é interrompida até terminar efectivamente (momento em que desaparece da pilha de transacções).

A operação *startrans* inicia uma nova transacção (de nível de topo ou sub-transacção) que passa a ser a nova transacção corrente.

A operação *endtrans* termina a transacção corrente. Normalmente, *endtrans* é chamado pela mesma operação que iniciou a transacção. Se a transacção não tiver sido abortada ou interrompida, é primeiro invocado o processamento de regras, que pode originar sub-transacções e o abortamento ou interrupção da transacção corrente. Quando uma transacção de nível de topo termina com sucesso (sem ter sido abortada), as alterações registadas no "log" de alterações são tornadas duradoiras e o "log" é esvaziado (ver operação *commit-changes* adiante). Quando uma sub-transacção termina com sucesso (sem ter sido abortada), as alterações ocorridas desde o início da sub-transacção permanecem registadas no "log" de alterações (como se fossem herdadas pela transacção invocadora), porque podem ter de ser desfeitas mais tarde (caso uma transacção invocadora seja abortada mais tarde).

A operação *abort* aborta a transacção corrente. O processamento de regras é interrompido e são desfeitas as alterações ocorridas desde o início da transacção (ver operação *rollback-changes* adiante). Se a transacção corrente não tiver "rollback" local, a transacção invocadora também é abortada, e assim sucessivamente. Uma transacção abortada tem na mesma de ser terminada com *endtrans* (o que normalmente acontece quando o controlo retorna à operação que iniciou a transacção). Não é possível iniciar uma nova transacção nem registar alterações de dados (que, em princípio, não devem acontecer, mas isso depende da aplicação) a partir de uma transacção abortada.

A operação *interrupt* interrompe a transacção corrente. O processamento de regras é interrompido, mas não são desfeitas as alterações ocorridas desde o início da transacção, como acontece com *abort*. A transacção invocadora não é considerada interrompida, contrariamente ao que pode acontecer com *abort*. Uma transacção interrompida tem na mesma de ser terminada com *endtrans*. Esta operação foi introduzida para suportar regras com a acção "doinstead" (a ver no capítulo 9).

8.3.2 Classes *Log* e *Log-Item*

A classe *Log* tem uma única instância (daí a restrição *{singleton}*) que constitui o "log" de alterações. Os itens (alterações) registados no "log" constituem as instâncias da classe *Log-Item*.

Conceptualmente, o "log" é uma pilha, porque os itens são inseridos e eliminados por uma ordem "last-in-first-out". Na prática, a pilha é implementada por um buffer de memória, geralmente com o topo da pilha, mais um ficheiro temporário para dados excedentes. Os itens são identificados pela sua posição relativa na pilha, que cresce por ordem cronológica.

Por uma questão de flexibilidade e generalidade, cada item do "log" tem uma sequência de bytes e um apontador para uma função que conhece o significado desses bytes. Esses bytes podem conter o valor antigo de um atributo global alterado, o estado antigo de um objecto modificado, o identificador de um objecto criado (porque basta saber o identificador do objecto para desfazer a criação do objecto), o estado antigo de um objecto eliminado, etc.

O "log" destina-se apenas a suportar "rollback" relativamente a itens de dados relevantes (o que depende da aplicação). Assim, não interessa registar no "log" todas as alterações. Por exemplo, se um objecto é inserido e depois modificado, sem que entretanto seja iniciada uma transacção com "rollback" local, basta registar o identificador do objecto inserido. Se um objecto é modificado várias vezes, sem

que entretanto seja iniciada uma transacção com "rollback" local, basta registar o estado do objecto antes da primeira modificação.

A operação *get-top* indica o topo corrente do "log". Interessa a *startrans*.

A operação *push-item* adiciona um item ao "log". Tem como argumentos um apontador para uma função, um apontador para uma sequência de bytes (em memória principal) e o n.º de bytes da sequência. Esses bytes e o apontador para função são copiados para o "log". A operação *push-item* retorna a posição do item no "log". Essa posição pode ser usada de várias formas. Quando se regista no "log" o estado antigo de um objecto ou atributo modificado, pode-se guardar junto com esse atributo ou objecto a posição do item inserido no "log". Através dessa posição e da operação *get-item* pode-se mais tarde facultar o acesso ao estado antigo desse objecto ou atributo, o que é útil para suportar regras que acedem a estados passados (a ver no capítulo 9). Quando o estado do atributo ou do objecto é de novo modificado, pode-se comparar a posição do último item inserido no "log" relativamente a esse atributo ou objecto com a atributo *start-log-top* da transacção corrente, para decidir se é necessário efectuar qualquer registo no "log".

A operação *get-item* permite aceder a um item do "log" pela sua posição. Os bytes desse item são colocados em memória principal (no "buffer" do "log" de alterações), e o seu endereço é retornado à aplicação para usar esses bytes (antes de ser chamada outra operação do log").

A operação *rollback-changes* destina-se a desfazer as alterações ocorridas desde um determinado ponto (definido por um argumento da operação). É usada pela operação *abort* da classe *Transaction*, para desfazer as alterações ocorridas desde o início da transacção. Percorre os itens do "log" até ao ponto indicado, por ordem inversa à ordem por que foram inseridos ("last-in-first-out"). Para cada item, chama a função associada ao item com a mensagem ROLLBACK, passando-lhe o endereço dos bytes desse item colocados em memória principal, e de seguida elimina o item do "log". Compete à função chamada desfazer as alterações que haja a desfazer.

A operação *commit-changes* destina-se a confirmar as alterações registadas no "log" e esvaziar o "log". É usada pela operação *endtrans* da classe *Transaction*, quando uma transacção de nível de topo termina com sucesso. Percorre todos os itens do "log", por ordem inversa à ordem por que foram inseridos ("last-in-first-out"). Para cada item, chama a função associada ao item com a mensagem COMMIT, passando-lhe o endereço dos bytes desse item colocados em memória principal, e de seguida elimina o item do 'log". Por exemplo, no caso ser mantido com cada objecto modificado a posição do estado antigo no "log", isto pode ser usado para fazer o "reset" (a zero) dessa posição. No caso da aplicação manter "backups" próprios de itens alterados, pode servir para libertar esses "backups".

8.4 Módulo de processamento de regras (*RuleProc*)

Este módulo trata da activação, ordenação e execução das regras, em coordenação com o módulo de processamento de transacções. O diagrama de classes deste módulo é apresentado na figura seguinte.

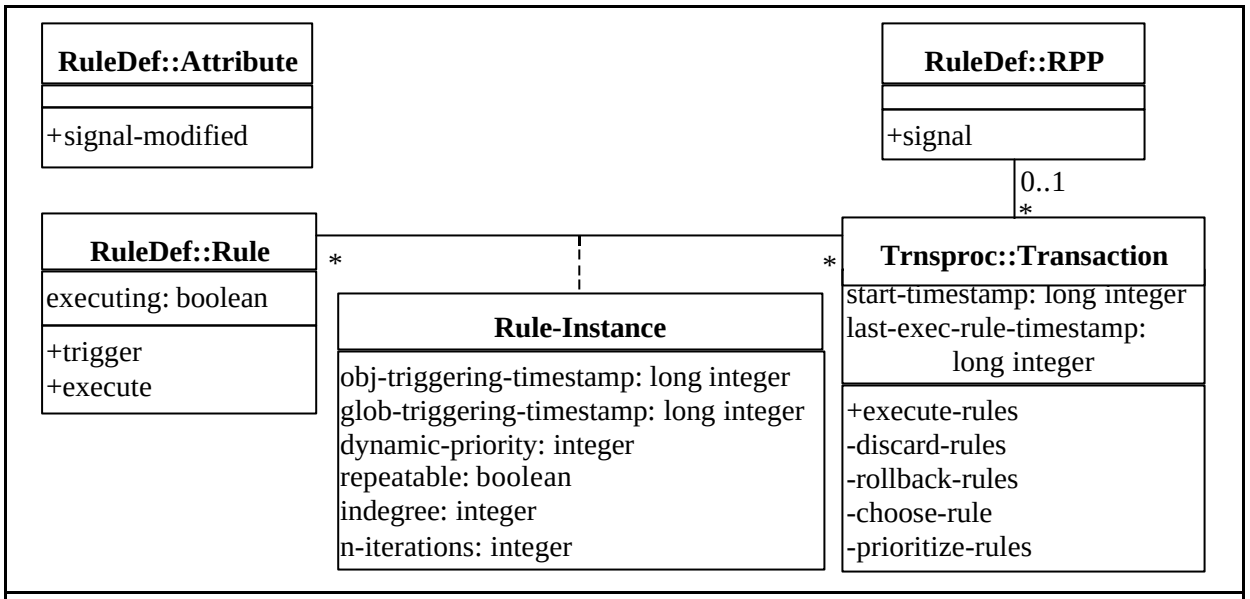


Fig. 8.4 Diagrama de classes do módulo de processamento de regras.

Consideram-se como fazendo parte deste módulo atributos e operações que são acrescentados a classes doutros módulos. Os elementos deste diagrama são descritos nas secções seguintes organizados por funções.

8.4.1 Sinalização de eventos (*signal* e *signal-modified*) e activação de regras (*trigger*)

A sinalização de eventos é efectuada através da invocação de operações acrescentadas a algumas classes anteriormente descritas. A sinalização dos eventos desencadeia a activação de regras para execução imediata ou diferida. Os eventos em si não são armazenados por razões de eficiência (a menos de possíveis alterações de dados registadas no "log" de alterações).

A operação *signal-modified* na meta-classe *Attribute* é invocada para sinalizar que o estado de um atributo acaba de ser alterado. Impõem-se as seguintes restrições:

- a alteração do estado de um atributo global só pode ser sinalizada quando está corrente uma transacção não abortada e não interrompida;
- a alteração do estado de um atributo intra-objecto só pode ser sinalizada quando está corrente uma transacção intra-objecto, não abortada nem interrompida, tendo como alvo o objecto que está a ser alterado.

Estas exigência servem para organizar a execução das regras. A segunda exigência não é penalizadora, porque, se a transacção corrente não tem como alvo o objecto que está a ser alterado, pode-se criar facilmente (até de forma implícita) uma transacção sem "rollback" local tendo como alvo esse objecto.

A criação e eliminação de objectos não deve ser sinalizada pela sinalização da alteração de atributos mas sim pela sinalização de PPR's.

As regras ligadas pela associação *On-Modify-Attribute* ao atributo alterado são activadas, de acordo com as restrições explicadas no módulo de definição de regras, ou seja:

- se uma regra ou a respectiva classe tem *enabled=false*, a regra não é activada;
- se uma ligação tem *self-triggering=false* e uma instância da regra está a ser executada (indicado pelo atributo *executing* da classe *Rule*), a regra não é activada;
- se uma regra tem *other-rpps=false* e o PPR em que a regra seria executada (de acordo com o modo de acoplamento) não está ligado explicitamente à regra pela associação *On-RPP*, a activação não tem efeito.

A operação *signal* na meta-classe *RPP* origina uma instância do PPR indicado na transacção corrente. Impõem-se as seguintes restrições:

- tem de estar corrente uma transacção não abortada e não interrompida;
- não pode estar em curso um PPR na transacção corrente;
- um PPR intra-objecto só pode ocorrer (ser instanciado) numa transacção intra-objecto sobre o mesmo objecto.

Estas restrições servem para organizar a execução das regras. As duas últimas restrições não são penalizadoras, porque facilmente se pode criar (até de forma implícita) uma sub-transacção sem "rollback" local.

As regras ligadas pela associação *On-RPP* a um PPR sinalizado com *mandatory=true*, são activadas, de acordo com as restrições explicadas no módulo de definição de regras, ou seja:

- se uma regra ou a respectiva classe tem *enabled=false*, a regra não é activada;
- se uma ligação tem *self-triggering=false* e a regra está a ser executada, a regra não é activada;
- se uma regra tem *other-rpps=false* e o PPR em que a regra seria executada (de acordo com o modo de acoplamento) não está ligado explicitamente à regra pela associação *On-RPP*, a activação não tem efeito (isto só pode acontecer com o modo de acoplamento diferido).

Quando uma regra é activada, é também determinada a transacção em que a regra será executada (no PPR em curso ou no PPR mais próximo dessa transacção), de acordo com os modos de acoplamento explicados no módulo de definição de regras.

No caso de *other-rpps=false* e de não estar em curso, no momento em que a regra é activada, um PPR na transacção em que a regra deve ser executada, a regra é activada condicionalmente. No início de cada PPR, é confirmada a activação anterior de regras com *other-rpps=false* associadas explicitamente ao PPR, e é cancelada a activação anterior das restantes regras com *other-rpps=false*.

No caso da granularidade de activação ser intra-objecto (*granularity=object* na associação *On-Modify-Attribute* ou *On-RPP*), o modo de acoplamento é sempre imediato e a regra é instanciada para o objecto-alvo da transacção corrente, de acordo com as restrições impostas até ao momento.

A activação propriamente dita é realizada pela operação *trigger* da classe *Rule*. A operação *trigger* também pode ser chamada explicitamente pela aplicação. Essa operação cria uma ocorrência na classe *Rule-Instance* (instância de regra), preenchendo a estampa temporal correspondente à granularidade da activação, a qual interessa para efeito de "rollback". Uma regra pode ter várias instâncias diferentes, mas não para a mesma transacção. Se já existia uma instância com granularidade *object* e a granularidade pedida é *class*, a segunda prevalece. No entanto, são mantidas duas estampas temporais, para o caso de ser necessário desfazer uma promoção de granularidade aquando de "rollback". Se já existia uma instância activada com granularidade *class* e a granularidade pedida é *object*, a segunda é ignorada. Se já existia uma instância activada com a granularidade pretendida, a estampa temporal não é alterada.

8.4.2 Execução das regras (*execute-rules*)

A operação *execute-rules* da classe *Transaction* executa as regras associadas à transacção corrente, a pedido de *endtrans* (quando a transacção não se encontra abortada ou interrompida) ou de *signal*. As regras activadas são executadas de forma sequencial, como no algoritmo 3.1. A activação da variável de erro *e* referida nesse algoritmo deve ser interpretada como o abortamento ou interrupção da transacção corrente. Note-se que a transacção corrente também é abortada quando é abortada uma sub-transacção sem *rollback* local. A invocação de *rollback* também sai do algoritmo de execução de regras. Assim, o algoritmo fica:

Algoritmo 8.1 (*execução sequencial de regras associadas a uma transacção t*):

1. Enquanto existirem regras activadas acopladas à transacção *t* e *t* não tiver sido abortada ou interrompida,
   1.1. Escolher uma regra activada *r* de acordo com um critério de resolução de conflitos (ou critério de ordenação).
   1.2. Desactivar *r*.
   1.3. Executar *r*.

Um PPR é, por definição, um ponto de invocação da operação *execute-rules*. O fim de uma transacção (*endtrans*) também é um PPR (implícito).

É mantido um contador do n.º de vezes que cada regra é executada em cada PPR (atributo *n_iterations* na classe *Rule-Instance*). Se for excedido o n.º máximo permitido (configurável), a transacção é abortada.

Durante a execução de uma regra *r* pode ocorrer uma sub-transacção *t'* com um PPR *p'* onde é de novo invocado (recursivamente) o algoritmo de execução de regras. No entanto, em cada transacção (de nível de topo ou sub-transacção), só pode estar em curso um PPR. Por outras palavras, o algoritmo não pode ser invocado recursivamente sem se iniciar entretanto uma sub-transacção.

8.4.3 Escolha da próxima regra a executar (*choose-rule*)

Nesta secção explica-se como é que se processa a escolha da próxima regra a executar (também chamada resolução de conflitos), quando há várias regras activadas num dado PPR, determinando portanto a ordem por que as regras são executadas.

Essa escolha é determinada pelas seguintes prioridades das regras, com pesos decrescentes:

- prioridades relativas estáticas, não necessariamente transitivas, estabelecidas pela associação *Rule-Precedence*;
- prioridades numéricas *dinâmicas* únicas (no sentido de que não há duas regras com a mesma prioridade) calculadas dinamicamente em cada PPR (e guardadas no atributo *dynamic-priority* da classe *Rule-Instance*).

As prioridades numéricas dinâmicas podem variar de PPR para PPR, mas mantêm-se fixas em cada PPR. Além disso são estabelecidas apenas para as regras que participam em cada PPR.

Dadas as prioridades dos dois tipos, a escolha da próxima regra a executar num dado PPR (nomeadamente, quando há várias regras activadas), processa-se conceptualmente em duas fases:

1º) do conjunto A de regras activadas, é seleccionado o conjunto A' de regras activadas que não são precedidas (de acordo com as prioridades relativas estáticas) por outras regras activadas, i. e., $A' = \{r \hat{I} A$: não existe nenhuma regra $r' \hat{I} A$ com prioridade relativa estática sobre $r\}$;

2º) do conjunto A' é escolhida a regra com prioridade dinâmica máxima (existe só uma regra nestas condições porque regras diferentes têm prioridades dinâmicas diferentes).

8.4.3.1 Obtenção das prioridades numéricas dinâmicas (*prioritize-rules*)

As prioridades numéricas dinâmicas são calculadas com base na seguinte informação, com influências decrescentes:

- grafo de dependências entre regras calculado dinamicamente em cada PPR, com o significado explicado mais adiante na secção 8.4.3.3;
- prioridades numéricas estáticas das regras (dadas pelo atributo *priority* da classe *Rule*), não necessariamente únicas (isto é, podem existir várias regras com a mesma prioridade);
- ordem de activação das regras (indicada pela estampa temporal).

Mais precisamente, as prioridades numéricas dinâmicas (equivalentes a uma ordenação total das regras) resultam da conjugação dos seguintes critérios, dispostos por forças decrescentes:

- Se existe um caminho de *r* para *r'*, e nenhum caminho em sentido contrário, no grafo de dependências, então *r* deve ter prioridade sobre *r'*.
  - Este critério corresponde ao critério 5.1 (ordenação das regras pelo princípio "calcular antes de usar") do capítulo 5, só que baseado num grafo de dependências que não corresponde exactamente ao grafo r-r.
- Ordenar as regras por forma a induzir um conjunto mínimo de arestas de realimentação (inverter um conjunto mínimo de arestas) no grafo de dependências.
  - Este critério corresponde ao critério 5.5 do capítulo 5, só que baseado num grafo de dependências que não corresponde exactamente ao grafo r-r. Este critério e o anterior complementam-se sem contradição. Os dois critérios equivalem-se quando o grafo de dependências é acíclico. Quando o grafo de dependências tem ciclos, este critério serve para ordenar as regras dentro dos componentes fortemente conexos (CFC's) não triviais.
- Ordenar as regras de acordo com as suas prioridades numéricas estáticas.
  - Este critério interessa quando os critérios anteriores não são suficientes para ordenar totalmente as regras. A utilidade de prioridades numéricas estáticas com pouca força foi já referida.
- Dar prioridades às regras activadas mais recentemente.
  - Este critério é introduzido para resolver os casos em que os critérios anteriores não chegam para determinar uma ordenação total das regras. Corresponde a usar a estampa temporal de activação de cada regra como uma prioridade numérica por omissão. Assim, pode ser aplicado ao mesmo tempo que o critério anterior (na prática, isso é feito numa função de comparação de prioridades de pares de regras). É baseado na ideia de que, na falta de informação adicional, a ordem de execução das regras deve estar relacionada deterministicamente com a ordem de activação das regras. O critério inverso (dar prioridade às regras activadas há mais tempo) também podia ser usado, mas é menos conveniente em termos de facilidade de implementação (conforme se verá adiante).

Por razões de eficiência, estes critérios são conjugados de uma forma que satisfaz totalmente aos dois primeiros critérios e satisfaz parcialmente aos dois últimos critérios. Isto é, pode ser produzida uma ordenação total *O* quando existe outra ordenação *O'* que satisfaz igualmente aos dois primeiros critérios, e tem menos violações aos dois últimos critérios. Há uma violação das prioridades estáticas quando:

prior. dinâmica de r_1 > prior. dinâmica de r_2 $\wedge$ prior. estática de r_1 < prior. estática de r_2

O algoritmo concreto utilizado é descrito na secção 8.5.

As prioridades numéricas dinâmicas são calculadas imediatamente antes de escolher a próxima regra a executar (no passo 1.1 do algoritmo 8.1). As prioridades são calculadas não só para as regras que se encontram activadas, mas também para todas as regras que podem ser atingidas a partir das regras activadas por caminhos do grafo de dependências, as quais são adicionadas (num estado de não activadas) ao conjunto de regras ligadas ao PPR. Isso é feito por duas razões: essas regras também têm de ser consideradas para efeito da aplicação dos dois primeiros critérios acima enunciados; essas regras podem vir a ser activadas em resultado da execução de regras que se encontram já activadas (atendendo ao significado do grafo de dependências).

Mesmo assim, após a execução da 1ª regra podem ser activadas regras que não tinham ainda sido consideradas (e prioritizadas). Por razões de eficiência, em vez de voltar a recalcular de novo todas as prioridades, são calculadas prioridades apenas para as regras do conjunto constituído pelas regras activadas que não tinham sido ainda prioritizadas, mais todas as regras não prioritizadas que podem ser atingidas a partir das primeiras por caminhos do grafo de dependências.

Formalmente, sejam (com significado local a um PPR):

A_i - conjunto de regras que se encontram activadas, para executar no PPR em causa, no momento de escolher a *i*-ésima regra a executar (na *i*-ésima iteração

do passo 1.1 do algoritmo 8.1) ($i \geq 1$);

$G_i=(R,E_i)$ - grafo de dependências entre regras, válido para esse PPR, no mesmo momento (utiliza-se o índice i porque o grafo de dependências pode variar no decurso do PPR);

$F_i(S)$ - fecho transitivo de um conjunto S de regras pelo grafo G_i, ou seja, conjunto de regras que podem ser atingidas a partir de S por caminhos de comprimento 0 ou mais do grafo G_i (incluindo portanto S);

C_i - conjunto de regras cujas prioridades dinâmicas são calculadas no mesmo momento;

D_i - conjunto de regras cujas prioridades dinâmicas são calculadas ao longo das iterações $1, ..., i$.

Tem-se então que:

$$D_0=\varnothing \qquad C_i = F_i(A_i-D_{i-1})-D_{i-1} \qquad D_i = D_{i-1} \cup C_i$$

As prioridades calculadas numa iteração são combinadas com as prioridades calculadas nas iterações anteriores da seguinte forma: as prioridades calculadas na i-ésima iteração começam (em sentido crescente) na prioridade mais alta calculada até ao momento (nas iterações anteriores). Isto é feito por facilidade de implementação e porque, normalmente, o grafo de dependências não muda. Se o grafo de dependências não mudou desde a 1ª iteração, não existem caminhos em G_i de regras já prioritizadas (D_{i-1}) para as novas regras a prioritizar (C_i), pelo que a atribuição de prioridades crescentes é consistente com os 2 primeiros critérios que se pretendem seguir. Além disso, é também consistente com o último critério (o que aliás justifica a utilização desse critério em vez do inverso).

Após a última iteração, todas as instâncias de regras ligadas à transacção corrente são eliminadas (não transitam para outro PPR da mesma transacção).

8.4.3.2 Algoritmo incremental para a combinação das prioridades relativas estáticas com as prioridades numéricas dinâmicas

A combinação das prioridades relativas estáticas (com mais força) com as prioridades numéricas dinâmicas (com menos força) é efectuada de forma incremental.

Para cada regra r associada a um dado PPR (o que, de acordo com a secção anterior, inclui não só as regras activadas mas também as regras cujas prioridades numéricas dinâmicas foram calculadas), é mantido um contador (atributo *indegree* da classe *Rule-Instance*) com o número de regras activadas associadas ao mesmo PPR com prioridade relativa sobre r (pela associação *Rule-Precedence*).

As regras activadas com $indegree=0$ são inseridas num "heap" que implementa uma fila de prioridades por ordem das prioridades numéricas dinâmicas. Os algoritmos de gestão de "heaps" (também usados na fase de prioritização) são idênticos aos referidos na secção 8.5.2.

Considere-se a seguinte notação:

Pred(r) - conjunto de regras que têm prioridade relativa estática sobre a regra r

Succ(r) - conjunto de regras sobre as quais a regra r tem prioridade relativa estática

O passo 1.1 do algoritmo de execução de regras é detalhado da seguinte forma (usando a notação introduzida na secção anterior):

Algoritmo 8.2 (*escolha da próxima regra a executar na i-ésima iteração do algoritmo 8.1*):

1. Calcular as prioridades numéricas dinâmicas das novas regras (conjunto C_i).
2. Para cada $r \in C_i$ /* prioritizada nesta iteração */, $indegree(r) \leftarrow 0$

3. Se i>1 então
   para cada $r \hat{I} C_i$,
   para cada $r' \hat{I} Pred(r)$,
   se $r' \hat{I} A_i \text{ Ç } A_{i-1}$ /* activada mas não de novo */ então
   indegree(r) ¬ *indegree(r)+1*
4. Para cada $r \hat{I} A_i\text{-}A_{i-1}$ /* activada de novo */,
   para cada $r' \hat{I} Succ(r)$,
   se $r' \hat{I} D_i$ então *indegree(r')* ¬ *indegree(r')+1*
5. Para cada $r \hat{I} A_i\text{-}A_{i-1}$ /* activada de novo */,
   se *indegre(r)=0* então *HEAP_INSERT(r)*
6. Se o "heap" está vazio, sair e terminar o processamento de regras.
7. Extrair do "heap" a regra *r* com prioridade numérica dinâmica máxima, i.e.,
   r ¬ *HEAP_DELMAX*
8. Se *indegree(r)*¹*0* /* devido a ter sido activada uma regra de *Pred(r)* */, voltar ao passo 6.
9. Para cada $r' \hat{I} Succ(r)$,
   se $r' \hat{I} D_i$ então
   indegre(r') ¬ *indegree(r')-1*
   se *indegree(r')=0* então *HEAP_INSERT(r')*
10. Retornar a regra *r* (regra escolhida para executar).

O acesso sequencial e o teste de pertença aos conjuntos de regras referidos podem ser implementados eficientemente através de (sub)listas ligadas (ver secção 8.4.3.4), "flags" e "timestamps", conforme se indica a seguir:

- para cada $r \hat{I} C_i$ - acesso eficiente com apontador da transacção para a cabeça da sub-lista de instâncias de regras prioritizadas de novo;
- para cada $r \hat{I} A_i\text{-}A_{i-1}$ - acesso eficiente com apontador da transacção para a cabeça da sub-lista de instâncias de regras activadas de novo;
- se $r' \hat{I} D_i$ - teste eficiente com apontador de regra para a cabeça da lista de instâncias da regra ordenada por transacções de níveis decrescentes (recorde-se que o algoritmo acima é executada na transacção corrente, que é, por definição, a que tem nível mais alto);
- se $r' \hat{I} A_i \text{ Ç } A_{i-1}$ - teste eficiente com apontador como no caso anterior, e comparação de "timestamps" (*triggering-timestamp* e *last-exec-rule-timestamp*).

Note-se que, normalmente, não há nenhuma regra ou há poucas regras a considerar no passo 3 do algoritmo acima, devido à natureza do grafo de dependências.

8.4.3.3 Grafo de dependências dinâmico

O grafo de dependências considerado no cálculo das prioridades dinâmicas traduz, de forma aproximada, a seguinte relação entre pares de regras r_1 e r_2: "r_1 pode activar r_2 para o mesmo PPR por alterar dados (directamente, ou indirectamente por intermédio de uma regra r_3 executada de forma encaixada durante a execução de r_1) que são usados por r_2".

A obtenção deste grafo tem várias dificuldades associadas:

- A relação traduzida neste grafo varia de PPR para PPR, mesmo entre diferentes ocorrências do mesmo tipo de PPR. Isso acontece principalmente porque algumas regras só podem ser executadas em alguns PPR's. Por exemplo, as regras com *other-rpps=false* só podem ser executadas nos PPR's indicados explicitamente pela associação *On-RPP*.

- O facto de se considerarem também as alterações produzidas por regras executadas de forma encaixada, introduz uma dificuldade acrescida. Normalmente, não existe informação suficiente para determinar que sub-transacções podem ocorrer durante a execução de uma regra. A única pista nesse sentido tem a ver com o facto de alterações intra-objecto terem de ocorrer em sub-transacções intra-objecto. Mesmo supondo que certas alterações ocorrem em sub-transacções, pode não ser possível prever com rigor a transacção a que uma regra activada por uma dessas alterações é acoplada.
- Embora isso seja desaconselhado, uma regra pode criar ou eliminar outras regras (directamente ou indirectamente através de operações invocadas), fisicamente (através de *register* e *unregister*) ou logicamente (através de *enable* e *disable*). Em consequência disso, o grafo de dependências associado a um PPR pode variar no decurso do PPR.

Estas dificuldades, levam a que o grafo seja obtido de forma aproximada e de passagem. Isto é, o grafo é apenas calculado e nunca é armazenado. O cálculo é efectuado através de um função *visitsucc(r,visitfunc)* que, dada uma regra *r* e uma função *visitfunc*, calcula os sucessores imediatos de *r* e chama a função *visitfunc* para cada sucessor.

8.4.3.4 Estruturas de dados usadas na implementação

As regras associadas a uma transacção (instâncias de regras) são acedidas de duas formas:

- lista de instâncias de regras por transacção: lista duplamente ligada com apontadores para a cabeça, cauda e pontos intermédios que delimitam sub-listas com instâncias em diferentes estados;
- lista de instâncias de regras por regra: lista duplamente ligada com apontadores para a cabeça e cauda, ordenada por transacções de níveis decrescentes; esta lista é útil para implementar os critérios de activação, acoplamento e ordenação.

Para minimizar a chamada a rotinas de alocação dinâmica de memória, é mantida também uma lista de itens livres.

Quando um PPR termina (o que pode acontecer porque não há mais regras activadas, ou a transacção foi abortada ou interrompida), as instâncias de regras da transacção corrente são eliminadas (passam para uma lista de itens livres) através da operação *discard-rules*.

Adicionalmente, quando uma transacção é abortada, são removidos (pela operação *rollback-rules*) todos as instâncias de regras criadas noutras transacções em curso desde o início da transacção abortada. Essas instâncias encontram-se à cabeça da lista de instâncias de cada transacção ordenadas pela estampa temporal da activação, a qual é comparada com a estampa temporal de início da transacção. Além disso, são também desactivadas (mas não eliminadas) as instâncias de regras activadas desde o início da transacção abortada, mas que tinham sido criadas antes do início da transacção abortada. Distinguem-se das anteriores pelo facto de estarem prioritizadas.

Para minimizar a chamada a rotinas de alocação dinâmica de memória, os vários "heaps" que implementam as filas de prioridade são mantidos num único array, que funciona como uma pilha de "heaps". Cada transacção sabe onde começa e acaba o seu "heap", através do "offset" inicial e tamanho. Isto é possível porque só é alterado o "heap" da transacção corrente. Em particular, quando uma transacção é abortada, as instâncias eliminadas em transacções anteriores não se encontram ainda no "heap".

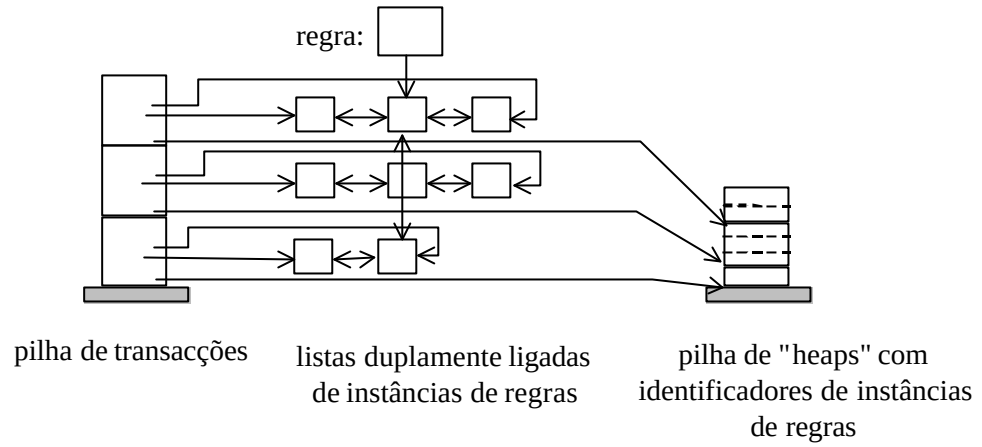


Fig. 8.5 Visão parcial das estruturas de dados usadas na implementação.

8.4.3.5 Discussão

Uma alternativa ao método de ordenação de regras apresentado consistiria na geração de um único conjunto de prioridades relativas, armazenado na associação *Rule-Precedence*, em que algumas das prioridades seriam definidas pela aplicação e outras seriam geradas automaticamente (tal como se faz com os eventos activadores de cada regra). Para esse efeito, algoritmos semelhantes aos apresentados seriam executados aquando da definição das regras e não aquando da execução das regras. Em tempo de execução das regras, poderia interessar continuar a combinar as prioridades relativas com prioridades numéricas dinâmicas correspondentes à estampa temporal de activação das regras.

As vantagens seriam: maior eficiência e simplicidade em tempo de execução; possibilidade de aplicar algoritmos com melhores resultados mas mais pesados, como o referido no anexo 1 a este documento; o programador poderia consultar e refinar as prioridades geradas automaticamente. Uma desvantagem seria o possível desaparecimento de algumas optimizações para PPR's específicos.

8.5 Ordenação dos vértices de um grafo por ordem topológica de componentes fortemente conexos minimizando inversões de arestas e de prioridades dos vértices

O problema de cálculo das prioridades numéricas dinâmicas levantado na secção 8.4.3.1 pode ser formulado da seguinte forma genérica:

Problema 8.1: Dado um grafo dirigido $G=(V,E)$, em que cada vértice $v \hat{I} V$ tem uma prioridade *priority(v)* associada, não necessariamente única (i.e., podem existir dois vértices com a mesma prioridade), atribuir um número único *num(v)* a cada vértice $v \hat{I} V$ (o que equivale a obter uma ordenação total das regras), de acordo com os seguintes critérios, dispostos por forças decrescentes:

1º) se existe um caminho de u para v em G, e nenhum caminho em sentido contrário, então deve ser $num(u) > num(v)$;

2º) inverter um conjunto mínimo de arestas de G (diz-se que uma aresta $u \to v$ é invertida quando $num(u) \pounds num(v)$);

3º) inverter um conjunto mínimo de prioridades de vértices de G (diz-se que ocorre uma inversão de prioridade quando $num(u) < num(v)$ e $priority(u) > priority(v)$).

No caso de G ser acíclico, a aplicação do 1º critério corresponde a numerar os vértices por ordem topológica inversa. No caso de G ser cíclico, uma forma eficiente de satisfazer a este critério é através

da ordenação dos vértices por ordem topológica de componentes fortemente conexos (CFC's), conforme foi referido no capítulo 5 e confirmaremos a seguir. Ver exemplos nas figuras 5.1 e 5.2.

O 1º e 2º critério complementam-se sem contradição, e equivalem-se quando *G* é acíclico. Quando *G* tem ciclos, o 2º critério serve para ordenar os vértices dentro de cada CFC não trivial. Se for escolhida uma ordenação dos vértices por ordem topológica de CFC's para satisfazer ao 1º critério, o 2º critério pode ser aplicado eficientemente (em tempo linear) ao mesmo tempo que se obtêm os CFC's, conforme veremos a seguir.

Se for escolhida uma ordenação dos vértices por ordem topológica de CFC's para satisfazer ao 1º critério, o 3º critério pode ser aplicado de uma forma local, que pode não ser óptima mas é eficiente (em tempo *n log n*), durante a ordenação topológica dos CFC's, conforme veremos a seguir.

Estas observações sugerem um método eficiente para ordenar os vértices através destes critérios, obedecendo completamente aos 2 primeiros critérios e parcialmente (porque de forma local) ao último. O método é baseada na combinação e adaptação de algoritmos conhecidos, conforme se explica por passos nas secções seguintes.

8.5.1 Ordenação topológica

É sabido que a ordenação topológica ("topological sorting") dos vértices de um grafo dirigido acíclico *G=(V,E)* pode ser efectuada em tempo *O(|V|+|E|)*, em que *|V|* designa o número de vértices e *|E|* designa o número de arestas.

São conhecidos dois algoritmos de ordenação topológica:

- um algoritmo recursivo baseado na visita em profundidade ("depth-first search"), conforme descrito por exemplo em [RNN77] ou [AU92];
- um algoritmo não recursivo, mas que exige uma estrutura de dados auxiliar, conforme descrito por exemplo em [K97].

Relembramos de seguida estes dois algoritmos. Os algoritmos são organizados de tal forma que o acesso aos sucessores de um vértice *v* pode ser facilmente implementado através de uma função *visitsucc(v,visitfunc)* que chama a função *visitfunc* para cada sucessor de *v*. Isto é útil quando o grafo é calculado e não armazenado, conforme foi explicado anteriormente.

O algoritmo recursivo pode ser descrito em pseudo-código da forma a seguir indicada.

Algoritmo 8.3 (*ordenação topológica baseada na visita em profundidade):*

```
/*
Entradas:
    V - conjunto de vértices
    Para cada vértice v∈ V,
        Succ(v) - conjunto de sucessores imediatos de v, i.e., Succ(v)={u∈ V: (v,u)∈ E}
Saídas:
    Para cada vértice v∈ V,
        num(v) - numeração por ordem topológica inversa
Dados temporários:
    Para cada vértice v∈ V,
        visited(v) - indica se v já foi visitado
*/

/* Procedimento principal */
procedure TOPSORTa
begin
    n ← 0
```

```
    for v∈ V do visited(v) ← false
    for v∈ V do DFS(v)
end

/* Ordenação topológica recursiva a partir do vértice v */
procedure DFSa(v)
begin
    if ¬visited(v) then
        visited(v) ← true
        for u∈ Succ(v) do DFS(u)
        n ← n+1
        num(v) ←n
end
```

O segundo algoritmo pode ser descrito da seguinte forma:

Algoritmo 8.4 (*ordenação topológica genérica):*

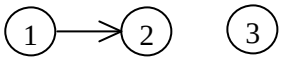
```
/*
Entradas: idem
Saídas: idem
Dados temporários:
    Para cada vértice v∈ V,
        indegree(v) - número de predecessores imediatos de v por ordenar
    S - conjunto de vértices não ordenados com indegree=0
*/

procedure TOPSORTb
begin
    n ← |V|
    S ← {}
    for v∈ V do indegree(v) ← 0
    for v∈ V do for u∈ Succ(v) do indegree(u) ← indegree(u)+1
    for v∈ V do if indegree(v)=0 then S ← S∪{v}
    while S≠{} do
        escolher um vértice v∈ S
        S ← S-{v}
        num(v) ← n
        n ← n-1
        for u∈ Succ(v) do
            indegree(u) ← indegree(u)-1
            if indegree(u)=0 then S ← S∪{u}
end
```

Este último algoritmo pode ser implementado em tempo $O(|V|+|E|)$, usando por exemplo um critério “last in first out” (pilha) ou um critério "first in first out" (fila) para as inserções e eliminações em *S*.

O segundo algoritmo é mais genérico, porque permite gerar todas as ordens topológicas, através de diferentes escolhas no ponto de escolha sublinhado. Em contrapartida, o primeiro algoritmo (com os pontos de escolha também sublinhados) não permite gerar todas as ordens topológicas.

Exemplo

No caso do grafo



o primeiro algoritmo apenas permite gerar as ordens topológicas 1-2-3 e 3-1-2, mas não a ordem topológica 1-3-2.

Assim, o segundo algoritmo é mais útil para combinar com outros critérios, nomeadamente com prioridades, conforme veremos na secção seguinte.

Em contrapartida, o primeiro algoritmo tem a vantagem de poder ser aplicado a um grafo com ciclos (o que não acontece com o segundo) com o seguinte resultado:

Teorema 8.1: A ordenação dos vértices de um grafo *G=(V,E)* produzida pelo algoritmo 8.3 inverte um conjunto mínimo de arestas em *G*.

Demonstração: Seja $v \rightarrow u$ uma aresta invertida. Essa aresta corresponde a uma chamada recursiva *DFS(v)*→*DFS(u)* em que *u* se encontra na pilha de chamada de *DFS* [AU92]. Sejam ... *u*, w_1,... , w_m, *v*, *u* os vértices que se encontram na pilha de chamada nesse momento. Esta sequência de vértices na pilha de chamada corresponde à existência do ciclo $u \rightarrow w_1 \rightarrow \ldots \rightarrow w_m \rightarrow v \rightarrow u$ em *G*. É fácil de constatar que apenas a última aresta do ciclo fica invertida. O conjunto de arestas invertidas é mínimo porque não é possível deixar de inverter a aresta $v \rightarrow u$ sem inverter uma das outras arestas do ciclo (em cada ciclo tem de existir pelo menos uma aresta invertida). Δ

Assim, o primeiro algoritmo é útil para implementar de forma eficiente o 2º critério de ordenação indicado. Note-se, no entanto, que nem todas as ordenações que invertem um conjunto mínimo de arestas podem ser obtidas por esse algoritmo. O conjunto mínimo de arestas invertidas é também um conjunto de arestas de realimentação ("feedback edge set"), isto é, um conjunto mínimo de arestas cuja remoção torna o grafo acíclico [AHU74].

Qualquer dos algoritmos apresentados é facilmente adaptável à ordenação de apenas uma parte dos vértices, correspondente ao fecho transitivo de um conjunto inicial de vértices.

8.5.2 Ordenação topológica minimizando inversões de prioridades

Nesta secção desenvolve-se um algoritmo eficiente (embora não óptimo) para, de entre as possíveis ordenações topológicas dos vértices de um grafo acíclico *G=(V,E)*, escolher uma ordenação topológica que minimize violações das prioridades dos vértices.

Este algoritmo interessa para escolher uma ordenação topológica dos CFC's de um grafo cíclico que minimize violações das prioridades numéricas das regras. O grafo acíclico a considerar para esse efeito é o grafo condensado dos CFC's.

Conforme foi referido, o segundo algoritmo de ordenação topológica é mais flexível para refinar a ordenação topológica de acordo com prioridades. Para esse efeito, basta na linha sublinhada do algoritmo base, escolher um elemento de *S* com prioridade máxima (isto é, tal que não existe outro elemento de *S* com prioridade superior). Assim, *S* corresponde a um tipo abstracto de dados conhecido por *fila de prioridades* (ver por exemplo [AU92]). Uma fila de prioridades é um conjunto de elementos cada um dos quais tem associada uma prioridade. As duas operações associadas a uma fila de prioridades são: inserir um elemento no conjunto (*INSERT*), e encontrar e eliminar do conjunto um elemento com prioridade máxima (*DELETMAX*). Uma fila de prioridades pode ser implementada eficientemente com um "heap". Com base no algoritmo de ordenação topológica anteriormente apresentado e nas rotinas de manipulação de "heaps" apresentadas em [AU92], obtém-se o seguinte algoritmo:

Algoritmo 8.5 (*ordenação topológica minimizando inversões de prioridades):*
/*
Entradas:
    V - conjunto de vértices

```
    Para cada vértice v∈ V,
        Succ(v) - conjunto de sucessores imediatos de v, i.e., Succ(v)={u∈ V: (v,u)∈ E}
        priority(v) - prioridade de v
Saídas:
    Para cada vértice v∈ V,
        num(v) - numeração por ordem topológica inversa minimizando violações de prioridades
                 de forma local (sem garantia de optimalidade global)
            (há uma violação de prioridade quando num(v)>num(u) e priority(v)<priority(u))
Dados temporários:
    Para cada vértice v∈ V,
        indegree(v) - número de predecessores imediatos de v por ordenar
    H - "heap" com vértices não ordenados com indegree=0
    m - tamanho do "heap"
*/

/* Procedimento principal */
procedure TOPSORTPRI
begin
    n ← |V|
    ERASEHEAP
    for v∈ V do indegree(v) ← 0
    for v∈ V do for u∈ Succ(v) do indegree(u) ← indegree(u)+1
    for v∈ V do if indegree(v)=0 then INSERTRAW(v)
    HEAPIFY
    while m > 0 do
        v ¬ DELETEMAX
        num(v) ← n
        n ← n-1
        for u∈ Succ(v) do
            indegree(u) ← indegree(u)-1
            if indegree(u)=0 then INSERT(u)
end

/** Seguem-se rotinas de gestão de "heaps" **/

/* Função de comparação de dois elementos do heap usada pelas rotinas de gestão de "heap".
   Retorna >0, 0 ou <0 conforme o 1º elemento é maior, igual ou menor do que o segundo. */
function COMPARE(u, v)
begin
    return priority(u) - priority(v)
end

/* Esvazia o "heap" */
procedure ERASEHEAP
begin
    m ← 0
end

/* Insere um elemento no fim sem organizar o "heap" */
procedure INSERTRAW(e)
begin
```

```
    m ← m+1
    H_m ← e
end

/* Organiza o "heap" */
procedure HEAPIFY
begin
    for i ← ⌊m/2⌋, ...,2, 1 do HEAPIFYDOWN(i)
end

/* Extrai e retorna o 1º elemento do "heap" e reorganiza o "heap" */
function DELETEMAX
begin
    e ← H_1
    H_1 ← H_m
    m ← m-1
    HEAPIFYDOWN(1)
    return e
end

/* Insere um novo elemento no fim e reorganiza o "heap" */
procedure INSERT(e)
begin
    m ← m+1
    H_m ← e
    HEAPIFYUP(m)
end

/* Sobe elemento no "heap" (em direcção a índice 1) por trocas com o pai */
procedure HEAPIFYUP(i)
begin
    while i>1 ∧ COMPARE(H_i, H_⌊i/2⌋) > 0 do
        H_i ↔ H_⌊i/2⌋
        i ← ⌊i/2⌋
end

/* Desce elemento no "heap" (em direcção a índice m) por trocas com o filho */
procedure HEAPIFYDOWN(i)
begin
    while true
        k ← 2i
        if k>m then return
        if k<m ∧ COMPARE(H_k+1, H_k) >0 then k ← k+1
        if COMPARE(H_i, H_k) ≥ 0 then return
        H_i ↔ H_k /* swap */
        i ← k
end
```

É sabido [AU92] que *INSERT* e *DELETMAX* têm complexidade temporal O(*log m*), que *HEAPIFY* tem complexidade temporal *O(m)* e é óbvio que *INSERTRAW* tem complexidade temporal *O(1)*. Na construção inicial do "heap" usa-se uma sequência de *INSERTRAW's* seguido de HEAPIFY em vez

duma sequência de *INSERT's*, porque a complexidade temporal é *O(k)* no 1º caso e *O(k log k)* no 2º caso, em que *k* é o número de elementos inseridos. Como as operações *INSERT* (ou *INSERTRAW*, que é ainda melhor) e *DELETEMAX* são efectuadas uma vez para cada vértice e o tamanho máximo do "heap" é $|V|$, o algoritmo completo tem complexidade temporal $O(|V| \log |V| + |E|)$, no pior caso. Se não existirem arestas, a ordenação corresponde ao método "heapsort". Se o grafo for muito conexo, o tamanho máximo do "heap" é pequeno e a complexidade temporal aproxima-se de $O(|V| + |E|)$.

8.5.3 Obtenção dos componentes fortemente conexos por ordem topológica

Um algoritmo para obter os CFC's de um grafo por ordem topológica, baseado em [RNN77] com pequenas optimizações e adaptações, é apresentado a seguir.

Algoritmo 8.6 (*obtenção de componentes fortemente conexos por ordem topológica):*

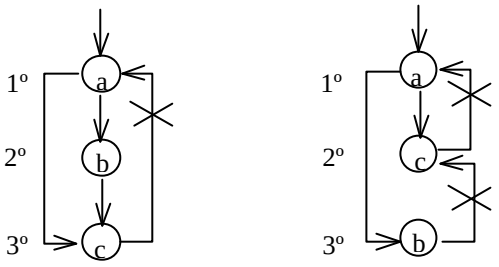
```
/*
Entradas:
    V - conjunto de vértices
    Para cada vértice v∈ V,
        Succ(v) - conjunto de sucessores imediatos de v, i.e., Succ(v)={u∈ V: (v,u)∈ E}
Saídas:
    Para cada vértice v∈ V,
        num(v) - numeração dos vértices por ordem topológica inversa de CFC's
        isroot(v) - indica se v é raiz (início) de um CFC (os restantes vértices estão a seguir, com
                números decrescentes)
Dados temporários:
    S - pilha auxiliar com vértices visitados mais ainda não ordenados
    Para cada vértice v∈ V,
        visited(v) - indica se o vértice v já foi visitado
        lowlink(v) - número de v por ordem de visita ou de um vértice acessível a partir
                de v que se encontra na pilha de chamada antes de v (ver [RNN77])
*/

/* procedimento principal */
procedure TOPSORTSCC
begin
    n ← 0
    for v∈ V do visited(v) ← false
    for v∈ V do if ¬visited(v) then
        i ← 0
        S ← pilha vazia
        DFSSCC(v)
end

/* visita em profundidade a partir do vértice v */
procedure DFSSCC(v)
begin
    i ← i+1
    lowlink(v) ← i
    visited(v) ← true
    isroot(v) ← true
    push(v,S)
    for u ∈ Succ(v) do
        if ¬visited(v) then DFSSCC(u)
```

```
            if u∈ S ∧ lowlink(u)<lowlink(v) then
                isroot(v) ← false
                lowlink(v) ← lowlink(u)
        if isroot(v) then /* retira este CFC de S e numera os vértices */
            repeat
                u ← pop(S)
                n ← n+1
                num(u) ← n
            until u=v
end
```

Algumas partes do algoritmo são executadas uma vez por cada vértice e outras são executadas uma vez por cada aresta. Assim, a complexidade temporal do algoritmo é *O(|V|+|E|)*.

8.5.4 Minimização de inversões de arestas nos componentes fortemente conexos

O algoritmo da secção anterior não garante que seja invertido um conjunto mínimo de arestas em cada CFC. Isso acontece porque os vértices de cada CFC são ordenados por ordem de visita. Na figura seguinte são apresentadas duas ordens de visita possíveis do mesmo CFC, partindo da mesmo raiz (vértice 1).



No caso da esquerda, é invertido um conjunto mínimo de arestas, mas no caso da direita isso já não acontece.

Para inverter um conjunto mínimo de arestas em cada CFC (e, consequentemente, no grafo completo, porque os CFC's são ordenados topologicamente), basta (de acordo com o teorema 8.1) proceder como no 1º algoritmo de ordenação topológica. Isso corresponde a efectuar a operação push(v,S) só depois de visitar os sucessores de v no algoritmo 8.6. Há que ter o cuidado em saber delimitar, para cada vértice v, os sucessores de v na pilha. O algoritmo modificado fica:

Algoritmo 8.7 (*ordenação por ordem ordem topológica de CFC's minimizando inversões de arestas):*

```
/*
Entradas: idem
Saídas: idem (induzindo conjunto mínimo de arestas de realimentação)
Dados temporários: idem, mais:
    Para cada vértice v∈ V,
        pending(v) - indica que v foi visitado mas ainda não foi ordenado; corresponde a
                     v∈ S na versão anterior
        old_S_size(v) - tamanho de S antes de visitar sucessores de v
                     (pode ser implementado por uma variável local)
*/

/* procedimento principal */
```

```
procedure TOPSORTSCCFES
begin
    n ← 0
    for v∈ V do
        visited(v) ← false
        pending(v) ¬ false
    for v∈ V do if ¬visited(v) then
        i ← 0
        S ← pilha vazia
        DFSSCCFES(v)
end

/* visita em profundidade a partir do vértice v */
procedure DFSSCCFES(v)
begin
    i ← i+1
    lowlink(v) ← i
    visited(v) ← true
    isroot(v) ← true
    pending(v) ¬ true
    old_S_size(v) ¬ |S| /* tamanho da stack antes de inserir sucessores de v */
    for u ∈ Succ(v) do
        if ¬visited(v) then DFSSCCFES(u)
        if pending(u) ∧ lowlink(u)<lowlink(v) then
            isroot(v) ← false
            lowlink(v) ← lowlink(u)
    push(v,S)
    if isroot(v) then /* retira v e sucessores de v do topo de S */
        n ¬ n + |S| - old_S_size(v)
        i ¬ n
        repeat
            u ← pop(S)
            num(u) ← i
            i ¬ i-1
            pending(u) ¬ false
        until |S| = old_S_size(v)
end
```

8.5.5 Algoritmo final

Basta agora combinar o algoritmo 8.7 com o algoritmo 8.5, aplicado ao grafo condensado acíclico dos CFC's, sem chegar a construir este último. Escolhe-se para prioridade de um CFC a prioridade do vértice raiz (porque é o 1º a ser escolhido do CFC). A numeração produzida pelo algoritmo 8.7 é usada para desempatar os casos de vértices com igual prioridade. Assim, se todos os vértices têm igual prioridade, é mantida a numeração produzida pelo algoritmo 8.7.

Algoritmo 8.8 (*ordenação por ordem ordem topológica de CFC's minimizando inversões de arestas e de prioridades):*

```
/*
Entradas: idem
Saídas: idem
```

```
Dados temporários: idem, mais:
    C - conjunto de raízes de CFC's
    Para cada vértice v∈ C,
        indegree(v) - número de arestas que partem de CFC's anteriores por ordenar para
                      o CFC de v
    H - "heap" com raizes de CFC's não ordenados com indegree=0
    m - tamanho do "heap"
*/

/* procedimento principal */
procedure TOPSORTSCCFESPRI
begin

    /* obtenção de CFC's e cálculo de indegree */
    C ¬ {}
    n ← 0
    for v∈ V do
        visited(v) ← false
        pending(v) ← false
    for v∈ V do if ¬visited(v) then
        i ← 0
        S ← pilha vazia
        DFSSCCFESPRI(v)

    /* ordenação de CFC's baseado em indegree que vai sendo decrementado */
    n ¬ |V|
    ERASEHEAP
    for v Î C do if indegree(v)=0 then INSERTRAW(v)
    HEAPIFY
    while |H| > 0 do
        v ¬ DELETEMAX
        for u such that root(u)=v /* por ordem de num(v) antigo decrescente */ do
            num(u) ¬ n /* renumera */
            n ¬ n-1
            for w Î Succ(u) do if root(w)¹ root(u) then
                indegree(root(w)) ¬ indegree(root(w))-1
                if indegree(root(w))=0 then INSERT(root(w))
end

/* visita em profundidade a partir do vértice v */
procedure DFSSCCFESPRI(v)
begin
    i ← i+1
    lowlink(v) ← i
    visited(v) ← true
    isroot(v) ← true
    pending(v) ← true
    old_S_size(v) ← |S|
    for u ∈ Succ(v) do
        if ¬visited(v) then DFSSCCFESPRI(u)
        if Øpending(u) then indegree(root(u)) ¬ indegree(root(u))+1
```

```
            if pending(u) ∧ lowlink(u)<lowlink(v) then
                isroot(v) ← false
                lowlink(v) ← lowlink(u)
        push(v,S)
        if isroot(v) then /* retira v e sucessores de v do topo de S */
            n ← n + |S| - old_S_size(v)
            i ← n
            repeat
                u ← pop(S)
                num(u) ← i
                i ← i-1
                root(u) ¬ v
                pending(u) ← false
            until |S| = old_S_size(v)
            indegree(v) ¬ 0
            C ¬ CÈ{v}
end

/* Função de comparação de dois elementos do heap usada pelas rotinas de gestão do heap.
   Retorna >0, 0 ou <0 conforme o 1º elemento é maior, igual ou menor do que o segundo. */
function COMPARE(u, v)
begin
    if priority(u) = priority(v)  then return num(u) - num(v)
    else return priority(u) - priority(v)
end

/** Seguir-se-iam rotinas de gestão do "heap" como em TOPSORTPRI **/
```

Em consequência dos resultados anteriores, facilmente se conclui que a complexidade temporal deste algoritmo é $O(|V| \log |V| + |E|)$, no pior caso.

9 Integração numa ferramenta de desenvolvimento de aplicações

Neste capítulo descreve-se a integração do motor de regras descrito no capítulo anterior (o SAGRES) e de regras activas dirigidas pelos dados numa ferramenta integrada de desenvolvimento rápido e execução de aplicações interactivas de bases de dados relacionais (o SAGA). As aplicações construídas com o SAGA são constituídas essencialmente por vistas interactivas com regras activas. As vistas são as classes da aplicação controladas pelo motor de regras. Algumas regras são geradas automaticamente pela ferramenta a partir de propriedades de mais alto nível das vistas e das tabelas da base de dados nelas referenciadas. A parte de acção dessas regras é implementada por funções em C codificadas na ferramenta, por razões de eficiência. O mecanismo de regras serve, neste caso, para automatizar a gestão de dependências ("dependency tracking"). Outras regras são definidas pelo programador de aplicações numa linguagem de regras e comandos disponível para o efeito. As regras deste tipo servem para especificar restrições de integridade, dados derivados, e comportamento reactivo em geral que não pode ser especificado por propriedades de mais alto nível das vistas. O capítulo termina com a apresentação de algumas conclusões tiradas da experiência de utilização da ferramenta e a identificação de algumas linhas de evolução para o futuro, numa perspectiva de Engenharia de Software.

9.1 Características gerais da ferramenta

O SAGA - Sistema Assistido de Geração e Gestão de Aplicações - é uma ferramenta destinada a automatizar o desenvolvimento de aplicações interactivas de bases de dados relacionais, constituídas essencialmente por formulários de ecrã, relatórios e alguns processamentos. Enquadra-se no âmbito das ferramentas "lower" CASE ("Computer Aided Software Engineering") e dos ambientes de desenvolvimento de aplicações de 4ª geração [P97].

O SAGA foi criado no INESC-Porto por uma pequena equipa liderada pelo autor e por João Vasco Ranito [MFR89][F90][FR91]. Tem sofrido evoluções significativas ao longo dos anos, da responsabilidade do autor, algumas delas documentadas em [F93a][F93b][F93c] e outras em relatórios de alterações. Conta presentemente com cerca de 135.000 linhas de código fonte em C. O SAGA tem sido usado quase exclusivamente no desenvolvimento de aplicações comerciais para o mercado Autárquico, de que se salienta o pacote SIGMA [MLF+89][VMF93][FLMB95], embora o SAGA em si como ferramenta de desenvolvimento não seja comercializado de forma alargada.

Descrevem-se de seguida algumas características importantes do SAGA.

9.1.1 Portabilidade e conectividade

O SAGA funciona sobre múltiplas plataformas:

- sobre vários SGBD's relacionais: Informix, Oracle, SQL Server, DISAM [2], outros via ODBC;

[2] O DISAM não é um verdadeiro SGBD, mas apenas um gestor de ficheiros baseado no método de acesso sequencial indexado (ISAM), com código fonte disponível gratuitamente, ao qual foram acrescentadas algumas funções de gestão de bases de dados.

- sobre vários sistemas operativos: Unix ou Windows;
- sobre vários sistemas de interface para o utilizador: alfanumérico (usando o pacote Curses em Unix) ou gráfico (em Windows);
- em diferentes arquitecturas: cliente-servidor ou centralizada.

O SAGA oferece independência da plataforma:

- uma aplicação desenvolvida sobre uma plataforma pode ser executada em qualquer outra plataforma suportada, sem qualquer rescrita;
- diferentes sessões da mesma aplicação, partilhando a mesma base de dados, podem estar a correr simultaneamente em diferentes sistemas de interface para o utilizador (alfanuméricos e gráficos).

9.1.2 Vistas

No SAGA, formulários e relatórios são unificados através do conceito de *vista interactiva da aplicação* ou simplesmente *vista*. As *vistas* são os componentes principais das aplicações. As vistas são assim designadas porque permitem visualizar, agregar e relacionar de múltiplas formas a informação armazenada na base de dados de uma forma normalizada não redundante. O conceito de vista do SAGA corresponde ao conceito de *vista da aplicação* em [M97].

Destacam-se as seguintes facilidades:

- todas as vistas se podem manipular interactivamente no ecrã (para consulta ou alteração) e imprimir, havendo poucas variações entre a formatação para o ecrã ou para a impressora, segundo o lema "what-you-see-is-what-you-get";
- podem-se efectuar pesquisas flexíveis por quaisquer campos das vistas, incluindo campos calculados, campos de sumário e campos de sub-vistas, sem necessidade de qualquer programação;
- vistas simples podem ser geradas automaticamente e refinadas depois através de um utilitário de desenho ("paint");
- podem-se manipular várias tabelas da base de dados através da mesma vista, sem necessidade de qualquer programação, porque são geradas automaticamente as condições de junção, os critérios de actualização da base de dados e as regras de integridade relacional correspondentes;
- podem-se construir vistas mais complexas contendo (sub)vistas mais simples, de que é exemplo o formulário apresentado na figura 7.1;
- uma vista pode estar organizada hierarquicamente, com secções dentro de secções, em que as secções correspondem a agrupamentos de registos;
- mecanismos automáticos de selecção incremental de dados da base de dados para as vistas permitem navegar eficientemente através de grandes quantidades de dados.

9.1.3 Regras

As vistas são refinadas através da adição de regras e comandos, escritas numa denominada linguagem de regras e comandos semelhante às linguagens que se encontram noutros ambientes de 4ª geração. Os comandos definem acções que o utilizador pode invocar explicitamente através de botões ou opções de menus. Em contrapartida, as regras são executadas automaticamente pelo sistema em resposta à ocorrência de eventos de manipulação de dados, servindo para diversas finalidades:

- manutenção de dados derivados (materializados): campos calculados em função de outros campos, registos seleccionados em função de outros registos, etc.;
- verificação de restrições de integridade: restrições envolvendo um único campo, vários campos do mesmo registo ou vários registos;
- realização de acções e validações no início (“before”) ou no fim (“after”) de operações pré-definidas (inserir, actualizar ou eliminar registos, salvar alterações, etc.) para as aumentar ou redefinir;

- definição de vistas que não tiram partido dos mecanismos automáticos (mas limitados) de selecção e actualização;
- definição de valores iniciais e por omissão;
- conversão e correcção de entradas;
- controlo de autorizações e obrigações;
- monitorização e alerta de situações.

A programação através de regras tem várias vantagens: é mais modular, está mais próxima da especificação e está mais adaptada a sistemas interactivos dirigidos por eventos.

9.1.4 Integração de ambientes de desenvolvimento e execução de aplicações

A integração entre os ambientes de desenvolvimento e de execução das aplicações assume diversos aspectos:

- as especificações das aplicações (tabelas, vistas, regras, menus, utilizadores, permissões, etc.) são armazenadas numa base de dados relacional (em tabelas ditas de sistema), tal como os dados manipulados por essas aplicações;
- na mesma sessão de trabalho (através de um único executável), pode-se passar uma vista do modo de execução para o modo de desenho e vice-versa, e podem coexistir vistas em modo de desenho e em modo de execução;
- grande parte do trabalho de desenvolvimento de aplicações é efectuado através de vistas criadas com o SAGA (ditas de sistema), assemelhando-se assim ao trabalho de utilização final das aplicações;
- o próprio desenho de "layout" das vistas pode ser efectuado através de vistas com um mapeamento para o ecrã diferente do habitual, em que os campos das vistas correspondem a atributos de objectos gráficos e não a objectos gráficos (os registos das vistas é que passam a corresponder a objectos gráficos).

A integração tem diversas vantagens:

- o desenvolvimento é mais expedito, pois o ciclo codificação-teste é encurtado;
- os utilizadores finais ou administradores de sistemas podem realizar pequenas acções de manutenção;
- o desenvolvimento de aplicações beneficia dos serviços fornecidos pelos SGBD's relacionais, nomeadamente serviços de interrogação e de controlo de concorrência;
- a ferramenta é mais pequena e mais fácil de manter.

9.2 Definição e manipulação de vistas

Apesar de serem normalmente definidos ao mesmo tempo, convém distinguir as seguintes componentes na definição de uma vista no SAGA:

- definição da estrutura de dados;
- definição do mapeamento para a camada de armazenamento persistente (base de dados);
- definição do mapeamento para a camada de apresentação e interacção com o utilizador (interface com o utilizador).

A terceira componente não será abordado aqui, porque a sua compreensão não é importante para a compreensão do mecanismo de regras.

9.2.1 Estrutura de dados básica de uma vista

Em termos do seu conteúdo de dados, uma vista do SAGA é basicamente uma tabela (conjunto de registos do mesmo tipo, i.e., com os mesmos campos *intra-registo*) mais um conjunto de campos *globais*. Todos os campos são atómicos. Podem existir vistas apenas com campos globais e vistas

apenas com campos intra-registo (com estrutura semelhante às vistas que se definem em SQL). É principalmente sobre vistas do seguinte tipo que concentraremos a nossa atenção.

A estrutura de dados básica de uma vista é especificada por uma lista de campos, com (no mínimo) o nome, tipo de dados e granularidade (global ou intra-registo) de cada campo.

As vistas materializam-se parcialmente quando são abertas durante uma sessão de trabalho, de forma temporária e privativa a essa sessão de trabalho. Em cada sessão de trabalho só pode estar aberta uma "instância" de cada "tipo" de vista.

Uma vista pode ter associado um *critério de selecção* (correspondente ao conceito de filtro no Microsoft Access) que restringe os registos da vista (entendida como tabela virtual) que são *seleccionados* (materializados). O critério de selecção contém um valor ou uma expressão de selecção (string com valores, meta-caracteres e operadores relacionais) para cada campo intra-registo. O critério de selecção não se refere aos campos globais. No SAGA, o critério de selecção também é chamado *registo de critério*. Para desfazer confusões, os registos "normais" são chamados *registos de dados*.

Uma vez que as alterações efectuadas numa vista podem não ser reflectidas imediatamente na base de dados, os registos de dados de um vista podem dividir-se em:

- registos *seleccionados* - registos que foram seleccionados e não foram posteriormente modificados (correspondem aos registos válidos no Designer/2000);
- registos *modificados* - registos que foram seleccionados e posteriormente modificados, e a sua modificação ainda não foi reflectida na base de dados;
- registos *inseridos* - registos que foram inseridos na vista, possivelmente modificados posteriormente, e a sua inserção ainda não foi reflectida na base de dados;
- registos *eliminados* (pendentes) - registos que foram seleccionados e posteriormente eliminados da vista (podendo ter sido entretanto modificados), e a sua eliminação ainda não foi reflectida na base de dados.

Os registos do último tipo são mantidos numa área de registos escondidos, até que a sua eliminação seja reflectida na base de dados. Quando se salvam as alterações, isto é, quando as alterações na vista são reflectidas (traduzidas por alterações) na base de dados, os registos eliminados desaparecem, e os registos modificados e inseridos passam ao estado de seleccionados.

9.2.2 Mapeamento para a base de dados

No que se refere ao modo de mapeamento para a base de dados, podem-se definir dois tipos de vistas:

- vistas actualizáveis, cujo mapeamento para a base de dados é definido (de forma bidireccional) através da indicação de tabelas-alvo e campos-alvo, podendo ser complementado através de regras;
- vistas não actualizáveis (pelo menos automaticamente), em que o mapeamento para a base de dados está embebido em regras que podem conter expressões de selecção genéricas.

É de vistas do primeiro tipo (*"targeted views"*) que se trata aqui com algum detalhe, porque a sua compreensão é importante para a compreensão das regras geradas automaticamente pelo SAGA.

Uma vista do primeiro tipo tem uma *tabela-alvo* na base de dados. A cada registo da vista corresponde um registo da tabela-alvo (chamado registo-alvo), e a cada registo da tabela-alvo corresponde virtualmente (não necessariamente em forma materializada) zero ou um registo da vista.

Um campo duma vista pode ter um *campo-alvo* na base de dados, que é o campo da base de dados de onde são seleccionados valores para o campo da vista. O campo-alvo pode ser da tabela-alvo ou doutra tabela, chamada *tabela de extensão*, ou cópia lógica de tabela (tratada também como tabela de extensão). Podem existir campos da vista sem campo-alvo na base de dados; normalmente são campos calculados por regras.

As condições de junção entre a tabela-alvo e as tabelas de extensão são determinadas automaticamente pelo SAGA, com base no conhecimento das chaves primárias e chaves estrangeiras das tabelas da base de dados, conforme se explica a seguir.

Para cada tabela referenciada (tabela-alvo ou tabela de extensão) é determinada automaticamente uma *chave de acesso*, para acesso à chave primária da tabela, constituída por campos ou parâmetros da vista. Os campos da vista escolhidos para o acesso à chave primária da tabela-alvo têm de ter como alvo campos correspondentes dessa chave primária. Os campos da vista escolhidos para o acesso à chave primária de uma tabela de extensão têm apenas de ter como alvo campos com a mesma *raiz* que os campos correspondentes dessa chave primária. A raiz de um campo da base de dados é definida recursivamente da seguinte forma: é o próprio campo, se não participar numa chave estrangeira; é a raiz do campo correspondente na chave primária referenciada pela chave estrangeira, no caso contrário (esta definição pressupõe algumas restrições à forma como as chaves estrangeiras são definidas). No caso de não existir na vista um campo nas condições requeridas, é criado um parâmetro automático da vista que deve ser ligado ("bound") dinamicamente a uma constante ou a um campo doutra vista com uma raiz apropriada na base de dados. A chave de acesso pode também ser definida pelo programador de uma forma mais livre.

As chaves de acesso determinam as condições de junção entre as tabelas referenciadas e o sentido em que são definidas as junções. O sentido é importante porque se usa, em geral, o método de junção externa assimétrica esquerda (*left outer join*) [R98] partindo da tabela-alvo, e só em casos particulares (relacionados com restrições do tipo *not null*) se usa o método de junção interna (*inner join*) mais tradicional. A ideia base é a seguinte: se um campo da vista tem como alvo um campo *t1.c1*, e é usado para aceder a um campo *t2.c2* da chave primária de *t2* (tabela de extensão), então a condição de junção é *t1.c1 = t2.c2* e o sentido da junção externa é de *t1* para *t2*. Esta junção externa significa que são seleccionados também os registos de *t1* que não têm nenhum registo correspondente em *t2* (os campos correspondentes do resultado são preenchidos com valores nulos).

O método completo de obtenção das condições de junção (que envolve alguns detalhes e restrições que seria fastidioso estar aqui a descrever, os quais podem ser encontrados em [F93b]), faz com que as junções sejam definidas segundo uma *cadeia de junções por extensão* [U88] partindo da tabela-alvo. Numa cadeia deste tipo, o número de registos (linhas) do resultado nunca aumenta à medida que se efectua a junção com a tabela seguinte da cadeia. Assim, garante-se efectivamente uma correspondência de zero ou um para um entre os registos duma vista e os registos da sua tabela-alvo. Diferentemente do que se passa com a tabela-alvo, a cada registo da vista pode corresponder zero ou um registo de cada tabela de extensão. Em sentido inverso, consoante as condições de junção, há casos em que a um registo de uma tabela de extensão pode corresponder mais do que um registo da vista, e há casos em que a um registo de uma tabela de extensão não corresponde mais do que um registo da vista. Por omissão, apenas é permitido actualizar através da vista a tabela-alvo e as tabelas de extensão que se enquadram no segundo caso (as outras tabelas podem também ser actualizadas com segurança se a sincronização com a base de dados for a um registo da vista de cada vez). Os métodos exactos de actualização não interessam aqui porque não são tratados através de regras.

Exemplo

Sejam o formulário (vista mestre) de manutenção de facturas e o sub-formulário (sub-vista ou vista de detalhe) apresentados na figura 7.1. Suponhamos que a base de dados manipulada através dessas vistas tem o seguinte esquema relacional:

Fornecedor(código, nome)
Factura(número, data, código_fornecedor → Fornecedor)
Artigo(código, nome)
LinFact(número_factura → Factura, código_artigo → Artigo, quantidade, preço_unitário)

Os atributos sublinhados constituem a chave primária. A notação *campo→tabela* significa que o atributo do lado esquerdo constitui uma chave estrangeira que referencia a chave primária da tabela (relação) do lado direito.

Por sua vez, os esquemas da vista mestre (*VFactura*) e da vista de detalhe (*VLinFact*) podem ser definidos da seguinte forma:

VFactura[Factura] (número [Factura.número], data [Factura.data],
código_fornecedor [Factura.código_fornecedor], nome_fornecedor [Fornecedor.nome],
preço_total)

VLinFact [LinFact] (código_artigo [Artigo.código], nome_artigo [Artigo.nome],
quantidade [LinFact.quantidade], preço_unitário [LinFact.preço_unitário], preço_total)

Os alvos são indicados entre parêntesis rectos.

No caso da vista *VFactura*, a chave de acesso à tabela-alvo (*Factura*) é obviamente constituída pela campo *número*, porque tem como alvo a chave primária da tabela. A chave de acesso a *Fornecedor* (tabela de extensão) é constituída pelo campo *código_fornecedor*, porque tem como alvo um campo da base de dados (*Factura.código_fornecedor*) com a mesma raiz (*Fornecedor.código*) do que a chave primária de *Fornecedor* (*Fornecedor.código*).

No caso da vista *VLinFact*, a chave de acesso à tabela-alvo (*LinFact*) é constituída por:

- um parâmetro da vista (com nome automático *import0*) a ligar a uma constante ou a um campo doutra vista que tenha como alvo um campo com a mesma raiz (*Factura.número*) do que o primeiro campo da chave primária de *LinFact* (*LinFact.número_factura*);
- o campo *código_artigo*, porque tem como alvo o segundo campo da chave primária de *LinFact* (*LinFact.código_artigo*).

Note-se que, ao ligar a vista *VLinFact* à vista *VFactura*, o parâmetro *import0* é ligado ao campo *VFactura.número*. A chave de acesso a *Artigo* (tabela de extensão) é constituída pelo campo *código_artigo*, porque tem como alvo um campo da base de dados (*LinFact.código_artigo*) com a mesma raiz (*Artigo.código*) do que a chave primária de *Artigo* (*Artigo.código*).

Dadas as chaves de acesso anteriormente definidas, e a menos de campo calculados e da possível má utilização do parâmetro ":import0" (que é legítima pelo menos em expressões de SQL embebido em C), as vistas *VFactura* e *VLinFact* poderiam ser definidas em SQL-92 [R98] da seguinte forma:

```
CREATE VIEW VFactura(número, data, código_fornecedor, nome_fornecedor)
  AS SELECT Factura.número, Factura.data, Factura.código_fornecedor, Fornecedor.nome
  FROM Factura LEFT OUTER JOIN Fornecedor
  WHERE Factura.código_fornecedor = Fornecedor.código

CREATE VIEW VLinFact(código_artigo, nome_artigo, quantidade, preço_unitário)
  AS SELECT LinFact.código_artigo, Artigo.nome, LinFact.quantidade, LinFact.preço_unitário
  FROM LinFact LEFT OUTER JOIN Artigo
  WHERE LinFact.número_factura = :import0 AND LinFact.código_artigo = Artigo.código
```

Note-se, no entanto, que o SAGA não usa a facilidade de definição de vistas em SQL. Em vez disso, gera dinamicamente expressões de selecção do tipo acima indicado, reorganizadas de forma mais adequada a cada caso (dependendo do SGBD usado e do critério de selecção definido pelo utilizador).

Para tornar as actualizações mais eficientes (por acesso directo) e controlar os "locks" ao nível do registo, são também seleccionados os identificadores internos ("rowid") dos registos da base de dados acedidos (registo-alvo e registos de extensão). Os atributos da vista que armazenam os identificadores do registo-alvo e dos registos de extensão são designados pelo nome da tabela da base de dados, seguido de um número de ordem para distinguir possíveis cópias lógicas de uma tabela.

9.2.3 Vistas com sub-vistas

Uma vista pode ter *sub-vistas*. Por sua vez, cada sub-vista pode ter as suas próprias "sub-sub-vistas", e assim sucessivamente. Um exemplo simples é o caso da vista de manutenção de facturas apresentada na figura 7.1 e tratada no exemplo anterior, com uma sub-vista de manutenção de linhas

de facturas. A relação entre uma vista e uma sua sub-vista é uma *relação mestre-detalhe*, em que a primeira assume o papel de *vista mestre* e a segunda assume o papel de *vista de detalhe*. Em geral, é possível definir uma árvore cujos nós são vistas e os ramos representam relações mestre-detalhe. A raiz da árvore é a vista de nível de topo (nível 1). Os outros nós da árvore são sub-vistas possivelmente de vários níveis. A árvore completa é chamada uma *vista encaixada* ("nested") ou *vista hierárquica*.

Tanto em termos visuais como em termos lógicos, existem diferentes graus de acoplamento entre vistas mestre e vistas de detalhe. Em termos visuais, a vista de detalhe pode ocupar uma sub-janela da vista mestre ou pode ocupar uma janela separada. Em termos lógicos, certas operações de manipulação de vistas (abrir e fechar, alterar e salvar alterações, interrogar, etc.) podem ser realizadas em conjunto ou separadamente.

Em termos do conteúdo de dados, a ligação entre as vistas envolvidas numa vista encaixada é efectuada normalmente através da definição de um ou mais parâmetros em cada sub-vista (também chamados *parâmetros de ambiente*), os quais são ligados ("bound") a campos de vistas que se encontram em linha ascendente na árvore (também chamadas *vistas de ambiente*), definido assim condições de junção entre vistas. Um caso simples foi descrito no exemplo da secção anterior.

Por razões de eficiência e de facilidade de implementação (relacionada com o facto de só poder estar aberta uma "instância" de cada "tipo" de vista), uma sub-vista é materializada apenas para o(s) registo(s) corrente(s) das vistas em linha ascendente na árvore, isto é, para uma única combinação de valores dos parâmetros da vista. Normalmente, isto não faz qualquer diferença para o utilizador (mesmo ao imprimir), porque à medida que se percorrem os registos de uma vista as suas sub-vistas (directas ou indirectas) vão sendo automaticamente (re)materializadas. Quando tal limitação não é satisfatória, existe a possibilidade de usar uma única vista organizada em secções hierárquicas. De qualquer forma, seria interessante remover esta limitação no futuro (aliás comum às ferramentas comerciais mais importantes).

9.2.4 Vistas com seccões hierárquicas

Em cada vista é possível especificar um critério de ordenação (correspondente a "order by" em SQL), na forma de uma lista de campos da vista, também chamada *chave de ordenação*.

O critério de ordenação pode ser usado simultaneamente como critério de seccionamento de registos, campos e "layout" da vista. Suponhamos que a chave de ordenação é $c_1, c_2, ..., c_n$. Então o seccionamento processa-se da seguinte forma:

- *Seccionamento dos registos*: Para qualquer i entre 2 e $n+1$, é possível dividir os registos da vista em subconjuntos (ou grupos) de nível i constituídos por registos com os mesmos valores nos campos $c_1, c_2, ..., c_{i-1}$. Por definição, o nível 1 (também chamado nível de vista) engloba todos os registos. No caso da chave de ordenação não ser suficiente para distinguir registos individuais, considera-se um nível $n+2$ (nível de detalhe). Os subconjuntos do nível de detalhe ($n+1$ ou $n+2$, conforme o caso) têm apenas um registo.
- *Seccionamento dos campos*: Cada campo c da vista tem um nível i associado, dado pelo nível mais baixo tal que o valor de c é constante dentro de qualquer subconjunto de registos de nível i. Este seccionamento tem implicações no significado das regras que calculam campos de sumário e na forma como os dados são armazenados fisicamente.
- *Seccionamento do "layout"*: Para cada nível i de 1 até ao nível de detalhe menos 1, pode-se definir um cabeçalho ("header") e um rodapé ("footer") de nível i. Adicionalmente, claro, pode-se definir uma área de detalhe. Ao imprimir, o cabeçalho/rodapé de nível i é apresentado no primeiro/último registo de cada subconjunto de registos de nível i.

9.2.5 Restrições de integridade e de acesso

Podem-se definir zero ou mais chaves numa vista. Uma chave é um grupo de campos tal que não podem existir dois registos diferentes com os mesmos valores nesses campos. Pelo menos a chave primária da tabela alvo é automaticamente herdada.

É possível associar a cada campo uma lista de valores permitidos (definida por uma expressão na linguagem de regras e comandos que retorna uma lista).

Podem-se definir campos de preenchimento obrigatório ("mandatory") e campos não alteráveis pelo utilizador ("locked"), de forma estática (através de propriedades estáticas dos campos) ou de forma dinâmica (através de regras).

9.2.6 Operações

Os dois quadros seguintes descrevem resumidamente as operações de manipulação de vistas mais importantes. As operações do primeiro quadro incidem sobre uma vista na globalidade.

Operação	Descrição
open	Abre uma vista num modo à escolha (interrogar, adicionar ou modificar).
query	Passa uma vista para o modo de interrogar (editar registo de critério). Normalmente, os registos de dados são libertados quando se passa ao modo de interrogar.
load	Selecciona os registos que obedecem ao critério de selecção e passa ao modo de modificar (editar registos de dados). Normalmente o registo de critério é libertado quando se passa ao modo de modificar.
updglob	Engloba modificações de dados duma vista (modificação de campos globais e/ou alteração de conjuntos de registos com operações do quadro seguinte).
save	Salva as alterações efectuadas numa vista. Se o modo de sincronização com a base de dados for a um registo de cada vez (em vez de a um conjunto de registos de cada vez), esta operação é automaticamente efectuada cada vez que se muda de registo corrente. Pode causar alteração de dados da vista (nomeadamente identificadores de registos correspondentes da base de dados).
close	Fecha uma vista.

No caso de vistas encaixadas, algumas destas operações podem referir-se à vista encaixada na globalidade (dependendo do tipo de acoplamento), sendo invocadas para a vista de nível de topo e propagadas para as sub-vistas.

As operações do quadro seguinte incidem sobre um registo duma vista.

Operação	Descrição
insrec	Insere um registo numa vista, o qual fica corrente.
updrec	Modifica valores de campos do registo corrente duma vista.
delrec	Elimina o registo corrente duma vista.

Note-se que as acções realizadas interactivamente pelo utilizador (na camada de apresentação e interacção com o utilizador) são mapeadas para chamadas a estas operações.

Operações de consulta serão descritas a propósito da linguagem de regras e comandos.

9.2.7 Transacções

No SAGA, as alterações nas vistas são também protegidas por transacções, mesmo quando essas alterações não são imediatamente salvas na base de dados. Assim, há que distinguir as transacções nas vistas das transacções na base de dados. Para efeito da compreensão do mecanismo de regras,

interessam-nos essencialmente as primeiras. No entanto, devido à estreita relação que existe entre os dois tipos de transacções, abordam-se também as transacções do segundo tipo.

9.2.7.1 Transacções na base de dados

Existe um certo conflito entre a filosofia de funcionamento do SAGA, que permite trabalhar continuamente numa vista salvando alterações periodicamente, e o pouco ou nenhum suporte que existe nos SGBD's para suportar transacções de longa duração (que são reconhecidamente indesejáveis em ambientes de grande concorrência, mas são úteis em ambientes mais pessoais). Na realidade, o modelo de transacções adequado para suportar este modo de funcionamento seria um modelo de transacções encadeadas. Uma transacção encadeada ("chained transaction") é uma sequência de transacções, uma imediatamente a seguir à outra, possivelmente com recursos transferidos de uma transacção para a seguinte, e possivelmente com persistência tal que uma falha do sistema não quebre a cadeia [GR93]. A propriedade importante aqui é a transferência de recursos, em particular "locks" e cursores, de uma transacção para a seguinte e não o aspecto da persistência. O que o SAGA faz é basicamente simular transacções encadeadas à custa de transacções de curta duração (desencadeadas só durante "save") e "locks" de longa duração (desde que um registo é seleccionado até que é libertado), sempre que possível.

A situação é mais simples no caso de vistas com um modo de sincronização com a base de dados registo a registo. Nesse caso, quando um registo da vista fica corrente, é (re)seleccionado nesse momento a partir da base de dados (com base apenas no seu "rowid") e os registos da base de dados passíveis de serem alterados através da vista são bloqueados ("locked"). Quando o registo vai deixar de ficar corrente, as alterações são salvas na base de dados e os bloqueios ("locks") são retirados. Apesar de poder começar logo a seguir outra transacção (se outro registo ficar corrente), não há necessidade de transferir recursos (nomeadamente "locks") de uma transacção para a seguinte.

Um ponto de conflito menos importante tem a ver com o facto de o SAGA permitir trabalhar em várias vistas ao mesmo tempo na mesma sessão de trabalho (mais precisamente, podem existir alterações por salvar em mais do que uma vista). O suporte adequado para este modo de funcionamento seria dado por transacções encadeadas concorrentes solicitadas a partir da mesma sessão de trabalho.

9.2.7.2 Transacções nas vistas

Todas as alterações nas vistas são englobadas em transacções ou sub-transacções locais a cada sessão de trabalho (na sua memória de trabalho), apenas com as propriedades de atomicidade e preservação de consistência (conforme o conceito de transacção descrito no capítulo 8). Estas transacções interessam, principalmente, para estabelecer um contexto adequado para o processamento de regras e recuperação de erros (conforme explicado no capítulo 8).

As transacções são desencadeadas pelas operações de manipulação de dados acima referidas segundo as seguintes regras gerais:

- se não está em curso nenhuma transacção, é criada uma transacção global (de nível de topo) que engloba a execução da operação;
- se uma operação é intra-registo (caso de *updrec* e de sub-operações intra-registo nas operações *insrec*, *query*, *load* e *save*), e não está corrente uma sub-transacção intra-registo sobre o mesmo registo, é criada uma sub-transacção intra-registo;
- são criadas também sub-transacções intra-registo ou globais para englobar pontos de processamento de regras (PPR's) de forma a cumprir as exigências do motor de regras e limitar o âmbito de "rollback".

Estas regras tentam minimizar o número de sub-transacções criadas, mas introduzem uma sensibilidade ao contexto de chamada de cada operação que complica a sua compreensão, pelo que devem ser revistas no futuro.

As transacções nas vistas estão relacionadas com as transacções na base de dados de várias formas:

- quando uma transacção nas vistas é abortada, é reposta também a situação anterior em termos de "locks";
- se a transacção de curta duração na base de dados criada por uma invocação de *save* falhar, a transacção na vista que engloba a invocação de *save* é também abortada, e vice-versa.

Note-se ainda que nem todas as operações nas vistas podem ser desfeitas (caso de *close*, por exemplo), por razões de eficiência e facilidade de implementação.

9.2.8 Classes e atributos

Cada vista é registada como classe da aplicação no motor de regras.

Os campos globais e parâmetros de cada vista são registados como atributos globais (porque só se admite uma "instância" aberta de cada "tipo" de vista). Os campos intra-registo são registados como atributos intra-objecto.

Para cada campo *c*, são também registados atributos *c->locked* e *c->mandat*, com a mesma granularidade do campo, para possibilitar o controlo através de regras das restrições ("locked" e "mandatory") referidas na secção 9.2.5.

Os atributos que armazenam os identificadores dos registos acedidos da base de dados (ver secção 9.2.2) são também registados com a granularidade apropriada. Estes atributos são usados nas regras geradas automaticamente.

Adicionalmente, para efeito de activação e ordenação de algumas regras, são criados os seguintes atributos globais em cada vista:

Atributo	**Significado**
record_set	Designa o conjunto de identificadores internos dos registos existentes, sem incluir os registos marcados como eliminados.
record_current	Designa o identificador interno do registo corrente.

9.2.9 Pontos de processamento de regras

Os dois quadros seguintes descrevem os pontos de processamento de regras (PPR's) mais importantes existentes nas operações de manipulação de vistas, e a sua classificação para efeito de registo no motor de regras. Estes PPR's podem ser vistos como eventos com execução imediata de regras.

Operação	**PPR**	**Âmbito**	**Tipo de Operação**	**Notas**
open	after_open	vista	create	
query	before_query	vista	modify	
	after_query	registo	create	Ocorre em sub-transacção intra-registo para inicialização do registo de critério.
load	before_load	vista	modify	
	before_load_record	vista	modify	O âmbito é vista porque o registo ainda não existe.

	after_load_record	registo	create	Ocorre em sub-transacção intra-registo.
	after_load	vista	modify	
save	before_save	vista	modify	
	before_save_record	registo	modify	Ocorrem em sub-transacção intra-registo.
	after_save_record	registo	modify	
	after_save	vista	modify	
updglob	before_updglob	vista	modify	
	after_updglob	vista	modify	
close	before_close	vista	delete	

Operação	**PPR**	**Âmbito**	**Tipo de operação**	**Notas**
insrec	before_add	vista	modify	O âmbito é vista porque o registo ainda não existe.
	after_add	registo	create	Ocorre em sub-transacção intra-registo para inicialização do registo inserido.
updrec	before_updrec	registo	modify	
	after_updrec	registo	modify	
delrec	before_delete	registo	delete	Caso de eliminação de registo seleccionado ou modificado.
	after_delete	registo	delete	
	before_cancel_add	registo	delete	Caso de eliminação de registo inserido.
	after_cancel_add	registo	delete	

9.3 Regras definidas na linguagem de regras e comandos

A definição de uma vista pode ser refinada através da definição de regras numa linguagem de regras e comandos.

A definição de uma regra tem as seguintes partes obrigatórias:

- o nome da vista a que a regra pertence;
- um número de ordem da regra dentro da vista, usado para a atribuição de nomes às regras (letra "r" seguida do número) e de prioridades numéricas fracas (sendo que regras com número mais baixo têm prioridade mais alta);
- uma expressão na linguagem de regras e comandos que define a parte de acção da regra e, opcionalmente, a parte de eventos;

e as seguintes partes opcionais:

- o âmbito (granularidade) da regra - *registo* ou *vista*;

- os modos de operação da vista em que a regra não está inibida (propriedade "modos"): I - interrogar (editar registo de critério), M - modificar (editar registos de dados), B - buscar (durante *load*), G - gravar (durante *save*);
- uma lista de regras (identificadas por nome de vista e número de regra) com precedência sobre a regra em causa;
- uma lista de regras (identificadas por nome de vista e número de regra) sobre as quais a regra em causa tem precedência.

Caso não sejam omitidos, o âmbito e a lista de eventos são inferidos pelo sistema.

A linguagem de regras e comandos é assim designada porque também é usada na definição de comandos associados a uma vista. Diferentemente das regras, os comandos são invocados explicitamente pelo utilizador, através de botões, opções de menus ou teclas.

As regras podem ser definidas interactivamente através de uma vista encaixada como a indicada na figura seguinte (constituída por uma vista de nível de topo e duas sub-vistas sobre a mesma tabela para a introdução das listas acima referidas).

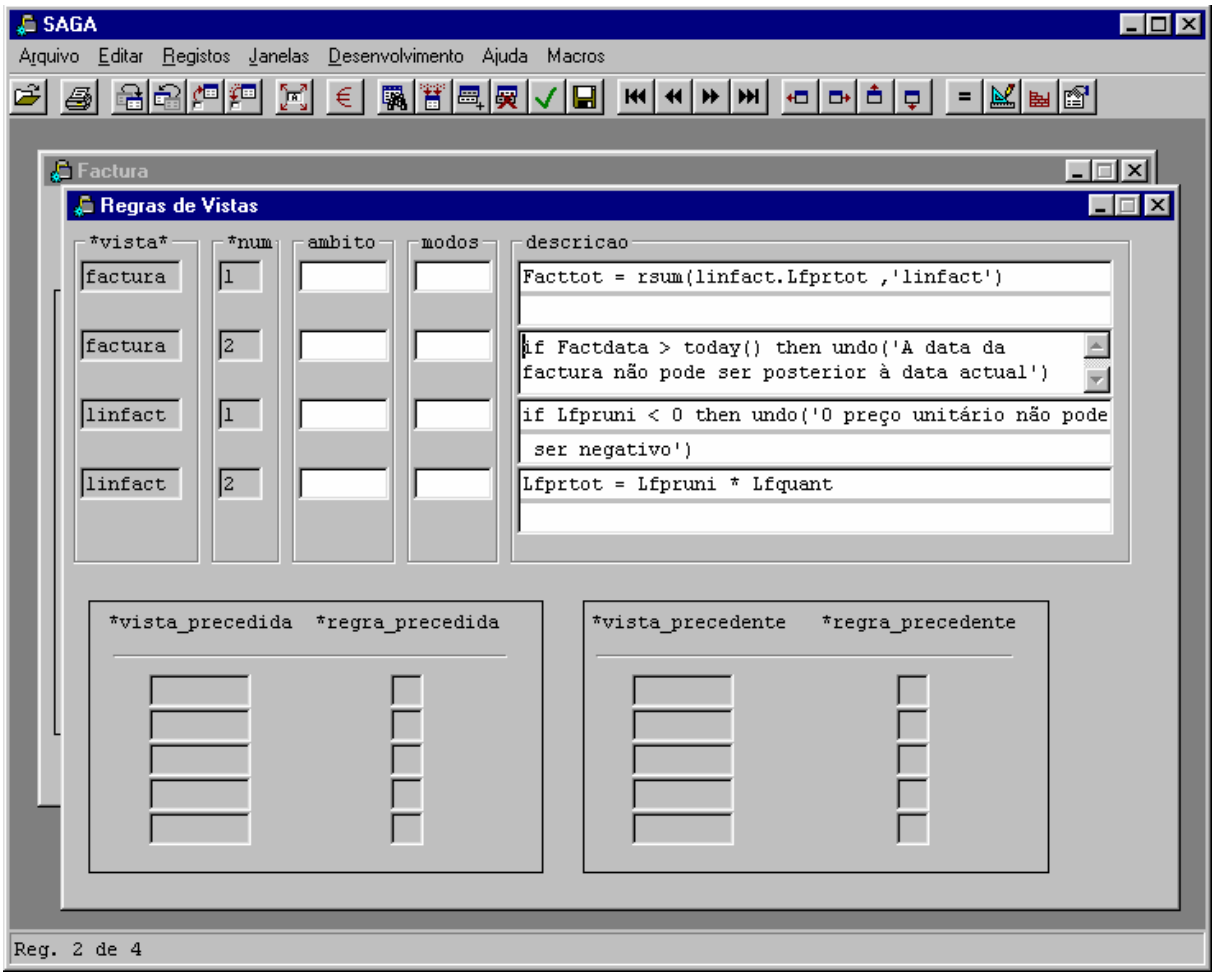


Fig. 9.1 Regras definidas pelo programador para impor algumas restrições da fig. 7.3 relativamente ao formulário da fig. 7.1. As restantes restrições são mantidas por regras geradas automaticamente.

9.3.1 Linguagem de regras e comandos

Não se pretende fazer aqui uma descrição completa desta linguagem, a qual se pode encontrar (numa versão antiga) em [F93c], mas apenas salientar alguns aspectos da sua concepção, e aspectos sintácticos (descritos informalmente) que permitem a apresentação de regras escritas nesta linguagem.

A linguagem de regras e comandos permite definir expressões com constantes, operadores, variáveis, chamadas de funções e algumas construções especiais.

As constantes podem ser dos seguintes tipos: inteiros, reais, strings (entre pelicas) e listas de constantes dos tipos anteriores (entre chavetas, com elementos separados por vírgulas). A lista vazia (`{}`) é uma constante especial com o significado de valor nulo, diferente de `0` e da string vazia [3]. O valor nulo é absorvente na maioria das operações.

A conversão entre números (inteiros ou reais) e strings é automática. Na conversão de string para número são entendidos formatos de representação de datas, horas e valores monetários. Por exemplo, a expressão `'1999/1/30' - '1999/1/20'` subtrai duas datas, e dá `10`. O operador unário "^" promove o valor nulo para string vazia (`''`) ou `0` (conforme o tipo preferido no ponto de invocação do operador), e mantém inalterados outros valores (de forma semelhante à função `Nz` do Microsoft Access).

Os operadores aritméticos, lógicos e de comparação (relacionais) têm a mesma notação e precedência que na linguagem C, embora nem todos os operadores de C estejam disponíveis. Nos operadores aritméticos e de comparação, o valor nulo é absorvente. O quadro seguinte indica o significado das expressões lógicos e condicionais mais importantes. Note-se a complicação perniciosa devida aos valores nulos (também encontrada noutros sistemas).

Expressão	**Descrição**
exp1 **||** *exp2*	Dá o valor de *exp1* (sem que *exp2* seja avaliada), se *exp1* tiver o valor nulo ou um valor não nulo diferente de 0; dá o valor de *exp2* no caso contrário.
exp1 **&&** *exp2*	Dá o valor de *exp1* (sem que *exp2* seja avaliado) se *exp1* tiver o valor 0 ou nulo; dá o valor de *exp2* no caso contrário.
!*exp*	Dá o valor nulo, 0 ou 1 conforme o valor de *exp* seja nulo, diferente de 0 ou 0, respectivamente.
*exp1***?** *exp2* **:** *exp3*	Dá o valor nulo (sem que *exp2* ou *exp3* sejam avaliados), o valor de *exp2* (sem que *exp3* seja avaliado) ou o valor de *exp3* (sem que *exp2* seja avaliado), conforme o valor de *exp1* seja nulo, diferente de 0 ou 0, respectivamente.
if *exp1* **then** exp2 **else** *exp3*	Mesmo que anterior, a menos da precedência.
if *exp1* **then** exp2	Mesmo que **!***exp1* **||** *exp2*, a menos da precedência.

[3] O autor prefere a designação de "valor vazio" em vez de "valor nulo", mas a segunda designação já está consagrada. No interface para o utilizador o valor nulo é apresentado da mesma maneira do que a string vazia, pelo que é muito discutível a distinção interna entre valor nulo e string vazia.

O operador "**~**" (ler "semelhante a") é usado para testar se a string do lado esquerdo obedece ao padrão (com meta caracteres) do lado direito. O operador "**!~**" é a negação de "**~**". O operador "**..**" concatena strings.

O operador "**=**" é o operador de atribuição. Tal como em C, a expressão "*variável* **=** *expressão*" tem como efeito lateral a atribuição do valor da expressão do lado direito à variável do lado esquerdo, e dá como resultado o valor atribuído.

A expressão "*exp1*; *exp2*" (expressão de sequenciação) tem significado semelhante è expressão "*exp1*, *exp2*" em C. As expressões são avaliadas da esquerda para a direita e o resultado é o valor da última expressão avaliada.

A linguagem de regras e comandos partilha com o motor de regras as meta-classes *Class* e *Attribute* (ver secção 8.2). As variáveis da linguagem de regras e comandos correspondem aos atributos (globais ou intra-objecto) definidos na meta-classe *Attribute*. Uma vez que nem todas as classes e atributos devem ser controlados através do motor de regras, é introduzida a distinção entre classes e atributos *activos* e *passivos*. Só os primeiros interessam ao motor de regras.

Tanto as vistas da aplicação como as tabelas da base de dados são representadas por classes na meta-classe *Class*. Uma vez que podem existir vistas com o mesmo nome de tabelas da base de dados, os nomes de tabelas da base de dados são precedidos do caracter '**%**'. As tabelas da base de dados são consideradas passivas e as vistas são consideradas activas.

Podem-se usar variáveis auxiliares, sem tipo e sem declaração prévia, que são tratadas como atributos passivos globais de uma classe (excepto de uma tabela da base de dados).

Um atributo pode ser referenciado pelo nome do atributo (quando a classe é implícita) ou pelo nome da classe seguido do caracter "**.**" e do nome do atributo.

A seguinte construção especial (iterador) permite percorrer os objectos de uma ou mais classes (as partes opcionais são indicadas entre parêntesis rectos):

```
foreach lista-de-classes
[where condição]
[order by lista-de-atributos]
do expressão
```

As listas referidas são nomes separados por vírgulas. São seleccionadas as combinações de objectos das classes referidas na parte de "**foreach**" que obedecem à condição especificada na parte de "**where**", com a ordem indicada na parte de "**order by**". Para cada combinação de objectos seleccionada, é avaliada a expressão indicada na parte de "**do**". Essa expressão pode conter a instrução **break** para interromper o ciclo "**foreach"**. Nessa expressão, a referência, para leitura ou escrita, a um atributo intra-objecto de uma classe mencionada na parte de **foreach** refere-se ao objecto correntemente seleccionado. Esta construção é usada principalmente para aceder a tabelas da base de dados, caso em que é processada através de uma expressão de selecção em SQL e um cursor. Quando inserida dentro de chavetas (operador de construção de listas), retorna a lista de valores constituída pelos valores da expressão a seguir a "**do**".

Existe um conjunto de funções pré-definidas às quais se podem acrescentar outras funções através de um interface em C. As funções podem receber argumentos avaliados (constantes) ou por avaliar (expressões). Na realidade, todos os operadores e construções especiais são tratados internamente como funções.

Algumas funções utilitárias importantes são listadas no quadro seguinte.

Função	Significado
`undo([`*`mensagem`*`])`	Aborta a transacção corrente. O argumento opcional é uma mensagem de erro a mostrar ao utilizador. É usada nas regras de restrição.
`doinstead(`*`exp`*`)`	Avalia a expressão argumento e a seguir interrompe a transacção corrente (no sentido explicado no capítulo 8). É útil para usar em regras que redefinem operações.
`msg(`*`mensagem`*`)`	Afixa uma mensagem de alerta (string) no ecrã.
`input(`*`prompt`*`)`	Pede uma string ao utilizador, e retorna a string introduzida. Existem outras variantes mais sofisticadas. É útil para usar em regras que seguem um curso dependente da vontade do utilizador.
`system(`*`comando`*`)`	Executa um comando da *shell* do sistema operativo (Unix ou DOS).
`isnull(`*`exp`*`)`	Testa se o valor da expressão argumento é o valor nulo.

9.3.1.1 Funções de manipulação de vistas

As operações de manipulação de vistas referidas na secção 9.2.6 correspondem a funções da linguagem de regras e comandos.

As funções de agregação indicadas no quadro seguinte são úteis para definir campos de sumário.

Função	Significado
`rsum(`*`exp`*`[,`*`vista`*`])`	Soma o valor da expressão para todos os registos da vista, ignorando os registos em que dá valor nulo.
`rmin(`*`exp`*`[,`*`vista`*`])`	Obtém o valor máximo da expressão para todos os registos da vista, ignorando os registos em que dá valor nulo.
`rmax(`*`exp`*`[,`*`vista`*`])`	Obtém o valor mínimo da expressão para todos os registos da vista, ignorando os registos em que dá valor nulo.
`rcount(`*`exp`*`[,`*`vista`*`])`	Obtém o número de registos da vista em que a expressão não tem valor nulo.

A função `numrec([vista])` retorna o número de registos existentes numa vista.

A função `modo([vista])` retorna um caracter que designa o modo em que se encontra uma vista ou o registo corrente dessa vista, com o significado indicado no quadro seguinte:

Modo	Ler	Estado da vista	Estado do registo corrente
`'I'`	interrogar	Está em curso a edição do registo de critério.	
`'B'`	buscar	Está em curso a operação *load*.	
`'G'`	gravar	Está em curso a operação *save*.	
`'E'`	eliminar	Está em curso a edição de registos de dados.	eliminado (pendente)
`'M'`	modificar		seleccionado ou modificado

'A'	adicionar		inserido
''	vazio	Nenhum dos anteriores.	

Esta função é útil para usar na condição das regras, em alternativa (ou em conjugação) com a utilização da propriedade "modos".

9.3.1.2 Funções de manipulação da base de dados

A função `sql()` executa um comando SQL. Tem a seguinte sintaxe:

`sql(`*`string`* `[,` *`parâmetros`*`] [,` *`expressão`*`])`

O 1º argumento é uma string com uma expressão em SQL dinâmico embebido em C [R98]. A expressão pode ter parâmetros (também chamados variáveis) indicados com "`:`*`identificador`*" (em SQL-92); o nome usado é irrelevante, apenas a posição conta; os argumentos seguintes fornecem os valores para os parâmetros. As expressões com parâmetros são úteis porque são reutilizáveis. No caso de expressões começadas com "`select`", pode existir um argumento adicional com uma expressão que é avaliada para cada linha de resultado produzida pelo "`select`"; nessa expressão interessa normalmente usar uma função

`sqlget(`*`nº-de-coluna-do-resultado`*`)`

para ir buscar colunas do resultado (na linha corrente).

A função `dbacc(`*`nome-de-tabela, exp, ...`*`)` efectua o acesso por chave primária a uma tabela da base de dados indicada no 1º argumento, usando os valores indicados nos argumentos seguintes. No caso de existir um registo com a chave indicada, uma cópia do registo fica corrente em memória para leitura e a função retorna 1. Senão, retorna 0.

9.3.1.3 Eventos explícitos

Existe a seguinte construção especial para definir regras com eventos explícitos:

on *lista-de-eventos* **do** *expressão*

Os eventos da lista são separados por vírgulas. Os eventos são do tipo

nome-de-PPR

ou do tipo

`modify(`*`nome-de-atributo`*`)`

O nome de um PPR ou atributo doutra vista (diferente da vista a que a regra pertence), deve ser precedido do nome da vista e do separador "`.`".

A seguir ao nome de cada evento podem indicar-se as seguintes opções (separadas apenas por espaços), correspondentes a propriedades suportadas pelo motor de regras (ver secção 8.2):

`deferred immediate once repeatable self_triggering`

As opções **`deferred`** e **`immediate`** são modos de acoplamento mutuamente exclusivos. O modo de acoplamento por omissão no caso de ocorrências de PPR's é **`immediate`**. As opções **`once`** e **`repeatable`** (opção por omissão) são também mutuamente exclusivas.

9.3.2 Compilação das regras

Por compilação de uma regra entende-se a compilação da respectiva expressão para o formato interno, seguida da análise dessa expressão para determinar informação exigida pelo motor de regras, ou seja:

- o âmbito (granularidade) da regra, quando omisso;
- as listas de atributos acedidos para leitura e escrita, e as granularidades dos acessos;
- os eventos activadores da regra, quando omissos.

São determinados os atributos de vistas referidos explicitamente para leitura ou escrita, com base numa análise sintáctica e no conhecimento de algumas propriedades das funções, operadores e construções da linguagem de regras e comandos.

As regras que lêem conjuntos de registos, nomeadamente através de funções de agregação ou da construção `foreach`, dependem adicionalmente do atributo `record_set`. A modificação de `record_set` é sinalizada no fim de operações que inserem ou eliminam registos. As regras que inserem ou eliminam registos têm, adicionalmente, como atributo de saída, o atributo `record_set`.

Conforme foi referido, nas relações mestre-detalhe a vista de detalhe é materializada apenas para o registo corrente da vista mestre. Assim, as regras que relacionam registos de uma vista mestre com registos de uma vista de detalhe devem referenciar apenas um registo da vista mestre, que se considera implicitamente ser o registo corrente da vista mestre. Por essa razão, a regra fica a depender adicionalmente do atributo `record_current` da vista mestre, para ser executada quando muda o registo corrente da vista mestre (no sentido de ficar outro registo corrente e não no sentido de ser modificado o conteúdo do registo). Neste caso, o âmbito por omissão da regra é "vista".

Fora esse caso, o âmbito por omissão de uma regra que referencia atributos ou operações intra-registo relativamente a um registo não especificado (não instanciado) é sempre "registo".

Os eventos activadores são gerados automaticamente conforme foi explicado no capítulo 8.

9.3.3 Exemplos

Indicam-se de seguida alguns exemplos de aplicação típicos. Ver também os exemplos da figura 9.1.

1) Expressão geral de regra para definir um campo calculado incondicionalmente:

```
campo = expressão
```

No caso de um campo intra-registo, a regra é executada quando é criado um registo (por `load`, `query` ou `insrec`) ou é modificado o valor dum campo referenciado na expressão do lado direito. A regra não é executada se o valor do campo do lado esquerdo for alterado directamente pelo utilizador, mas existe uma propriedade estática de cada campo para impedir essas alterações.

Em muitas casos, não interessa calcular o campo no registo de critério, pelo que se define a propriedade "modos" da regra (ver figura 9.1) com "BM". Dessa forma, a regra só é desinibida (*enabled*) no decurso da operação `load` (modo "B") ou durante a edição de registos de dados (modo "M").

2) Expressão geral de regra para definir um campo calculado em certas condições e preenchido pelo utilizador noutras condições:

```
if (campo->locked = condição) then campo = expressão
```

que equivale a:

```
if condição
then (campo->locked = 1; campo = expressão)
else campo->locked = 0 /* para permitir utilizador preencher */
```

Neste caso a permissão de alteração varia dinamicamente.

3) Expressão de regra que obriga a preencher um campo (A) quando outro campo (B) está por preencher:

```
A->mandat = isnull(B)
```

4) Expressão de regra na sub-vista de linhas de factura da figura 7.1 para verificar se o código do artigo introduzido pelo utilizador existe na tabela de artigos e, em caso afirmativo, obter o respectivo nome (restrições R5 e R6 da figura 7.3) e preço unitário por omissão:

```
if isnull(codigo_artigo) then
  nome_artigo = ''
else if dbacc('artigo', codigo_artigo) then
(
  nome_artigo = %artigo.nome;
  preco_unitario = %artigo.preco_unitario
)
else
  undo('Não existe um artigo com o codigo ' .. codigo_artigo)
```

A propriedade "modos" da regra deve tomar o valor "M" (edição de registos de dados). Note-se que os nomes dos campos da base de dados são precedidos de "%". Trata-se de uma regra mista (de restrição e de derivação) que lê o campo `codigo_artigo` e actualiza os campos `nome_artigo` e `preco_unitario`. Note-se que a mensagem de erro produzida pela regra não é constante.

Note-se que as tabelas da base de dados são tratadas como passivas, pelo que a alteração de dados na base de dados não desencadeia a execução da regra. Normalmente, o programador não tem que definir regras deste tipo, pois são geradas automaticamente regras em C com o mesmo efeito e com controlo de "locks" (conforme se explica na secção seguinte).

5) Expressão geral de regra para definir um valor por omissão para um campo nos registos inseridos:

```
if modo()~'A' then campo = expressão
```

Adicionalmente, pode-se definir a propriedade "modos" da regra com o valor "M" (edição de registos de dados).

O efeito é o mesmo que:

```
on after_add do campo = expressão
```

6) Expressão geral de regra para definir um condição de selecção para um dado campo:

```
campo = expressão /* string com condição de selecção,
                     como por exemplo '>0', '1-10', 'v*', etc.
*/
```

com "I" na propriedade "modos" da regra.

O efeito é o mesmo que

```
on after_query do campo = expressão
```

mas só no caso da expressão não depender doutros campos.

7) Expressão geral de regra (intra-registo) para impedir a selecção dos registos que não satisfazem a uma dada condição:

```
if condição then undo()
```

com "B" na propriedade "modos" da regra.

O efeito é o mesmo que:

```
if after_load_record && condição then undo()
```

Estas regras são processadas em `after_load_record`, numa sub-transacção intra-registo com "rollback" local. A falha da sub-transacção (invocada com `undo`) é tratada por `load` como uma recusa da selecção desse registo, e não doutros registos. Este tipo de selecção *a posteriori* é de evitar porque é ineficiente. Sempre que possível, é preferível definir condições de selecção como no exemplo anterior. No futuro, espera-se poder gerar automaticamente condições de selecção de regras deste tipo.

8) Expressão de regra numa vista `v1` para calcular uma coluna de acumuladas (`ValAcum`) a partir de uma coluna da valores simples (`Valor`) numa vista só para consulta:

```
on after_load
do (aux=0; foreach v1 do (ValAcum = aux += ^Valor))
```

Notar a utilização de "`^`" para promover o valor nulo para 0, do operador "`+=`" com o mesmo significado que em C e da variável auxiliar "`aux`". "`foreach`" percorre os registos da vista pela ordem por que são apresentados ao utilizador.

9) Expressão geral de regra para impedir a eliminação de registos que obedecem a uma determinada condição:

```
on before_delete
do if condição
   then undo('Nao pode eliminar este registo porque ... !')
```

10) Expressão de regra para impedir a gravação de alterações na base de dados enquanto se verifica uma determinada condição:

```
on before_save
do if condição
   then undo('Não pode gravar alterações porque .... !')
```

Regras executadas em `before_save` são normalmente usadas para efectuar validações que não podem ser feitas incrementalmente.

11) Expressão de regra para completar a gravação de uma linha de factura com a actualização do último preço do artigo na base de dados:

```
on after_save_record
do sql('update artigo set ultimo_preço = :a where codigo = :b',
       código, preço_unitário)
```

Regras executadas em `after_save_record` ou `after_save` são normalmente usadas para completar a gravação de alterações na base de dados (em relação ao que o SAGA já faz automaticamente a partir da especificação de propriedades de mais alto nível referidas na secção 9.2.2).

12) Expressões de duas regras para manter uma segunda vista aberta (`v2`) enquanto a vista a que a regra pertence está aberta:

```
on after_open do open(´v2´)
on before_close do close(´v2´)
```

13) Expressão geral de regra (chamada *regra de agregação*) para definir um campo de sumário:

campo = *função-de-agregação*(*expressão* [, *nome-de-vista*])

A função de agregação pode ser uma das indicadas na secção 9.3.1.1 (`rsum`, `rmax`, `rmin` ou `rcount`). No caso do nome da vista ser omitido ou ser o nome da vista a que a regra pertence, a agregação é implicitamente efectuada por conjuntos de registos do nível do campo do lado esquerdo (o conceito de nível é explicado na secção 9.2.4) através de regras automáticas indicadas na secção 9.4.3.

9.4 Regras geradas automaticamente

São geradas automaticamente algumas regras para a manutenção incremental das vistas. A parte de acção destas regras é realizada por funções em C codificadas na ferramenta, por razões de eficiência. As regras geradas automaticamente têm uma prioridade numérica mais alta do que a das regras definidas pelo programador através da linguagem de regras e comandos.

9.4.1 Regra de reinicialização

Para cada vista com parâmetros ligados ("bound") a campos de vistas ascendentes (conforme explicado na secção 9.2.3), é criada uma regra de reinicialização da vista, que é activada quando é modificado o valor de um desses campos ou o identificador do registo corrente (atributo `record_current`) de uma dessas vistas.

9.4.2 Regras de acesso por chave a tabelas da base de dados

Para cada tabela da base de dados ou cópia lógica de tabela acedida para leitura (tabela alvo e tabelas de extensão referidas na secção 9.2.2) e/ou acedida para verificar a integridade referencial é criada uma regra de acesso pela chave primária que efectua a leitura e/ou testa a integridade referencial e trata de "locks".

São também criadas regras de acesso a tabelas por chaves alternativas quando existem campos da vista que têm como alvo campos de chave alternativa.

Estas regras interessam essencialmente para dar um "feedback" imediato ao utilizador durante o modo de edição de registos de dados.

No modo de busca da base de dados (*load*), é gerada uma expressão de selecção abrangendo o maior número possível de tabelas da base de dados, e são inibidas as regras de acesso por chave correspondentes a essas tabelas. Todas as regras de acesso por chaves alternativas e de teste de integridade referencial são inibidas durante *load*.

9.4.3 Regras de agregação

As regras de agregação definidas pelo programador que são totalmente internas a uma vista são substituídas por uma única regra de agregação total e, opcionalmente, uma regra de agregação incremental.

Numa única passagem, a regra de agregação total calcula os valores de todos os campos de sumário. Além disso, quando executada em `after_load`, verifica as restrições de selecção relativas aos campos de sumário, removendo os grupos de registos que não obedecem às restrições.

Opcionalmente, os dados de uma vista podem ser armazenados de uma forma hierárquica, de acordo com a divisão de campos e registos por níveis explicada na secção 9.2.4. Nesse caso, a ordenação dos registos é efectuada incrementalmente durante o modo de edição de dados, e é criada uma regra de agregação incremental que substitui a regra de agregação total durante o modo de edição de dados. A regra de agregação incremental é activada em modo imediato quando é inserido, eliminado ou modificado um registo. No último caso, a regra é activada apenas quando são modificados os campos envolvidos nas expressões-argumento das funções de agregação ou campos de ordenação. O registo modificado pode mudar de grupo de registos de um ou mais níveis, pelo que pode ser necessário recalcular os campos de sumário nos grupos antigos de registos e nos novos grupos de registos. Actualmente, apenas é possível efectuar a manutenção incremental das funções *rcount* e *rsum*. A manutenção incremental das funções *rmax* e *rmin* poderá ser efectuada no futuro com o recurso a índices de ordenação auxiliares.

9.5 Experiência de utilização e linhas de evolução

Existe alguma experiência acumulada na utilização da ferramenta SAGA para o desenvolvimento de aplicações comerciais com alguma complexidade (com centenas de vistas e milhares de regras por aplicação) para a administração pública em Portugal [FLMB95], embora usando versões anteriores que não incorporam muitas das novas facilidades de definição e processamento de regras descritas neste capítulo e no anterior.

A experiência tem revelado que se podem construir rapidamente com o SAGA aplicações flexíveis, eficientes e, sobretudo, de fácil manutenção, após um período de aprendizagem razoável. A rapidez de desenvolvimento está estreitamente ligada com a variedade de automatismos disponíveis, tanto ao nível da gestão de vistas como ao nível da gestão de regras, importando salientar aqui, como aspecto mais inovador, os automatismos inerentes à utilização de regras activas dirigidas pelos dados com eventos e prioridades implícitos.

A experiência tem revelado também alguns aspectos a melhorar, de que se destacam os que têm mais a ver com o subsistema de regras. A principal dificuldade encontrada tem a ver com o comportamento complexo das regras em algumas situações. Em situações mais complexas, a abundância de automatismos e aspectos implícitos é contraproducente, e é necessário dar mais controlo ao programador. Algumas das soluções descritas neste capítulo e no anterior (como é o caso da explicitação de prioridades relativas e a explicitação mais clara dos eventos activadores), vão nesse sentido. Importa também clarificar e simplificar alguns aspectos do modelo de execução do SAGA. Algumas limitações referidas deviam ser levantadas (admitir várias instâncias do mesmo tipo de vista e sub-vistas materializadas de forma diferente) e deviam ser uniformizados outros aspectos, como é o caso do tratamento diferente de vistas com secções hierárquicas e vistas com sub-vistas. Faltam ainda ferramentas de "design", análise estática e análise dinâmica das regras, cuja necessidade é largamente reconhecida em todos os sistemas de regras activas (ver por exemplo [PD99]). Os instrumentos de análise e resultados dos capítulos 4 a 7 teriam aí lugar importante, com as devidas extensões. Um aspecto que coloca desafios interessantes para o futuro próximo tem a ver com a geração de interfaces "finos" para a Web incorporando algumas regras activas a partir das especificações de vistas do SAGA.

10 Conclusões

10.1 Resultados alcançados

O modelo proposto de regras activas dirigidas pelos dados para a manutenção de restrições de integridade e dados derivados, com eventos activadores e prioridades inferidos pelo sistema, combina:

- flexibilidade, declaratividade e modularidade ao nível da definição das regras para a imposição de restrição de integridade e o cálculo de dados derivados;
- eficiência ao nível da execução das regras, garantindo a imposição eficiente das restrições de integridade e o cálculo eficiente dos dados derivados;
- boa integração com regras activas de mais baixo nível (dirigidas por eventos) especificadas directamente pelo utilizador ou geradas automaticamente a partir de especificações de alto nível (como na secção 9.4);
- facilidade de implementação, porque os eventos activadores e prioridades se baseiam numa mera análise sintáctica das regras e, opcionalmente, em propriedades de alto nível indicadas pelo utilizador (como a propriedade de idempotência).

Foram identificadas condições a que um conjunto de regras activas dirigidas pelos dados deve obedecer para garantir a terminação e o determinismo do processamento de regras. Foram identificadas condições mais conservadoras, cuja verificação exige apenas uma análise sintáctica das regras (sendo, portanto, fácil de automatizar), e condições menos conservadoras, cuja verificação exige também uma análise semântica das regras. Conforme se previa, a natureza especializada das regras activas dirigidas pelos dados permitiu obter condições de terminação e determinismo menos conservadoras do que as que são conhecidas para regras activas genéricas.

Foi estabelecida uma relação entre a sintaxe e a semântica de uma regra activa dirigida pelos dados (na secção 4.1.4), através da noção de variáveis de entrada e saída, que permitiu demonstrar a correcção dos critérios de obtenção de eventos activadores e prioridades e das condições de terminação e de determinismo que se baseiam apenas em informação de natureza sintáctica.

Foi analisada a combinação com o modelo de regras proposto de diversas estratégias conhecidas para a manutenção eficiente de restrições de integridade e dados derivados em conjuntos de dados complexos, com destaque para a diferenciação de regras orientadas a conjuntos e o encapsulamento de regras em objectos. O encapsulamento de regras em objectos é, na opinião do autor, a estratégia chave para lidar eficazmente com dados e sistemas complexos, porque num sistema complexo não é viável considerar um único conjunto global de regras. A manutenção de restrições e derivações locais a um objecto com atributos atómicos, através de regras dirigidas pelos dados encapsuladas no objecto (regras dirigidas pelos dados intra-objecto), não exige qualquer refinamento do modelo de regras proposto (no modelo de regras proposto inicialmente os dados são representados abstractamente por variáveis de estado cujo estrutura interna é ignorada; no caso de regras intra-objecto, as variáveis de estado são os atributos do objecto). Em contrapartida, para a manutenção eficiente de restrições e derivações que envolvem vários objectos (inter-objecto), são necessários alguns refinamentos, alguns dos quais de difícil implementação, pelo que as regras activas com eventos explícitos terão aí um papel tendencialmente mais importante.

Foi apresentada uma implementação concreta de um sistema de regras activas integrado numa ferramenta de desenvolvimento de aplicações de bases de dados que, apesar de algumas limitações da implementação (até por razões de compatibilidade com versões anteriores da ferramenta), demonstra a aplicabilidade da abordagem proposta no contexto pretendido.

10.2 Sugestões para trabalho futuro

Apresenta-se de seguida uma pequena lista, de modo algum exaustiva, de problemas para os quais as soluções encontradas neste trabalho não são plenamente satisfatórias.

As garantias de terminação e determinismo do processamento de regras são, na prática, extremamente importantes para o desenvolvimento de aplicações confiáveis. Vimos que a simples presença de regras condicionais num conjunto de regras não ordenado (sem prioridades) pode ser suficiente para causar o não determinismo. Felizmente, o não determinismo causado por regras condicionais é afastado através da ordenação (prioritização) das regras pelo princípio "calcular antes de usar", que parece ser de aplicação inquestionável por razões de semântica e eficiência. Afastado este tipo de não determinismo, vimos que a não terminação poderia ser causada por regras conflituosas ou por regras recursivas, enquanto que o não determinismo podia ser causado apenas por regras recursivas. Para resolver estes problemas, é importante no futuro: i) refinar as condições de terminação e determinismo relativamente a regras deste tipo; ii) automatizar a verificação dessas condições; iii) estabelecer claramente as medidas que o utilizador deve tomar para resolver as situações de potencial não terminação ou não determinismo (prioridades, rescrita de regras, etc.).

Por outro lado, na análise de terminação e determinismo do processamento de regras, não se entrou em conta com a possibilidade de as regras activas dirigidas pelos dados serem executadas em conjunto com outras regras activas. Esta é, obviamente, uma lacuna que é importante resolver no futuro. Provavelmente, o que há a fazer é combinar da melhor maneira as condições menos conservadoras obtidas para o caso específico de regras activas dirigidas pelos dados, com as condições mais conservadoras conhecidas para regras activas genéricas.

No modelo de execução proposto, sempre que é invocado o processamento de regras, todos os itens de dados derivados que possam estar desactualizados são actualizados, antes de se aceitarem novos comandos do utilizador. Trata-se de um modo de processamento que se justifica na presença de restrições de integridade nos dados derivados. No entanto, no caso de itens de dados derivados só para consulta que não estão sujeitos a restrições de integridade, deveria ser considerado um modo de processamento mais "preguiçoso" ("lazy"), como descrito por exemplo em [H91], à medida das necessidades, e possivelmente em paralelo com os comandos do utilizador.

No modelo de regras proposto, todas as regras têm de ser satisfeitas (isto é, as regras são processadas até ser atingido um ponto fixo para todas as regras). Em alternativa podia-se admitir que algumas regras mais "fracas" (regras por omissão, regras gerais, etc.) não fossem satisfeitas quando entram em conflito com outras regras mais "fortes" ou até com os comandos do utilizador. As regras fracas poderiam ser usadas, por exemplo, para especificar valores por omissão de uma forma mais flexível. Critérios que poderiam ser usados para automaticamente atribuir forças a regras conflituosas são descritos em [IS89].

O problema da manutenção eficiente e automática de restrições e derivações inter-objecto foi abordado apenas de forma superficial, e foram apresentadas apenas soluções limitadas. Em particular, a forma como as regras são encapsuladas em objectos precisa de ser definida com maior precisão.

11 Referências

[AHU74] Alfred V. Aho, J. E. Hopcrotf e Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.

[AHW95] Alexander Aiken, Joseph Hellerstein e Jennifer Widom. Static Analysis Techniques for Predicting the Behavior of Active Database Rules. *ACM Transactions on Database Systems*, vol. 20, nº 1, pp. 3-41, Março de 1995.

[AU92] Alfred V. Aho e Jeffrey D. Ullman. *Foundations of Computer Science*. Computer Science Press, New York, 1992.

[AWH92] Alexander Aiken, Jennifer Widom e Joseph Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 59-68, California, USA, Junho de 1992.

[BCP95] Elena Baralis, Stefano Ceri e Stefano Paraboschi. Improved rule analysis by means of triggering and activation graphs. In *Proceedings Rules in Database Systems - Second International Workshop – RIDS'95*, pp. 165-181, Atenas, Grécia, Setembro de 1995.

[BW95] Elena Baralis e Jennifer Widom. Using Delta Relations to Optimize Condition Evaluation in Active Databases. In *Proceedings Rules in Database Systems - Second International Workshop – RIDS'95*, pp. 292-308, Atenas, Grécia, Setembro de 1995.

[C98] Christine Collet. NAOS. In *Active Rules in Database Systems*, N. Paton (editor), pp. 279-296, Springer-Verlag, 1998.

[CC95] Thierry Coupaye e Christine Collet. Denotational Semantics for an Active Rule Execution Model. In *Proceedings Rules in Database Systems - Second International Workshop – RIDS'95*, pp. 36-50, Atenas, Grécia, Setembro de 1995.

[CFPB96] Stefano Ceri, Pietro Fraternali, Stefano Paraboschi e Leticia Branca. Active Rule Management in Chimera. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 151-176, Morgan Kaufmann, 1996.

[CW90] Stefano Ceri e Jennifer Widom. Deriving Production Rules for Constraint Maintenance. In *Proceedings of the 16th VLDB Conference*, pp. 566-577, Brisbane, Austrália, Agosto de 1990.

[CW91] Stefano Ceri e Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proceedings of the 17th VLDB Conference*, pp. 577-589, Barcelona, Espanha, Setembro de 1991.

[CW94] Stefano Ceri e Jennifer Widom. Deriving Production Rules for Deductive Data. In *Information Systems*, vol. 19, nº 6, pp. 467-490, Elsevier Science Ltd, 1994.

[CW96] Stefano Ceri e Jennifer Widom. Standards and Commercial Systems. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 232-258, Morgan Kaufmann, 1996.

[D88] Umeshwar Dayal. Active Database Management Systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, Israel, Junho de 1988.

[D95] C. J. Date. *An Introduction to Database Systems*. 6ª edição, Addison-Wesley, 1995.

[DBC96] Umeshwar Dayal, Alejandro Buchmann e Sharma Chakravarthy. The HiPAC Project. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 175-206, Morgan Kaufmann, 1996.

[DGG95] Klaus Dittrich, Stella Gatziu e Andreas Geppert. The Active Database Management System Manifesto: A Rulebase of ADBMS Features. In *Proceedings of the Second International Workshop on Rules in Database Systems*, pp. 3-17, Atenas, Grécia, Setembro de 1995.

[E93] Opher Eztion. PARDES - A Data-Driven Oriented Active Database Model. In *SIGMOD RECORD*, vol. 22, nº 1, pp. 7-14, Março de 1993.

[EGS93] Opher Eztion, Avigdor Gal e Arie Segev. Data Driven and Temporal Rules in PARDES. In *Proceedings of the First International Workshop on Rules in Database Systems*, pp. 92-108, Edimburgo, Escócia, Agosto de 1993.

[F90] João P. Faria. *SAGA - Uma Ferramente Interactiva para a Geração e Gestão de Aplicações baseadas em Vistas e Regras*. Trabalho de síntese elaborado para as provas de aptidão pedagógica e capacidade científica. FEUP, Outubro de 1990.

[F93a] João P. Faria. *SAGA 1.15 - Manual do Utilizador*, INESC-Porto, Maio de 1993.

[F93b] João P. Faria. *SAGA 1.15 - Manual do Programador: Parte I - Dicionário de vistas*. INESC-Porto, Junho de 1993.

[F93c] João P. Faria. *SAGA 1.15 - Manual do Programador: Parte II - Linguagem de Regras e Comandos*. INESC-Porto, Julho de 1993.

[FLMB95] João P. Faria, Mário J. Leitão, José M. Moreira e António V. Bouça. Aspectos Tecnológicos do SIGMA e do Projecto de Informatização Municipal. In *Informação e Informática*, Revista do Instituto de Informática do Ministério das Finanças, 1995.

[FR91] João P. Faria, João V. Ranito. SAGA - Uma Ferramenta Interactiva para o Desenvolvimento Expedito e Fácil Manutenção de Aplicações de Bases de Dados. In *ENDIEL'91, ST2-Indústrias do Software e da informação, Proceedings*, pp. 33-40, Lisboa, Portugal, Junho de 1991.

[FV99] João P. Faria e Raul M. Vidal. Data-driven Active Rules for the Maintenance of Derived Data and Integrity Constraints in User Interfaces to Databases. Aceite para publicação nos *Proc. XIV Simposium Brasileiro de Bases de Dados - SBBD'99*, Florianopolis, Santa Catarina, Brazil, Outubro de 1999.

[G95] Avigdor Gal. *TALE - A Temporal Active Language and Execution Model.* Research Thesis. Technion - Israel Institute of Technology, Haifa, Israel, Maio de 1995.

[GE95] Avigdor Gal and Opher Eztion. Maintaining Data-driven Rules in Databases. In *IEEE Computer*, 28(1):28-38, Janeiro de 1995.

[GJ91] Narain Gehani, H.V. Jagadish. Ode as an Active Database: Constraints and Triggers. *In Proceedings of the 17th International Conference on Very large Data Bases*, pp. 327-336, Barcelona, Setembro de 1991.

[GJ96] Narain Gehani, H.V. Jagadish. Active Database Facilities in Ode. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 207-232, Morgan Kaufmann, 1996.

[GR93] Jim Gray e Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[H80] Gérard Huet. Confluent Reductions: Abstract Porperties and Applications to Term Rewriting Systems. In *Journal of the ACM*, vol. 27, nº 4, pp. 797-821, Outubro de 1980.

[H91] Scott E. Hudson. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. In *ACM Transactions on Programming Languages and Systems*, vol. 13, nº 3, pp. 315-341, Julho de 1991.

[H92] Eric N. Hanson. Rule condition testing and action execution in Ariel. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 49-58, San Diego, Califórnia, Junho de 1992.

[H96] Eric N. Hanson. The Ariel Project. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 63-86, Morgan Kaufmann, 1996.

[IS89] Yannis E. Ioannidis, Timos K. Sellis. *Conflict Resolution of rules assigning values to virtual attributes.* In *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 205-214, 1989.

[K97] Donald E. Knuth. *The Art of Computing Programming, Volume 1: Fundamental Algorithms*. Third Edition. Addison-Wesley, 1997.

[KMC98] Krishna Kulkarni, Nelson Mattos e Roberta Cochrane. Active Database Features in SQL3. In *Active Rules in Database Systems*, N. Paton (editor), pp. 197-219, Springer-Verlag, 1998.

[KR88] Brian W. Kernighan e Dennis M. Ritchie. *The C Programming Language*. Second edition. Prentice-Hall, 1988.

[M97] Robert J. Muller. *Oracle Developer/2000 Handbook*. Second Edition. Osborne/McGraw-Hill, 1997.

[MFR89] Vladimiro Miranda, João P. Faria e João V. Ranito. *Alguns Parâmetros de Qualidade no Desenvolvimento e Utilização de Aplicações Informáticas*. Encontro da Associação Portuguesa de Qualidade, Dezembro de 1989.

[MLF+89] Vladimiro Miranda, João C. Lopes, João P. Faria, João C. Ferreira, Eduardo Silva, João V. Ranito e Jorge Serro. SIGMA - Gestão Municipal em UNIX/C e Bases de Dados em Implantação na Região Norte. In *ENDIEL'89*, Porto, 1989.

[OF93] *Oracle Forms Ô Reference Manual - Version 4.0, Volume 1.* Oracle Corporation, 1993.

[P97] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* Fourth Edition. McGraw-Hill, 1997.

[PD99] Norman Paton e Oscar Diaz. Active Database Systems. To be published in *ACM Computing Surveys* (url: http://www.cs.man.ac.uk/~norm/papers/surveys.ps).

[PS96] Spyros Potamianos e Michael Stonebreaker. The POSTGRES Rule System. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 43-61, Morgan Kaufmann, 1996.

[R91] J. Rumbaugh *et al. Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[R98] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.

[RNN77] E. M. Reingold, J. Nieverge e D. Narsingh. *Combinatorial Algorithms: Theory and Practice*. Prentice-Hall, 1977.

[RS98] Tore Risch e Martin Sköld. Monitoring Complex Rule Conditions. In *Active Rules in Database Systems*, N. Paton (editor), pp. 81-102, Springer-Verlag, 1998.

[RSS90] R. Ramakrishnan, D. Srivastava e S. Sudarshan. Rule ordering in Bottom-up Fixpoint Evaluation of Logic Programs. In *Proceedings of the 16th VLDB Conference*, pp. 359-371, Brisbane, Austrália, 1990.

[S97] Bjarne Stroustrup. *The C++ Programming Language*. Third Edition. Addison-Wesley, 1997.

[SK96] Eric Simon e Jerry Keirnan. The A-RDL System. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 111-149, Morgan Kaufmann, 1996.

[SM96] Michael Stonebreaker e Dorothy Moore. *Object-Relational DBMSs: The next great wave*. Morgan Kaufmann Publishers, Inc., 1996.

[SMBB93] Michael Sannela, John Maloney, Bjorn Freeman-Benson e Alan Borning. Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm. In *Software Practice and Experience*, vol. 23(5), pp. 529-566, Maio de 1993.

[TS92] K. Thulasiraman e M. N. S. Swamy. *Graphs: Theory and Algorithms*. John Wiley & Sons, 1992.

[U88] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*. Volumes I e II. Computer Science Press, Inc., USA, 1988.

[UML97] *UML Notation Guide - Version 1.1*. Rational Software e outras empresas, Setembro de 1997.

[VMF93] Raul M. Vidal, José M. Moreira, João P. Faria. Implementação de Novas Tecnologias da Informação na Administração Local e Regional. PIM (Projecto de Informatização Municipal): Um exemplo de aplicação. In *ENDIEL'93*, Porto, 1993.

[W96] Jennifer Widom. The Starbust Rule System. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 87-109, Morgan Kaufmann, 1996.

[WC96] Jennifer Widom e Stefano Ceri (editores). *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, 1996.

[WC96a] Jennifer Widom e Stefano Ceri. Introduction to Active Database Systems. In *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom e S. Ceri (editores), pp. 1-41, Morgan Kaufmann, 1996.

[WF90] Jennifer Widom e Sheldon Finkelstein. Set-Oriented Production Rules in Relational Database Systems. *In Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pp. 259-270, 1990.

[ZCF+97] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V.S. Subrahmanian e Roberto Zicari. *Advanced Database Systems*. Morgan Kaufmann, 1997.

ANEXO 1

"An algorithm to find feedback edge sets with one edge per cycle"

An algorithm to find feedback edge sets with one edge per cycle

João Carlos Pascoal Faria

Abstract: An algorithm is presented to check if a directed graph has a feedback edge set with only one edge per cycle and, if so, produce it, without actually computing the cycles. The topological order of the vertices of the reduced graph without the feedback edges is called a cycle-preserving vertex order (CPVO). CPVO's are important for efficient bottom-up fixpoint evaluation of logic programs [1]. It is shown that the worst case running time of the algorithm is of the same order as the time needed to find all the cycles. Several conditions are identified under which expensive steps of the algorithm can be avoided, leading to a polynomial execution time in many cases, including the case of reducible flow graphs. Results of experiments with randomly generated graphs are presented, showing an average running time of the algorithm approximately linear in the size of the graph.

1. Introduction

We first recall some basic graph concepts and notation, in order to clearly state the problem being addressed in the paper.

A directed graph G is a pair (V,E) of a set V of vertices and a set E of directed edges (also called arcs). The number of vertices is denoted by $|V|$ and the number of edges is denoted by $|E|$.

Throughout this paper, we assume *simple* directed graphs, that is, directed graphs without self-loops and parallel edges. However, the results presented might easily be extended to other graphs. All the graphs, edges, paths and cycles are directed. It is also assumed that all the vertices of a path are distinct (following the definition of path found in [2]) and that all except the end vertices of a cycle are distinct.

A *strongly connected component* (*SCC* for short) H of a graph $G=(V,E)$ is a maximal sub-graph such that, for every distinct vertices v and w in H, there are paths from v to w and from w to v in G (and, consequently, in H). A SCC with a single vertex is called *trivial.*

A *linear ordering* (or simply an ordering) L of the vertices of a graph $G=(V,E)$ is as an assignment of distinct consecutive order numbers 1, 2, ..., $|V|$ to its vertices. It may be represented by a sequence of the vertices. A *topological ordering* of the vertices of an acyclic graph is an ordering such that all the edges are directed from a vertex with a lower order number to a vertex with a higher order number.

A *feedback edge set* (*FES* for short) of a graph $G=(V,E)$ is a subset $F \acute{I} E$ of edges such that every cycle of G contains an edge in F or, equivalently, such that the graph $G'=(V,E-F)$ is acyclic. Usually, one is interested in *minimal* FES's, that is, FES's that do not contain other FES's with fewer elements. The problem of testing if a graph has a FES with k elements is a well-known NP-complete problem [3][4].

Vertex orderings and feedback edge sets are related. Every ordering L of the vertices of a graph G has a corresponding (or induced) FES F constituted by the edges of G directed from vertices with higher order numbers to vertices with lower order numbers under L. Given a FES F, let G' be the graph obtained from G by reversing the orientation of the edges contained in F, and let G'' be the graph obtained from G by removing the edges contained in F. If G' has cycles, then F has no corresponding vertex orderings. If G' has no cycles (which is always true when F is minimal), then F has one or more corresponding vertex orderings given by the topological orderings of G' or G''.

According to [1], an ordering *L* of the vertices of a graph *G* is said to *break* a cycle C by degree *k* (*k ³1*), if *C* has *k* edges directed from vertices with higher order numbers to vertices with lower order numbers under *L*. A vertex ordering that breaks a cycle by degree 1 is said to *preserve* that cycle. A vertex ordering that preserves all the cycles is called a *cycle-preserving vertex ordering* (*CPVO* for short). CPVO's are important for efficient bottom-up fixpoint evaluation of logic programs [1].

A CPVO has a corresponding *feedback edge set with one edge per cycle F*, also called here a *cycle-preserving feedback edge set (CPFES* for short*)*, in the sense that every cycle of *G* has exactly one edge in *F*.

Notice that a CPFES is minimal, unless it has edges that are not contained in cycles. Usually, one is interested in minimal CPFES's, and in the corresponding CPVO's.

Also notice that a CPVO may not exist and may not be unique. For example, graph G_1 in Fig. 1 has a CPVO (e.g. 1-2-3-4), while graph G_4 has not.

The rest of this paper addresses the problem, left open in [1], of finding an efficient algorithm to solve the following problem:

Problem 1: Given a directed graph *G*=(*V*,*E*), check if it has a CPFES *F* (or, equivalently, a CPVO *L*), and, if so, produce it.

2. Basic algorithm

In this section we introduce a basic iterative algorithm to solve problem 1. Since cycles are contained in non-trivial SCC's, the idea is to cut a minimal set of edges from each non-trivial SCC, excluding edges belonging to cycles already "cut", in order to be able to decompose it, and then to proceed in the same way with the resulting sub-SCC's. More precisely, it is proposed the following algorithm:

Algorithm 1 *(construction of a minimal CPFES F for a graph G=(V,E))*:

1. Initialize *F*, the set of feedback edges, with the empty set.
2. Initialize *W*, the working set of the non-trivial SCC's to decompose, with the non-trivial SCC's of *G*.
3. If *W* is empty, stop with success (*F* is a CPFES).
4. Select and remove from *W* an arbitrary element *H* (a non-trivial SCC).
5. Find a vertex *v* in *H* whose incoming or outgoing edges in *H* are not traversed by cycles of *G* that are not fully contained in *H*, and move from *H* to *F* the incoming or outgoing edges of *v*, respectively. If such a vertex does not exist, stop and fail (*G* has no CPFES).
6. Insert into *W* the non-trivial (sub) SCC's of *H* without the edges moved to *F*.
7. Go to step 3. Δ

The decomposition of each non-trivial SCC is done in steps 5 and 6. Step 5 is sufficient to break all the cycles that cross the selected vertex and to disconnect it (in the strong sense) from the other vertices of the same SCC. In terms of the corresponding vertex ordering, this has the effect of pushing the selected vertex into either the first or the last position of the topological ordering of the resulting sub-SCC's.

A few definitions follow. The cycles of *G* that are not fully contained in a sub-graph *H* of *G* are called *external cycles* (with respect to *H*). Any vertex with the property mentioned in step 5 (whose

incoming or outgoing edges in H are not traversed by external cycles) is called a *critical vertex* of H, and the selected vertex (v) is called the *decomposing-vertex* of H. The initial SCC's are also called *top level* SCC's and the sub-SCC's are also called *nested* SCC's. The *level* of nesting is quantified as follows. Level 1 is assigned to the top-level SCC's. The decomposition of a non-trivial SCC of level k originates nested SCC's of level k+1. For convenience, level 0 is assigned to the entire graph. With respect to a decomposition process, the *level of a vertex, edge, cycle or sub-graph* is the level of the smallest nested SCC, top level SCC or entire graph that contains it.

Theorem 1: Algorithm 1 has the following properties:

i) it always terminates;

ii) if it succeeds then F is a minimal CPFES (*correctness*);

iii) any minimal CPFES F may be generated by it (*completeness*);

iv) if it fails then G has no CPFES (*irrevocability*).

Proof:

i) We assume each step terminates, so we only have to prove that the number of iterations of steps 3 to 7 is finite. In each iteration of step 6, at least one vertex - the decomposing-vertex - is excluded from further consideration, because it becomes a trivial sub-SCC. So, after at most $|V|$ iterations of steps 3 to 7, the algorithm must terminate.

ii) If algorithm 1 succeeds, then F is a CPFES, because the criteria followed in step 5 guarantees that all the cycles are broken and a cycle is never broken twice. Since all the edges in F belong to cycles of G and it has only one edge per cycle, it is minimal.

iii) Step 5 must be done for some vertex v in the process of reducing H to an acyclic graph, because an acyclic graph must have vertices with zero in-degree and zero out-degree. However, not all the (unconstrained) choices of v would ensure minimal FES's. The constraint imposed only eliminates those FES's that have more than one edge per cycle. It also guarantees that F is minimal as noted in ii).

iv) See Appendix A. Δ

Since any failure in step 5 is irrevocable, it is not necessary to backtrack to a previous choice-point (a previous iteration of step 5) to try an alternative (select a different critical vertex). Apart from the non-trivial step 5, the time complexity is given by:

Theorem 2 (*optimistic complexity*): Algorithm 1 may be run in time $O[|V|\cdot(|V|+|E|)]$ if it is possible to execute step 5 in time not exceeding $O(|V|+|E|)$.

Proof: In step 2, the initial SCC's may be obtained in time $O(|V|+|E|)$ (see [5] for example). In addition, in step 6, each non-trivial SCC H may be decomposed in time not exceeding $O(|V|+|E|)$ (and the same in average). The maximum number of non-trivial (top-level and nested) SCC's is $|V|$-1. Multiplying and summing we get $O[|V|\cdot(|V|+|E|)]$. Δ

3. Finding the edges traversed by external cycles

In this section we refine step 5 of algorithm 1 in two different ways.

First method

In order to apply step 5 in algorithm 1, it is necessary to determine the edges of each SCC that are traversed by external cycles. This may be done by refining step 5 in the following way:

Algorithm 1a (*with the exploration of internal cycles*):

(... previous steps as in algorithm 1)

5.1. Select from H a vertex v whose incoming or outgoing edges in H have not been marked. If such a vertex does not exist, stop and fail.

5.2. Mark the edges of the cycles of H that cross v.

5.3. Move from H to F the incoming or outgoing edges of v, respectively.

(... the rest as in algorithm 1) Δ

It's assumed that all the edges are initially unmarked.

To solve step 5.2, we may adapt an algorithm to find all the cycles of a graph that cross a given vertex, like the one described in [5], that takes a time $O(|V|+|E|)$ between consecutive cycles. Instead of listing the cycles, we only have to mark their edges. Although this procedure is excessive, as we shall see, it allows us to derive the following result:

Theorem 3 (*worst-case complexity*): Problem 1 may be solved in time $O[(|V|+|E|)(c+1)]$, where c is the number of cycles in G.

Proof: If it does not fail, algorithm 1a will find all the cycles of G. Using the above mentioned algorithm, the time spent between consecutive cycles in the same iteration of step 5.2 is $O(|V|+|E|)$, and there is always at least one cycle. The time spent between consecutive iterations of step 5.2, before the first iteration and after the last iteration is of the same order. Consequently, the overall time is $O[(|V|+|E|)(c+1)]$. Δ

The time above is not polynomial because the number of cycles may be exponential in $|V|$.

Second method

The edges traversed by the cycles of H (a non-trivial SCC) that cross the decomposing-vertex v may be divided into two groups: those that link different sub-SCC's and those that are internal to the sub-SCC's. Since any edge of a strongly connected graph must belong to at least one cycle, the first ones necessarily belong to the cycles that cross v. Besides that, they are no longer relevant for algorithm 1. So, we may ignore them, provided that we are able to find the second ones a sub-SCC at a time. To do it, we introduce the concept of *gate*.

A vertex of a SCC of level n is called an *in-gate* [*out-gate*] *of level* k ($0 \le k < n$) if it has an incoming [outgoing] edge of level k (necessarily external to that SCC). For this purpose, feedback edges are also considered. A vertex that is simultaneously an in-gate and an out-gate of level k is called a *bi-gate of level* k. A vertex may be an in-gate and an out-gate of several levels. For instance, with respect to graph G_1 in Fig. 1, vertex 2 is an in-gate of level 1 of H_1 and an in-gate of levels 1 and 2 and an out-gate of level 2 of H_2. Gates and cycles are related in the following way:

Theorem 4 (*gates and cycles*): Every existing path P_{xy} from an in-gate x of level k to an out-gate y of level k in a SCC H_n of level n ($n>k>0$) is part of a cycle of level k. Moreover, if $n=k+1$, no more paths inside H_n are part of cycles of level k. It is assumed that P_{xy} is a single vertex if x and y are identical.

Proof: See Appendix B. Δ

The in-gates and out-gates of level k may be easily obtained together with the sub-SCC's of level $k+1$ in step 6 of algorithm 1, without any significant overhead. After theorem 4, algorithm 1 may now be refined in the following way:

Algorithm 1b (*with the exploration of external cycles*):

(...)

5. Move from H to F either the incoming or the outgoing edges of a vertex v without those edges marked. If such a vertex does not exist, *fail*.

6.1. Obtain the non-trivial sub-SCC's H_1, H_2, ..., H_n (of level $k+1$) of H (of level k) and determine its new in-gates and out-gates (of level k).

6.2. For each sub-SCC H_i of H, mark the edges that belong to the paths connecting the new in-gates to the new out-gates (of level k) in H_i.

6.3. Insert each H_i into W.

(...) Δ

We state the problem to solve in step 6.2 in the following general form (the in-gates and out-gates of level k become start and stop vertices and the SCC H_i becomes a general graph G):

Problem 2 (*connecting edges*): Given a directed graph $G=(V,E)$ and sets of start and stop vertices, not necessarily disjoint, find the edges that belong to some path connecting a start vertex to a stop vertex, in this direction. It is assumed that the path that connects a start vertex to an identical stop vertex is simply that vertex.

Algorithm 2 (*finding the connecting edges*):

1. Define an augmented graph $G'=(V',E')$ with an extra vertex w, extra edges from w to each start vertex and extra edges from each stop vertex to w.
2. Find the cycles that cross w. These cycles minus the extra vertex and edges are the connecting paths and their edges are the connecting edges. Δ

Theorem 5: Algorithm 2 may be run in time $O[(|V|+|E|)(p+1)]$, where p is the number of connecting paths.

Proof: Clearly, $|V'|=|V|+1$ and $|E|<|E'|\leq|E|+2|V|$ (the upper limit occurs when all the vertices are start and stop vertices). The number of cycles that cross the extra vertex w is the number p of paths. Since we may use an algorithm that takes time $O(|V'|+|E'|)$ between consecutive cycles and step 1 may be executed in time $O(|V|)$, the overall time is $O[(|V'|+|E'|)(p+1)+|V|]$. Substituting and removing constant factors, this is the same as $O[(|V|+|E|)(p+1)]$. Δ

Notice that, like the number of cycles, the number p of paths may be exponential in the number of vertices

After these results, let's compare step 5.2 of algorithm 1a and step 6.2 of algorithm 1b. If p_i is the number of paths in each non-trivial sub-SCC $H_i=(V_i,E_i)$, c_H the number of cycles in the parent SCC $H=(V_H,E_H)$ and O(*T'*) and O(*T''*) the running times of step 5.2 of algorithm 1a and step 6.2 of algorithm 1b, then

$$T'' = \sum_i [(|V_i| + |E_i|)(p_i+1)] \leq \sum_i [(|V_i| + |E_i|)(c_H+1)] < (|V_H| + |E_H|)(c_H+1) = T'$$

Consequently, theorem 3 still applies as an upper bound. Algorithm 1b may save a significant work. If the sub-SCC's are linked sequentially, $p_1 \times ... \times p_n$ ($=c_H$) cycles are inspected by algorithm 1a, while only $p_1+...+p_n$ paths (or cycles) are inspected by algorithm 1b.

Complexity bound

Algorithm 2 still does redundant work, because it finds the paths that connect the start to the stop vertices, while we only need to know what edges belong to those paths. However, we cannot improve it dramatically, because even the following elementary sub-problem is NP-complete, as we shall see.

Problem 3 (*connecting edge*): Given a directed graph $G=(V,E)$, a start vertex s, a distinct stop vertex t and an edge (v,w) not incident on s or t (that is, with distinct s, t, v and w), is there some path connecting s and t and traversing (v,w)?

We next show that this problem is equivalent to the following well known problem:

Problem 4 (*distinct connecting paths* [4]): Given a graph $G=(V,E)$ and a collection of disjoint vertex pairs (s_1,t_1), (s_2,t_2), ..., (s_k,t_k), does G contain k mutually vertex-disjoint paths, one connecting s_i and t_i for each i, $1 \leq i \leq k$? We assume here the particular case where $k=2$ and G is directed.

Problem 3 may be transformed into problem 4 for the same graph in linear time, since any path connecting s and t and traversing (v,w) is the union of the edge (v,w) with two vertex-disjoint paths, one connecting s and v and the other connecting w and t. Problem 4 may be transformed into problem 3 in linear time, since two vertex-disjoint paths, one connecting s_1 and t_1 and the other connecting s_2 and t_2, exist iff there is a path connecting s_1 and t_2 and traversing (t_1,s_2) in the graph augmented with this edge or, equivalently, iff there is a path connecting s_2 and t_1 and traversing (t_2,s_1) in the graph augmented with this edge.

Problem 4 is NP-complete [4]. Since problem 3 may be transformed into problem 4 in linear time, problem 3 is also NP-complete. Since problem 3 is a sub-problem of problem 2, problem 2 is also NP-complete.

Polynomial time algorithms to solve problem 4 are known only for acyclic graphs [6], and they are also applicable to solve problems 2 and 3. However, such algorithms are useless here because we have to solve problem 2 for strongly connected (sub)graphs.

4. Additional failure conditions

In this section we introduce additional conditions to anticipate failures in algorithm 1.

Theorem 6 (*failure conditions*): Let H be a non-trivial nested SCC of level $k>1$ in algorithm 1, and let H' be the sub-graph of H traversed by external cycles. Then, G has no CPFES if any of the following conditions holds:

i) H' has a cycle;

ii) H has no in-gate [out-gate] without incoming [outgoing] internal edges traversed by external cycles;

iii) H has no vertex without incoming [outgoing] internal edges traversed by external cycles;

iv) H has multiple bi-gates of the same level.

Furthermore, conditions i), ii) and iii) are equivalent, in the sense that, if k_i, k_{ii} and k_{iii} are the levels of the nested SCC where they first occur, the occurrence of one of them implies the occurrence of the others with $k_i \le k_{ii} \le k_{iii}$; the occurrence of condition iv) implies the occurrence of the others with $k_{iv}=k_i$, but the converse is not true.

Proof: Beforehand, notice that H' cannot be empty. By theorem 4, all the gates of H are traversed by external cycles. Since k>1, H must have at least one in-gate and one out-gate (not necessarily distinct).

i) Obvious, since such a cycle cannot be broken without producing extra feedback edges on cycles already broken.

ii $\Rightarrow$ i) Since the only possible vertices of H' with zero in-degree [out-degree] are the in-gates [out-gates] of H, condition ii) implies that H' has no vertex with zero in-degree [out-degree]. Since H' cannot be empty, this implies i).

iii $\Rightarrow$ ii) Obvious.

iii) This is the condition of failure in algorithm 1. According to theorems 1, this is a sufficient failure condition.

i $\Rightarrow$ iii) Since condition iii) is a sufficient failure condition, if i) is true, condition iii) must be true for the same SCC or for a sub-SCC of level$>k$.

iv $\Rightarrow$ i) Let $z_1 \neq z_2$ be two bi-gates of some level $h<k$. By the definition of SCC, there are inside H paths from z_1 to z_2 and from z_2 to z_1. Those paths necessarily contain a cycle. By theorem 4, those paths are part of external cycles of level h and, consequently, are contained in H'. This implies i). Δ

Consequently, condition i) is the most important. In practice, condition ii) is also useful because it may be detected less expensively and, hence, more frequently than i); condition iv) is also useful because it does not require the actual computation of H'.

Condition ii) also allows us to limit the choice of critical vertices to in-gates and/or out-gates, although not all the CPFES's may be found this way.

In order to allow the detection of failures earlier, the algorithm that marks the edges traversed by external cycles should privilege the inspection of edges not yet marked. We propose *preliminary depth-first searches* starting from each in-gate. This allows the detection of failures in polynomial time in many cases.

5. Single in-gates and out-gates

Next we introduce some important conditions that allow a trivial selection of a critical vertex, without an expensive identification of the edges traversed by external cycles.

Theorem 7 (*trivial critical vertex*): Given a nested SCC H of level $k>1$ in algorithm 1,

i) if H has is a single in-gate [out-gate], then none of its incoming [outgoing] internal edges are traversed by external cycles (and, consequently, they may be safely cut);

ii) if H has a single gate, then none of its edges is traversed by external cycles (and, consequently, any vertex may be selected and either its incoming or its outgoing edges may be safely cut).

Proof: i) Let v be a single in-gate. The external cycles that traverse H will traverse an edge (u,v) for some external vertex u. Consequently, they cannot also traverse an edge (w,v) for any internal vertex w. A similar reasoning applies to a single out-gate.

ii) Obvious, since only the single gate is intersected by external cycles. Δ

Condition i) is a special case of condition ii). However, condition ii) enables the construction of *any* CPFES, while condition i) may not.

Corollary 7.1 (*single in-gates or out-gates*): Problem 1 may be solved with a positive answer in polynomial time $O[|V|(|V|+|E|)]$ if every nested SCC in algorithm 1 has a single in-gate or a single out-gate.

Proof: Combination of theorems 2 and 7. Δ

Notice that, for a given graph, this may happen only for an appropriate choice of the first decomposing-vertex and appropriate choices between single in-gates and single out-gates, which are not known in advance.

The conditions stated in theorem 7 may not apply for a sub-SCC of level $n>k$, in which case its edges traversed by external cycles have to be found. Fortunately, the following theorem shows that we only have to inspect the paths connecting the in-gates and out-gates of the SCC of level n, and have not to return to the SCC's of previous levels.

Theorem 8: Let H_k, H_{k+1}, ..., H_n $(0<k<n)$ be a sequence of related (parent-child) non-trivial SCC's such that each H_{k+1}, ..., H_{n-1} has a single in-gate or a single out-gate that is selected for decomposing-vertex and its incoming or outgoing edges are cut, respectively. Then,

i) The intersection of the external cycles of levels k, $k+1$, ..., $n-1$ with H_n are the paths in H_n that connect its in-gates to its out-gates of levels k, $k+1$, ..., $n-1$ (with the levels of the in-gate and out-gate possibly different). It is assumed that, if the in-gate and out-gate are the same vertex, the path reduces to that vertex.

ii) If H_n has more than one vertex that is simultaneously an in-gate of level i and an out-gate of level j, with i and j between k and $n-1$ and not necessarily equal, then G has no CPFES.

Proof: See Appendix C. Δ

In practice, it is not necessary to record the levels of the in-gates and out-gates; it is sufficient to distinguish between *old* and *new* ones. The new ones are determined in step 6 of algorithm 1, together with the sub-SCC's. In step 5, whenever there are multiple in-gates and multiple out-gates (including old and new ones), the paths connecting the new in-gates to the new out-gates must be found before the selection of the decomposing-vertex, after which those in-gates and out-gates become old ones.

The failure condition iv) of theorem 6 (multiple bi-gates of the same level) is replaced by the more general failure condition ii) of theorem 8, that need only be tested for the new gates. After theorems 6, 7 and 8, algorithm 1 may be improved in the following way:

Algorithm 1c (*improved version of algorithm 1*):

1. Initialize F, the set of feedback edges, with the empty set.
2. Initialize W, the working set of the non-trivial SCC's to decompose, with the non-trivial SCC's H^1_i (of level 1) of G.
3. If W is empty, stop with success (F is a CPFES).
4. Select and remove from W an arbitrary element H^k_i (a non-trivial SCC of level k).

5.1. If $k=1$, move from H^k_i to F either the incoming edges or the outgoing edges of an arbitrary vertex and skip to step 6.

5.2. If H^k_i has more than one vertex that is simultaneously a new in-gate and a new out-gate, fail.

5.3. If H^k_i has a single in-gate [out-gate], move its incoming [outgoing] edges from H^k_i to F and skip to step 6.

5.4. Find and mark (as traversed by external cycles) all the edges that belong to the paths that connect the new in-gates to the new out-gates in H^k_i and mark the new in-gates and out-gates as old ones. Optionally, perform preliminary depth-first searches starting in each new in-gate and, only if conditions 5.5 and 5.6 don't apply, continue to perform exhaustive searches.

5.5. If H^k_i has no in-gate [out-gate] without incoming [outgoing] internal edges marked, fail. If possible, test this before and during step 5.4.

5.6. If there are cycles in the marked edges of H^k_i, fail. If possible, test this during step 5.4.

5.7. Move from H^k_i to F either the incoming or the outgoing edges of a vertex (possibly an in-gate or an out-gate, respectively) without those edges marked.

6. Obtain the non-trivial (sub) SCC's H^{k+1}_j of H^k_i without the edges moved to F, determine and mark as new the respective in-gates and out-gates of level k, and insert each H^{k+1}_j into W.
7. Go to step 3. Δ

Examples are given in Fig. 1. All the graphs presented are strongly connected. Hence, there is initially a single working non-trivial SCC (the entire graph). In all the cases, vertex *1* is initially selected and its incoming edges are cut. The remaining non-trivial nested SCC's processed by algorithm 1c are shown in the figure. In the case of graphs G_1 and G_2, the analysis of in-gates and out-gates is sufficient. In the case of G_1, H_1 has a single out-gate (vertex *4*), and H_2 has a single in-gate (vertex *1*). In the case of G_2, H_3 has a single out-gate (vertex *4*), and H_4 has two new bi-gates (vertices *1* and *3*); hence, G_2 doesn't have a CPFES. In the case of G_3, H_5 has neither a single in-gate (both vertices *2* and *3* are in-gates) nor a single out-gate (both vertices *3* and *4* are out-gates); hence, the edges traversed by external cycles have to be found and marked (edges *(2,3)* and *(3,4)*); the remaining non-trivial nested-SCC H_6 has a single in-gate (vertex 4). In the case of graph G_4, H_7 has neither a single in-gate (both vertices *2* and *4* are in-gates) nor a single out-gate (both vertices *3* and *4* are out-gates); hence, the edges traversed by external cycles have to be found and marked (edges *(2,3)*, *(3,4)* and *(4,3)*); since there is a cycle in these edges, G_4 doesn't have a CPFES.

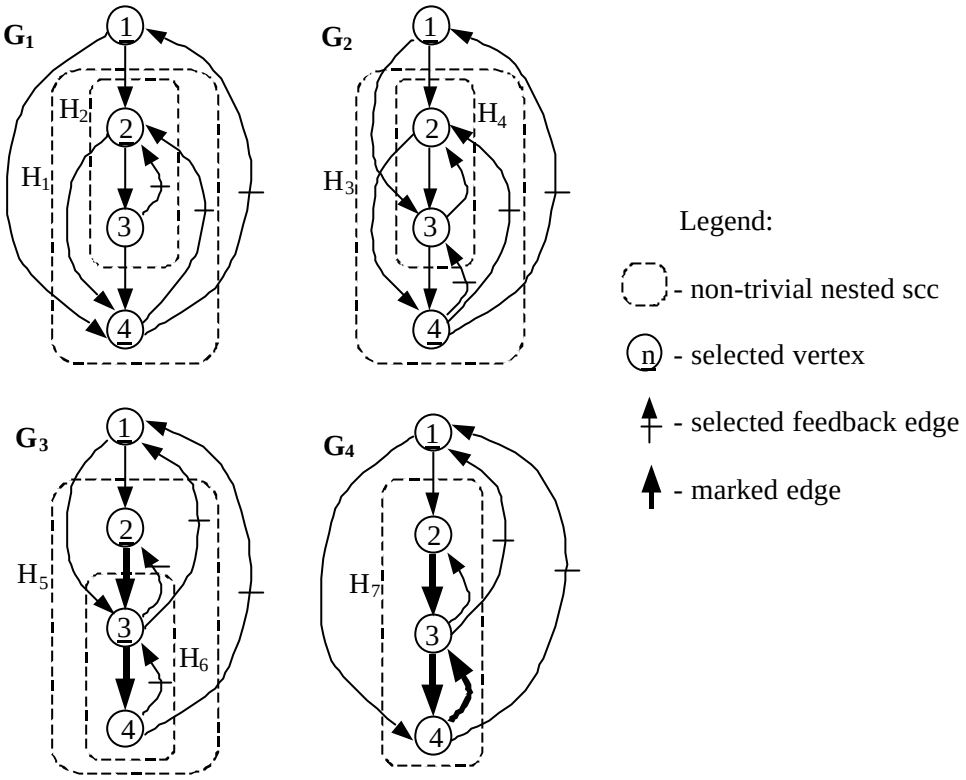


Fig. 1 - Examples of graphs, with illustrations to explain how they are processed by algorithm 1c.

6. Polynomial time algorithm for reducible flow graphs

The existence of a single in-gate mentioned in corollary 7.1 is closely related to the concept of a reducible flow graph. Recall that a *flow graph* (or *program graph*) G is a directed graph with a distinguished start vertex s from which any other vertex may be reached (see [2], pg. 361). It is known that a flow graph G is *reducible* iff there do not exist distinct vertices $v \neq s$ and $w \neq s$, directed paths P_1 from s to v and P_2 from s to w, and a cycle C containing v and w, such that C has no edges and only one vertex in common with each P_1 and P_2 (see [2], pg. 363). That is, iff every cycle not traversing s may be reached from s at a single vertex. A property of reducible flow graphs already presented in [7] may be restated as:

Theorem 9: Every non-trivial SCC H of a reducible flow graph G that does not contain the start-vertex s has a single in-gate.

Proof: Assume H has multiple in-gates $x_1, x_2, \ldots$ (of level 0). By the definition of in-gate, there must exist vertices $v_1, v_2, \ldots$ outside H, not necessarily distinct, and edges (v_1,x_1), (v_2,x_2), ... From the definition of a flow graph, there must exist paths from s to $v_1, v_2, \ldots$, and those paths cannot cross H (otherwise the vertices would belong to H). Consequently, all the in-gates are reachable separately from s. By the definition of SCC, H contains a directed path P_{12} from x_1 to x_2 and a directed path P_{21} path from x_2 to x_1. Let w be the vertex common to P_{12} and P_{21} nearer to x_1 (w may be x_2), and let C be the cycle made by the parts of P_{12} and P_{21} between x_1 and w. But then the cycle C may be reached from s at x_1 and w (through x_2) and G is not reducible. Since we assumed G is reducible, we conclude that H must have a single in-gate. Δ

As a generalization of this result, we may conclude:

Theorem 10: Problem 1 may be solved positively in polynomial time O[(|*V*|+|*E*|)|*V*|] if *G* is a reducible flow graph.

Proof: We give a method to obtain *F*. Let *H* be a non-trivial SCC (of level 1) of *G*. If *H* does not contain *s*, we have already shown that it has a single in-gate *r* (of level 0). So, we select it and remove its incoming edges. If *H* contains *s*, we just select *r=s*. Let H_2 be a non-trivial sub-SCC of level 2 obtained from *H*. It cannot have any in-gate of level 0, because there is only *r (*and it becomes a trivial SCC of level 2). It has a single in-gate of level 1. The argument is similar as for *H* (see the proof of theorem 9), except that the paths from *s* traverse *r* (in case they are different). The incoming edges may be cut according to theorem 7. The same reasoning applies recursively to nested SCC's. Δ

7. Implementation and performance results

A detailed algorithm (*CPSORT*) in structured pseudo-code, with space requirements of order O(|*V*|+|*E*|), was derived directly from algorithm 1c (see Appendix D) and then implemented in C.

In order to estimate the average and worst-case performance of *CPSORT*, three experiments were conducted. The experiments run in a Sun4c machine running SunOS and the *prof* utility was used to measure the average time spent per call to *CPSORT*.

In the first experiment, the following algorithm was used to generate strongly connected graphs of moderate density at random:

Algorithm 3 (*generate a random strongly connected graph with n vertices and d "expected" out-degree per vertex*):

1. Initialize the vertex set *V* with the set of numbers {*1, 2, ..., n*}
2. Repeat the following until a strongly connected graph *G=(V,E)* has been generated:
   2.1. Initialize the edge set *E* with the empty set *{}*
   2.2. For each vertex *u* of *V*, repeat the following until *u* has at least an outgoing-edge:
      2.2.1. For each vertex *v* of *V*, distinct from *u*, generate a (pseudo) random number *r*, and,
         if *r* lies in an interval with probability *d/n*, add the edge *(u,v)* to *E*. Δ

In fact, *d* is not rigorously the expected out-degree per vertex, because a probability *d/n* is used instead of *d/(n-1)* (remember that each vertex may have *n-1* outgoing edges), and because non strongly connected graphs are rejected.

For each number of vertices (*n*) from 4 to 28, 1700 strongly connected graphs (100 for each value of *d* in the range 0.8, 1.0, ..., 4.0, possibly with repetitions) were generated according to the above algorithm, and then processed by *CPSORT*. Higher numbers of vertices (above 28) where not tried because of the time spent by algorithm 3 (and not because of *CPSORT* itself). The performance results obtained are summarized in Fig. 2.

From Fig. 2, we conclude that, in experiment 1, the average execution time of *CPSORT* is approximately linear in the number of vertices. Since, in this experiment, the expected number of edges per cycle is approximately proportional to the number of vertices (approximately (0.8+4.0)/2×*n* edges expected), we can also conclude that, in this experiment, the average execution time of *CPSORT* is approximately linear in the size of the graph, measured in number of vertices plus number of edges. By

contrast, the average number of cycles per graph, as well as the time spent to count them, using an algorithm similar to the one presented in [5] to generate all the cycles of a directed graph, presented an exponential growth with the number of vertices (see Appendix E).

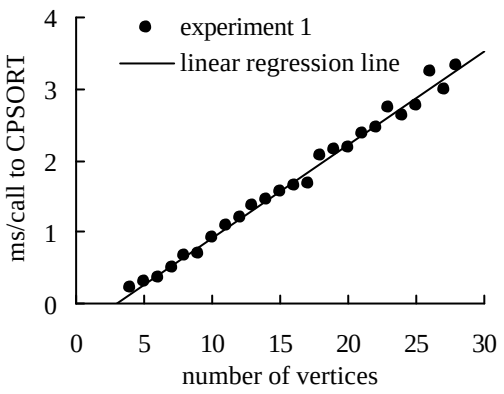


Fig. 2 Average performance of *CPSORT*, as a function of the number of vertices, in experiment 1 (expected number of edges linear in the number of vertices).

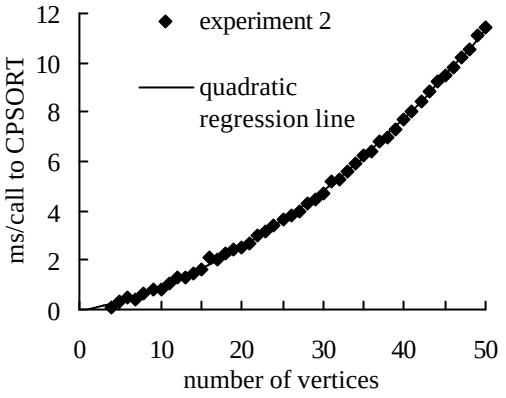


Fig. 3 Average performance of *CPSORT*, as a function of the number of vertices, in experiment 2 (expected number of edges quadratic in the number of vertices).

In the second experiment, for each number of vertices (n) from 4 to 50, $100 \times n$ graphs were selected at random (possibly with repetitions) from the set of all simple directed graphs with n vertices (without exclusion of isomorphic graphs), and then processed by *CPSORT*. Since a complete directed graph with n vertices has $n \times (n-1)$ edges, in order to generate a random graph with n vertices, it suffices to generate a random number (or several random numbers concatenated together) with $n \times (n-1)$ bits, and select or reject an edge according to the value of the corresponding bit. The performance results obtained are summarized in Fig. 3.

From Fig. 3, we conclude that, in experiment 2, the average execution time of *CPSORT*, is approximately quadratic in the number of vertices. Since, in this experiment, the expected number of edges per cycle is quadratic in the number of vertices ($n \times (n-1)/2$ edges expected), we can also conclude that, in this experiment, the average execution time of *CPSORT* is approximately linear in the size of the graph, measured in number of vertices plus number of edges, as in experiment 1.

In both experiments, the percentage of graphs with a CPVO, as a function of the number of vertices, presented an exponential decrease towards 0, stronger in the second experiment (see Appendix E).

In order to estimate the worst-case performance of *CPSORT*, we identified of a class of "difficult" graphs with a CPVO - see Fig. 4. The performance results obtained are summarized in Fig. 5. From Fig. 5, we conclude that the execution time of *CPSORT*, for graphs of this class, grows exponentially with the number of vertices (and also with the size of the graph, because the number of edges is $(n-1) \times (n-2)/2+4$). In this experiment, *CPSORT* took approximately the same time as the time needed to count the cycles (see Appendix E).

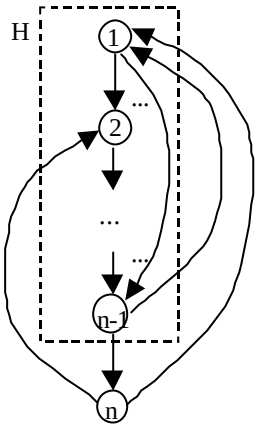


Fig. 4 Special class of graphs poorly processed by *CPSORT*. *H* is complete acyclic sub-graph of n-1 vertices, in topological order 1, 2,..., n-1.

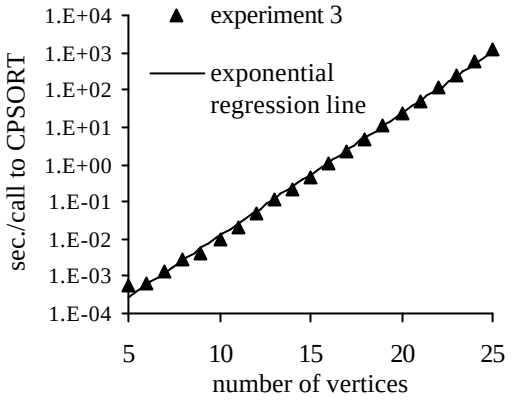


Fig. 5 Performance of *CPSORT*, as a function of the number of vertices, for the class of graphs in Fig. 4.

We feel that the kind of graphs shown in Fig. 4 is very close to the worst-case. Comparing the results in this experiment with the good average performance of *CPSORT* in the previous experiments, we conclude that difficult cases are very rare. For such difficult cases, *CPSORT* has been designed so that it may be interrupted at any time (e.g. when some kind of time-out is reached), except when two vertices are being interchanged, producing the best ordering obtained so far. This is advantageous when cycle-preserving vertex ordering is for efficiency and not for semantic, as it happens with some logic programs (e.g. pure datalog rules).

8. Conclusions and further work

In this paper, it was presented (by stepwise refinement) an algorithm to solve the following problem (problem 1): check if a directed graph $G=(V,E)$ has a CPFES (or equivalently, a CPVO) and, if so, produce it. CPVO's are relevant in the context of bottom-up fixpoint evaluation of datalog programs. Problem 1 was related with other known problems, and it was proved that its worst-case complexity is of order not greater than $O((|V|+|E|)(c+1))$, where c is the number of cycles, $|V|$ is the number of vertices and $|E|$ is the number of edges. The expensive steps of the algorithm were carefully identified, and several conditions were found under which those steps could be avoided, leading to a polynomial execution time in many cases. Classes of graphs for which problem 1 can be solved in polynomial time include reducible flow graphs.

Several experiments with randomly generated graphs have shown an average running time of the algorithm approximately linear in the size of the graph. It was also identified a special class of "worst-case" graphs, for which the running time of the algorithm grows exponentially with the size of the graph.

An important open problem is to prove whether problem 1 is NP-complete. The partial results obtained in this paper suggests so, but a definite proof was not given.

Acknowledgments

The author would like to thank João Canas Ferreira for his comments and suggestions.

References

1. R. Ramakrishnan, D. Srivastava, S. Sudarshan, Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs, *in* "Proceedings of the 16th VLDB Conference" (D. McLeod, R. Sacks-Davis, H. Schek, Ed.), pp. 359-371, Brisbane, Australia, 1990.
2. K. Thulasiraman, M. N. S. Swamy, "Graphs: Theory and Algorithms", John Wiley & Sons, 1992.
3. A. V. Aho, J. E. Hopcrotf, J. D. Ullman, "The Design and Analysis of Computer Algorithms", Addison-Wesley Publishing Company, Reading, Massachusetts, 1974.
4. M. R. Garey, D. S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness", Bell Laboratories, W. H. Freeman and Company, San Francisco, 1979.
5. E. M. Reingold, J. Nieverge, D. Narsingh, "Combinatorial Algorithms: Theory and Practice", Prentice-Hall, 1977.
6. Y. Perl, Y. Shiloach, Finding Two Disjoint Paths Between Two pairs of Vertices in a Graph, *J. Assoc. Comput. Mach.* **25** (1978), 1-9.
7. J. L. Szwarcfiter, On minimum Cuts of Cycles and Maximum Disjoint Cycles, *Contemp. Math.* **89** (1989), 153-166.

Appendix A - Proof of theorem 1 –iv

In order to be able to prove part iv of theorem 1, we first introduce a few lemmas about how to transform a vertex ordering into another equivalent vertex ordering, modulo the number of induced feedback edges per cycle.

Lemma A1 (*rotation*): Given an ordering L of the vertices of a graph G, the number of feedback edges induced by L on each cycle of G is not changed by a rotation of L.

Proof: This is already presented in [1]. Any rotation of L is done by interchanging two sub-sequences A (left) and B (right) into which L is divided. When A and B are interchanged, only the edges linking A and B are reversed. Every cycle has the same number of edges directed from A to B and from B to A. So, for every cycle of G, the number of additional feedback edges (directed from A to B) equals the number of feedback edges subtracted (directed from B to A), and the total is not changed. Δ

Lemma A2 (*local rotation*): Given an ordering L of the vertices of a graph G and two contiguous sub-sequences A and B of L such that there is no cycle simultaneously intersecting vertices in A, in B and outside A and B, the number of feedback edges induced on each cycle of G is not changed if A and B are interchanged in L.

Proof: The number of feedback edges induced on the cycles that do not intersect vertices outside A and B are not changed, as in lemma A1. The feedback edges induced on the cycles that intersect vertices outside A and B are not changed, since it's assumed that those cycles don't intersect simultaneously A and B, and so don't intersect any edge linking A and B. Δ

Lemma A3 (*local reordering*): Let L be a cycle-preserving ordering of the vertices of a graph G. Let S be a sub-sequence of L. Let v be a vertex in S such that there is no cycle in G simultaneously intersecting a vertex outside S, a vertex before [after] v in S and v itself. Then, there is a permutation R of S, without extra feedback edges, such that there is no cycle in G simultaneously intersecting a vertex outside R, a vertex before [after] v in R, and v or a vertex after [before] v in R.

Proof: We give a method to obtain a possible R from S. We number the steps for easier reference.

1. Let H be the sub-graph of G induced by S (that is, H has the vertices contained in S plus the edges of G whose end vertices are in S).

2. The cycles of G that are not fully contained in H are called *external cycles*.

3. For any external cycle C, the feedback edge f induced by L in C (linking the last to the first vertex of C in L) is not contained in H, because at least one of the end vertices of f is not contained in S (otherwise C would have no vertex outside S).

4. Hence, the intersection of an external cycle with H is a simple path.

5. Let H' be the graph obtained from H by removing the feedback edges induced by S, and let G' be the graph obtained from G by removing the same feedback edges (only those removed from H).

6. Any topological ordering of the vertices of H' cannot induce extra feedback edges, besides the ones induced by S. Furthermore, if S induces a minimal FES, then any topological ordering of the vertices of H' induces exactly the same feedback edges as S.

7. Given 4, the assumption that there is no cycle in G simultaneously intersecting a vertex outside S, a vertex before [after] v in S and v itself, is equivalent to the assumption that there is no vertex u and edge (u,v) [(v,u)] in H' intersected by an external cycle.

8. Assume that R is a topological ordering of the vertices of H'. According to 6, this guarantees that R does not induce extra feedback edges. According to 7, the assumption about S also applies to R, that is, there is no cycle in G simultaneously intersecting a vertex outside R, a vertex before [after] v in R and v itself. Given 4, to further guarantee that there is no cycle in G simultaneously intersecting a vertex outside R, a vertex before [after] v in R, and a vertex after [before] v in R, it is sufficient to require that, for any edge (u,w) of H' intersected by an external cycle, with $u \rightarrow v \rightarrow w$, both u and w must precede or follow v in R.

9. Let H'' be the graph obtained from H' by adding an edge (t,s) for each pair of distinct vertices s and t in H', such that there is an external cycle whose intersection with H (or H') has initial vertex s and terminal vertex t. External cycles that intersect a single vertex in H are ignored, because they don't intersect any edge of H.

10. Besides the edges previously added, only the edges of H (or H') that are intersected by external cycles are contained in SCC's of H''. In fact, any edge (u,v) of H intersected by an external cycle will be contained in a cycle $(s, ..., u,v,...,t,s)$ of H'' of length>1, and, consequently, will be contained in a non-trivial SCC of H''. Conversely, assume (u,v) is an edge of H contained in a SCC of H''. This means that there is a path from v to u in H''. Since the edges added in 9 correspond to paths in G', this also means that there is a (simple) path from v to u in G', completing a cycle C' in G', together with (u,v). Since G' does not contain any cycle of H, C' is an external cycle.

11. In order to obtain R we sort topologically the SCC's of H''. Inside each SCC, we remove the edges previously added (returning back to H'), and sort topologically its vertices. A special care has to be taken with respect to the SCC of H'' that contains v. Let J'' be the SCC of H'' that contains v, and let J' be the corresponding acyclic sub-graph, without the edges added in 9. Since there is no edge (u,v) [(v,u)] in H' intersected by an external cycle (according to 7), and we know that all the edges in J' are intersected by external cycles (according to 10), we conclude that there is no edge (u,v) [(v,u)] in J'. Consequently, when sorting topologically the vertices of J', v may be moved to the first [last] position.

12. With respect to the resulting ordering R, for any edge (u,w) of H' intersected by an external cycle, with $u \rightarrow v \rightarrow w$, both u and w will precede or follow v in R, because either the edge (u,w) belongs to a different SCC in H'' (other than J''), or it belongs to the same SCC (J''), and v is in the first [last] position within J''. Δ

Lemma A4 (*local reordering and rotation*): Given a vertex v in a sub-sequence S of a cycle-preserving ordering L of the vertices of a graph G, there is a permutation P of S, for the same number of feedback edges per cycle, with v in the first [last] position of P iff there is no cycle in G simultaneously intersecting a vertex outside S, a vertex before [after] v in S and v itself.

Proof: *Necessity*: Assume the condition is false, that is, assume there is a cycle C intersecting a vertex outside S, a vertex before [after] v in S and v itself. Let u be the vertex immediately before [after] v intersected by C in S. Since cycles are preserved, C has only one feedback edge induced by L, linking the first and the last vertex of C in L, one of which must be outside S (otherwise C would have no vertex outside S). The direction of this feedback edge is not changed by any permutation P of S, while the relative positions of u and v are interchanged in any permutation of S that moves v to the first [last] position, originating an additional feedback edge. So, the number of feedback edges induced on C is increased. Consequently, there is no such P if the condition is false.

Sufficiency: Assume the condition is true. According to lemma A3, we can reorder S to obtain a sequence R, without extra feedback edges, with v already in the first [last] position of R or with R dividable into two sub-sequences A and B such that v is in the first position of B [last position of A],

with *A* and *B* in the conditions of lemma A2. So, these two sub-sequences may be interchanged to obtain *P*, with the same number of feedback edges per cycle and with *v* in the first [last] position. Δ

We are now in conditions to prove:

Proof of theorem 1 – iv: We prove the equivalent converse statement. Let *G=(V,E)*. Assume *G* has a CPFES. Since any non-minimal CPFES of *G* is easily transformed into a minimal element by removing the edges that do not belong to any cycle of *G*, *G* has at least one minimal CPFES, say *F'*. Obviously, there exists a corresponding CPVO *L* of the vertices of *G*, such that the vertices of each SCC of *G* are kept together and the SCC's are topologically ordered in *L*. The construction of a CPFES *F* by algorithm 1, corresponds to a sequence of transformations of *L*, preserving the number of feedback edges per cycle in each transformation, such that, in the end, *L* corresponds to *F*. Let *H* be a non-trivial SCC (of level 1) of *G*, as referred in step 5 of algorithm 1. Let *S* be the corresponding sub-sequence in *L*. According to lemma A4, a vertex *v* of *H* may be moved to the first [last] position of *S* (still preserving cycles), if (and only if) there is no external cycle (with respect to *H*) intersecting an edge (*u*,*v*) [(*v*,*u*)] for some vertex *u* in *H*. This condition is exactly the one stated in step 5 of algorithm 1. Moving *v* to the first [last] position of *S*, corresponds, in terms of feedback edges, to add its incoming [outgoing] edges in *H* to the resulting feedback edge set *F*. In this case (SCC of level 1), there are no external cycles, and so any vertex may be selected and step 5 cannot fail. Consequently, assume *S* is reorganised as stated in lemma A4. Obviously, it may be further reorganised, without modification of the feedback edges, so that the vertices of each resulting sub-SCC are kept together and the sub-SCC's are topologically ordered. Let *H'* be a non-trivial sub-SCC of level 2 (with parent SCC *H*) and let *S'* be the corresponding sub-sequence in the reorganised vertex ordering. Again, any vertex *v* of *H'* in the conditions of lemma A4 or step 5 may be moved to the first [last] position of *S'*. At least the first [last] vertex of *S'* is already in those conditions, so step 5 cannot fail. If another vertex is selected in the required conditions, we may reorganise *S'* in a way similar to *S*, and proceed recursively. This is true also for nested SCC's of higher levels. So, we conclude that step 5 never fails, independently of the choices made in previous iterations. Δ

Appendix B - Proof of theorem 4

Proof: Let H_k be the corresponding ancestor SCC of level k and let v be its decomposing-vertex.

Assume the case where $n=k+1$ and H_n is not v. By the definition of in-gate and out-gate, there must exist vertices w and z (not necessarily distinct) and edges (w,x) and (y,z) inside H_k but outside H_n. By the definition of SCC, there must exist a (simple) path P_{zw} from z to w in H_k. This path must intersect v; otherwise, z and w would belong to the same SCC of level $k+1$ as x and y (that is, H_n). We next prove that P_{zw} doesn't intersect H_n. If $z=w$, the proof is done. Otherwise, assume that P_{zw} intersects H_n, and let s and t be the first and last vertex (not necessarily distinct) of P_{zw} in H_n. Let P_{zs} and P_{tw} be the portions of P_{zw} from z to s and from t to w. Without any loss of generality (the other case is symmetric), assume that it is P_{zs} that contains v. Since P_{zs} and P_{tw} are disjoint except possibly for s and t in case $s=t$, and s and t are contained in H_n, and H_n doesn't contain v, we conclude that P_{tw} doesn't contain v. By the definition of SCC, there is a path from x to t in H_n. Since there are paths from w to x (the edge (w,x)) and from x to w (through t) in H_k not traversing v, x and w must belong to the same SCC of level $k+1$ (that is, H_n). Since this contradicts the definition of x and w, we must conclude that P_{zw} cannot intersect H_n. Hence, the path P_{zw} concatenated with any path P_{xy} in H_n constitutes a cycle contained in H_k but not in H_n, that is, constitutes a cycle of k.

Now assume the case where $n=k+1$ and H_n is the decomposing-vertex v. In this case, $x=y=v$. Since there is at least one cycle crossing any vertex (namely v) of a non-trivial SCC, and all the cycles inside a SCC of level k (H_k in this case) that cross its decomposing-vertex (v in this case) have the same level k, we conclude that P_{xy} (that reduces to v in this case) is part of at least one cycle of level k.

Now assume the case where $n>k+1$. Let H_{k+1} be the (ancestor) SCC of level $k+1$ corresponding to H_n. Then, x and y are also gates of level k of H_{k+1} and the paths that connect x and y in H_n also connect x and y in H_{k+1}. So, the same conclusion holds.

To prove the second part of the theorem, notice that the cycles of level k that cross an SCC of level $n>k$ must enter and leave it through in-gates and out-gates of levels between k and $n-1$. If $n=k+1$ this reduces to k. Δ

Appendix C - Proof of theorem 8

Proof:

i) First, notice that no more paths in H_n might be traversed by external cycles of levels k, k+1, ..., n-1. So we only have to prove that each such path is traversed be one such cycle.

Let v_k, v_{k+1}, ..., v_{n-1} be the decomposing-vertices of H_k, H_{k+1}, ..., H_{n-1}. Since H_n is a non-trivial SCC, it cannot contain any of these vertices (the decomposing-vertex of a SCC of level i, is always a trivial SCC of level $i+1$).

Let x and y be an in-gate and an out-gate (not necessarily distinct) of H_n with the required levels (between k and $n-1$). By the definition of in-gate and out-gate, there must exist vertices w and z (not necessarily distinct) and edges (w,x) and (y,z) inside H_k but outside H_n. By the definition of SCC, there must exist a (simple) path P_{zw} from z to w in H_k. This path must intersect at least one of the decomposing vertices v_k, v_{k+1}, ..., v_{n-1}; otherwise, z and w would belong to the same SCC of level n as x and y (that is, H_n). Let i ($k£i£n-1$) be the index of the decomposing-vertex (v_i) with lowest index intersected by P_{zw}. We next prove that P_{zw} doesn't intersect H_n. If $z=w$, the proof is done. Otherwise, assume that P_{zw} intersects H_n, and let s and t be the first and last vertex (not necessarily distinct) of P_{zw} in H_n. Let P_{zs} and P_{tw} be the portions of P_{zw} from z to s and from t to w. Without any loss of generality (the other case is symmetric), assume that P_{zs} contains v_i. For any j, $i+1£j£n-1$, P_{zs} must contain a sub-path from v_i to an in-gate x_j of H_j and a sub-path from an out-gate y_j of H_j back to v_i (remember that H_j cannot contain v_i, because v_i becomes a trivial SCC of level $i+1$ and H_j is non-trivial). Since H_j has a single in-gate or a single out-gate that is selected for decomposing-vertex, one of these (x_j or y_j) must be v_j. Consequently, P_{zs} must also contain the vertices v_{i+1}, ..., v_{n-1}, besides v_i. Since all the decomposing vertices v_i, v_{i+1}, ..., v_{n-1} are contained in P_{zs}, and P_{zs} and P_{tw} are disjoint, except possibly for s and t (contained in H_n) in case $s=t$, and P_{zw} does not contain other decomposing vertex v_h, $k£h£i-1$, we conclude that P_{tw} does not contain any of the decomposing vertices v_k, v_{k+1}, ..., v_{n-1}. Since there are paths from w to x (the edge (w,x)) and from x to w (through t) in H_k not traversing any of the decomposing vertices v_k, v_{k+1}, ..., v_{n-1}, x and w must belong to the same SCC of level n (that is, H_n). Since this contradicts the definition of x and w, we must conclude that P_{zw} cannot intersect H_n. Hence, the path P_{zw} concatenated with any path P_{xy} in H_n constitutes a cycle of level i, $k£i<n$.

ii) This is similiar to condition iv) of theorem 6. Let $z_1 \neq z_2$ be two vertices in the conditions of the theorem. By the definition of SCC, there are inside H_n paths from z_1 to z_2 and from z_2 to z_1. Those paths necessarily contain a cycle C. By i), those paths are part of external cycles. So, C cannot be broken without producing extra feedback edges on (the external) cycles already broken. Δ

Appendix D - Detailed algorithm (*CPSORT*)

```
/*
Inputs:
   V - ordered set of vertices V1, V2,  ... V|V|
   For each vertex v Î V,
      Adj(v)  - set of the immediate successors of v, that is, {u Î V:  (v,u) Î E}
      Adj-1(v)        - set of the immediate predecessors of v, that is, {u Î V:  (u,v) Î E}
Outputs:
   failed   - true if the graph has no cycle-preserving ordering
   V        - reordered in-place

Comments: The algorithm works in-place by successive improvements. It may be easily modified
so that, after a failure, decomposition into nested SCC's still proceeds in order to minimize
feedback edges.
*/

/* Main procedure */
procedure CPSORT
begin
   /* initialize data */
   for i=1, ..., |V| do
      index(Vi) ← i
      current(Vi) ← startSCC(Vi) ← ingate(Vi) ← newingate(Vi) ← outgate(Vi) ←
newoutgate(Vi) ← false
      for w∈ Adj(Vi) do inextcycle(Vi,w) ← false
   failed ← false
   k ← 0 /* level */
   s0 ← 1 /* start index of the current SCC of level 0 */
   t0 ← |V| /* stop index  of the current SCC of level 0 */
   /* work recursively */
   CPSORTR
end

/* Recursively sorts the vertices of the current SCC of level k. */
procedure CPSORTR
begin
   /* Divide into (sub) SCC's topologically sorted */
   TOPSORTSCC
   /* Scan the ordered sub-components */
   k ← k+1
   sk ← sk-1
   while sk≤tk-1 ∧ ¬failed do
      /* find the end of the current SCC of level k */
      tk ← min{i: sk≤i≤tk-1 ∧ [i=tk-1 ∨ startSCC(Vi+1)]}
      /* only non-trivial SCC's have further cycles and need be analysed */
      if tk≠sk then
         SELECT_CRITICAL_GATE
         if ¬failed then CPSORTR
      sk ← tk+1
```

```
    /* return to the parent level */
    k ← k-1
end

/* Decomposes the current SCC of level k into sub-SCC's of level k+1 by topological order,
 and determine their new in-gates and out-gates. */
procedure TOPSORTSCC
begin
    S ← s_k /* increasing-decreasing index to place pending vertices */
    T ← t_k /* decreasing index to place vertices already sorted */
    d ← 0
    if k=0 then
        while S≤T do TOPOSRTSCCR(V_S)
    else
        /* exclude the selected critical vertex, which becomes a trivial sub-SCC*/
        if x_k=V_sk then S ← S+1 else T ← T-1
        startSCC(x_k) ← true
        lowlink(x_k) ← 0
        /* work recursively from its successors */
        for u ∈ Adj(x_k) such that s_k≤index(u)≤t_k do
            if S≤index(u)≤T then TOPSORTSCCR(u)
            ingate(u) ← true
            newingate(u) ← true
end

/* Recursive topological sort from a vertex v. */
procedure TOPSORTSCCR(v)
begin
    d ← d+1
    lowlink(v) ← d
    current(v) ← true
    startSCC(v) ← true
    for u ∈ Adj(v) such that s_k≤index(u)≤t_k do
        if k≠0 ∧ u=x_k then
            outgate(v) ← true
            newoutgate(v) ← true
        else
            if ¬current(u) ∧ S≤index(u)≤T then TOPSORTSCCR(u)
            if index(u)>T then /* u and v belong to different SCC's */
                if k>0 then
                    ingate(u) ← true
                    newingate(u) ← true
                    outgate(v) ← true
                    newoutgate(v) ← true
            else
                if lowlink(u)<lowlink(v) then
                    lowlink(v) ← lowlink(u)
                    startSCC(v) ← false
    if startSCC(v) then
        /* move (last in last out) the "top" of S onto the "top" of T */
```

```
        b ← S
        while b>s_k ∧ lowlink(V_{b-1})≥lowlink(v) do b ← b-1
        j ← b
        while j<S ∧ j<T do
            EXCHANGE(j,T)
            j ← j+1
            T ← T-1
        T ← T-(S-j)
        S ← b
        /* move v onto the "top" of T */
        EXCHANGE(index(v),T)
        T ← T-1
    else
        /* move v onto the "top" of S */
        EXCHANGE(index(v),S)
        S ← S+1
    current(v) ← false
end

/* Exchanges the positions of two vertices. */
procedure EXCHANGE(i,j)
begin
    V_i ↔ V_j
    index(V_i) ← i
    index(V_j) ← j
end

/* Selects a critical in-gate or out-gate (x_k) in the current SCC of level k and moves it to the first
or last position, respectively. Returns false if there is none. */
procedure SELECT_CRITICAL_GATE
begin
    /* level 1 is trivial */
    if k=1 then
        x_k ← V_{s_k}
        return
    /* single in-gate is trivial */
    ingates ← 0
    for i=s_k, ..., t_k do if ingate(V_i) then ingates ← ingates+1
    if ingates=1 then
        x_k ← V_{s_k} /* it must be already in the first position */
        return
    /* single out-gate is trivial */
    outgates ← 0
    for i=s_k, ..., t_k do if outgate(V_i) then
        outgates ← outgates+1
        indexoutgate ← i
    if outgates=1 then
        /* place the out-gate in the last position */
        EXCHANGE(t_k,indexoutgate)
        x_k ← V_{t_k}
        return
```

```
    /* multiple new bi-gates */
    newbigates ← 0
    for i=s_k, ..., t_k do if newingate(V_i) ∧ newoutgate(V_i) then newbigates ← newbigates+1
    if newbigates>1 then
        failed ← true
        return
    /* otherwise inspect the edges traversed by external cycles */
    MARKEDGES
    if Ø failed then
        /* move the critical in-gate to the first position */
        i ← min{i: s_k≤i≤t_k ∧ critingate(V_i)}
        EXCHANGE(s_k,i)
        x_k ← V_{s_k}
end

/* Marks the edges of the current SCC of level k that connect the new in-gates to the new out-
gates. Returns as soon as possible on a failure condition. */
procedure MARKEDGES
begin
    /* counts critical in-gates and out-gates */
    cyclicmarks ← false
    critingates ← 0
    critoutgates ← 0
    for i=s_k, ..., t_k do
        if ingate(V_i) then
            if ∃u∈ Adj^{-1}(V_i): s_k≤index(u)≤t_k ∧ inextcycle(u,V_i) then
                critingate(V_i) ← false
            else
                critingate(V_i) ← true
                critingates ← critingates+1
        else
            critingate(V_i) ← false
        if outgate(V_i) then
            if ∃u∈ Adj(V_i): s_k≤index(u)≤t_k ∧ inextcycle(V_i,u) then
                critoutgate(V_i) ← false
            else
                critoutgate(V_i) ← true
                critoutgates ← critoutgates+1
        else
            critoutgate(V_i) ← false
    if critingates=0 ∨ critoutgates=0 then
        failed ← true
        return
    /* preliminary depth-first search from each in-gate to rapidly detect failures (optional) */
    dfs ← true
    for i=s_k, ..., t_k do if newingate(V_i) then
        for j=s_k, ..., t_k do avail(V_j) ← true
        MARKEDGESR(V_i,f)
        if failed then return
    /* systematic exploration of the cycles */
    dfs ← false
```

```
    for i=sk, ..., tk do avail(Vi) ← true
    for i=sk, ..., tk do if newingate(Vi) then
        MARKEDGESR(Vi,f)
        if failed then return
    /* depth-first search of the sub-graph traversed by external cycles, to detect cycles in it
(optional) */
    for i=sk, ..., tk do avail(Vi) ← true
    for i=sk, ..., tk do if avail(Vi) then
        DFS_EXTCYCLES(Vi)
        if failed then return
    /* new become old */
    for i=sk, ..., tk do
        newoutgate(Vi) ← false
        newingate(Vi) ← false
end

/* Finds and marks the edges of the paths in the current SCC of level k, from v to the new out-
gates, without traversing previous vertices in the current path. If no such path exists, v remains
unavailable and parameter "exitfound" becomes false. In case of preliminary depth-first search,
all the vertices visited remain unavailable. */
procedure MARKEDGESR(v,exitfound)
begin
    current(v) ← true  /* "v" is in the current path */
    backedge ← false  /* "backedge" is a local variable */
    exitfound ← newoutgate(v)  /* "exitfound" is an output parameter */
    avail(v) ← false
    for w∈ Adj(v) such that sk≤index(w)≤tk ∧ ¬failed do
        if current(w) then
            if inextcycle(v,w) then
                backedge ← true
                if exitfound then failed ← true  /* there is a cycle in the sub-graph
                                                    traversed by external cycles*/
        else if avail(w) then
            MARKEDGESR(w,f)
            if f then
                exitfound ← true
                if backedge then failed ← true /* there is a cycle in the sub-graph traversed
                                                  by external cycles*/
                inextcycle(v,w) ← true
                if critingate(w) then
                    critingate(w) ← false
                    critingates ← critingates-1
                    if critingates=0 then failed ← true
                if critoutgate(v) then
                    critoutgate(v) ← false
                    critoutgates ← critoutgates-1
                    if critoutgates=0 then failed ← true
    if exitfound ∧ ¬dfs ∧ ¬failed /*otherwise unnecessary*/ then MAKE_AVAIL(v)
    current(v) ← false
end
```

```
/* Makes v available, as well as its unavailable predecessors that are not on the current path. */
procedure MAKE_AVAIL(v)
begin
    avail(v) ← true
    for w ∈ Adj^-1(v) such that s_k≤index(w)≤t_k do
        if ¬avail(w) ∧ ¬current(w) then MAKE_AVAIL(w)
end

/* Depth-first search of the sub-graph (of the current SCC of level k) traversed by external
cycles to detect cycles in it. Returns immediately on detection of a cycle. */
procedure DFS_EXTCYCLES(v)
begin
    current(v) ← true /* "v" is in the current path */
    avail(v) ← false
    for w ∈ Adj(v) such that s_k≤index(w)≤t_k ∧ inextcycle(v,w) ∧ ¬failed do
        if current(w) then failed ← true
        else if avail(w) then DFS_EXTCYCLES(w)
    current(v) ← false
end
```

Appendix E - Additional performance results

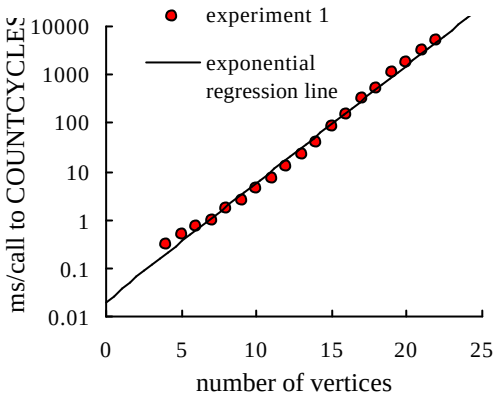


Fig. E1 Average performance of the procedure used to count the cycles, as a function of the number of vertices, in experiment 1.

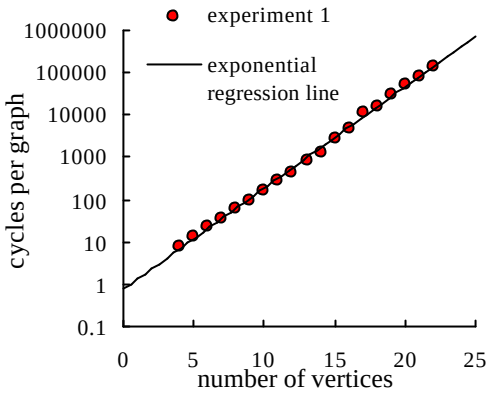


Fig. E2 Number of cycles per graph, as a function of the number of vertices, in experiment 1.

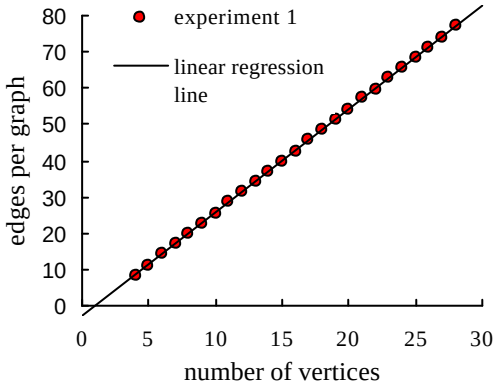


Fig. E3 Number of edges per graph, as a function of the number of vertices, in experiment 1.

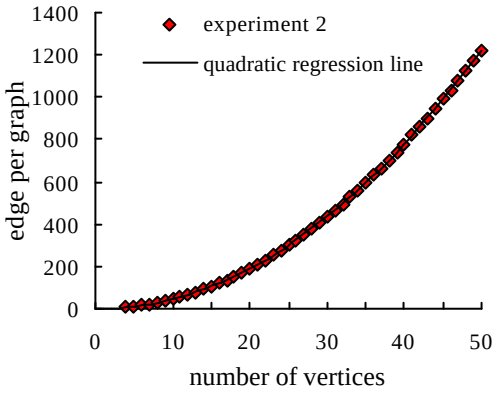


Fig. E4 Number of edges per graph, as a function of the number of vertices, in experiment 2.

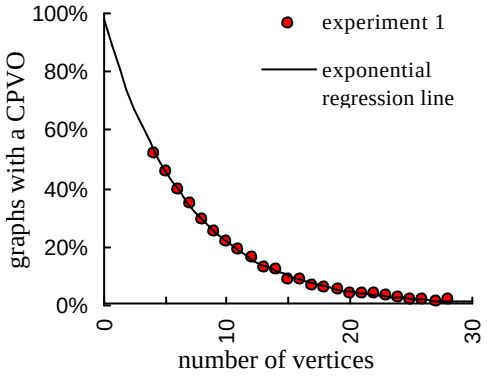


Fig. E5 Percentage of graphs with a CPVO, as a function of the number of vertices, in experiment 1.

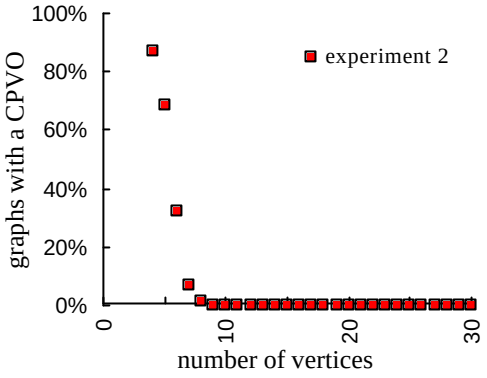


Fig. E6 Percentage of graphs with a CPVO, as a function of the number of vertices, in experiment 2.

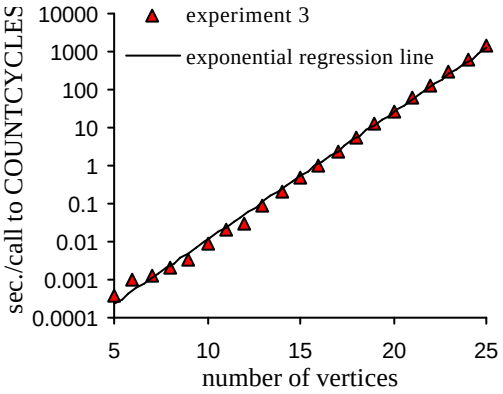


Fig. E7 Performance of the procedure used to count the cycles, as a function of the number of vertices, in experiment 3.

ANEXO 2

"On the equivalence of vertex orderings modulo the number of backward edges per cycle"

On the equivalence of vertex orderings modulo the number of backward edges per cycle

João Carlos Pascoal Faria

Abstract: It is analyzed the equivalence (called *α-equivalence*) between orderings of the vertices of a directed graph *G=(V,E)* modulo the number of "backward" edges induced in each cycle of *G*. α-equivalence is also defined with respect to undirected graphs by fixing an arbitrary orientation to each cycle. It is shown that two orderings of the vertices of a directed graph (undirected graph, respectively) are α-equivalent if and only if one of them may be obtained from the other by rotation or by interchanging consecutive vertices not joined by a cyclic edge (by any edge, respectively). Edge transformations (insertions and removals) that preserve α-equivalence between vertex orderings are identified. Using an undirected graph to represent the relationship "don't commute" between equations in a system of equations solved iteratively, it is shown that the maximum number of iterations required by different α-equivalent orderings differ at most by a number of iterations equal to the distance between any two connected equations.

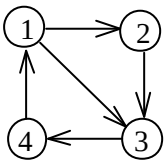
1. Definitions

Definition 1 *(backward and forward edges)*: Given an ordering *L* of the vertices of a directed graph *G=(V, E)*, any edge *(u, v)* of *G* such that *u* does not precede *v* under *L*, is called a *backward edge* induced by *L* in *G*; any other edge is called a *forward edge* induced by *L* in *G*.

We are not interested in the backward edges themselves, but rather in the number of backward edges induced in each cycle of *G* (also referred to as *number of backward edges per cycle*).

Example 1: With respect to the following directed graph, the following table shows all the vertex orderings that start in vertex 1, and the number of backward edges induced by them in each cycle of *G*. Other orderings may be obtained from the ones shown by rotation.

G:



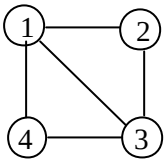
	vertex orderings and number of backward edges per cycle			
cycles	(1, 2, 3, 4)	(1, 2, 4, 3) (1, 4, 2, 3)	(1, 3, 2, 4) (1, 3, 4, 2)	(1, 4, 2, 3)
C_1 = (1, 3, 4, 1)	1	2	1	2
C_2 = (1, 2, 3, 4, 1)	1	2	2	3

Backward edges per cycle may also be defined for undirected graphs by fixing an arbitrary orientation to each cycle.[1] Notice that the same edge may get different orientations in different cycles.

Example 2: With respect to the following undirected graph, the table shows all the vertex orderings that start in vertex 1, and the number of backward edges induced by them in each cycle of G_u. Other orderings may be obtained from the ones shown by rotation.

[1] Alternatively, we could deal with undirected graphs by regarding each undirected edge as a pair of symmetric directed edges. However, for the purpose of comparing vertex orderings with respect to the number of backward edges they induce in each cycle of a graph, the two approaches are equivalent, as lemma 1 will show.

G_U:



	vertex orderings and number of backward edges per cycle			
cycles (clockwise oriented)	(1, 2, 3, 4)	(1, 2, 4, 3) (1, 4, 2, 3)	(1, 3, 2, 4) (1, 3, 4, 2)	(1, 4, 2, 3)
C_1 = (1, 3, 4, 1)	1	2	1	2
C_2 = (1, 2, 3, 4, 1)	1	2	2	3
C_3 = (1, 2, 3, 1)	1	1	2	1

Definition 2 *(α-equivalence)*: Two orderings L and L' of the vertices of a graph $G=(V,E)$, directed or undirected, are *α-equivalent* if they induce the same number of backward edges on any cycle of G.

For instance, in examples 1 and 2, vertex orderings in the same column are α-equivalent.

Definition 3 *(α-operation)*: An operation that transforms any vertex ordering L into a α-equivalent vertex ordering L' is called a *α-operation.*

Definition 4 (*β-equivalence)*: Two graphs $G=(V,E)$ and $G'=(V,E')$, with the same vertex set, are *β-equivalent* if any two vertex orderings L and L' that are α-equivalent with respect to one of the graphs are also α-equivalent with respect to the other graph.

For instance, the graphs of example 1 and 2 are β-equivalent.

Definition 5 (*β-operation)*: An operation that transforms any graph $G=(V,E)$ into a β-equivalent graph $G'=(V,E')$ is called a *β-operation*.

2. β-equivalence and β-operations

Lemma 1 *(β-equivalence between undirected and directed graphs)*: Any undirected graph $G_u=(V, E_u)$ is β-equivalent to the directed graph $G_d=(V, E_d)$ obtained from G_u by replacing each undirected edge of G_u by a pair of symmetric edges.

Proof: Cycles of length 1 or 2 have one backward edges for any vertex ordering. Consequently, they need not be considered to compare vertex orderings with respect to the number of backward edges per cycle.

The cycles of G_u and G_d with length greater than 2 are related in the following way: each cycle C of G_u with length >2 corresponds to a pair of symmetric cycles C_1 and C_2 in G_d. In order to count the number of backward edges induced in C, we have to choose an arbitrary orientation, say the one defined by C_1. Let l be the length of C, C_1 or C_2, and let f_1 and f_2 be the number of backward edges induced in C_1 and C_2 by a vertex ordering L. It's obvious that $f_1=l-f_2$. Any two vertex orderings that induce the same f_1, also induce the same f_2. Hence, it suffices to consider one cycle from each pair of symmetric cycles to compare vertex orderings with respect to the number of backward edges per cycle.

Since the cycles that need to be considered to compare vertex orderings with respect to the number of backward edges per cycle are the same in G_d and G_u, they are β-equivalent. Δ

Lemma 2 *(β-operations)*: Given a graph $G=(V, E)$, directed or undirected, the following edge insertion and edge removal operations are β-operations on G:

i) add or remove a self-loop [2];
ii) add or remove a parallel edge [3];
iii) add or remove an edge that is not contained in any cycle of G of length greater than 2;

[2] A self-loop is an edge with the same initial and terminal vertex [TS92].
[3] Parallel edges are edges that have the same pair of initial and terminal vertices [TS92].

iv) (for directed graphs only) add or remove the symmetric of an edge that is contained in a cycle of G other than the cycle defined by the two symmetric edges. [4]

Proof:

i) A self-loop has one backward edge with respect to any vertex ordering and it does not contribute to any other cycle. Hence, it may be ignored when comparing vertex orderings with respect to the number of backward edges per cycle.

ii) Let e_1 and e_2 be two parallel edges. First consider the case of a directed graph. For every cycle C_1 containing e_1 there is another cycle C_2 with e_1 replaced by e_2, with the same number of induced backward edges. Hence, it suffices to consider the cycles that contain e_1 to compare vertex orderings with respect to the number of backward edges per cycle. In the case of an undirected graph, there is an additional cycle to be considered: the cycle constituted by e_1 and e_2. This cycle has length 2 and has one backward edge with respect to any vertex ordering. Hence, it may be ignored when comparing vertex orderings with respect to the number of backward edges per cycle.

iii) In the case of an edge that is not contained in any cycle of G, the result is obvious, since such an edge is not important to count the number of backward edges per cycle. Cycles of length 1 or 2 have one backward edge with respect to any vertex ordering, and may also be ignored when comparing vertex orderings with respect to the number of backward edges per cycle.

iv) Let (u, v) be an edge that is contained in a cycle of G. Without loss of generality, assume that G does not contain the edge (v, u), and let G' be the graph obtained from G by adding the edge (v, u).

Let $p_1, p_2, ..., p_k$ be all the simple paths in G with initial vertex u and terminal vertex v. One of such paths, say p_1, is the edge (u,v) itself. Hence, $k \geq 1$.

Let $q_1, q_2, ..., q_l$ be all the simple paths in G with initial vertex v and terminal vertex u. Since (u, v) is contained in a cycle of G, there must exist at least one of such paths. Hence, $l \geq 1$.

Let L and L' be any two α-equivalent vertex orderings.

Let d be the variation on the number of backward edges of path p_1 (the edge (u, v)) from L to L'. The possible values of d are: 0, in case u and v have the same relative positions; 1, in case u moves after v; and -1, in case v moves after u.

The concatenation of any of the p_i's ($1 \leq i \leq k$) with any of the q_j's ($1 \leq j \leq l$) is a closed path, possibly with repeated vertices and edges. It's known that any closed path is a cycle or a combination of cycles, in the sense that the closed path may be built by starting with a single cycle, inserting a new cycle in some position of the path obtained so forth, and so on. Since L and L' induce the same number of backward edges in any cycle of G, they also induce the same number of backward edges in any closed path in G.

Applying this fact to any of the closed paths obtained by concatenating p_1 with any of the q_j's ($1 \leq j \leq l$), we conclude that the variation on the number of backward edges of any of the q_j's, from L to L', must be $-d$. Applying the same fact to any of the closed paths obtained by concatenating any of the p_i's with any of the q_j's, we conclude that the variation on the number of backward edges of any of the p_i's, from L to L', must be $+d$.

Consider now the graph G' with the additional edge (v, u). The variation on the number of backward edges induced on this edge, from L to L', is obviously $-d$. Hence, the number of backward edges induced in any of the new cycles of G', obtained by the concatenation of the edge (v, u) with any of the p_i's ($1 \leq i \leq k$), remains unchanged from L to L' ($-d$ due to the new edge, plus $+d$ due to any of the p_i's).

We conclude that, if two vertex orderings L and L' are α-equivalent with respect G, they are also α-equivalent with respect to G'. The converse is trivially true, because the cycles of G are a subset of the cycles of G'. Δ

Theorem 1 *(general b-equivalence)*: Two graphs $G=(V, E)$ and $G'=(V, E')$, directed or undirected, are β-equivalent if and only if any two vertices u and v that are joined by an edge contained in a cycle of length > 2 in one of the graphs are also joined by an edge contained in a cycle of length > 2 (not necessarily the same) in the other graph.

Proof:

First consider the case where both of the graphs are directed.

[4] Two edges are symmetric if the initial vertex of one of them is the terminal vertex of the other, and vice-versa.

Sufficiency: Assume that G and G' are two graphs with the same pairs of vertices in the conditions described. We first reduce G and G' to a "canonic form". We remove all the self-loops, parallel edges (leaving only one edge from each group of parallel edges) and edges not contained in cycles of length > 2. Then, for any edge (u, v), necessarily cyclic, without a symmetric edge (v, u), we add the edge (v, u). Let us name G_r and G'_r, respectively, the resulting graphs. According to lemma 2, G and G_r (as well as G' and G'_r) are β-equivalent. It's obvious that the pairs of vertices in the conditions described in the theorem statement are not affected. Since we assumed that G and G' have the same pairs of vertices in those conditions, the reduced graphs G_r and G'_r are identical and, consequently, β-equivalent. Hence, G and G' are β-equivalent also (β-equivalence is transitive).

Necessity: Let u and v be two vertices of G joined by an edge (u, v) contained in a cycle C of length > 2. Let L be a vertex ordering with u immediately preceding v. Let L' be the vertex ordering obtained from L by interchanging these two vertices. L' will induce one more backward edge in C than L. Hence, L and L' are not α-equivalent with respect to G. Notice that if C had length 1 (in case of a self-loop) or 2 (in case of two symmetric edges), L' would induce the same number of backward edges in C as L. Now assume that in G' the vertices u and v are not joined by an edge (u, v) or (v, u) contained in a cycle C' of G' with length > 2. However, u and v may be joined by edges belonging to a cycle of length 2 (two symmetric edges). A cycle of length 2 has always one backward edge with respect to any vertex ordering. When u and v are interchanged, no other cycles are affected. Hence, L and L' are α-equivalent with respect to G'. Since L and L' are α-equivalent with respect to G' but not with respect to G, these two graphs are not β–equivalent. Consequently G and G' are not β–equivalent either.

Now consider the case where one or both graphs are undirected. Let G_d and G'_d be the directed graphs obtained from G and G' by replacing any undirected edge by a pair of symmetric directed edges. It's obvious that G and G_d (as well as G' and G'_d) have the same pairs of vertices in the conditions described in the theorem statement. According to lemma B1, G and G_d (as well as G' and G'_d) are β-equivalent. As proved above, G_d and G'_d are β-equivalent if and only if they have the same pairs of vertices in the conditions described in the theorem statement. Hence, we may conclude that G and G' are β-equivalent if and only they have the same pairs of vertices in the conditions described in the theorem statement. Δ

3. α-equivalence and α-operations

Lemma 3 *(α-operations for directed graphs)*: The following are α-operations on the vertex orderings of a directed graph $G=(V, E)$:

i) rotation;

ii) interchanging pairs of consecutive vertices that are not joined by cyclic edges of G. [5]

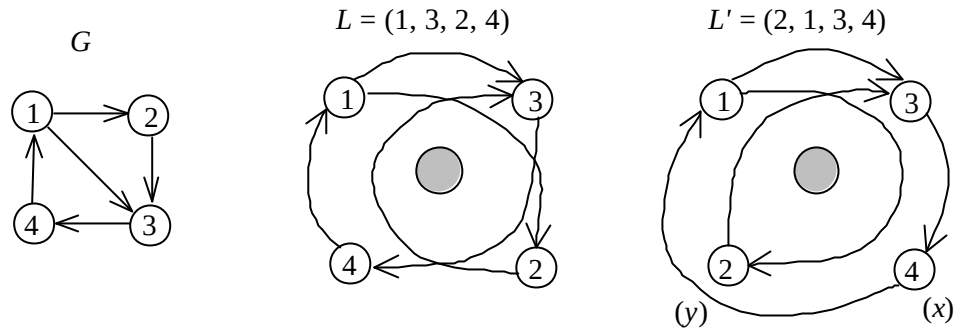
Proof:

i) This is already presented in [RSS90], in a different context. Let L' be a vertex ordering obtained from a vertex ordering L by rotation. Let's regard L and L' as vertex sequences. To rotate L is the same as to interchange two sub-sequences L_1 and L_2 into which L is divided. The orientation (forward or backward) of the edges of the cycles that intersect vertices in only one of these sub-sequences does not change, and, hence, the number of backward edges does not change either. Cycles that intersect vertices in L_2 and vertices in L_1, must have the same number of edges oriented from L_1 to L_2, as in the opposite direction. When L_1 and L_2 are interchanged, all such edges have their orientation reversed. Hence, the number of backward edges added equals the number of backward edges subtracted, and the total number of backward edges remains unchanged.

ii) Interchanging vertices in the conditions described does not affect the orientation of the edges that are important to count the number of backward edges per cycle. Δ

We next present an algorithm to transform an ordering L' of the vertices of a strongly connected directed graph $G=(V,E)$ into a given α-equivalent ordering L, by applying only the operations mentioned in Lemma 3. The algorithm is based on a circular representation of vertex orderings, as shown below for two orderings of the vertices of the strongly connected graph of example B1.

[5] An edge of a graph G is said to be a *cyclic edge* (or *circuit edge*) if there exists a cycle in G containing the edge [TS92].



The method is also based on the notion of *circular length* of a path P under a vertex ordering L, denoted *Len(P,L)*, defined as the length of P in the circular representation, measured in number of vertices. For instance, with respect to the above example, the path P = (4, 1, 2) has circular length 3 under L and circular length 5 under L'.

Algorithm 1 *(transform an ordering L' of the vertices of a strongly connected directed graph G=(V,E) into a given α-equivalent ordering L, by applying only the operations mentioned in Lemma B3*):

1. Repeat until L' and L differ at most by a rotation:
   - 1.1. Pick two vertices x and y obeying the following conditions:
     - 1.1.1. x and y are circularly consecutive vertices in L' (i.e., x is followed by y in the circular representation of L');
     - 1.1.2. x and y are not joined by any edge of G; (Note: in fact, this is a consequence of condition 1.1.3.)
     - 1.1.3. $Len(P_{xy},L) < Len(P_{xy},L')$, where P_{xy} is an arbitrary path from x to y in G. (Note: if this inequality holds for one simple path P_{xy}, it also holds for any other simple path from x to y, reason why the choice of P_{xy} is arbitrary.)
   - 1.2. Modify L' by interchanging x and y. (Note: in the linear representation, if x is the first vertex and y is the last vertex of L', this is accomplished by a combination of operations i) and ii) of Lemma B3).
2. Rotate L' to obtain L. Δ

Next we present some lemmas in order to prove the correctness of algorithm 1.

Lemma 4: For any closed path C (simple or not) in a directed graph $G=(V,E)$, and any two α-equivalent vertex orderings L and L', $Len(C, L) = Len(C, L')$.

Proof: A closed path is a cycle or a combination of cycles. Since L and L' induce the same number of backward edges on every cycle of G, they must also induce the same number of backward edges on every closed path in G. Let b be the number of backward edges induced by L or L' on C, and let n be the number of vertices of G. Then $Len(C, L) = nb = Len(C, L')$. Δ

Lemma 5: In case L and L' differ by more than a rotation, there must exist vertices x and y in the conditions mentioned in algorithm 1.

Proof:

Let $p_1, p_2, ..., p_n$ be simple paths that connect each two circularly consecutive vertices in L', where n is the number of vertices in G. These paths exist because we assume G is strongly connected. Let C be the concatenation of these simple paths. Obviously, C constitutes a closed path. According to lemma B4, $Len(C,L) = Len(C,L')$. It's also obvious that $Len(C,L)=Len(p_1,L)+...+ Len(p_n,L)$, and the same is true under L'. If, for any p_i $(1 \le i \le n)$, $Len(p_i,L) = Len(p_i,L')$, then L and L' would differ at most by a rotation. Consequently, there must exist some p_i $(1 \le i \le n)$ with $Len(p_i,L) \neq Len(p_i,L')$. Since $Len(p_1,L)+...+ Len(p_n,L) = Len(p_1,L')+...+ Len(p_n,L')$, there must exist some p_i $(1 \le i \le n)$ with $Len(p_i,L) < Len(p_i,L')$ and some p_j $(1 \le j \le n)$ with $Len(p_j,L) > Len(p_j,L')$. So, there is a simple path p_i with $Len(p_i,L) < Len(p_i,L')$. Let x be its initial vertex and let y be its terminal vertex.

We now prove that, for any simple path P_{xy} from x to y, $Len(P_{xy},L) < Len(P_{xy},L')$ and, for any simple path P_{yx} from y to x, $Len(P_{yx},L) > Len(P_{yx},L')$. For any simple path P_{yx} from y to x, the concatenation of P_{yx} with p_i is a closed path. By lemma B4, $Len(P_{yx},L) + Len(p_i,L) = Len(P_{yx},L') + Len(p_i,L')$. Since $Len(p_i,L) < Len(p_i,L')$, we get $Len(P_{yx},L) > Len(P_{yx},L')$. For any simple path P_{xy} from x to y (including p_i), the concatenation of P_{xy} with any simple path P_{yx} from y to x is a closed path. By lemma B4, $Len(P_{yx}, L) + Len(P_{xy}, L) = Len(P_{yx}, L') + Len(P_{xy}, L')$. Since $Len(P_{yx}, L) > Len(P_{yx}, L')$, we get $Len(P_{xy}, L) < Len(P_{xy}, L')$.

We now prove that vertices x and y are not adjacent in G. If there exists the edge (x, y), the path $P=(x, y)$ would have $Len(P, L) \geq Len(P, L') = 1$, which contradicts the hypothesis about x and y. If there exists the edge (y, x), the path $P=(y, x)$ would have $Len(P, L) \leq Len(P, L') = n-1$, which contradicts the hypothesis about x and y. Δ

Lemma 6: Algorithm 1 terminates.

Proof: By interchanging vertices x and y in L', we get a new ordering L'' that is closer to L. Let n be the number of vertices. To measure the distance between two α-equivalent vertex orderings, L and L', we use the circular length modulo n, that is, the number of completed turns of each path. For any simple path P_{xy} from x to y, we get $Len(P_{xy},L) \bmod n \leq Len(P_{xy},L'') \bmod n = Len(P_{xy},L') \bmod n - 1$. For any simple path P_{yx} from y to x, we get $Len(P_{yx},L) \bmod n \geq Len(P_{xy},L'') \bmod n = Len(P_{xy},L') \bmod n + 1$. Finally, for any other simple path P, we get $Len(P,L'') \bmod n = Len(P,L') \bmod n$. These imply a finite bound on the number of iterations of step 1 of algorithm 1. Δ

Algorithm 1 is easily extended to non strongly connected graphs. There is no difficulty in reordering the vertices of one strongly connected component of G at a time. For each strongly connected component S, in order to select the vertices x and y to exchange, we ignore all the vertices and edges that do not belong to S. If, between x and y, there are some intermediate vertices not belonging to S, it is necessary to move x and y together before interchanging them, which can be accomplished by interchanging x or y with one (the nearest) intermediate vertex at a time. In this case, the vertices interchanged are not joined by a cyclic edge of G, because they belong to different strongly connected components. After reordering all the strongly connected components, there is no difficulty in putting all the vertices in their final positions (as in L), by interchanging only pairs of consecutive vertices that are not joined by cyclic edges of G.

Hence, we conclude:

Theorem 3 *(complete set of α-operations for directed graphs)***:** Two orderings of the vertices of a directed graph $G=(V,E)$, L and L', are α-equivalent if and only if any of them may be obtained from the other by repeatedly applying the following operations:

i) rotation;

ii) interchanging pairs of consecutive vertices that are not joined by cyclic edges of G. [6]

Besides that, none of these two operations is sufficient by itself.

Proof: *Sufficiency*: If L and L' may be obtained from each other by applying these operations, then they are α-equivalent, according to lemma B3.

Necessity: If L and L' are α-equivalent, then one of them may be obtained from the other by applying these operations, according to algorithm 1.

To show that none of the two operations is sufficient by itself, we give two examples. Consider a complete graph with n vertices. In this case, operation ii) cannot be applied, and operation i) is necessary. Consider a graph with n vertices and without any edge (all the vertices are isolated vertices). In this case, all the vertex orderings are α-equivalent, but they cannot be obtained from each other by rotation only. Hence, operation ii) is also necessary. Δ

This can be translated for undirected graphs.

[6] An edge of a graph G is said to be a *cyclic edge* (or *circuit edge*) if there exists a cycle in G containing the edge [TS92].

Theorem 4 *(complete set of α-operations for undirected graphs)***:** Two orderings of the vertices of an undirected graph $G_u=(V, E_u)$, L and L', are α-equivalent if and only if any of them may be obtained from the other by repeatedly applying the following α-operations:

i) rotation;

ii) interchanging pairs of consecutive vertices that are not adjacent in G_u.

Besides that, none of these two operations is sufficient by itself.

Proof: Let G_d be the directed graph obtained from G_u by replacing each undirected edge of G_u by a pair of symmetric directed edges. Let L and L' be two α-equivalent vertex orderings with respect to G_u. According to lemma B1, L and L' are still α-equivalent with respect to G_d. According to theorem B2, L may be obtained from L by rotation or by interchanging pairs of consecutive vertices that are not joined by cyclic edges of G_d. To say that two vertices are not joined by cyclic edges of G_d is the same as to say that those vertices are not adjacent in G_u. Δ

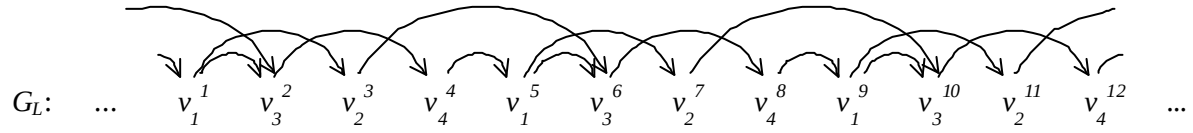
4. Iterated vertex sequences

Assume we represent vertex orderings as vertex sequences.

Theorem 5: Let L and L' be any two α-equivalent orderings of the vertices of a directed graph $G=(V,E)$, represented as vertex sequences. Let D be the maximum distance between any two strongly connected vertices in G, or 0 in case there are no strongly connected vertices in G. Then, there exist vertex sequences A and B, and integer h, with $0 \pounds h \pounds D$, such that, for any natural N, the vertex sequence L'^{N+h} (L' iterated $N+h$ times) may be transformed into the vertex sequence AL^NB by repeatedly interchanging consecutive vertices that are not joined by cyclic edges of G.

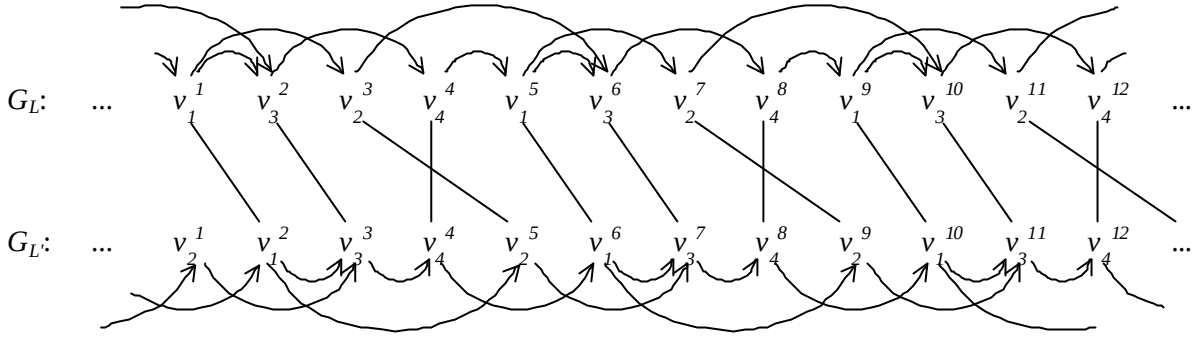
Proof:

1. We will use a method similar to the one exposed in the proof of the previous theorem to transform L' into an ordering L^* that differs from L at most by a rotation, by interchanging pairs of circularly consecutive vertices that are not joined by cyclic edges of G. At the same time, we give a method to transform the vertex sequence L'^{N+h} into a vertex sequence of the form AL^NB, by interchanging pairs of consecutive vertices (not circularly consecutive) that are not joined by cyclic edges of G.
2. Instead of the circular representation used in the previous theorem, we use "iterated graphs" defined as follows:
   2.1. Given G and L, we define a corresponding (acyclic) iterated graph $G_L=(V_L, E_L)$ as follows:
      2.1.1. To each vertex $v_i \hat{I} V$, with position p_i in L ($1 \pounds p_i \pounds n$), there correspond vertices $v_i^{p_i+kn}$ in V_L, with k integer (the iteration number). We use superscripts to number the vertices sequentially in G_L.
      2.1.2. To each cyclic edge (v_i, v_j) of G, there correspond edges $(v_i^{p_i+kn}, v_j^{p_j+kn})$ in E_L, in case $p_i<p_j$, or $(v_i^{p_i+kn}, v_j^{p_j+(k+1)n})$, in case $p_i>p_j$, with k integer. Each of these edges connects an occurrence of vertex v_i to the nearest following occurrence of vertex v_j in G_L. Consequently, for any edge $(v_i^a, v_j^b) \hat{I} E_L$, we have $1 \pounds b-a \pounds n$ ($b-a=n$ in case $v_i=v_j$, that is, in the case of a self-loop in G).
      2.1.3. Edges of G that are not contained in any cycle have no corresponding edges in G_L.
   2.2. For instance, with respect to the example already used in theorem B2, we get (with v_i instead of i):



   2.3. Given G and L', we define $G_{L'}$ in the same way.
3. Next, we establish a one-to-one correspondence between the vertices and edges of G_L and $G_{L'}$, as follows:
   3.1. To each vertex v_i^k in G_L there corresponds a vertex $v_i^{k+d_i}$ in $G_{L'}$. Hence, d_i ($1 \pounds i \pounds n$) is the displacement between corresponding occurrences of v_i from G_L to $G_{L'}$.
   3.2. To each edge (v_i^k, v_j^l) in G_L there corresponds an edge $(v_i^{k+d_i}, v_j^{l+d_j})$ in $G_{L'}$.

3.3. The following figure shows a possible correspondence for the example given above, with $d_1=1$, $d_2=2$, $d_3=1$, and $d_4=0$. Only the vertex correspondence is graphically represented.



3.4. We have to prove that it is always possible to establish such a correspondence.

3.4.1. Since the acyclic edges of G are ignored, and those edges are precisely the ones that are not contained in strongly connected components of G, we may establish a correspondence between G_L and $G_{L'}$ independently for each strongly connected component of G.

3.4.2. So, let S_x be a strongly connected component of G.

3.4.3. Let us choose an arbitrary path P in S_x, containing all its edges and vertices, possibly with repetitions. With respect to the foregoing example, we may choose:

$P = (v_1, v_2, v_3, v_4, v_1, v_3)$

3.4.4. Let P_L be a corresponding path in G_L that traverses vertices with the same subscripts as in P, starting in an arbitrary occurrence of the first vertex of P in G_L. With respect to the foregoing example, we may choose:

$P_L = (v_1^1, v_2^3, v_3^6, v_4^8, v_1^9, v_3^{10})$.

3.4.5. Let q_i, $1 \pounds i \pounds n$, be the superscript of the first occurrence of vertex v_i in P_L. For instance, with respect to the above P_L, we have $q_1=1$, $q_2=3$, $q_3=6$ and $q_4=8$.

3.4.6. We now prove that the superscripts of further occurrences of vertex v_i in P_L, $1 \pounds i \pounds n$, may be computed as a function of q_i and the number of backward edges per cycle induced by L in G (the complete knowledge of L is not required). The sub-path of P that goes from the first occurrence of vertex v_i to another occurrence of the same vertex in P, is a closed path. It is known that a closed path is a cycle or a combination of cycles. Summing up the number of backward edges induced by L in each of the composing cycles, we get the number of backward edges induced by L in the closed path. Multiplying by n (the number of vertices in G), we get the difference between the superscripts of the corresponding vertices in P_L, because a backward edge in G corresponds to an edge in G_L that crosses the barrier between two iterations. With respect to the foregoing example, we may rewrite

$P_L = (v_1^{q_1}, v_2^{q_2}, v_3^{q_3}, v_4^{q_4}, v_1^{q_1+2n}, v_3^{q_3+n})$, with $q_1=1$, $q_2=3$, $q_3=6$, and $q_4=8$,

because the cycle $(v_1, v_2, v_3, v_4, v_1)$ has two backward edges induced by L - the edges (v_2, v_3) and (v_4, v_1), and the cycle (v_3, v_4, v_1, v_3) has one backward edge induced by L - the edge (v_4, v_1).

3.4.7. Let $P_{L'}$ be defined in a way similar to P_L, with respect to L' instead of L. Since L' induces the same number of backward edges per cycle as L, $P_{L'}$ differs from P_L only in the way of fixing the values of the q_i's ($1 \pounds i \pounds n$). With respect to the foregoing example, we get

$P_{L'} = (v_1^2, v_2^5, v_3^7, v_4^8, v_1^{10}, v_3^{11})$

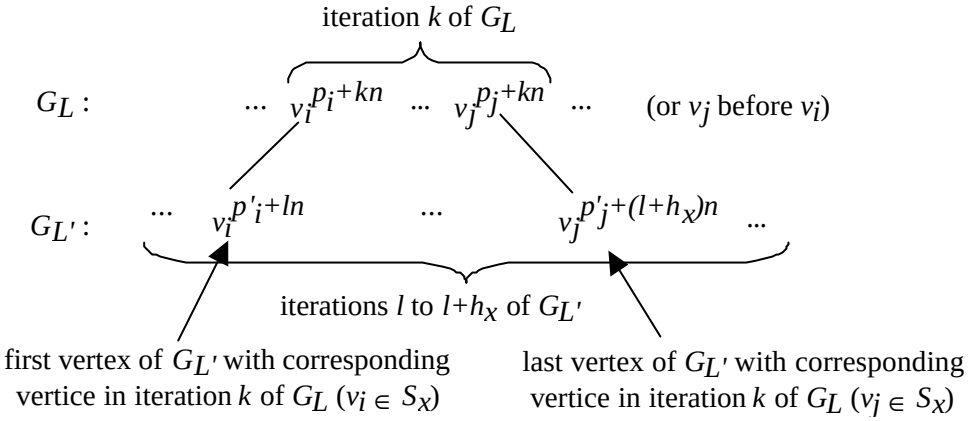
which is the same as

$P_{L'} = (v_1^{q_1}, v_2^{q_2}, v_3^{q_3}, v_4^{q_4}, v_1^{q_1+2n}, v_3^{q_3+n})$, with $q_1=2$, $q_2=5$, $q_3=7$, and $q_4=8$.

3.4.8. We choose d_i ($1 \pounds i \pounds n$) as the increment on the value of q_i from P_L to $P_{L'}$. With respect to the foregoing example, we get $d_1=1$, $d_2=2$, $d_3=1$ and $d_4=0$. Now we prove that this correspondence has the required properties. All the occurrences of edges of S_x in G_L are of the form (v_i^{k+an}, v_j^{l+an}), where (v_i^k, v_j^l) is an edge of P_L, a in an integer, and n is the number of vertices in G. The vertices v_i^{k+an} and v_j^{l+an} have corresponding vertices in $G_{L'}$ $v_i^{k+an+d_i}$ and $v_j^{l+an+d_j}$. The edge $(v_i^{k+d_i}, v_j^{l+d_j})$ is obviously contained in $P_{L'}$, and, hence, in

$G_{L'}$. Consequently, $(v_i^{k+an+d_i}, v_j^{l+an+d_j})$ is also an edge of $G_{L'}$. The same reasoning may be applied in the reverse direction.

4. We next show how to refine the above correspondence, so that the vertices in any iteration k of G_L will have corresponding vertices in iterations k to $k+h$ of $G_{L'}$, in a way that minimizes h. We also show that h does not exceed the maximum distance between any two strongly connected vertices of G (denoted as D in the theorem statement).
   4.1. Let S_x be a strongly connected component of G, and assume that its corresponding vertices in iteration k of G_L have corresponding vertices in iterations l to $l+h_x$ of $G_{L'}$ (see figure below).

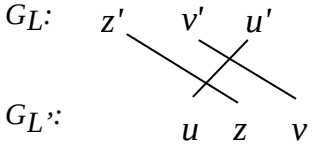


   4.2. We next show that h_x does not exceed the maximum distance between any two vertices of S_x (the diameter of S_x).[7]
   4.3. Let v_i and v_j be vertices of S_x, in the conditions shown in the above figure. Let P_{ij} be a simple path in G (or S_x) from v_i to v_j, with minimum length d_{ij} (the distance from v_i to v_j). Let $P_{ij,L}$ and $P_{ij,L'}$ be corresponding paths in G_L and $G_{L'}$, starting in corresponding occurrences of v_i, namely $v_i^{p_i+kn}$ and $v_i^{p'_i+ln}$, where p'_i is the position of vertex v_i in L' (see above figure). Let d and d' be the number of backward edges induced by L and L' in P_{ij}. Obviously, $0 \le d,d' \le d_{ij}$. The terminal vertices of $P_{ij,L}$ and $P_{ij,L'}$ will be $v_j^{p_j+(k+d)n}$ and $v_j^{p'_j+(l+d')n}$. Since these are corresponding vertices, as well as $v_j^{p_j+kn}$ and $v_j^{p'_j+(l+h_x)n}$ (see above figure), we get $h_x = d'-d$. Since $0 \le d,d' \le d_{ij}$, we get $h_x \le d_{ij}$ (and $h_x \ge 0$ by definition). The maximum value occurs when all the edges of P_{ij} are forward edges with respect to L, and backward edges with respect to L'.
   4.4. In case it does not occur already, there is no difficulty in redefining the correspondence between G_L and $G_{L'}$, for the vertices of S_x, so that $l=k$ (it suffices to shift one end of correspondence lines by an appropriate number of iterations).
   4.5. Doing the same for all the strongly connected components of G, we guarantee that the vertices in iteration k of G_L will all have corresponding vertices in iterations k to $k+h$ of $G_{L'}$, in a way that minimizes h. From the above discussion, we conclude that h does not exceed the maximum distance between any two strongly connected vertices of G. For instance, with respect to the foregoing example, we already got $h=1<D=2$.
   4.6. Notice that the correspondence established between G_L and $G_{L'}$ in this way is unique.

5. Assuming that L' differs from L by more than a rotation, we select and interchange two circularly consecutive vertices of L', obtaining a new ordering L'' that is closer to L, as explained next.
   5.1. If L' differs from L by more than a rotation, the correspondence lines between $G_{L'}$ and G_L cannot be all parallel to each other. The converse is also true.
   5.2. If those lines are not all parallel to each other, there must exist two lines that intersect each other and reach two vertices consecutive in $G_{L'}$, v_i^k and v_j^{k+1}, like vertices v_4^4 and v_2^5 in the foregoing example. Saying that the lines intersect each other, means that $k-d_i > k+1-d_j$, which is the same as $d_i < d_j-1$.
      5.2.1. Notice that there may exist pairs of lines that intersect each other and reach two vertices (say u and v) that are not consecutive in $G_{L'}$. However, in such a case, between those vertices, there must exist two consecutive vertices in the required conditions (such that

[7] Remember that the distance from u to v is the length (number of edges) of the shortest path from u to v.

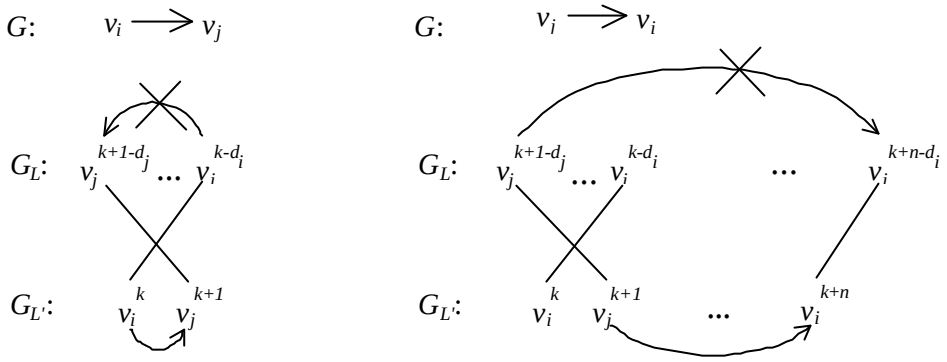
their lines intersect each other), as is illustrated in the following figure for vertices *u* and *z*.



5.3. We next prove by contradiction that, if there are vertices v_i^k and v_j^{k+1} in $G_{L'}$ in the above conditions ($d_i<d_j-1$), then there cannot exist any cyclic edge joining v_i and v_j in G.

5.3.1. Assume that there exists a cyclic edge (v_i, v_j) in G. Then, there would exist the edge (v_i^k, v_j^{k+1}) in $G_{L'}$ and the corresponding edge $(v_i^{k-d_i}, v_j^{k+1-d_j})$ in G_L. But such a vertex cannot exist in G_L because $k-d_i>k+1-d_j$.

5.3.2. A similar contradiction arises if we assume that there exists a cyclic edge (v_j, v_i) in G. Then there would exist the edge (v_j^{k+1}, v_i^{k+n}) in $G_{L'}$ and the corresponding edge $(v_j^{k+1-d_j}, v_i^{k+n-d_i})$ in G_L. But such a vertex cannot exist in G_L because $(k+n-d_i)-(k+1-d_j)>n$. Both of these situations are illustrated in the figure below.



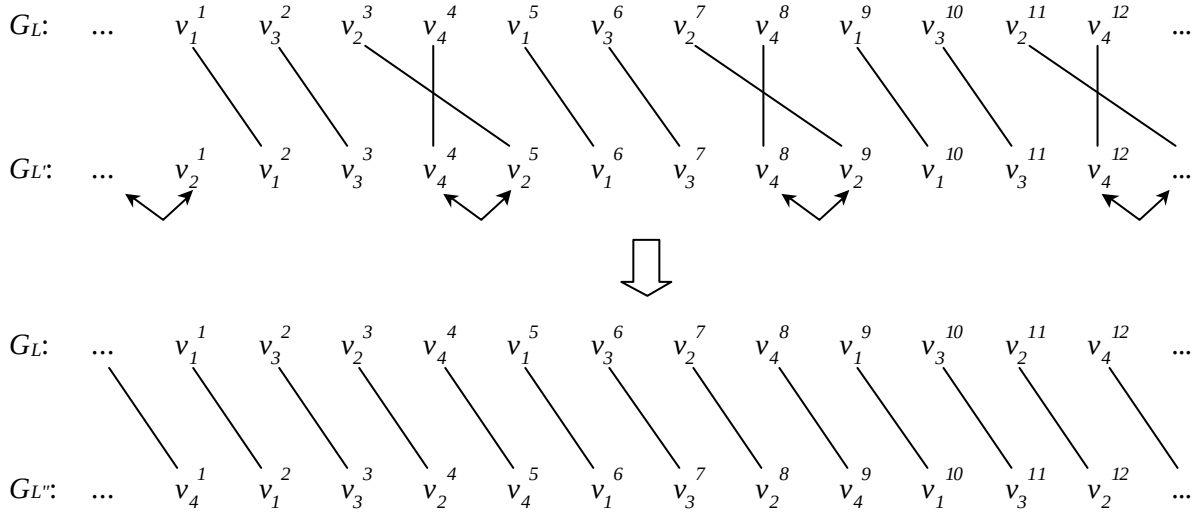
5.4. Having found vertices v_i^k and v_j^{k+1} in the required conditions, we exchange vertices v_i and v_j in L', getting a new vertex ordering L''. As we proved above, there is no cyclic edge connecting vertices v_i and v_j in G. Hence, L'' induces the same number of backward edges per cycle as L'.

5.5. With respect to the foregoing example, by exchanging vertices v_4 and v_2 in L', we get:

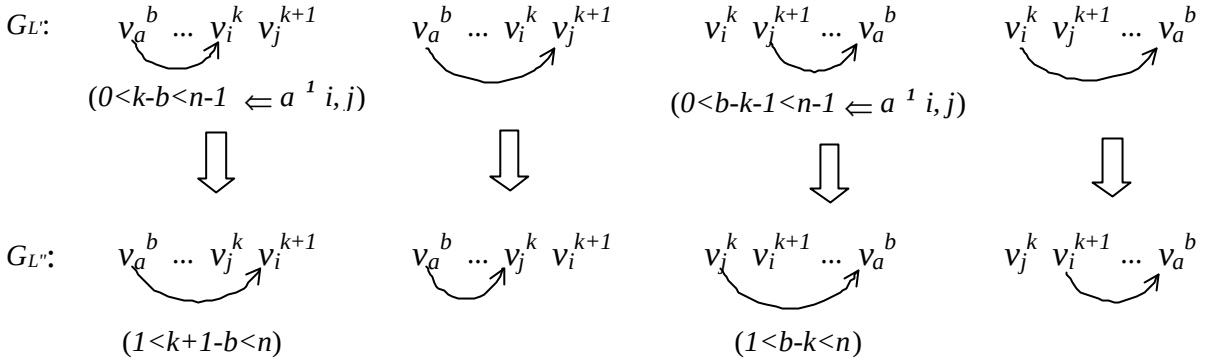
$L''=(v_4,v_1,v_3,v_2)$

6. We next show that the correspondence between $G_{L''}$ and G_L is obtained from the correspondence between $G_{L'}$ and G_L simply by incrementing d_i and decrementing d_j. In the pictorial representation, this means moving the lines that reach vertices v_i^k and v_j^{k+1} in $G_{L'}$ together with them when they are interchanged.

6.1.1. In the foregoing example, we get $d_1=1$, $d_2=2-1$, $d_3=1$, and $d_4=0+1$. The transformation is:



6.1.2. The fact that the new correspondence is valid with respect to the vertices, is fairly obvious. With respect to the edges, it suffices to notice that the edges of $G_{L''}$ may also be obtained from the edges of $G_{L'}$, simply be moving the ends of the edges incident on vertices v_i^k or v_j^{k+1} together with them, when the vertices are interchanged. The possible situations are illustrated in the figure below, assuming there are no self-loops. The resulting edges after this transformation are always valid.



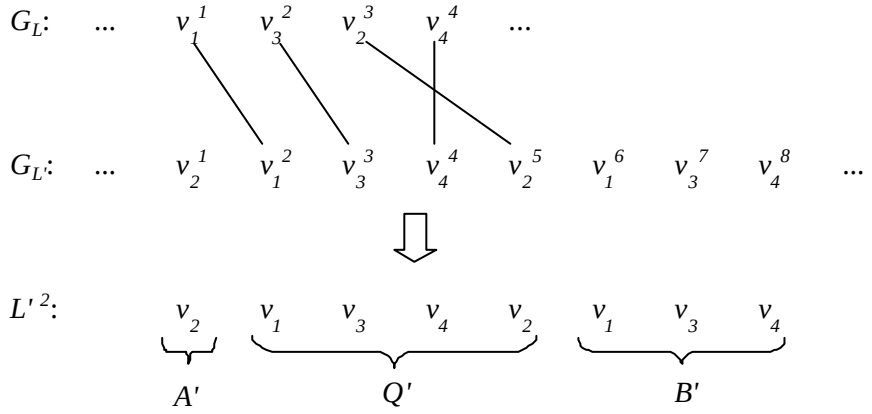
6.1.3. A situation that is not represented above has to do with self-loops, that originate edges (v_i^k, v_i^{k+n}) or (v_j^{k+1}, v_j^{k+1+n}) in $G_{L'}$. However, it's obvious that, after the transformation, the resulting edges (v_i^{k+1}, v_i^{k+1+n}) or (v_j^k, v_j^{k+n}) are valid (in $G_{L''}$).

6.1.4. Hence, the correspondence from G_L to $G_{L''}$ is the transitive combination of the correspondence from G_L to $G_{L'}$ plus the correspondence from $G_{L'}$ to $G_{L''}$.

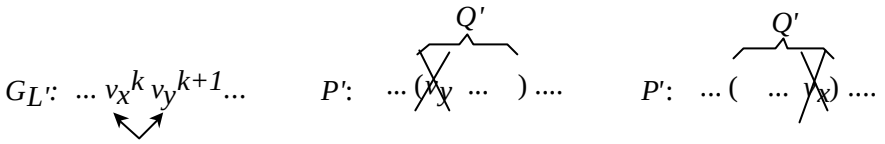
6.1.5. Notice, however, that the vertices in the iteration of G_L will not necessarily have vertices in a minimum number of iterations of $G_{L''}$.

6.2. Since, before interchanging the two vertices, we have $d_i \pounds d_j - 2$, after interchanging the vertices, d_i and d_j will be closer to each other. Hence, we get closer to the condition $d_1=d_2=\dots=d_n$, that corresponds to the situation where all the correspondence lines are parallel to each other.

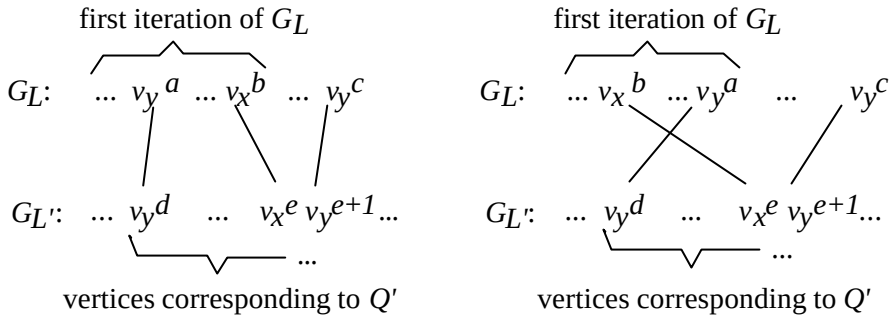
7. At the same time we transform L' and obtain L'', we also transform the initial vertex sequence L'^{N+h}. We rewrite this vertex sequence as $P' = L'^{N+h} = A'Q'B'$, where Q' corresponds to the vertices of $G_{L'}$ (without superscripts) comprised between the first and the last vertex of $G_{L'}$ that have corresponding vertices in the first N iterations of G_L. This vertex sequence will be transformed into a vertex sequence of the form $P'' = A''Q''B''$, where Q'' is defined in the same way as Q', with respect to $G_{L''}$ instead of $G_{L'}$. Since the first iteration of G_L is mapped to the first $1+h$ iterations of $G_{L'}$, the first N iterations of G_L are mapped to the first $N+h$ iterations of $G_{L'}$. For instance, with respect to the example given and for $N=1$, we get Q' as follows:



7.1. L'' was obtained from L' by interchanging two vertices v_x and v_y. We nest prove by contradiction that v_y cannot be the first element of Q' and v_x cannot be the last element of Q' (see figure below).

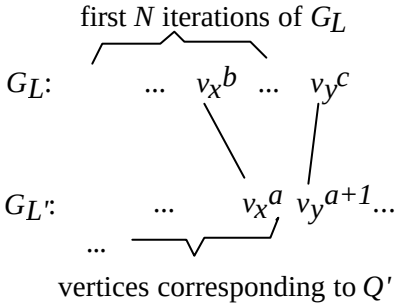


7.1.1. Assume v_y is the first element of Q'. Let v_y^a and v_x^b be the first occurrences of v_y and v_x in G_L, let v_y^d and v_x^e be the corresponding vertices in $G_{L'}$ (with $e>d$, because v_y^d is the first vertex of $G_{L'}$ with a corresponding vertex in the first iteration of G_L), let v_y^{e+1} be the vertex immediately after v_x^e in $G_{L'}$, and let v_y^c be the corresponding vertex in G_L (with $c>b$, because v_y^c cannot belong to the first iteration of G_L), as shown in the figure below for the two possible relative positions of v_y^a and v_x^b.



7.1.2. Hence, the correspondence lines that depart from v_x^e and v_y^{e+1} (consecutive occurrences of v_x and v_y) cannot intersect each other, and the pair $v_x\ v_y$ is not a valid candidate for interchanging.

7.1.3. Now assume that v_x is the last element of Q'. Let v_x^a be the last occurrence of v_x in $G_{L'}$ with a corresponding vertex in the first N iterations of G_L, let v_y^{a+1} be the next vertex in $G_{L'}$, and let v_x^b and v_y^c be the corresponding vertices in G_L, as shown in the figure below.



7.1.4. Since all the vertices in the first N iterations of G_L are already mapped to vertices that are not after v_x^a in $G_{L'}$, we conclude that $c>b$, and, again, the pair $v_x\ v_y$ is not a valid candidate for interchanging.

7.2. Since v_y cannot be the first element of Q' and v_x cannot be the last element of Q', all the occurrences of v_x and v_y in Q' exist in pairs that can be interchanged. After interchanging all the occurrences of v_x and v_y in Q', Q'' will be the resulting sequence, except possibly for the exclusion of the first and/or the last element in the following cases. If v_x is the first element of Q' and the following v_y is not mapped to an element of G_L, this occurrence of v_y is excluded from Q'' and included in A'' (we get $A'' = A'\ v_y$); otherwise, we get $A'' = A'$. If v_y is the last element of Q' and the preceding v_x is not mapped to an element of G_L, this occurrence of v_x is excluded from Q'' and included in B'' (we get $B'' = v_y\ B'$); otherwise, we get $B'' = B'$. The vertices in the first N iterations of G_L will still be mapped to vertices in the first $N+h$ iterations of $G_{L''}$, at most.

8. Vertices are interchanged as explained until there remain no intersecting lines. The final vertex ordering, say L^*, will differ from L at most by a rotation. At that point, we must have $Q^* = L^N$. Hence, we have transformed L^{N+h}, with $0 \pounds h \pounds D$, into a vertex sequence of the form AL^NB (with $A=A^*$, $B=B^*$ and $L^N=Q^*$) by interchanging pairs of consecutive vertices that are not joined by cyclic edge of G, as wanted. This concludes the proof. Δ

A similar theorem may be applied to undirected graphs.

Theorem 6 *(reordering vertex orderings for undirected graphs)*: Let L and L' be two α-equivalent orderings of the vertices of an undirected graph G_u, represented as vertex sequences. Let D be the maximum distance between any two connected vertices in G_u, or 0 in case G_u has no edges. Then, there are vertex sequences A and B, and integer h, with $0 \leq h \leq D$, such that, for any natural N, the vertex sequence L'^{N+h} (L' repeated $N+h$ times) may be transformed into the vertex sequence AL^NB by interchanging pairs of consecutive vertices that are not adjacent in G_u.

Proof: Let G_d be the directed graph obtained from G_u by replacing each undirected edge of G_u by a pair of symmetric directed edges. Let L and L' be two α-equivalent vertex orderings with respect to G_u. According to lemma B1, L and L' are still α-equivalent with respect to G_d. The maximum distance between any two strongly connected vertices of G_d is the same as the maximum distance (D) between any two connected vertices in G_u. According to theorem B3, L'^{N+h} may be tranformed into AL^NB by interchanging pairs of consecutive vertices that are not joined by cyclic edges of G_d. To say that two vertices are not joined by cyclic edges of G_d is the same as to say that those vertices are not adjacent in G_u. Δ

5. Application to the iterative resolution of systems of equations

Assume one wants to solve a system of equations $x=f_i(x)$, $i=1, ..., n$, where x denotes an arbitrary vector of variables and f_i denotes an arbitrary function. Assume one chooses an ordering of the equations. Let $p(i)$ be i-th equation to apply by the order chosen. One starts with an initial value x_0 and, in each round, compute new values of x by applying all the equations by the ordering previously chosen. That is, we compute the succession $x_{k+1}=F_p(x_k)$, with $F_p=f_{p(1)}\circ\ldots\circ f_{p(n)}$. The process stops when a solution is found (this may be checked when the value of x is not changed by any equation). Let $G=(V,E)$ be the undirected graph whose vertices are the functions and whose edges join non-commutative functions. According to theorem 6, if two orderings $p(i)$ and $p'(i)$ are α-equivalent, then $F_{p'}^{N+h}=A\circ F_p^N\circ B$. Hence, if a solution is always found (for any initial value) in at most N iterations under ordering $p(i)$, then a solution must also be found (for any initial value) in at most $N+h$ iterations under ordering $p'(i)$.

References

[RSS90] R. Ramakrishnan, D. Srivastava and S. Sudarshan. Rule Ordering in Bottom-Up Fixpoint Evaluation of Logic Programs. In *Proceedings of the 16th VLDB Conference* (D. McLeod, R. Sacks-Davis, H. Schek, Ed.), pages 359-371, Brisbane, Australia, 1990.

[TS92] K. Thulasiraman and M. N. S. Swamy. Graphs: Theory and Algorithms. John Wiley & Sons, 1992.

ANEXO 3

"Data-driven Active Rules for the Maintenance of Derived Data and Integrity Constraints in User Interfaces to Databases"

Data-driven Active Rules for the Maintenance of Derived Data and Integrity Constraints in User Interfaces to Databases

(Aceite para publicação nos Proceedings do XIV Simposium Brasileiro de Bases de Dados - SBBD'99, Florianopolis, Santa Catarina, Brazil, Outubro de 1999).

João Pascoal Faria
(jpf@fe.up.pt)

Raul Moreira Vidal
(rmvidal@fe.up.pt)

FEUP - Faculdade de Engenharia da Universidade do Porto
Rua dos Bragas – 4099 Porto Codex – Portugal

INESC - Instituto de Engenharia de Sistemas e Computadores
Praça da República, 93 R/C – Apartado 4433 - 4007 Porto Codex – Portugal

Abstract

It is presented an approach based on data-driven active rules for the maintenance of derived data (calculated data) and integrity constraints (data restrictions) in screen forms and other user interfaces to databases. The approach aims to improve the capabilities of current application development tools, by combining the power of active rules, as proposed in the context of active databases, with the simplicity and data-driven nature of spreadsheets. The proposed data-driven active rules define derived data items as functions of other data items, by explicitly reading and writing their values, conditionally or unconditionally. They are defined essentially as unordered action-only rules with implicit triggering events, and are executed essentially as ordered (prioritized) event-action (EA) rules, possibly in combination with other event-action (EA) or event-condition-action (ECA) rules originated from different sources. The specialized nature of the actions performed by these rules allows the automatic determination of the triggering events and priorities among rules, as well as the establishment of improved conditions on rule sets that guarantee termination and confluence (determinism) of rule execution. The maintenance of integrity constraints is essentially reduced to the maintenance of derived data. The approach presented is not tied to a particular rule definition language or data model.

1. Introduction

The spreadsheet paradigm is very attractive for the maintenance of derived data in user interfaces to databases, particularly screen forms and reports, due to its simplicity and data-driven nature. But spreadsheet engines and languages usually lack flexibility to support all the sorts of derived data (calculated data) and integrity constraints (data restrictions) found in complex screen forms and reports. They usually provide limited or no support for:

i) conditionally calculated data items, i.e. data items that are calculated in some circumstances and input by the user in other circumstances;
ii) calculation of multiple data items through the same formula, to avoid repeating expensive computations, like the ones that involve the access to an external database;
iii) calculation of the same data item by multiple non-contradictory formulas that naturally arise as a consequence of the support of the two previous kinds of calculations;
iv) formulas for input conversion (e.g. uppercase conversion);
v) multi-way constraints, i.e. constraints with multiple choices of derived (calculated) and primitive (input) data items (e.g. impose the constraint $c=b-a$ by allowing the user to input any two values and have the third one calculated);
vi) general integrity constraints on both primitive and derived data;
vii) procedural definition of complex derivations.

Some constraint-based graphical user interfaces use a high-level paradigm - constraint satisfaction by the perturbation model - that could be used to support nicely some of these features, specially multi-way constraints [1]. But the kinds of relationships between data that are supported are limited.

Current form managers and application development tools usually provide low-level event-driven mechanisms, such as triggers or event-action (EA) rules, that may be used to implement the features above mentioned. Triggers provide high flexibility, but place on the programmer the burden of determining the events upon which the derived data items should be recalculated or the integrity constraints be checked.

A similar situation occurs in most active database systems [2] [3] [4], where derived data and integrity constraints are maintained by means of active rules or triggers of the form event-condition-action (ECA) or event-action (EA). To overcome the disadvantages of a pure event-driven approach without compromising efficiency, several approaches have been proposed, namely the following extreme approaches:

- The definition of constraints and derivations in a high-level declarative language, and its automatic compilation into active rules (STARBUST system [5][6]). This approach has been developed mainly for the incremental maintenance of materialized views and integrity constraints. Maintenance of integrity constraints is essentially reduced to the maintenance of materialized views.
- The direct definition of derivations as executable data-driven rules (PARDES project [7][8][9]). Data-driven rules are there defined as "rules that define derived data-elements as functions of other data-elements and thus are triggered by modifications to certain data-elements". Data-driven rules are essentially unordered action-only rules, with implicit triggering events of a restricted type - modification of data items that participate in derivations, and explicit actions of a restricted type - the assignment of values to derived data items. This enables the automatic derivation of the triggering events and allows the definition of more accurate processing and analysis techniques. As in the first approach, maintenance of integrity constraints is similar to the maintenance of derived data. This approach is best suited for the maintenance of derived attributes and restrictions on the values of the attributes, and can accommodate complex derivations that are specified in a procedural language.

We think that data-driven rules, as proposed in the PARDES system, are also well suited for the maintenance of derived data and integrity constraints in user interfaces to databases. However, the concept of data-driven rules is interpreted in a limited way in the PARDES project. For example, features iii) (when different rules are involved) and iv) mentioned above are contrary to the restrictions of uniqueness and non-reflexiveness of the PARDES model. Our approach starts by removing such restrictions at the rule definition level.

2. Data model and rule definition language

2.1. Data model

For the purpose of rule processing, the data items controlled by rules are represented abstractly by *state variables*. The set of state variables is denoted by V. It's assumed that V is finite. The set of possible values of the state variables is called the *state space* and is denoted by S.

State variables may represent screen fields, dynamically varying properties of those fields (must-fill, locked, etc.), screen records, screen tables, etc., but its internal structure is ignored in the generic approach presented here.

2.2. Rule definition language

Details of a practical rule definition language are not important in the generic approach presented here, only its expressive power is important. As explained in the introduction, data-driven active rules are defined essentially as unordered action-only rules. So, only the definition of the action part is considered here.

It's assumed that rules are defined in one language with primitives to query (read) and update (write), conditionally or unconditionally, the values of the state variables. Rules that update the value of state variables are called *derivation rules.*

Rules may also conditionally call a primitive to abort rule processing and subsequently restore the system to the last quiescent state. Such rules are called *abort rules.* Without any loss of generality,

abort rules are hereafter handled as derivation rules that conditionally activate a special error variable, denoted *e*.

It's assumed that rules are individually *deterministic*, in the sense that, when a rule *r* is executed in state *s*, the state *s'* reached at the end of its execution is a function only of *s*, and does not depend on past states or time. Apart from that, no additional restrictions are placed about rule definition.

It's assumed that only the *net effect* of rules is important. That is, when a rule *r* is executed in state *s*, only the departure state *s* and the state *s'* reached at the end of *r*'s execution are important. Intermediate states are not important. Side effects (e.g. messages to the user) are allowed, but it's assumed that they play a secondary role. Rules that produce the same net effect are considered semantically equivalent. Consequently, a rule *r* may be defined abstractly as a *function* from *S* to *S*, with the same name of the rule. The state reached at the end of the execution of *r* from departure state *s* may be denoted by *r(s)*.

It's assumed that rules have *fixpoint semantics*, in the sense that a rule *r* need be executed in state *s* only if *s* is not a fixpoint for *r*. A *fixpoint* for a rule *r* is a state *s* where *r(s)=s*. The condition *r(s)=s* that gives the fixpoints for *r* is called the *underlying static constraint* of *r*, or the static constraint *enforced* by *r*.

In order to present some examples, it is used the following *abstract representation*: a rule *r* is represented by a conjunction for different *yy* of unconditional or conditional formulas of the type *y'=f(X)* or *p(Z) Þ y'=f(X)*, respectively. In these formulas, *y* is a state variable, *p(Z)* is an arbitrary condition on a set *Z* of state variables, and *f(X)* is an arbitrary expression on a set *X* of state variables. Primed variable references, such as *y'*, denote final values and unprimed variable references, such as *X* and *Z*, denote initial values. The symbols "*Þ*" and "=" have the usual logical meaning. Common conditions may be factored out. An example of a rule set is given in appendix A.

This abstract representation has the following advantages:

- To obtain the complete function mapping from *S* to *S*, it suffices to add the following *frame conditions*: for any state variable *v* without a primed reference, add *v'=v*; for any state variable *y* with a conditional primed reference *p(Z) Þ y'=f(X)*, add *Øp(Z) Þ y'=y*. The symbol "*Ø*" denotes logical negation.
- To obtain the underlying static constraint imposed by a rule, it suffices to remove the primes.
- It is easily translated into a practical rule language, referencing the same state variables in corresponding modes. Primed and unprimed variable references correspond to update (write) and query (read) operations. Auxiliary variables (local to each rule) may be required to store the initial values of some variables.

A rule with a common condition factored out is called *conditional*; it may be seen as a condition-action (CA) rule. However, it is not possible to defer action execution with respect to condition evaluation, and there is no obligation to execute the action *until* the condition is false (it is sufficient to reach a fixpoint for the pair condition-action). This is different from what usually happens in active rule systems that support CA rules [10]. Hence, the proposed rules are *essentially* action-only rules.

3. Rule execution model

Data-driven active rules, defined essentially as unordered action-only rules, are best executed as ordered (prioritized) event-action (EA) rules, because of:

- the need for their integration with other EA or ECA rules originated from different sources, with similar or different purposes;
- efficiency reasons.

Data-driven active rules, defined with the purpose of maintaining derived data and integrity constraints, are most safely executed *sequentially* (without nesting or concurrency), in order to keep rules atomic with respect to each other, and, consequently, keep the net effect of each rule unaffected by the existence of other rules.

Sequential execution of prioritized EA or ECA rules (with immediate coupling mode between condition and action) may be described by the following basic algorithm:

Algorithm 1 *(sequential rule execution)*:

1. While there are triggered rules and the error variable *e* has not been activated:

1.1. Select a triggered rule *r* such that there is no other triggered rule *r'* with precedence (priority) over *r*
1.2. Detrigger rule *r*
1.3. Execute rule *r*, i.e., evaluate the condition and, in case it is *true*, execute the action

2. If the error variable *e* has been activated, restore the system to the last quiescent state

Sequential rule execution may be represented abstractly as function composition.

The point in time (at the end of a transaction, at the end of a user update, on the occurrence of an event, etc.) where rule execution is invoked is called a *rule processing point* (*RPP*). The precise localization of RPP's is not important here. It is only assumed that, during rule execution, the rule set *R* does not change and user updates (or other updates not produced by rules) are not allowed.

Triggering criteria (triggering events) and ordering criteria (rule priorities) are developed next based on a preliminary analysis of the data dependencies existing between variables and rules.

4. Data dependency analysis

The sets of variables queried and updated by each rule constitute important information for rule processing and analysis. The variables queried (read) by a rule *r* are called the *input variables* of *r*, denoted *i-vars(r)*. The variables updated (written) by a rule *r* are called the *output variables* of *r*, denoted *o-vars(r)*. They correspond to the variables with unprimed and primed references, respectively, in the abstract representation. In a practical language, they can be obtained by a simple syntactic analysis.

A global picture of the relationship between variables and rules is given by a *rule-variable data dependency graph* (*RVDDG* for short). Due to obvious analogies, it's used the notation of dataflow diagrams. An example of a RVDDG is given in appendix A.

Since we assume that only the net effect of rules is important, any input and output variables of a rule *r* that could be eliminated by rewriting the rule while preserving its net effect, should be excluded from consideration. It can be shown that the minimal sets of input and output variables that can be achieved by rewriting a rule *r* while preserving its net effect are unique [11]. If no input or output variables of a rule *r* could be eliminated by rewriting *r* while preserving its net effect, it is said that *r* is in *canonical form.* For example, all the rules in the appendix are in canonical form. Rules with the same net effect of rules r_2 and r_6 from the appendix, but not in canonical form, are:

r_{2a}: $z \leftarrow x + y \Rightarrow z'=x+y$ and $\qquad r_{6a}$: $y' = \begin{cases} x, & \text{if } x \geq 0 \\ y, & \text{otherwise} \end{cases}$.

In some cases, it is not convenient to write rules in canonical form, for efficiency reasons. Two approaches are possible to obtain minimal input and output variables in such cases:

- they are explicitly provided by the programmer;
- the system performs a dataflow analysis during rule compilation to detect read and write operations that could be avoided by rule rewriting (e.g. a read operation after a write operation can be avoided by using an auxiliary local variable).

A global picture of the possible interferences between rules, via the variables queried and updated by them, is given by a *rule-rule data dependency graph* (*RRDDG* for short). The vertices of this graph are the rules. A directed edge $r_i \rightarrow r_j$, with $i \neq j$, means that r_i updates some variable that is referenced (queried or updated) by r_j. A self-loop $r_i \rightarrow r_i$ means that r_i updates some variable also queried by r_i. See the example in appendix A.

5. Rule types

As it will be seen, the following types of rules are potential sources of non-termination or non-determinism of rule execution: conditional rules, recursive rules and conflicting rules. Conditional rules have already been described.

A rule with overlapping sets of input and output variables is called *self-recursive*, because it defines the new value of a variable using the old value of the same variable. Examples of useful self-recursive rules are:

- rules for input conversion, like rule r_5 in appendix A;
- rules used in iterative methods, like
  r_{10}: $y' = exp(-y)$

Sets of rules that participate in a cycle of the RVDDG with two or more rules (otherwise it's the case of a single self-recursive rule) are called *mutually recursive*, because they define the new value of a variable using the old value of the same variable. Examples of useful mutually recursive rules are:

- rules with mutually exclusive conditions, like
  r_{11}: $x=0 \Rightarrow y' = z$ and r_{12}: $x \neq 0 \Rightarrow z' = y$;
- rules that enforce the same constraint in multiple ways, like rules r_8 and r_9 in appendix A.

Pairs of rules with common output variables are called *mutually conflicting* (or simply conflicting), because of the potential for contradictory updates. Examples of admissible conflicting rules (ones that in fact do not perform contradictory updates) are:

- rules with mutually exclusive conditions, like
  r_{13}: $x=0 \Rightarrow y' = z$ and r_{14}: $x \neq 0 \Rightarrow y' = w$;
- rules with identical actions, like
  r_{15}: $x < 0 \Rightarrow e' = true$ and r_{16}: $y < 0 \Rightarrow e' = true$ (abort rules);
- rules with cumulative actions, like
  r_{17}: $z' = z \cup x$ and r_{18}: $z' = z \cup y$;
- rules that act on different partitions of a state variable, like
  r_{19}: $z' = z \cup \{1\}$ and r_{20}: $z' = z - \{2\}$ (assuming z is set-valued).

Recursive rules are more powerful, because mutually conflicting rules may be converted into mutually recursive rules [11].

6. Triggering criteria

In this section it is presented triggering criteria (triggering and detriggering events) for action-only data-driven rules, needed for their translation into event-action rules. The primary purpose of the triggering criteria is to guarantee termination of rule processing after and only after a fixpoint for all the rules has been reached. The secondary purpose is to avoid unnecessary rule executions as much as possible. It is first presented a basic triggering criterion that only requires the knowledge of the input and output variables of each rule, and several improvements are subsequently discussed.

6.1. Basic triggering criterion

Conservative triggering criterion for data-driven rules may be defined based solely on the static knowledge of their input and output variables and on the dynamic monitoring of state variable modification events. A modification event occurs when a variable is updated with a different value from the one it currently holds.

In order to cope with the assumption that only the effect of rules is important, the following *net effect requirement* has to be obeyed: either each rule is forbidden to perform mutually compensating modifications or they are detected and ignored by the event monitoring mechanism.

The basic triggering criterion is:

i) all the rules are initially triggered (at the beginning of their existence);
ii) rules are detriggered at the beginning of their execution (see algorithm 1);
iii) a rule *r* being executed is triggered when an input variable of the rule is modified (by the rule itself);
iv) a rule *r* not being executed is triggered when an input or output variable of the rule is modified (by other rules or by the user).

The variables mentioned in iii) and iv) are called the *self-triggering variables* and *external-triggering variables* of *r*, denoted *st-vars(r)* and *et-vars(r)*, respectively.

We next give the rational behind point iii). The decision of what values are assigned to what variables when a rule *r* is executed depends only on the initial values of the input variables of *r*. If the

input variables of *r* are not modified during its execution, a second execution of *r* would assign exactly the same values to the same variables. Hence, there is no need to execute *r* again, reason why it may remain detriggered.

Now we give the rational behind point iv). Assume that a rule *r* was executed and became in a detriggered state. As long as neither the input nor the output variables of *r* are modified subsequently, a new execution of *r* would assign the same values to the same unmodified output variables, causing no change of the state of the system. Hence, there is no need to execute *r* again, reason why it may remain detriggered.

6.2. Static improvements based on the elimination of redundant triggering variables

The sets of triggering variables presented above may be too conservative.

Possible ways by which less conservative sets of external-triggering variables may be obtained are:

- Take for *et-vars(r)* the variables referenced in the underlying static constraint of *r*. This is the optimal static choice that can be made based only on the knowledge of *r* [11].
- In case it is known in advance that the updates performed by others (the user or other rules) on the output variables of *r* do not "contradict" the ones performed by *r*, those variables may be excluded from *et-vars(r)*.

Possible ways by which less conservative sets of self-triggering variables may be obtained are:

- In case it is known in advance that *r* is *idempotent*, take *st-vars(r)={}*. A rule *r* is called *idempotent* when $r^2(s)=r(s)$ for any state *s*. Rules with disjoint sets of input and output variables are trivially idempotent. A rule such that the execution of the rule's action always falsifies the rule's condition is also idempotent. In general, a semantic analysis is required to check if a rule is idempotent. For example, rule r_5 in appendix A is idempotent (the other rules are trivially idempotent).
- In case *r* is conditional, restrict *st-vars(r)* to the variables that are simultaneously input and output variables of the action part of *r* only (i.e., ignore the condition part).
- In a practical rule language, restrict *st-vars(r)* to the variables that may be updated after being read in the action part of *r*.

Example: Assume the following rule

r_{21}: $y>0 \Rightarrow [(x' = x+1) \vee (y' = 0)]$

By the basic triggering criterion, one would have $et\text{-}vars(r_{21})=st\text{-}vars(r_{21})=\{x, y\}$.

The underlying static constraint is

$y>0 \Rightarrow [(x = x+1) \vee (y = 0)]$

which can be simplified to

$y \leq 0$.

Hence, one takes $et\text{-}vars(r_{21})=\{y\}$. Since the rule is idempotent one takes $st\text{-}vars(r_{21})=\{\}$.

6.3. Dynamic improvements based on the value of the condition

Clearly, the external triggering variables of a conditional rule may be dynamically restricted in the following way:

- if a rule *r* is executed and its condition does not hold, subsequently take for *et-vars(r)* the variables referenced in the condition part;
- if a rule *r* is executed and its condition holds, subsequently take for *et-vars(r)* the variables referenced in the action part.

Initially, the value of the condition is unknown, but that's not a problem because all the rules are initially triggered.

Example: Assume the following rule from appendix A:

r_3: $y=1 \Rightarrow w' = 1$

If r_3 is executed and the condition holds, one subsequently takes $et\text{-}vars(r_3)=\{w\}$. Otherwise, one takes $et\text{-}vars(r_3)=\{y\}$.

6.4. Dynamic improvements based on the monitoring of read and write events

In practice, an improved triggering criterion may be defined by monitoring read and write events, besides the modification events. Prior static knowledge of the input and output variables of each rule is not even required!

Dynamic input and output sets are associated with each rule and maintained as follows:

- initially, the input and output sets are empty;
- the input and output sets are emptied at the beginning of a rule execution;
- a variable *x* is added to the input set when it is read during the rule execution, but only if it is not already in the output set (i.e., ignore a *read* after a *write*);
- a variable *y* is added to the output set when it is written during the rule execution (this is done before the eventual modification event is handled);
- in the case of conditional rules, if the condition holds, the input set is emptied before the action is executed.

The basic triggering criterion still applies, but with respect to the dynamic input and output sets instead of the static input and output sets.

7. Termination analysis

Rules may trigger each other (in *cascade*) indefinitely, causing non-termination of rule processing. We next give sufficient conditions on rule sets that guarantee termination of rule processing, for any initial state and any order of rule execution (i.e., any order by which rules are selected for execution when there are multiple triggered rules). Such conditions are useful in termination analysis of particular rule sets and in the development of ordering criteria to prevent non-termination.

7.1. Conservative termination analysis based on the triggering graph

Conservative termination analysis of active rules is usually based on the construction of a *triggering graph* [12]. The triggering graph (*TG*) for a rule set *R* is defined as follows:

- The vertices of the TG are the rules in the rule set *R*.
- A directed edge $r_i \rightarrow r_j$ (possibly with $i=j$) in the TG means that r_i may generate an event that triggers r_j.

It is known that, if the TG for a rule set *R* is acyclic, then rule processing always terminates for any initial state and any order of rule execution [12].

The TG for a set of data-driven active rules, processed according to the triggering criteria presented in section 6, is trivially obtained from the triggering variables and the output variables of each rule, as follows:

- A rule r_i may trigger itself only if $st\text{-}vars(r_i) \neq \{\}$.
- A rule r_i may trigger a distinct rule r_j only if $o\text{-}vars(r_i) \cap et\text{-}vars(r_j) \neq \{\}$.

With the basic triggering criterion, the TG is identical to the RRDDG. If some triggering variables are eliminated due to further information, the TG may have fewer edges. Consequently, if the RRDDG is acyclic (i.e., if the rule set *R* contains no recursive or conflicting rules), rule processing always terminates for any initial state and any order of rule execution.

7.2. Improved termination analysis based on the activation graph

Let's define the *activation graph* (*AG*) for a rule set *R* in the following way:

- The vertices of the AG are the rules in *R*.
- A directed edge $r_i \rightarrow r_j$ (with $i \neq j$) in the AG means that r_i may activate r_j, in the sense that the execution of r_i in a fixpoint for r_j may produce a state that's not a fixpoint for r_j.
- A self-loop $r_i \rightarrow r_i$ in the AG means that r_i is not self-disactivating, in the sense that the final state at the end of r_i' execution may be not a fixpoint for r_i. In other words, r_i is not idempotent.

See an example of an AG in appendix A.

It can be shown that, if the AG of a rule set *R* is acyclic and the net effect requirement is satisfied, then rule processing always terminates for any initial state and any order of rule execution [11]. The net effect requirement is important to ensure the termination of rule processing after a fixpoint for all the rules has been reached, with at most one extra execution of each rule.

Since the edges of the AG are a subset of the edges of the TG, this condition is less conservative than the one based on the TG. By contrast, the construction of the AG requires a semantic analysis that cannot be automated in general, while the construction of the TG only requires a syntactic analysis. So, both conditions are useful.

Improved termination analysis of active rules has also been proposed in [13], by combining the TG with an analysis of the effect of rules actions on the truth-value of other rules conditions. However, it can be shown that, for the specific case of data-driven active rules, the conditions obtained in [13] are more conservative than the ones we've obtained here [11]. They use an activation graph that coincides with our in case the condition of each rule is precisely the complement of its underlying static constraint.

8. Confluence analysis

The order by which rules are considered for execution when there are multiple triggered rules may affect the final state reached, at termination of rule processing. If, for a given rule set, the final state reached at termination of rule processing is independent of the order by which rules are executed, for any initial state, the rule set is called *confluent* [12]. We next give sufficient conditions that guarantee confluence. Such conditions are useful in confluence analysis of particular rule sets and for the development of ordering criteria.

8.1. Rule commutativity

Confluence analysis of active rules is usually based on the notion of rule commutativity. Two rules r_i and r_j are *commutative* (or r_i and r_j *commute*) if, for any state *s*, applying rule r_i and then rule r_j from state *s* produces the same state as applying rule r_j and then rule r_i (i.e., $r_i \circ r_j = r_j \circ r_i$, where "$\circ$" denotes function composition). Non-commutative rules are also called *order dependent* rules in this paper. For a given rule set *R*, order dependencies between rules are depicted by an *order dependency graph* (*ODG*) with vertex set *R* and undirected edges joining pairs of order dependent rules.

The following properties are useful in the construction of the ODG:

- The ODG has no self-loops, because every rule commutes with itself.
- If two rules are not adjacent in the RRDDG, then they commute [11].
- If two rules are adjacent in the AG, then they do not commute [11].

From the first two properties, a conservative ODG may be obtained from the RRDDG by replacing directed edges by undirected edges and removing parallel edges and self-loops. An example of an ODG is given in appendix A.

8.2. Confluence of unordered rules

A confluence requirement for generic partially ordered active rules has been presented in [12]. Once applied to unordered active rules, the resulting requirement is that the rule set is terminating and that every two rules in the rule set commute, which is too conservative. In fact, using classic results about confluence [14], it can be shown that, if the rule set is terminating and every two rules in the rule set are confluent, then it is confluent [11].

There is another important difference with respect to the requirement presented in [12]. The definition of rule commutativity in [12] is based on *execution states* that also include the triggered rules and their corresponding "transition tables" besides the database state (corresponding here to the state of the state variables). The specialized nature of data-driven active rules dispenses the inclusion of the triggered rules and the transition tables.

The following properties are useful to check if two rules r_i and r_j are confluent:

- If two rules are commutative and terminating, then they are obviously confluent.
- If two rules r_i and r_j have an acyclic AG, then they are confluent iff $r_i \circ r_j \circ r_i = r_j \circ r_i \circ r_j$ [11].

- In particular, if two rules r_i and r_j are not recursive and not conflicting, and the execution of any of the rules cannot *falsify* (change value from *true* to *false*) the condition upon which the value of a variable is derived by the other rule, then they are confluent [11]. For example, with respect to the rule set in the appendix, the execution of rule r_6 can falsify the condition ($y=1$) upon which the value of w is derived by rule r_3.

Consequently, if a rule set R has no recursive rules, no conflicting rules, and no rules whose execution may falsify the conditions upon which the values of variables are derived by other rules, then it is terminating and confluent.

For example, the rule set in the appendix has two pairs of non-confluent rules. In fact, it is not confluent.

9. Ordering criteria

The order by which rules are considered for execution when there are multiple triggered rules may have impact on the *semantics* (the termination of rule processing and, in case of termination, the final state reached) and the *efficiency* (the number of executions of each rule). We next present ordering criteria aimed to improve termination, efficiency, confluence and controllability of rule processing.

9.1. Basic ordering criterion

Intuitively, the principle "compute before use" should be used whenever possible. That is, the following ordering criterion should be used:

- A rule r_i takes precedence over a rule r_j if there is a path from r_i to r_j and no path from r_j to r_i in the RVDDG.

The precedence means that if both are triggered, r_j cannot be selected for execution. The absence of a path from r_j to r_i is required to avoid the generation of cyclic precedences.

This criterion has the following advantages:

- It requires only a syntactic analysis of rules.
- It usually provides the expected semantic. In particular, abort rules only check final values of their input values. In general, non-recursive rules are executed only after their input variables have reached final values [11].
- Rules that are neither recursive nor conflicting are executed at most once [11].
- In the absence of recursive rules, this criterion fixes the semantic of the rule set. That is, further ordering choices have no impact on the termination of rule processing and, in case of termination, the final state reached [11].

9.2. Ordering of recursive rules

Mutually recursive rules are not ordered by the basic criterion. Several heuristics that can be used in isolation or in combination to order such rules are:

- Either statically or dynamically eliminate a minimal set of cyclic edges from the RVDDG, and then apply the basic ordering criterion. This tends to minimize the number of rule executions.
- Use the order by which rules are defined. This gives the programmer higher control over the system.
- Give precedence to rules with input variables most recently updated by the user. This tends to preserve the most recent user updates, giving the user higher control over the system (see example below). It has the disadvantage of being dynamic.

Example: Assume that one wants to impose the static constraint $b=a+c$ in multiple ways by the following mutually recursive rules (from appendix A):

r_8: $b' = a+c$ $\qquad$ r_9: $c' = b-a$

Assume that the user updates c and then rules are processed. Rule r_8 is first selected for execution. The user update is preserved and b is recalculated. Rule r_9 is executed with no effect. A symmetric situation occurs if the user updates b. If the user updates a, the value of the variable (b or c) most recently updated by the user will be preserved.

9.3. Additional criteria

Since rules may not become totally ordered by the basic criterion, there is room for additional secondary criteria. For instance, abort rules may be given higher priority, to be executed as earlier as possible. The order by which rules are defined may also be used.

For example, two total orderings of the rules presented in appendix A that result from the criteria discussed are $r_1 r_4 r_6 r_5 r_2 r_3 r_7 r_8 r_9$ and $r_1 r_4 r_6 r_5 r_2 r_3 r_7 r_9 r_8$.

10. Dynamic detection of contradictory updates

Although it is unusual or may even be considered a programming error, conflicting rules may perform contradictory updates (assign different values to the same variable) that may cause non-termination of rule processing. When conflicting rules do not appear in combination with recursive rules and the "principle compute before use" is followed, once contradictory updates are performed by rules during a RPP, it can be concluded that rule processing will not terminate [11]. Hence, rule processing may be immediately aborted. In the presence of recursive rules, contradictory updates may also be performed in different executions of the same rule. Such contradictory updates may converge, so rule processing should not be aborted immediately. In practice, it may be established a bound on the number of times that each rule may be executed or the number of times that each variable may be modified.

By contrast, it is assumed that the user updates may be contradicted by rules. This is necessary for input conversion, to establish departure values for some iterative methods, and to support error repair rules (derivation rules used with the purpose of repairing input errors). Mandatory user updates that cannot be contradicted by rules are best treated as temporary rules.

11. Conclusions and further work

It was presented a data-driven rule model suited for the maintenance of integrity constraints and derived data in screen forms and other user interfaces to databases. The specialized nature of the actions performed by those rules allowed us to derive triggering events and priorities among rules for efficient rule execution, as well as improved properties on rule sets that ensure termination and confluence of rule execution. More conservative conditions require only a syntactic analysis of the rules, while less conservative conditions require also a semantic analysis. A rule system based on the approach described is being re-implemented as a core component of a software tool that automates the development of interactive database applications [15], benefiting from the experience acquired in the development of successful applications with thousands of rules in previous implementations [16].

Several assumptions related to rule definition and execution are too restrictive in many practical applications and might be relaxed.

We've assumed that rules do not reference past states. Under some conditions, rules that reference past states may be accommodated into the formalism presented, such as:

- Rules that reference the last quiescent state of the system, e.g. to check dynamic integrity constraints.
- Rules that reference the last state of the system consistent with the rule (the state at the end of its last execution, or the last quiescent state of the system, according to which is most recent) for optimization purposes (e.g. the incremental evaluation of summary fields).

We've assumed that rules do not reference time. Under some conditions, rules that reference time may be accommodated into the formalism presented, such as:

- Rules that reference the time given by the system clock, e.g. to animate user interfaces or to automatically collect data from the database (or other external data sources). The clock time may be represented by a state variable that is periodically updated by means of an alarm or a timer. Such an update is similar to a user update. The only restriction is that it should be prevented (buffered) during rule processing, as it is required with respect to the user updates.
- Rules that compare times such as the time of creation and last modification of state variables. For comparison purposes, the real time need not be used; a sequential timestamp generated by

the system is sufficient. Such rules may be used to provide a more explicit support for multi-way constraints.

We've ignored the internal structure of the state variables and considered only simple modification events. But, in the case of state variables with a complex internal structure, specialized modification events should be considered for optimization purposes (such as insertion, modification and deletion of elements of a set-valued state variable).

We've assumed that, in each RPP, rules are executed until a new quiescent state is reached, before user updates are allowed again. Such an *eager* mode of evaluation is important in the presence of abort rules. But a *lazy* mode of evaluation (*on demand*) of some data items, particularly output-only data items, could be more effective.

We've assumed a net effect policy that imposes an implementation overhead. The implications of removing such net effect policy should be investigated.

We've assumed that a quiescent state of the system must be a fixpoint for *all* the rules, even in the presence of conflicting rules. Alternative approaches could be useful in practice (e.g. allow *default* and *exception* rules).

The special nature of abort rules should be taken in consideration to improve termination and confluence analysis.

We've assumed a sequential mode of execution, without nesting or concurrency. To deal with more complex user interfaces and systems, such as master-detail and hierarchical screen forms, at least a combination of sequential and nested rule execution (by means of nested RPPs) has to be considered. Such richer modes of executions are found in object-oriented databases [17]. Data-driven active rules may be integrated in such environment according to the following scenario (the details are left to [15]):

- A complex system is viewed as a collection of interacting composite objects.
- A composite object is viewed as a small-scale system, with its own *intra-object state variables* and *intra-object rules*. The intra-object state variables represent component-objects that do not interact directly with each other. Attributes of an object and references to other objects are viewed as components-objects. These non-interacting component-objects are integrated via the methods (operations) and rules attached to the composite object. Data-driven rules attached to the composite object (intra-object rules at the composite object level) may be used to maintain derivations and constraints between the component-objects. Such rules are executed in RPPs in the methods of the composite object (including methods for object creation and deletion), and are triggered by events generated inside the composite object.
- Derivations and constraints that involve objects that interact directly with each other may be managed either via intra-object data-driven rules attached to a composite object of a sufficient high level of granularity, or via partial intra-object data-driven rules attached to each of the objects involved (essentially as proposed in the Ode system [18]). In the latter case, accesses to other objects involved are viewed as *side effects* to external systems.

Important issues related to the distribution of reactive processing (by means of active rules) between the user interface layer (our main target) and the database layer, in a centralized, client-server or three-tier architecture, have not been addressed here. Some of them are analyzed in [15].

12. References

[1] M. Sannela, J. Maloney, B. Freeman-Benson and A. Borning. Multi-way versus one-way constraints in user interfaces: experience with the DeltaBlue algorithm. In *Software Practice and Experience*, Vol. 23(5), pp. 529-566, May 1993.

[2] U. Dayal. Active database management systems. In *Proc. of Third International Conference on Data and Knowledge Bases*, Jerusalem, Israel, June 1998.

[3] S. Ceri and J. Widom. Active Database Systems. Triggers and Rules For Advanced Database Processing. Morgan Kaufmann, 1996.

[4] C. Zaniolo, S. Ceri, C. Faloutsos, R. Snodgrass, V. Subrahmanian and R. Zicari. Advanced Database Systems. Morgan Kaufmann, 1997.

[5] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *Proc. of the 16th VLDB Conference*, pp. 566-577, Brisbane, Australia, August 1990.

[6] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the 17th VLDB Conference*, pp. 577-589, Barcelona, Spain, September 1991.

[7] O. Eztion. PARDES - A Data-Driven Oriented Active Database Model. In *SIGMOD RECORD*, Vol. 22, No. 1, March 1993.

[8] O. Eztion, A. Gal and A. Segev. Data driven and temporal rules in PARDES. In *Proc. of the First International Workshop on Rules in Database Systems*, pp 92-108, Edinburgh, Scotland, August 1993.

[9] A. Gal and O. Eztion. Maintaining Data-driven Rules in Databases. IEEE Computer, 28(1):28-38, January 1995.

[10] E. Hanson. Rule condition testing and action execution in Ariel. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 49-58, San Diego, California, June 1992.

[11] J. Faria. Data-driven rules for the maintenance of derived data and integrity constraints in user interfaces to databases. *INESC Technical Report RI 1/99*. Porto, Portugal, May 1999 (url: http://www.fe.up.pt/~jpf/research/tr3.ps.zip).

[12] A. Aiken, J. Widom and J. Hellerstein. Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 59-68, USA, June 1992.

[13] E. Baralis, S. Ceri and S. Paraboschi. Improved rule analysis by means of triggering and activation graphs. In *Proc. Rules in Database Systems - Second International Workshop – RIDS'95*, pp. 165-181, Athens, Greece, September 1995.

[14] Gérard Huet. Confluent Reductions: Abstract Porperties and Applications to Term Rewriting Systems. In *Journal of the ACM*, Vol. 27, No. 4, October 1980, pp. 797-821.

[15] J. Faria. Concepção de um sistema de regras activas para a manutenção de restrições de integridade e dados derivados em aplicações interactivas de bases de dados. FEUP, Porto, Portugal, September 1999 (*Ph.D. thesis to be submitted in portuguese*).

[16] J. Faria and J. Ranito. SAGA - Uma ferramenta interactiva para o desenvolvimento expedito e fácil manutenção de aplicações de bases de dados. In *Proc. ENDIEL'91, ST2-Indústrias do Software e da Informação*, pp. 33-40, Lisboa, Portugal, June 1991 (*in portuguese*).

[17] N. Paton and O. Diaz. Active Database Systems, 1999(?). (survey to be published in ACM Computing Surveys, url: http://www.cs.man.ac.uk/~norm/papers/surveys.ps).

[18] N. Gehani and H. Jagadish. Ode as an Active Database: Constraints and Triggers. *In Proc. of the 17th International Conference on Very large Data Bases*, pp. 327-336, Barcelona, September, 1991.

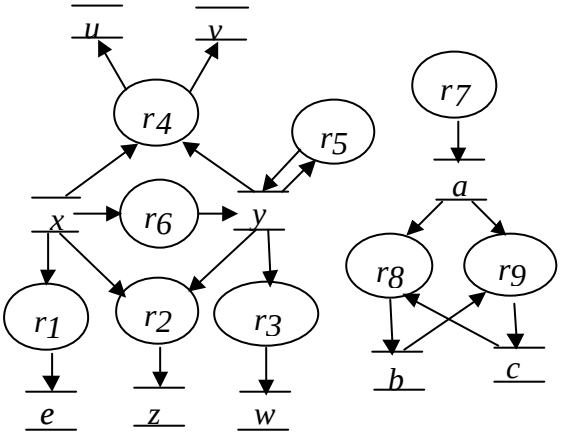
Appendix A – Example of a rule set and corresponding graphs

a) Rule set

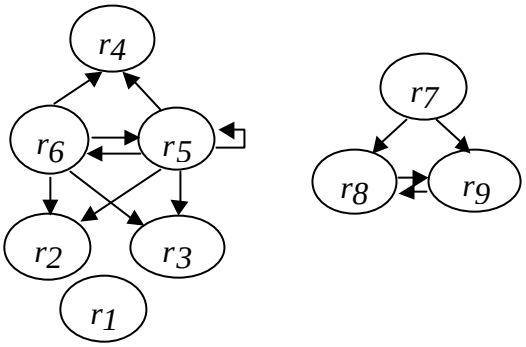
(intended to illustrate several types of rules and not a concrete application)

r_1: $\|x\|>100 \Rightarrow e'=true$	*(abort rule handled as a derivation rule)*
r_2: $z' = x+y$	*(unconditional derivation rule)*
r_3: $y = 1 \Rightarrow w'=1$	*(conditional derivation rule)*
r_4: $u'=max(x,y) \vee v'=min(x,y)$	
r_5: $y' = \|y\|$	*(self-recursive)*
r_6: $x \geq 0 \Rightarrow y'=x$	*(mutually conflicting with r_5)*
r_7: $a' = 1$	
r_8: $b' = a+c$	
r_9: $c' = b-a$	*(mutually recursive with r_8)*

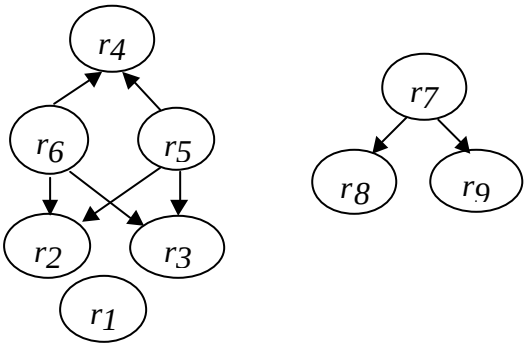
b) Rule-variable data dependency graph



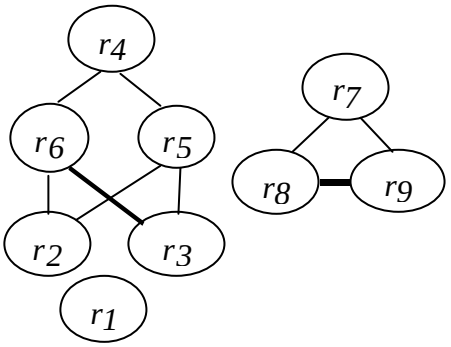
c) Rule-rule data dependency graph



d) Activation graph



e) Order dependency graph



(heavier edges link pairs of non-confluent rules)