Un modèle d'exécution paramétrique pour systèmes de bases de données actifs

Thierry Coupaye

Thèse

présentée par

Thierry COUPAYE

pour obtenir le grade de Docteur
de l'Université Joseph Fourier - Grenoble 1
(arrêté ministériel du 30 Mars 1992)
Spécialité : **Informatique**

Un modèle d'exécution paramétrique pour systèmes de bases de données actifs

---

Date de soutenance : 14 novembre 1996

Composition du jury :

Président	:	Paul Jacquet
Rapporteurs	:	Nicole Bidoit
		Éric Simon
Directeur	:	Christine Collet
Examinateurs	:	Michel Adiba
		Ahmed Bouajjani

Thèse préparée au sein du laboratoire
Logiciels Systèmes Réseaux – IMAG

"Non comme le Chaos, où tout s'entasse et s'écrase,
Mais comme l'harmonie confuse du monde :
Où dans la diversité un ordre se voit,
Et où, quoique tout diffère, tout s'accorde."

Alexander Pope. "Windsor Forest".

Je voudrais ici remercier :

Paul JACQUET, *Professeur à l'Institut National Polytechnique de Grenoble et directeur du laboratoire Logiciels Systèmes Réseaux (LSR-IMAG) qui me fait l'honneur de présider ce jury de thèse.*

Nicole BIDOIT, *Professeur à l'Université de Paris XIII et Éric* SIMON, *Directeur de Recherches à l'INRIA Rocquencourt qui ont bien voulu porter leur jugement sur ce travail.*

Michel ADIBA, *Professeur à l'Université Joseph Fourier, responsable de la formation doctorale en informatique et responsable de l'équipe STORM (LSR-IMAG) qui a accepté de faire partie de ce jury. Je voudrais lui exprimer ma plus sincère gratitude pour m'avoir "aiguillé" sur les systèmes d'informations et les bases de données puis pour m'avoir fait confiance en m'acceuillant dans son équipe.*

Christine COLLET, *Maître de Conférences à l'Université Joseph Fourier et co-responsable de l'équipe STORM qui est à l'origine et à l'aboutissement de ce travail. Elle m'a conseillé, guidé et formé à la recherche plus que quiconque. Notre configuration en tandem n'est plus à présenter et je voudrais simplement lui signaler que si je suis "son premier thésard", elle restera quant à elle, "ma première (et unique) directrice de thèse" !*

Ahmed BOUAJJANI, *Maître de Conférences à l'Université Joseph Fourier, qui a accepté de participer à ce jury. Je le remercie plus particulièrement pour ses remarques et suggestions qui ont grandemement amélioré la lisibilité de la sémantique formelle présentée dans le cinquième chapitre de ce document – mettant ainsi en valeur ce travail de formalisation.*

Je voudrais également remercier tous les membres de l'équipe STORM avec qui mes rapports furent aussi divers qu'enrichissants. Je pense plus particulièrement à Christian ESCULIER, *Pierre* HABRAKEN *et Claudia Lucia* RONCANCIO *qui se sont, je crois, réellement intéressés à ce travail. Leurs commentaires, questions et suggestions ont notablement amélioré cette thèse tant sur le fond que sur la forme. Merci à Jean-Pierre* GIRAUDIN *et Monique* CHABRE-PECCOUD *pour avoir suggérer chacun une idée présente dans cette thèse ! Je ne voudrais certainement pas omettre Hervé* MARTIN *qui a souvent joué le rôle de catalyseur dans mon "entreprise personnelle de dédramatisation".*

Merci à Corinne, Marie-Claude et Catherine qui ont accepté de se métamorphoser momentanément en correcteurs orthographiques.

Merci à ma famille – Jean-Pierre, Colette, Yvonne et EvelYne – qui m'a offert une nouvelle preuve de son amour en se relayant pour veiller, elle aussi, à l'orthographe de ce document.

Merci a Philou, Rosine, Claire et Juliette pour leur présence au cours de ces derniers mois.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Baromètre *n. Instrument ingénieux qui indique la sorte de temps que nous sommes en train de subir.*
Ambrose Bierce - "Le dictionnaire du Diable"

1.1 Des SGBD conventionnels aux SGBD actifs

Face à la complexité grandissante des nouvelles applications – telles que la conception assistée par ordinateur (CAO), le Génie Logiciel, la gestion de trafic aérien ou ferroviaire, les applications géographiques, bancaires, financières, etc. – la recherche dans le domaine des bases de données est en constante ébullition et répond à ces nouveaux besoins en proposant de nouveaux types de SGBD – toujours plus performants, bien évidemment.

Ces nouveaux systèmes [Kim95, Kim90, AC93, OV91, ODV94, GV91] : objets, temporels, multimédias, parallèles, distribués, répartis, etc., offrent un large éventail de modèles de données et de services qui permettent de résoudre des problèmes spécifiques. Cependant, ils se révèlent souvent très limités pour gérer explicitement la dynamique des bases de données, et en particulier, pour réagir aux changements d'états des bases de données. En effet, dans ces systèmes, toutes les opérations de manipulation de données sont effectuées sur la demande explicite d'un utilisateur ou d'une application. On qualifie souvent ces systèmes de *passifs*, par opposition aux

systèmes *actifs*, qui eux, sont capables d'effectuer automatiquement des actions prédéfinies, en réponse à des *événements spécifiques*, lorsque certaines conditions sont vérifiées.

La notion de *déclencheur* (*trigger* en anglais) a été introduite pour décrire de façon déclarative une action qui doit être exécutée en réaction à un événement et non à la demande d'un utilisateur ou d'une application. Ce n'est pas, à proprement parler, une notion récente puisqu'elle était déjà proposée dans les années soixante dix, dans le cadre des systèmes relationnels, principalement pour garantir et maintenir l'intégrité des bases de données. Les déclencheurs sont apparus beaucoup plus récemment (quelques vingts ans plus tard) dans les SGBD commerciaux [Ing91, Syb92, Ora92] ; principalement à cause d'un problème chronique de performances et surtout d'une sémantique d'exécution qui rend souvent leur utilisation hasardeuse.

Au cours des dernières années, les *bases de données actives* ont connu un regain d'interêt certain de la part de la communauté des chercheurs en base de données et la notion de *règle active* a été proposée et unanimement acceptée comme mécanisme de base des SGBD actifs. Les règles actives généralisent la notion de déclencheur, elles utilisent généralement le formalisme *Événement-Condition-Action* – ou plus simplement E-C-A. La sémantique la plus générale d'une règle E-C-A est la suivante : *lorsque survient un événement du type E, si la Condition C est satisfaite, alors exécuter l'action A*. Les règles actives sont parfois affiliées aux *règles de production* des systèmes déductifs, généralement de la forme *Événement-Action*. Dans la suite, nous considérons uniquement les règles actives (règles E-C-A).

Systèmes et prototypes

L'intégration "harmonieuse" des règles actives dans différents modèles de données : relationnel, relationnel étendu, objet ou déductif a fait l'objet de nombreux travaux. Ces travaux ont abouti à la proposition de langages qui décrivent principalement les types d'Événements susceptibles de déclencher l'exécution des règles, la structure des parties Condition et Action, de même que le passage d'informations entre les trois parties constituantes d'une règle. L'intégration de ces langages dans les langages d'accès des SGBD ou dans les langages de programmation de bases de données (LPBD) a également suscité de nombreuses recherches.

L'architecture et l'implantation des systèmes de règles actives ont également fait l'objet de nombreux travaux. On distingue, en général, deux types d'architectures logicielles : les *architectures en couches* dans lesquelles un système de règles est im-

planté au-dessus d'un SGBD qui est vu comme une "boîte noire" ; et les *architectures intégrées* dans lesquelles l'exécution des règles est prise en compte au niveau même du noyau d'exécution du SGBD. La première approche offre d'avantage de portabilité, tandis que l'on escompte de meilleures performances de la part de la seconde. Cette seconde approche nécessite cependant d'intervenir directement au niveau du noyau du SGBD, ce qui, dans le cas de recherches universitaires, relève souvent de contextes particuliers (coopérations, projets nationaux ou internationaux) ; et n'est pas envisageable pour les SGBD commerciaux. Des solutions intermédiaires entre architectures en couches et arhitectures intégrées ont également été proposées.

La recherche de performances et l'optimisation des systèmes de règles actives sont des thèmes récurrents du domaine. Ils révèlent d'ailleurs une situation relativement ambigüe. D'une part, les règles actives ont été souvent introduites comme un moyen d'optimiser les applications – puisqu'elles permettent d'automatiser et de factoriser des traitements "gourmands" en temps d'exécution : c'est notamment le cas la vérification des contraintes d'intégrité qui reste l'utilisation la plus classique des règles actives. D'autre part, l'une des raisons majeures mise en avant en ce qui concerne la réticence des utilisateurs potentiels des systèmes de règles actives est justement l'inefficacité présumée de ces systèmes en terme de performance. Notons que nous sommes ici face à un réel problème. L'optimisation des SGBD actifs est, en effet, rendue extrêmement ardue par le modèle d'exécution de ces systèmes qui implique une exécution mélée des applications (transactions) et des règles actives. Nous aurons évidemment l'occasion de revenir amplemement sur les modèles d'exécution des SGBD actifs puisqu ils sont au cœur de nos travaux.

Utilisations des règles actives et applications

En contrepoint de ces travaux sur les mécanismes de règles actives, d'autres travaux ont été menés quant à *l'utilisation des règles actives*, c'est-à-dire à l'intégration de règles actives dans des *applications bases de données*. On peut distinguer deux types d'applications ([SKD95]) :

- les *applications passives* qui n'utilisent pas explicitement les règles actives même si le SGBD sous-jacent les utilise – on parle alors d'*utilisations internes* – comme la gestion des vues, des versions, des droits d'accès,le calcul de valeurs par défaut, ou certaines formes de vérification de contraintes d'intégrité ;
- les *applications actives* qui utilisent explicitement les règles actives pour assurer certaines fonctions des applications considérées. Ce sont ici principalement les

nouvelles applications que nous avons déjà évoquées. Les principales utilisations des règles actives sont alors la notification, la communication et la coopération entre les applications et les utilisateurs ou entre différentes parties des applications.

1.2 Problématique et objectifs de notre travail

1.2.1 Un nouveau paradigme de programmation

On prête généralement un énorme potentiel aux mécanismes de règles actives, c'est-à-dire de nombreux domaines d'utilisation, et ce dans de nombreux types d'applications. Certains SGBD commerciaux proposent d'ores et déjà de tels mécanismes et le concept même de *règle active* est en passe de devenir un argument de vente pour les industriels qui commercialisent ces SGBD [Pic95]. Ce phénomène dénote donc une certaine maturité du domaine.

Cependant, bien que les fonctions remplies par les systèmes incluant un mécanisme de règles actives restent encore limitées, en regard des propositions émanant de la recherche, l'utilisation intensive des règles actives dans des applications réelles est très problématique[1] en raison de leur sémantique complexe. Cette sémantique rend les systèmes actifs difficiles à appréhender et surtout à maîtriser – pour les raisons données ci-dessous.

Tout d'abord, les règles actives incarnent un nouveau paradigme de programmation qui se démarque des *programmation impérative (procédurale), fonctionnelle* ou *logique*. Ce nouveau style de programmation, peu familier, suffit déjà à rendre leur utilisation difficile.

Ensuite, maîtriser ce style de programmation ne suffit pas ! En effet, les règles sont déclenchées par l'exécution d'une application qui utilise, la plupart du temps, un autre style de programmation (le plus souvent impératif). Il s'ensuit des interactions entre exécutions d'applications, et plus précisémment entre exécutions de transactions et exécutions de règles qui sont parfois difficiles à contrôler : dans le cadre d'une application, le programmeur exécute une transaction sans toujours bien se rendre compte que cette transaction va être "augmentée" par l'exécution des règles.

Enfin, si la plupart des sytèmes de règles actives sont basés sur le formalisme E-C-A[2], on pourrait presque dire, en caricaturant, que leurs ressemblances s'arrêtent là, du moins en ce qui concerne leur *comportement*. En effet, si les SGBD actifs diffèrent par

1. Elle est même, pour cette raison, souvent déconseillée [SKD95] !
2. Nous reviendrons ultérieurement sur les systèmes qui considèrent des règles EC-A et E-CA.

le modèle de données qui les sous-tend, leur intégration dans ce modèle ou par l'expressivité des langages utilisés pour exprimer les événements, conditions et actions ; c'est, à notre avis,principalement dans leurs modèles d'exécution que l'on rencontre les différences les plus importantes. C'est le modèle d'exécution qui va dicter le comportement du système, c'est-à-dire quand et comment (dans quel ordre, dans quelle transaction, etc.) vont s'exécuter un nombre arbitrairement grand de règles interreliées (règles déclenchées simultanément, règles déclenchées par l'exécution d'autres règles, etc.).

1.2.2 Vers un modèle d'exécution paramétrique

Lorsque l'on considère la recherche dans le domaine des bases de données actives au cours des dix dernières années, on peut isoler deux périodes distinctes : une première période euphorique, au cours de laquelle, de nombreux systèmes ont été proposés, assortis de modèles d'exécution divers et variés ; et une seconde période "de repli", suite à la mise en évidence des problèmes quant à l'*utilisabilité* de ces systèmes – problèmes en grande partie dûs à la très grande diversité, voire à l'antagonisme de leurs modèles d'exécution. On distinguait alors assez mal la puissance et la pertinence des fonctions et sémantiques proposées. C'est au cours de cette seconde période que l'on a réellement commencé à s'intéresser aux domaines d'utilisations des règles actives. On a alors, a posteriori, tenté d'établir des *typologies des règles actives* afin d'établir des correspondances entre tel type d'utilisation ou d'application et telle fonction du modèle d'exécution – ces typologies devant ensuite servir de guides pour proposer des modèles d'exécution ad-hoc pour telle ou telle utilisation des règles [DGG95].

Nous ne croyons guère à cette approche. Tout d'abord, parce que nous ne croyons pas à la pertinence de ces typologies – aussi bien sur le fond que sur la forme. Ensuite, parce qu'on peut découvrir chaque jour une nouvelle utilisation des règles actives, et on ne peut décemment pas construire un modèle d'exécution pour chacune ! Enfin, et surtout, parce que nous pensons que ces différentes utilisations sont, et certainement seront, de plus en plus présentes simultanément au sein d'une même application.

L'objectif de notre travail est donc de proposer **un modèle d'exécution flexible** qui puisse s'adapter aux diverses utilisations et applications des règles actives. Notre modèle est souple, puissant et facile à appréhender. Il permet d'adapter le comportement de modules de règles destinés chacun à une utilisation particulière des règles au sein d'une même application. Il est basé sur une approche *boîte-à-outils* (toolkit en anglais) : le comportement (modèle d'exécution) des règles et des modules de règles est spécifié par *l'instantiation* de paramètres qui peuvent prendre des valeurs parmi

un ensemble prédéfini. Il est également générique (portable en termes d'implantation) puisqu'indépendant, autant que faire se peut, des modèles de données et de modèles de règles. Ceci facilite l'implantation de notre modèle qui peut être vu comme un *moteur d'exécution* pour SGBD actif.

1.3 Démarche et contribution de notre travail

Après avoir défini le cadre de notre étude : les modèles d'exécution des SGBD actifs, nous proposons une taxonomie de ces modèles. Cette taxonomie est constituée de dimensions qui caractérisent l'exécution des règles actives. Nous proposons également une représention graphique des modèles d'exécution des SGBD actifs.

Nous proposons ensuite un modèle d'exécution paramétrique. Pour se faire, nous considérons les dimensions de notre taxonomie et surtout nous étudions les dépendances et interactions entre ces dimensions afin de les transformer en constantes et paramètres qui constituent le modèle d'exécution paramétrique. Celui-ci permet, par instanciation des paramètres, de définir le modèle d'exécution d'une règle prise isolément puis la stratégie d'exécution d'ensembles de règles – ou *modules* – (destinés chacun à une utilisation particulière des règles actives).

Nous présentons ensuite une sémantique dénotationnelle du modèle d'exécution paramétrique. Nous montrons également que le formalisme retenu permet de fournir une spécification implantable du moteur d'exécution.

Nous décrivons ensuite l'expérimentation que nous menons autour de NAOS afin de remplacer son moteur d'exécution par le nôtre. Le projet NAOS – mené depuis 1992 à l'Université de Grenoble (LSR-IMAG) – auquel je participe, vise à fournir un mécanisme de règles actives pour le SGBD O_2. Le prototype NAOS a été principalement développé dans le cadre du projet européen GOODSTEP (Esprit III - no. 6115). Le travail effectué pour NAOS est la source de bon nombre d'observations, de reflexions et d'idées présentées dans cette thèse.

1.3.1 Taxonomie et représentation graphique des modèles d'exécution

Comme nous l'avons souligné, les SGBD actifs se distinguent principalement par leur comportement qui est décrit par leur modèle d'exécution. Celui-ci exhibe un nombre de dimensions variable. Il est difficile, sans une méthodologie adaptée, d'évaluer et de comparer les nombreuses propositions qui ont été faites, afin de déterminer les dimensions essentielles, de celles plus anecdotiques mais surtout afin de déterminer

les interactions entre ces dimensions.

L'approche classique pour étudier, évaluer et comparer les modèles d'exécution des SGBD actifs consiste à étudier un certain nombre de points jugés caractéristiques qui sont principalement : les modes de couplages, les déclenchements multiples d'une règle et les exécutions de règles en cascade.

Notre première contribution est la proposition d'une taxonomie rigoureuse des modèles d'exécution des SGBD actifs. Cette taxonomie est constituée de vingt et une dimensions – avec les valeurs qu'elles peuvent prendre – qui caractérisent l'exécution d'une règle et d'un ensemble de règles. Nous isolons également les dimensions des qui sont dépendantes des modèles de transactions et des modèles de règles.

1.3.2 Un modèle d'exécution paramétrique et sa sémantique formelle

Le principal résultat de notre travail un modèle d'exécution paramétrique pour SGBD actifs nommé **Fl'are** (**Fl**exible **a**ctive **r**ule **e**xecution). Ce modèle a une structure en trois couches : la couche I gère l'exécution d'une règle, la couche II gère l'exécution d'un module de règles, c'est-à-dire les interactions entre les règles d'un module et la couche III gère les interactions entre modules de règles.

La spécification d'un modèle d'exécution pour une utilisation particulière des règles actives consiste simplement à regrouper les règles concernées dans un module puis à instancier les paramètres des couche I et II pour spécifier le modèle d'exécution de chaque règle et la stratégie d'exécution du module.

La sémantique formelle du modèle permet d'en donner une description précise, complète et sans ambiguïté, ce que ne permet pas une description en langue naturelle, aussi soignée soit-elle. Dans la description en langue naturelle du modèle, nous abordons tous les paramètres séparément puis nous tentons de montrer les interactions entre ces paramètres. Évidemment, nous ne pouvons traiter tous les cas, ce qui serait extrêmement long et fastidieux. Nous aurions de plus, toutes les chances d'omettre certains cas. La sémantique dénotationnelle, au contraire, par sa concision, son élégance et sa simplicité (on ne manipule que deux concepts : les ensembles et les fonctions) permet de traiter tous les cas. Un autre avantage, et non des moindres, de la sémantique dénotationnelle est, de part son caractère fonctionnel, de fournir une spécification implantable du modèle.

1.3.3 Exploitation du modèle et de sa sémantique formelle

Une autre caractéristique essentielle du modèle est sa généricité. Il s'agit d'un modèle d'exécution et uniquement un modèle d'exécution. En particulier, il ne décrit pas comment sont représentés les données, les événements et les règles. Il décrit uniquement l'exécution de règles actives. Cette séparation des données et des traitements, qui rapelle les fondements des SGBD modernes, permet une implantation (un *portage*) aisée du moteur Fl'are, ou plus précisément un échange standard du moteur d'exécution d'un SGBD actif par Fl'are.

En effet, en terme d'implantation, Fl'are étant complêtement spécifié par sa sémantique dénotationnelle, "il ne reste presque plus qu'à" spécifier les interfaces avec le SGBD et les autres composants du système de règles actives considérés (le détecteur d'évenements en particulier). Un échange standard du moteur de NAOS est ainsi esquissé.

1.4 Organisation du document

La suite de ce document est organisée de la manière suivante :

- Le chapitre 2 est une présentation du domaine des bases de données actives. Nous résumons tout d'abord les caractéristiques d'un SGBD actif. Nous présentons ensuite le modèle de connaissances qui décrit comment sont représentées et manipulées les données et les règles, en particulier les Événements, les Conditions et les Actions. Le modèle d'exécution décrit quant à lui l'exécution de règles déclenchées par l'exécution d'une transaction.

- Le chapitre 3 présente une étude approfondie des modèles d'exécution des SGBD actifs. Nous isolons les dimensions qui caractérisent l'exécution d'une règle, d'un ensemble de règles ; de celles qui dépendent des modèles de transactions et des modèles de données des SGBD sous-jacent. Nous proposons alors une taxonomie constituée des vingt et une dimensions mises en lumière. Enfin nous proposons une représentation graphique des modèles d'exécution – représentation basée sur la taxonomie proposée – qui permet de comparer aisément les SGBD actifs.

- Dans le chapitre 4, après avoir présenté différentes approches visant à fournir des modèles d'exécution flexibles, à savoir EXACT [DJ94] et Heraclitus [HJ91, SGJ93], nous présentons Fl'are, notre modèle d'exécution paramétrique. Nous décrivons successivement les trois couches qui composent le modèle puis les

interactions entre ces couches. Nous montrons ensuite comment "instancier" Fl'are afin de couvrir les modèles d'exécution de différents systèmes existants.

- Le chapitre 5 définit une sémantique dénotationnelle de Fl'are. Nous introduisons les domaines représentant les événements, les règles, les modules de règles, les transactions, etc. ainsi que les fonctions qui manipulent ces domaines. Nous comparons ensuite notre travail à d'autres travaux de formalisation des modèles d'exécution des SGBD actifs et nous montrons la puissance et la concision de la sémantique dénotationnelle qui présente, en outre, l'avantage de fournir une spécification implantable de Fl'are.

- Le chapitre 6 décrit l'expérimentation que nous menons autour de NAOS afin de remplacer son moteur d'exécution par Fl'are. Nous décrivons tout d'abord l'architecture et l'implantation de NAOS, en insistant sur les interfaces entre ses différents composants. Nous proposons ensuite une implantation de Fl'are qui permet de (ré)utiliser bon nombre des ces composants et interfaces.

- Le chapitre 7 conclut ce document en mettant l'accent sur les apports de notre travail et en donnant quelques perspectives pour des recherches futures : outils d'analyse du comportement des SGBD actifs (traceurs, débogueurs, etc.) et prise en compte de modèles de transactions plus évolués afin d'adapter Fl'are aux applications des SGBD répartis, distribués, ou temps réel, qui se dessinent comme les cibles privilégiées des mécanismes de règles actives.

Chapitre 2

Caractéristiques des SGBD actifs

> ***Généalogie*** *n. Liste des ascendants de quelqu'un, à partir d'un lointain ancêtre qui ne se souciait guère en son temps d'établir la sienne.*
> ***Ambrose Bierce - "Le dictionnaire du Diable"***

Les systèmes de gestion de bases de données exécutent des requêtes et des transactions lancées par les utilisateurs et les programmes d'application. Les systèmes traditionnels sont *passifs* en ce sens que les opérations, calculs, requêtes, manipulations de données et transactions sont exécutés exclusivement à la demande explicite des utilisateurs et des programmes d'application. A contrario, les systèmes *actifs* sont eux capables de détecter des situations ou des faits d'intérêt et de *réagir* pour exécuter automatiquement certaines actions. Cette capacité de réaction des systèmes actifs est exprimée par des *règles actives* ou *règles Événement-Condition-Action*.

Les systèmes actifs suscitent un vif intérêt, aussi bien de la part des chercheurs et des concepteurs de SGBD, que de la part des utilisateurs de ces systèmes. On peut, d'ailleurs, parler d'un renouveau des recherches dans ce domaine depuis le début des années 90. Cette renaissance est visible à la fois par le nombre des systèmes proposés et implantés, et par le nombre de travaux visant à évaluer et comparer ces systèmes. Les raisons de ce regain d'intérêt sont, elles, principalement liées à l'énorme potentiel que l'on prête à ces systèmes, en particulier, dans le cadre des nouvelles applications qui nécessitent des SGBD la capacité de gérer explicitement la dynamique des bases de données – l'émergence de ces nouvelles applications étant elle-même due aux progrès de l'informatique en général et, des bases de données en particulier.

Ce chapitre est une présentation générale du domaine des SGBD actifs. Son objectif est de défricher le domaine, de donner les définitions et références nécessaires à la compréhension de la suite du document, et donc, en définitive, d'esquisser le cadre de notre étude.

La section 2.1 introduit les caractéristiques et les constituants fondamentaux des SGBD actifs. Les sections 2.2 à 2.4 détaillent, cette fois, les constituants primaires des SGBD actifs. Ces sections introduisent une grande part de la terminologie que nous employons dans la suite de ce document. Enfin, la section 2.5 conclut ce chapitre en montrant le besoin d'étudier spécifiquement les modèles d'exécution des SGBD actifs.

2.1 Caractéristiques fondamentales

Un Système de Gestion de Bases de Données (SGBD) actif a deux caractéristiques primaires [DGG95]: premièrement, c'est un SGBD (!) et deuxièmement, il a la capacité de **réagir** à des événements spécifiques pour entreprendre des actions prédéfinies. Cette capacité de réaction est réalisée par l'adjonction d'un système de règles actives à un SGBD.

SGBD : Un SGBD est un système logiciel qui permet d'interagir avec une base de données [DA82] – une Base de Données (BD) étant une collection d'informations inter-dépendentes. Un SGBD permet de modéliser, stocker et manipuler, de manière cohérente et efficace en temps et en mémoire, de très gros volumes d'informations, accessibles simultanément par des nombreux utilisateurs [Ull88], et ce de manière sélective.

Dans une base de données, l'agencement des informations obéit à certaines contraintes. Par exemple, dans une agence de voyage, le nombre de réservations doit être inférieur ou égal au nombre de places disponibles. Dans une application bancaire, le montant des opérations de crédit est égal au montant des opérations de débit. Ce sont ces *contraintes d'intégrité* qui définissent la cohérence de la base.

Un des rôles d'un SGBD est d'assurer la liaison entre l'utilisateur et les données : résultats d'une requête, modifications de données, etc. Le SGBD doit traiter simultanément les accès des différents utilisateurs alors que chacun d'eux raisonne comme s'il était seul à accéder à la base : il doit permettre ces accès concurrents, tout en maintenant la cohérence de la base.

Système de règles actives : Un système de règles actives (SRA) est un système logiciel qui permet de **définir** et d'**exécuter** des règles actives. Un SRA est la "com-

posante active" d'un SGBD actif.

Les règles actives partagent une structure très simple – en trois parties – basée sur le *formalisme Evénement-Condition-Action (ECA)*. La sémantique la plus générale d'une règle utilisant ce formalisme est la suivante :

lorsque un événement de type E se produit
/ on dit que la règle est déclenchée */*
si la condition C est satisfaite
alors exécuter l'action A

Afin d'illustrer le concept de règle active, considérons une application de *gestion de stocks en flux tendu*. Le stockage – de matières premières ou de fournitures – est extrêmement coûteux pour les entreprises. Certaines cherchent alors a réduire au maximun ce coût en reduisant la quantité de chaque article stocké. Pour tenir cette politique, l'entreprise doit être capable de réagir très vite pour réapprovisionner ses stocks afin que la chaîne de production ne soit pas interrompue. Dans un tel contexte, l'utilisation de règles actives telles que la règle NAOS Réapprovisionnment_Stock définie dans la figure 2.1 semble particulièrement appropriée.

Cette règle est déclenchée à chaque mise à jour de la quantité d'un article en stock. Si cette quantité passe sous un certain seuil – ce qui est testé dans la partie Condition de la règle – un procédure de réapprovisionnement est lancée automatiquement. stock (sans majuscule !) représente l'environnement de l'événement qui a déclenché la règle, c'est-à-dire la valeur – avant et après la mise à jour – de l'objet de la classe Stock concerné par la mise à jour.

```
rule Reapprovisonnement_Stock
on update Stock->quantite with stock
if stock->quantite < stock->article->seuil_commande
do {
      Reapprovisionner(stock->article);
}
```

FIG. 2.1 – *Une règle active NAOS*

Cette vision d'un SGBD actif comme un SGBD muni d'un système de règles

correspond à une vision fonctionnelle, architecturale. En terme de représentation et de modèles, un SGBD actif a trois constituants primaires :

1. le *modèle de connaisances* qui décrit comment sont définies, représentées et manipulées les données et les règles ;
2. le *modèle d'exécution* qui décrit comment sont exécutées les règles dans le cadre d'une *application* et ;
3. le *modèle de détection et de production des événements* qui décrit comment les événements sont reconnus et signalés en vue de leur prise en compte pour l'exécution des règles.

Le modèle de connaissances représente, en quelque sorte, l'aspect statique d'un SGBD actif ; tandis que le modèle de détection et de production des événements et le modèle d'exécution en représentent l'aspect dynamique, mécanique.

Application : Une application bases de données est un ensemble de programmes qui manipulent des données stockées dans des bases de données. Les données sont créées en concordance avec un *schéma de données* et stockées dans une ou plusieurs bases associées à ce schéma. Un schéma de données est un ensemble de *types de données*. Une application définie sur un schéma manipule des données qui sont des instances des types définis dans ce schéma.

Une *application bases de données actives* est un ensemble de programmes définis sur un schéma et ses règles associées.

Nous attirons l'attention du lecteur sur la nuance entre les termes *application* bases de données actives, qui correspond à la définition donnée ci-dessus ; et *domaine d'utilisation* des règles actives qui désigne une utilisation particulière des règles actives : intégrité, notification, communication, etc. En clair, on peut trouver des règles destinées à différents domaines d'utilisation au sein d'une même application.
□

Avant de nous consacrer à la description détaillée des différents modèles, nous émettons les deux remarques suivantes, afin d'éviter d'éventuels malentendus ou ambiguïtés :

- **Bases de données actives et objets actifs :** Dans certains travaux, on parle de *base de données actives* – le "s" final sous-entendant que ces sont les données qui sont actives. Ceci concerne principalement les systèmes de règles actives pour des SGBD à objets (SGBDO) dans lesquels on parle également d'*objets actifs* [Buc94] (et [TC92] d'une certaine manière). Nous nous plaçons dans un

cadre plus général et considérons, pour notre part, que c'est le système qui est actif et non les données qu'il manipule. Aussi, dans la suite du document, nous ne parlons plus – sauf erreur de notre part – de bases de données actives, ni d'objets actifs mais de SGBD actifs.

– **Règle de production :** Il existe des similitudes [Wid93] certaines entre *systèmes déductifs* [Bid92] et *systèmes actifs*. Dans les deux cas, le système est capable d'exécuter automatiquement des actions prédéfinies. Les différences majeures entre les deux types de systèmes résident dans leur modèle d'exécution. Un système déductif est sensible aux états de la base : il réagit sur une requête explicite d'une application pour inférer de nouvelles données. Un système actif est sensible aux changements d'états de la base : il réagit automatiquement lorsque survient un événement dans la base ou l'environnement de celle-ci. La ressemblance vient du fait que dans les deux cas, l'exécution d'une règle peut déclencher l'exécution d'autres règles. On considère tantôt le domaine des bases de données actives comme une dégénérescence des bases de données déductives [HW92] – probablement parce que les systèmes déductifs ont été plus étudiés et depuis plus longtemps – tantôt comme un paradigme différent. Notre travail étant justement centré sur les modèles d'exécution des SGBD actifs, nous adoptons – sans polémique aucune – le second point de vue et nous ne nous intéresserons plus à la mécanique d'exécution des systèmes déductifs.

2.2 Modèle de connaissances

Le modèle de connaissances d'un SGBD actif décrit comment sont définies, représentées et manipulées les données et les règles. Le modèle de connaissances a deux éléments constitutifs : le *modèle de données* et le *modèle de règles*.

Modèle de données : Le modèle de données définit la structure des données et les opérations applicables à ces données. Un *schéma* est constitué d'un ensemble de définitions de *types (de données)*. Les données créées en concordance avec un schéma sont stockées dans une (ou plusieurs) base(s) associées à ce schéma. Les modèles de données les plus courants sont : le *modèle relationnel* [Cod70, DA82, DLR91], le *modèle relationnel étendu* [DLR91] et le *modèle à objets* [ABD+92, DLR91]. Nous supposons le lecteur familier du domaine des bases de données et nous ne décrivons pas plus ici ces différents modèles.

Modèle de règles : Le modèle de règles définit la structure des règles et les opérations applicables à ces règles. Le modèle de règles définit ainsi la structure des parties Événement, Condition et Action. Les parties Condition et Action étant généralement très simples, on ne parle pas de *modèle de conditions* ou de *modèle d'actions* ; en revanche, on parle de *modèle d'événements*.

Les opérations applicables aux règles sont en nombre très restreint dans tous les systèmes que nous avons étudiés. Elles se limitent à la *désactivation* et la *ré-activation* d'une règle et au *déclenchement implicite ou explicite* d'une règle. Ces points sont abordés dans le chapitre 3.
□

Enfin, le modèle de connaissances exprime les interactions entre le modèle de données et le modèle de règles, c'est-à-dire : l'intégration du modèle de règles dans le modèle de données et l'influence du modèle données sur la structure des parties Événement, Condition et Action.

2.2.1 Modèle de règles et modèle de données

Les règles d'un système construit au-dessus d'un modèle de données peuvent être attachées à deux niveaux de définition de données :

- la relation pour le modèle relationnel, la classe pour le modèle objet, on parle alors de *règles internes* ;
- le schéma de relations ou de classes, on parle alors de *règles externes*.

Le fait qu'une règle soit interne ou externe prend beaucoup plus d'importance dans le modèle à objets vis-à-vis des concepts d'*héritage*, *encapsulation*, etc. En effet, dans un système à objets, l'espace de visibilité d'une règle interne est l'ensemble des propiétés (attributs et méthodes) publiques et privées de l'objet tandis que l'espace de visibilité d'une règle externe se limite aux propriétés publiques. Par contre, une règle externe étant définie au niveau d'un schéma, elle s'exécute dans le contexte global d'une application manipulant une base de données instance de ce schéma tandis qu'une règle interne s'exécute dans le contexte de sa classe et ne "voit" pas les objets des autres classes du schéma.

Les règles sont internes dans Ode[1], Starburst, Postgres, externes dans Hipac et NAOS, internes ou externes dans SAMOS. Notons que la cohabitation des deux types de règles au sein d'un même système peut rendre difficile pour le programmeur la maîtrise des bases de règles qui seront créées avec ce système.

1. Les références des systèmes que nous évoquons ici sont présentées dans l'annexe A

Un programmeur de règles a une visibilité sur l'ensemble du schéma de données. Il semble donc plus naturel qu'il puisse définir des règles manipulant plusieurs classes (ou plusieurs relations en relationnel), c'est-à-dire des règles externes qui offrent un usage plus universel d'un système de règles.

2.2.2 Événements

Un événement est *"tout se qui se produit, arrive, ou apparaît et qui a une importance toute particulière"* (dictionnaire *Larousse*). Dans les SGBD actifs, les événements ont effectivement une importance toute particulière puisqu'ils ont le pouvoir de *déclencher* une règle, c'est-à-dire de provoquer l'évaluation de la Condition de cette règle – et l'exécution de sa partie Action si la Condition est satisfaite.

Dans le cadre de notre étude[2], la définition donnée ci-dessus et incomplète car elle ne prend pas en compte la notion de *temps*. La notion d'événement est pourtant intrinsèquement liée à la notion de temps. Aussi, de manière pragmatique, redéfinissons-nous ce concept par : *"un événement est quelque chose qui se produit à un instant donné et qui a une importance toute particulière : le pouvoir de déclencher une règle (ou plusieurs)"*.

Événement : Moins prosaïquement, un événement est une occurrence d'un *type d'événement* auquel on peut associer un point de temps – que nous appelerons *instant d'occurrence*.

Instant : Nous introduisons ici la notion de temps de manière très simplifiée sachant qu'il s'agit d'un thème très vaste dont l'approfondissement dépasse le cadre de ce document. Le lecteur peut se référer à [IEE88, TCG+93, Mac86, Adi93] pour des travaux sur le temps et les bases de données temporelles.

Dans la suite, nous considérons que la granularité de la représentation du temps (jour, heure, minute, seconde, etc.) est déterminée par le système ou l'application. Quelle que soit la représentation, il est toujours possible de ramener cette représentation qui correspond au temps du calendrier Grégorien, en un élément du domaine du temps discret ayant comme origine 0 (zéro) et comme extrémité $+\infty$, c'est-à-dire $\mathbb{N}$ (l'ensemble de entiers naturels). On dit que $\mathbb{N}$ est isomorphe au calendrier Grégorien. Un *instant* est un *point de temps* qui appartient à $\mathbb{N}$ (et qui est donc représenté par un entier naturel).

2. Nous utilisons également [CHR96] dans cette section.

Type d'événement : Un type d'événement est une expression qui caractérise une classe de faits significatifs pour le déclenchement d'une règle. Par exemple, un type d'événement peut être "la mise à jour de l'adresse d'une personne". Une occurrence de ce type – ou plus simplement un événement de ce type – sera "la mise à jour de l'adresse de Pierre, le 15 août 1996 à 17h".

Modèle d'événements : Pour un système donné, le modèle d'événements spécifie l'ensemble des types d'événements représentables dans ce système. Les événements (resp. les types d'événements) peuvent être classés en *primitifs* et *composites* (resp. types d'événements primitifs et composites).

Événement primitif : Les événements primitifs sont eux-mêmes classés en quatre catégories : les *événements internes*, les *événements temporels*, les *événements utilisateurs* et les *événements externes*.

1. Les événements internes sont constitués par tous les événements émanant directement du SGBD. Ils sont liés pour la plupart aux manipulations des données et des transactions. Les événements internes sont basés sur les *transitions* entre les différents états d'une base de données. Dans les systèmes relationnels, ces transitions sont causées par les opérations insert, update, delete, parfois retrieve appliquées à un ou plusieurs n-uplets. Dans les systèmes orientés-objet, un événement interne est associé à un appel de méthode ou à une manipulation (lecture, mise-à-jour) de la valeur d'un objet (ou d'une partie d'un objet) ou d'une collection d'objets (insertion et supression). D'autres événements sont associés à la gestion des transactions (début, validation, abandon) : *événements transactionnels* ; et des programmes ou applications (début et fin) : *événements applicatifs*.
2. Les événements temporels sont des événements dont le type fait explicitement référence au temps. Ils ont un large domaine d'utilisation dont les applications basées sur la gestion d'historiques de la base de données et les applications temps réel. Ils peuvent se classer en deux catégories :
   (a) les *événements temporels absolus* (ex: "le 19 décembre 1965") et les *événements temporels périodiques* (ex: "tous les premiers lundis du mois") qui sont réellement des événements primitifs ;
   (b) les *événements temporels relatifs* (ex: "10 minutes après le début d'une transaction") qui sont parfois considérés comme des événements composites (cf. paragraphe suivant).

3. Les événements utilisateurs ne sont pas nécessairement liés à des opérations sur une base. Ils ne peuvent donc pas être directement détectés par le système et doivent être *signalés* explicitement dans le code des applications.

4. Les événements externes ne sont liés à aucun phénomène opératoire du SGBD mais proviennent de l'environnement de celui-ci et peuvent être capturés par le système de règles. Des exemples d'événements externes sont l'arrivée d'un courrier électronique (e-mail) ou l'arrivée de relevés en provenance d'un thermomètre, une alarme, etc.

La prise en compte des événements utilisateurs et externes permet, d'une part, d'étendre à volonté le modèle d'événement et, d'autre part, d'étendre le champ d'application des règles actives vers des domaines qui sortent du cadre des bases de données.

Événement composite : Les événements composites sont construits récursivement à partir d'événements primitifs grâce à des opérateurs de composition dont les plus répandus sont : la disjonction, la conjonction, la négation et la séquence. Chaque système offrant des types d'événements composites propose un ensemble d'opérateurs plus ou moins riche – NAOS propose, par exemple, deux types de disjonctions et deux types de séquences [Cou93, CC96a] – avec des sémantiques diverses et variées.

Les modèles et la gestion (c'est-à-dire le processus de détection) d'événements composites constituent, à eux seuls, un domaine de recherche dont l'approfondissement dépasse le cadre de notre travail. Nous invitons le lecteur à se référer directement aux travaux de Ode [GJS92, NG92], SAMOS [GGD93, GD94], Snoop [CM93], Chimera [MPC96] et NAOS [CC96a, CC96b].

Signalons enfin, que plus encore que pour les événements primitifs, la notion d'*occurrences simultanées* d'événements est intrinsèquement liée à celle d'événement composite. Nous verrons dès le chapitre 3 que cette notion a une certaine importance vis-à-vis des modèles d'exécution des SGBD actifs.

Environnement d'un événement : Tout événement véhicule des informations qui le distinguent et renseignent sur les conditions dans lesquelles il s'est produit. Ces informations constituent l'*environnement de l'événement*. Si l'on considère à nouveau le type d'événement "mise à jour de l'adresse d'une personne", l'environnement d'un événement de ce type contiendra la valeur du n-uplet ou de l'objet représentant la personne concernée – et en particulier la valeur de l'adresse avant et après l'opération de mise à jour.

2.2.3 Conditions

Une condition est *"un état, une situation ou un fait, dont l'existence est indispensable pour qu'un autre état, un autre fait existe"* (dictionnaire *Le petit Robert*). Dans le cas des règles actives, une condition est une formule qui doit être *évaluée* à `vrai` pour que l'action soit *exécutée*.

Une Condition est une formule composée de prédicats sur l'état de la base de données pour les systèmes relationnels ou sur l'ensemble des objets temporaires et persistants pour les systèmes à objets. Les prédicats sont construits à partir d'expressions du langage de requêtes ou d'expressions logiques. Dans ces expressions, il est souvent possible d'accéder aux n-uplets ou objets insérés ou supprimés, à l'ancienne et à la nouvelle valeur des n-uplets ou objets modifiés – c'est-à-dire à l'environnement des événements. Dans le cas des SGBDO, les expressions peuvent, en outre, comporter des appels de méthodes. Le résultat de l'évaluation de la condition doit être `vrai` pour que la partie Action de la règle soit exécutée.

Au niveau des langages de règles, la partie Condition est souvent optionnelle. On considère dans ce cas que la Condition est toujours satisfaite.

2.2.4 Actions

Une action est *"ce que fait quelqu'un ou quelque chose et par quoi il réalise une intention ou une impulsion"* (dictionnaire *Le petit Robert*).

Dans les SGBD relationnels, le langage utilisé pour spécifier une Action est souvent le langage de requêtes du SGBD, éventuellement étendu avec des constructions permettant d'accéder aux n-uplets insérés ou supprimés, à l'ancienne et à la nouvelle valeur des n-uplets modifiés. Dans le cas des SGBDO, la partie Action est comparable à une procédure du langage d'accès du SGBD, elle peut évidemment comporter des appels de méthodes et peut accéder aux objets insérés ou supprimés, à l'ancienne et à la nouvelle valeur des objets modifiés. Il est aussi souvent possible d'accéder au résultat de l'évaluation de la Condition associée à la règle.

Dans la plupart des systèmes, les opérations possibles sur les transactions, dans la partie Action d'une règle se limite à l'abandon (`abort`) de la transaction dans laquelle s'exécute cette règle.

2.2.5 Environnements d'évaluation et d'exécution

Nous avons vu précédemment qu'un environnement pouvait être associé à toute occurrence d'événement. Dans la plupart des systèmes, le modèle de règles offre un mé-

canisme pour accéder à ces environnements dans les Conditions et Actions des règles. *L'environnement d'évaluation* d'une règle R – accessible dans la partie Condition de R – contient l'environnement de l'événement déclenchant de R. *L'environnement d'exécution* de R – accessible dans la partie Action de R – contient l'environnement de l'événement déclenchant de R plus éventuellement le résultat de la requête qui constitue la partie Condition.

Les mécanismes offerts pour accéder aux environnements d'évaluation et d'exécution sont implicites ou explicites. À notre connaissance, aucun n'inclut l'aspect temporel des événements (accès à l'instant d'occurrence par exemple). Dans les systèmes relationnels, ces mécanismes (implicites) sont souvent basés sur l'utilisation de *mots clés* du langage de règles : inserted, deleted, old_updated, new_updated pour A-RDL et Starburst ; current, new, old pour Postgres. Dans les systèmes à objets, les mécanismes sont plus souvent explicites et reposent sur la possibilité pour le programmeur de désigner – par le langage de règles – de structures de données spécifiques, et de manipuler ces structures par des opérateurs prédéfinis. C'est, par exemple, la solution adoptée dans NAOS. Dans la figure 2.1, e est une *delta structure* qui désigne l'environnement d'évaluation de la règle. e est manipulée grâce aux opérateurs new et current.

Ces environnements n'ont de rapport avec les modèles d'exécution que dans la mesure ou le moteur de règles, qui implante le modèle d'exécution, doit les gérer, c'est-à-dire les stocker et les utiliser pour l'évaluation et l'exécution des règles. Nonobstant, nous considérons que ce point des modèles d'exécution n'a pas de réelle influence sur les modèles d'exécution, c'est-à-dire sur la *mécanique* des exécutions et, dans la suite du document, nous ne traiterons plus des environnements d'événement, d'évaluation ou d'exécution.

2.3 Modèle d'exécution

L'étude des modèles d'exécution des SGBD actifs constitue le cœur de notre travail. Notre objectif, dans cette section, est simplement de donner une vision générale et d'introduire un certain nombre de termes et concepts[3] nécessaires à une bonne compréhension des chapitres suivants.

Modèle d'exécution : Le modèle d'exécution d'un SGBD actif spécifie quand et comment est exécuté, au cours d'une application, un nombre arbitrairement grand de

3. La plupart de ces termes et concepts seront redéfinis, de manière plus précise dans le chapitre suivant.

règles inter-reliées. Une application bases de données (en l'absence de règles actives) s'exécute selon le *modèle d'exécution du SGBD*. De la même manière, les règles s'exécutent selon le *modèle d'exécution du système de règles actives* (SRA).
□

En réalité, on ne peut pas réellement considérer de manière indépendante ces deux modèles car le modèle d'exécution du SGBD influence fortement le modèle d'exécution des règles sur certains points. Aussi, nous considérons que le modèle d'exécution d'un SGBD actif décrit, à la fois, le modèle d'exécution du SGBD, le modèle d'exécution du système de règles et les interactions entre ces deux modèles.

2.3.1 Exécution d'applications

Une caractéristique essentielle d'une application base de données est que les opérations de manipulation de données sont placées dans des *transactions*. On peut alors considérer que le modèle d'exécution du SGBD – qui décrit l'exécution des applications – est confondu avec le modèle d'exécution des transactions – ou plus simplement le *modèle de transactions* du SGBD.

Une transaction [DA82, AE92] est une séquence d'opérations qui doit conserver la cohérence de la base, c'est-à-dire la "faire passer" d'un état cohérent vers un autre état cohérent. Durant une transaction, la base peut passer temporairement par des états incohérents. Une transaction doit assurer également que les résultats obtenus vont être correctement enregistrés dans la base. C'est elle qui va définir les instants où les modifications effectuées doivent devenir définitives. Enfin, une transaction doit s'exécuter indépendamment des autres transactions. Il faut pour cela que les transactions aient des propriétés telles que les programmeurs puissent ignorer l'existence des autres transactions.

La gestion des transactions [GR93, Pap86] a principalement pour objet :

- de contrôler les accès concurrents et synchroniser les transactions en conflit ;
- de gérer la reprise après panne, c'est-à-dire de remettre la base dans le dernier état cohérent connu et éventuellement relancer les transactions interrompues ;
- enfin – et c'est le point qui nous concerne le plus – d'initialiser et de contrôler les transactions. Une transaction démarre par une opération *début* (start) ; elle se termine par une *validation* (commit) ou un *abandon* (abort). La validation termine une transaction en rendant ses résultats définitifs ; l'abandon, au contraire, remet la base dans l'état ou elle se trouvait avant la transaction. L'abandon d'une transaction peut-être provoquée par la transaction ou par des erreurs ou des pannes, etc.

Les transactions peuvent être manipulées par le programmeur : les opérations *début*, *validation* et *abandon* sont accessibles dans le langage d'accès du SGBD. Ce sont d'ailleurs ces opérations qui peuvent produire des événements transactionnels si les types d'événements correspondants existent dans le modèle d'événements.

Le mécanisme d'exécution des transactions représente le modèle d'exécution du SGBD (où modèle d'exécution des applications). Le modèle de transactions auquel nous ferons le plus fréquemment référence est le *modèle de transactions classique* dans lequel toutes les transactions ont les propriétés ACID : Atomicité, Cohérence, Isolation, Durabilité [DA82, AE92]. Dans ce modèle, il n'y a pas de transactions emboîtées ou séparées – les modèles offrant de telles transactions sont appelés *modèles de transactions emboîtées* [Mos85, BK91].

2.3.2 Exécution de règles

Le modèle d'exécution d'un système de règles actives spécifie quand et comment sont exécutées les règles. L'extrême simplicité de la sémantique du modèle Événement-Condition-Action ("lorsqu'un événement du type E survient, si la condition C est vérifiée alors exécuter l'action A") n'est qu'apparente ; elle masque en réalité de nombreuses questions auxquelles le modèle d'exécution doit répondre :

1. Quand exécuter une règle par rapport à l'événement qui l'a déclenchée ? Doit-on évaluer sa partie Condition et exécuter sa partie Action immédiatement ? Doit-on différer l'évaluation ? L'exécution ? Les deux ? Différer jusqu'à quand ?
2. Doit-on suspendre le processus qui a produit l'événement qui a déclenché une règle ? Doit-on attendre la fin de ce processus ? Doit-on exécuter la règle en parallèle de ce processus ?
3. Que faire lorsqu'une règle est déclenchée par plusieurs événements ?
4. Que faire des événements après qu'ils aient occasionné l'exécution d'une ou plusieurs règles ?
5. Que faire lorsque plusieurs règles sont déclenchées simultanément par un même événement ? Par des événements différents ?
6. Que faire lorsque l'exécution d'une règle produit des événements qui, à leur tour, déclenchent cette même règle ou d'autres règles ? Doit-on toutes les exécuter ? Dans quel ordre ? L'exécution de ces règles doit-elle interrompre l'exécution de celle qui les a déclenchées ?

Au vrai, ces questions sont au centre de notre travail et constitue le principal objet de ce document.

2.3.3 Exécution d'applications avec des règles

L'introduction d'un système de règles actives dans un SGBD va sensiblement – et c'est un euphémisme – modifier le modèle d'exécution de ce SGBD. Le modèle d'exécution d'un SGBD actif est, en réalité, une *alchimie* entre modèle d'exécution des applications et modèle d'exécution des règles. Les deux modèles cohabitent au sein du SGBD et il est difficile [4] de parler d'exécution de règles sans utiliser les termes d'*application* ou de *transaction*.

Dans une première approximation, nous considérons l'exécution d'une application bases de données active comme l'interaction de deux processus. Un processus application exécute, en séquence, les opérations qui composent l'application ("non active"). Lorsqu'un événement est produit par une opération, le second processus : le moteur d'exécution est alors réveillé pour l'exécution des règles actives.

Point d'exécution : Lorsque l'application s'exécute sans produire d'événements, le moteur d'exécution est endormi. Il est réveillé lorsqu'une règle est *évaluable* ou *A-exécutable*. Une règle est évaluable lorsque sa partie Condition est susceptible d'être évaluée [5] ; elle est A-exécutable lorsque sa partie Action est susceptible d'être exécutée.

Le moteur d'exécution doit, *à certains instants*, "s'arrêter pour examiner" les règles afin de déterminer celles qui sont évaluables ou A-exécutables pour ensuite sélectionner, parmi elles, la prochaine règle à traiter (évaluation de la partie Condition et/ou exécution de la partie Action). Ces instants particuliers sont appelés des *point d'exécution*. Un point d'exécution est un point de temps (cf.section 2.2.2, paragraphe Instant) associé :

1. à l'instant d'occurrence d'un événement produit par l'application ;
2. à la fin de l'exécution d'une règle ;
3. à l'instant d'occurrence d'un événement produit par l'exécution d'une règle.

Nous verrons dans la suite que le comportement d'un moteur de règles est très différent en cours de transaction et en fin de transaction. Compte tenu de ce fait, on peut considérer qu'il existe six types de points d'exécution :les trois types énumérés ci-dessus en cours et en fin de transaction.

Unités de production et d'exécution : Comme nous l'avons dit précédemment, les opérations de manipulation des données d'une base sont placées dans des transactions. Bien que certains événements, temporels ou externes par exemple, peuvent

4. Comme l'a peut-être noté le lecteur, dans la section précédente!

5. Il faut évidemment qu'elle soit déclenchée mais ce n'est pas la seule condition comme nous le verrons dans le chapitre 3.

être produits en dehors d'une transaction, nous considérons que l'*unité de production (des événements)* est la transaction. On appelle une telle transaction une *transaction déclenchante*.

De la même manière, nous considérons que les règles sont toujours exécutées dans des transactions. On dira que *l'unité d'exécution (des règles)* est la transaction et on parlera de *transaction déclenchée*.

Ces précisions nous permettent de spécifier le cadre de notre étude : quand nous décrivons le processus application, nous ne considérons pas, en réalité, une application dans son ensemble, mais les transactions qui la composent. Ceci offre un cadre à la fois plus réduit et plus solide – puique la notion de *transaction* est mieux définie que celle un peu floue d'*application*.

En conclusion, le modèle d'exécution d'un SGBD actif spécifie quand et comment sont exécutées des règles actives déclenchées par des événements produits par l'exécution d'une transaction.

2.4 Modèle de détection et de production des événements

Dans la section précédente, nous avons défini l'exécution d'une application BD actives comme l'interaction de deux processus que nous avons nommés application et moteur d'exécution : le moteur d'exécution étant réveillé suite à des événements produits par l'application. Ce "réveil" ne se fait pas tout seul, un SGBD classique ne produit pas d'événement ! Pour cela, il est nécessaire de disposer d'un troisième processus, que nous appelons détection et production des événements.

C'est ce processus qui "surveille" l'exécution des applications, qui *détecte* ou *reconnait*[6] les événements et qui les *signale* au moteur d'exécution. C'est donc lui seul qui, réellement, *produit* les événements qui déclenchent les règles. Ceci est d'autant plus vrai si l'on considère les événements qui ne se sont pas associés à des opérations sur les données d'une base (événements temporels, utilisateurs et externes).

Le modèle de détection et de production des événements spécifie notamment la *granularité du déclenchement* et le *mode de production des événements composites*.

La granularité du déclenchement spécifie à quelle granularité de l'application les événements sont produits. La granularité du déclenchement a donc une influence sur les points d'exécution puisqu'elle spécifie quand le moteur d'exécution va être ré-

6. On parle généralement de *détection* pour les événements primitifs et de *reconnaissance* pour les événements composites.

veillé. Les granularités possibles sont, par ordre décroissant, la *session*, la *transaction*, l'*opération*, l'*opération élémentaire* et l'*opération de lecture/écriture*. La session correspond à l'exécution complète d'une application. Dans les systèmes relationnels, une opération peut être, par exemple, une requête, décomposée en opérations élémentaires (update, insert, delete, etc.), elles-mêmes divisées en opérations de lecture/écriture. Dans les système à objets, une opération peut être un appel de méthode ; méthode composée d'opérations élémentaires (création d'objets, mises à jour d'objets, etc.), elles-mêmes composées d'opérations de lecture/écriture. Cette notion de *granularité* pose certains problèmes, elle est à manier avec précaution. Dans les systèmes à objets, par exemple, une méthode (granularité : opération) peut elle-même être composée d'appels de méthodes (granularité : opération) qui ont la même granularité qu'elle.

Le mode de production des événements composites a été introduit dans Snoop [CM93] – sous le nom de *parameter context*. Ils sont au nombre de quatre : *continu*, *récent*, *chronique* et *cumulatif* ; et indiquent les combinaisons d'événements primitifs constituant un événement composite. Ils offrent ainsi la possibilité au programmeur de préciser la sémantique des types d'événements composites.

Exemple Le tableau 2.1 illustre, sur une exemple, les différents modes de production. Le type d'événement composite E considéré est : E = E1 ; E2 dans lequel E1 et E2 sont des types d'événements primitifs, ";" est l'opérateur de séquence, e12, e13, e15 sont des événements du type E1 et e24, e26, e27 sont des événements du type E2.

		Modes de production			
IO	EP	continu	récent	chronologique	cumulatif
1	e12				
2	e13				
3	e24	(e12;e24) (e13;e24)	(e13;e24)	(e12;e24)	({e12,e13};e24)
4	e15				
5	e26	(e12;e26) (e13;e26) (e15;e26)	(e15;e26)	(e13;e26)	({e12,e13,e15};e26)
6	e27	(e12;e27) (e13;e27) (e15;e27)	(e15;e27)	(e15;e27)	({e12,e13,e15};e27)

TAB. 2.1 – *Modes de production des événements composites*

La première colonne indique les Instants d'Occurrence (IO) des événements primitifs : colonne EP, et composites : colonnes suivantes.

□

Comme nous l'avons déjà souligné, les modèles et les processus de détection des événements constituent à eux seuls un vaste domaine de recherche. Les résultats obtenus ne sont pas utilisables uniquement dans le domaine des bases de données actives mais au contraire dans de nombreux autres domaines qui utilisent les notions d'*événement* et de *temps* : BD temporelles, BD temps réel, applications temps réel (sans BD), programmation événementielle, etc.

Par ailleurs, les modèles d'exécution de règles actives constituent égalemment un vaste domaine de recherche et comme nous le montrons dans la suite de ce document, il est possible d'étudier le processus d'exécution de règles actives en faisant presque complétement abstraction de la nature des événements et de leur processus de détection.

Dans la plupart des travaux sur les bases de données actives (cf. Annexe A), on considère que le modèle de détection et de production des événements fait partie du modèle de règles (plus précisément du modèle d'événements) ; et que le moteur de règles est responsable de la détection des événements.

Dans la suite de ce document, nous focalisons notre attention sur l'exécution des règles actives. Nous supposons l'existence d'un modèle et d'un processus de détection des événements tel que nous venons de le décrire mais nous considérons, pour les deux raisons indiquées ci-dessus, qu'il ne fait pas partie du modèle d'exécution des SGBD actif.

2.5 Conclusion

Ce chapitre a défini les principaux composants des SGBD actifs et introduit la terminologie que nous employons dans la suite. Les aspects essentiels pour la construction d'un SGBD actif sont la spécification du modèle de connaissances (modèle de données et modèle de règles), du modèle de détection et de production des événements, et finalement la spécification du modèle d'exécution.

Nous pensons que les SGBD actifs se distinguent principalement par leur comportement, qui est décrit par leur modèle d'exécution. La richesse des possibilités offertes dans la combinaison des différents aspects des modèles d'exécution fait qu'il est difficile, sans une méthodologie adaptée, d'évaluer et de comparer les nombreuses propositions qui ont été faites. Ceci est pourtant de première importance si l'on cherche à déterminer les dimensions essentielles, de celle plus anecdotiques. Et plus encore si l'on cherche à déterminer les interactions entre ces dimensions. Cet aspect est développé

dans les chapitres suivants dans lesquels nous étudions les modèles d'exécution dans le détail afin de proposer une taxonomie de ces modèles puis un modèle d'exécution paramétrique.

Nous n'avons pas encore traité de l'architecture et de l'implantation des SGBD actifs. Les solutions proposées sur ces aspects sont souvent déterminantes pour les performances des systèmes actifs, qui comme nous l'avons déjà souligné, semble être un aspect crucial pour le succès de ces systèmes – en vue d'un transfert vers les SGBD commerciaux. Cet aspect fait l'objet du chapitre 6.

Enfin, signalons que nous n'avons pas mené d'étude particulière en ce qui concerne "l'optimisation des SGBD actifs". Afin de mener des travaux dans cette direction, il conviendrait tout d'abord, selon nous, de définir très précisément ce que l'on entend par *optimisation des systèmes de règles actives*. En effet, une certaine ambiguïté entoure ce terme – ambiguïté qui provient peut-être de l'ascendance des systèmes déductifs sur les systèmes actifs ?

Chapitre 3

Taxonomie des modèles d'exécution pour SGBD actifs

> ***Connaisseur*** *n. Spécialiste qui sait tout à propos d'une chose et rien à propos de tout le reste.*
> *Un vieil amateur de vin avait été sérieusement blessé lors d'une collision ferroviaire, un peu de vin lui fut donné à boire pour le ranimer. "Pauillac 1873", murmura-t-il dans son dernier soupir.*
> ***Ambrose Bierce - "Le dictionnaire du Diable"***

Le modèle d'exécution d'un SGBD actif détermine l'exécution d'un ensemble de règles actives, c'est-à-dire quand et comment ces règles sont exécutées au cours d'une application. Si les différentes propositions de SGBD actifs diffèrent par leur modèle de connaissances, c'est – à notre avis – essentiellement leur modèles d'exécutions qui présentent les différences les plus profondes et les plus difficiles à appréhender pour un utilisateur de ces systèmes.

Dans ce chapitre – qui est une étude approfondie des modèles d'exécution des SGBD actifs – nous dressons, dans les sections 3.1 et 3.2, l'inventaire d'un certain nombre de *dimensions* qui caractérisent respectivement l'exécution d'**une** règle puis d'**un ensemble** de règles.

Nous prenons le terme *dimension* dans son sens le plus commun qui est : *un aspect significatif d'une chose* (dictionnaire petit Robert). Nous ne considérons pas son sens "géométrique", que l'on trouve, par exemple, dans "espace à trois dimensions". En

clair, les dimensions que nous mettons à jour dans ce chapitre ne sont pas indépendantes. L'étude des dépendances et interactions entre nos dimensions fait, en partie, l'objet du chapitre 4.

Les dimensions que nous proposons nous permettent de comparer les modèles d'exécution des principaux SGBD actifs existants. Les références de ces systèmes, et plus généralement, les principaux travaux que nous avons utilisé dans cette thèse sont présentés dans l'annexe A. Aussi, dans la suite de ce chapitre, nous indiquerons des références bibliographiques uniquement lorsque les informations mentionnées se trouvent dans une publication particulière.

Dans la section 3.3, nous traitons des liens entre le modèle d'exécution d'un système de règles et le modèle de transactions du SGBD sous-jacent. Dans la section 3.4, nous proposons une taxonomie des modèles d'exécution pour SGBD actifs. Enfin, la section 3.5 conclut ce chapitre.

3.1 Exécution d'une règle

Dans cette section, nous nous intéressons à l'exécution d'**une** règle. Nous ne tenons donc pas compte des possibles interactions de cette règle avec d'autres règles. Ce point fait l'objet de la section 3.2.

La figure 3.1 met en évidence les trois phases de l'exécution d'une règle :

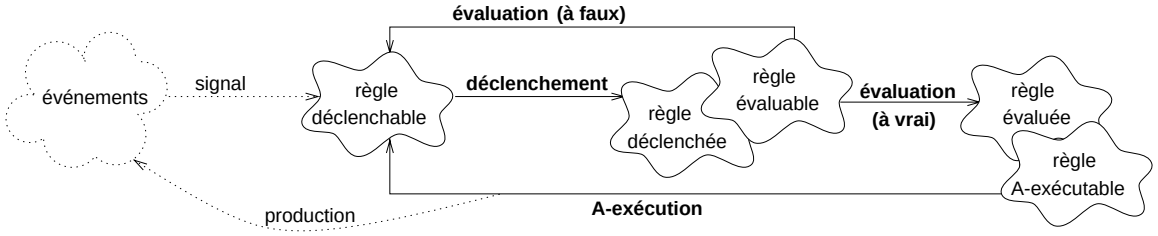

FIG. 3.1 – *Exécution d'une règle*

1. le **déclenchement** est le processus par lequel une règle passe d'un état *déclenchable* – c'est-à-dire un état que l'on peut qualifier de "repos" ou de "sommeil" – à un état *déclenchée*, suite à l'occurrence d'un ou plusieurs événements dont au moins un est un événement déclenchant ;
2. l'**évaluation** correspond à l'évaluation de la partie Condition de la règle. Si la condition est satisfaite, la règle passe dans un état *évalué* – sinon elle revient à l'état *déclenchable* ;

3. l'**A-exécution** correspond à l'exécution de la partie Action de la règle. Après exécution[1], la règle revient à l'état *déclenchable*.

Remarque : Une règle *déclenchable* devient *déclenchée* dès l'occurrence d'un événement déclenchant. En revanche, une règle *déclenchée* n'est pas forcément évaluée immédiatement. De la même manière, une règle *évaluée* n'est pas forcément A-exécutée immédiatement. Pour qu'une règle soit évaluée (resp. A-exécutée), il faut qu'elle soit *évaluable* (resp. *A-exécutable*).

Pour illustrer ce phénomène, nous avons schématisé dans la figure 3.1 deux états distincts pour indiquer qu'une règle peut être *déclenchée* mais non *évaluable*.

Les transitions vers les états *évaluable* ou *A-exécutable* dépendent du type de point d'exécution "auquel on se trouve" lorsque l'on traite la règle ; et des dimensions de l'exécution d'une règle : *mode de préemption* et *mode de déclenchement*, que nous allons détailler dans la suite de ce chapitre.

3.1.1 Déclenchement

3.1.1.1 Traitement des événements

Lors de sa définition, une règle R est associée à un type d'événement E. Lorsque plusieurs occurrences (ou instances) de E sont produites dans une même unité de production, il existe deux stratégies de prise en compte de ces événements qui influent sur le déclenchement de R :

1. la première stratégie consiste à traiter les événements instance par instance et donc à exécuter R autant de fois qu'il y a d'événements déclenchants, on parle alors de *sémantique d'instance* ;
2. la seconde stratégie consiste à n'exécuter R qu'une fois pour l'ensemble des événements, on parle alors de *sémantique ensembliste*.

Les systèmes relationnels Starburst et Ariel adoptent une sémantique ensembliste, Postgres – ainsi que la norme SQL3 et certains produits commerciaux comme Informix ou Oracle – une sémantique d'instance. Dans les systèmes à objets, la sémantique d'instance est la plus utilisée : Exact, Ode, Reach, Samos, Sentinel, TriGS, Beeri & Milo, à l'exception de Chimera. Dans NAOS, on trouve les deux sémantiques.

1. Le terme *exécution* désigne le processus complet déclenchement/évaluation/A-exécution. La fin de l'A-exécution consitue donc égalemment la fin de l'exécution de la règle.

Instant de déclenchement : Le mode de traitement des événements détermine l'*instant de déclenchement* d'une règle, c'est-à-dire l'instant associé à la transition de l'état *déclenchable* vers l'état *déclenché*. Pour une règle ayant une sémantique d'instance, l'instant de déclenchement est l'instant d'occurrence de l'événement déclenchant. Pour une règle ayant une sémantique ensembliste, l'instant de déclenchement de la règle est l'instant d'occurrence de son premier (dans le temps) événement déclenchant.

Occurrence d'une règle : Une *occurrence d'une règle* est un triplet $< r, t, e >$ dans lequel r est la règle considérée, t est son instant de déclenchement et e son environnement (d'évaluation et d'exécution).

La notion d'occurrence d'une règle est importante en ce sens qu'en fonction du mode de traitement des événements et du mode de consommation des événements (décrits dans la section suivante), il est possible qu'à un point de considération, il existe plusieurs occurrences d'une même règle (la règle a été déclenchée plusieurs fois).

3.1.1.2 Consommation des événements

Comme nous venons de le voir, une règle est susceptible d'être exécutée plusieurs fois dans une même unité de production. Chaque événement déclenchant e_i est conservé jusqu'à la première exécution de la règle qu'il a déclenché. Un événement est donc pris en compte au moins une fois au cours de cette exécution. En ce qui concerne les éventuelles exécutions ultérieures de cette même règle, deux politiques de traitement de l'événement peuvent être adoptées :

- *Consommation :* l'événement e_i est consommé par la première exécution de la règle et ignoré dans les exécutions ultérieures (il n'apparait pas dans l'environnement de la règle lors de ces exécutions ultérieures).
- *Préservation :* après avoir été pris en compte une première fois, l'événement e_i est préservé et donc pris en compte dans les exécutions ultérieures de la règle (il reste visible dans l'environnement de la règle lors de ces exécutions ultérieures).

La consommation des événements peut être *locale*. Dans ce cas, l'événement e_i peut déclencher d'autres règles non encore considérées. La consommation peut également être *globale*. Dans ce cas, e_i est ignoré dans les exécutions ultérieures de la règle mais aussi dans celles de toutes les autres règles (e_i aura donc provoqué l'exécution d'une seule règle).

La plupart des systèmes ont adopté une politique de consommation locale des événements. Postgres, en revanche, a adopté une politique de consommation globale. A-RDL a adopté une politique de préservation. Enfin, Chimera laisse à l'utilisateur la possibilité de spécifier la politique qu'il désire pour chaque règle.

La consommation peut avoir lieu soit à l'évaluation de la règle, soit, de manière plus restrictive, à l'A-exécution de la règle. Cette dernière possibilité implique que l'événement est réellement consommé uniquement si la condition est satisfaite. Seul Starburst dans [WF90] avait adopté cette politique mais lors de l'implantation du système, cette politique fut révisée[2].

3.1.1.3 Effet net

L'*effet net* d'une séquence d'opérations est le bilan de ces opérations qui perdurent à la fin de cette séquence. Si, par exemple, une séquence d'opérations crée puis détruit une entité, l'effet net sera considéré comme nul. Si une séquence d'opération crée deux entités `a` et `b` puis détruit `a`, l'effet net sera la création de `b`.

L'effet net peut être ou ne pas être pris en compte pour l'exécution d'une règle. Il constitue par là une dimension importante du modèle d'exécution mais cette dimension est difficile à appréhender car il n'existe pas de réel consensus quant à (1) la définition et le calcul de l'effet net, et (2) son utilisation, c'est à dire son influence sur l'exécution des règles.

Calcul de l'effet net L'effet net est déterminé selon des critères qui varient d'un système à l'autre. La notion d'effet net a été largement traitée dans le cas des systèmes s'appuyant sur des SGBD relationnels [KMS90, AWH92]. Dans ces systèmes, les opérations élémentaires sur les données sont limitées à (1) l'insertion, (2) la suppression et (3) la modification de n-uplets. Ceci facilite l'expression de l'effet net. Il suffit de donner l'effet résultant de toute séquence d'opérations élémentaires (insertion, suppression, mise-à-jour) prise deux à deux sur le même n-uplet.

L'effet net des actions dans les systèmes à objets a été moins traité que dans les systèmes relationnels. La grande différence vient du fait que ces derniers supportent un modèle de données basé sur des valeurs (ensembles de n-uplets de valeurs atomiques), alors que dans les systèmes à objets la notion de valeur joue souvent un rôle secondaire par rapport à celui de la notion d'objet. Il faut noter que la plupart des travaux sur les règles actives se sont restreints à prendre en compte les événements issus des exécutions des méthodes en contournant ainsi les aspects liés à la structure et à la

2. Les raisons de ce changement n'ont pas été explicitées.

valeur des données et ne proposant pas de support à l'effet net. Pour un modèle de règles plus complet, il est nécessaire de prendre en compte les notions propres à l'objet telles que l'identificateur d'objet, l'encapsulation ainsi que la structure complexe des données, ce qui complique la définition de l'effet net.

Influence de l'effet net La prise en compte de l'effet net se manifeste principalement sur deux aspects de l'exécution d'une règle :

- Le premier aspect concerne le déclenchement lui-même de la règle. Prenons l'exemple d'une règle avec une sémantique ensembliste et avec effet net. L'ensemble des événements qu'elle va traiter est formé de ceux qui n'ont pas été annulés par d'autres événements. Si tous les événements s'annulent, certains systèmes n'exécutent pas cette règle qui avait pourtant été déclenchée.
- Le second aspect concerne la construction de l'environnement d'exécution de la règle. La prise en compte de l'effet net permet, d'une certaine manière, de construire des environnements d'exécution plus "robuste". Prenons l'exemple d'une règle déclenchée par la création d'une entité, avec une sémantique d'instance et qui ne prendrait pas en compte l'effet net. L'entité concernée peut avoir été détruite entre le déclenchement et l'A-exécution de la règle. La manipulation de cette entité dans la Condition ou l'Action de la règle peut alors se révéler très dangereuse, car susceptible d'entrainer incohérences et erreurs.

La plupart des systèmes de règles actives basés sur des SGBD relationnels que nous avons pu étudier (A-RDL, Ariel, Starburst) offre un modèle d'exécution dans lequel l'effet net est systématiquement pris en compte à la fois pour les déclenchements et pour la construction des environnements d'exécution. Postgres, dans lequel la notion même d'effet net n'existe pas, constitue une exception notable sur ce point.

En ce qui concerne les systèmes de règles actives basés sur des SGBD à objets, la situation est moins consensuelle :

- dans Ode, l'effet net est systématiquement pris en compte pour le déclenchement de toutes les règles ;
- dans NAOS, l'effet net est systématiquement pris en compte à la fois pour le déclenchement et la construction des environnements d'exécution de toutes les règles ;
- dans Chimera, il est possible de spécifier pour chaque règle si l'effet net doit être pris en compte ou non, et il l'est uniquement au travers des opérateurs du langage de règles qui permettent l'accès aux environnements d'exécution.

3.1.2 Évaluation

Nous nous intéressons ici à l'évaluation d'une règle, c'est-à-dire à l'évaluation de sa partie Condition.

Le modèle d'exécution a évidemment une influence sur le **résultat** de l'évaluation d'une règle puisque c'est lui qui détermine quand et comment cette règle va être exécutée ; donc dans quel environnement, dans quel état de la base de données, cette règle va être exécutée. Toutefois, les modèles d'exécutions des SGBD actifs n'exhibent aucune dimension relative au **processus** d'évaluation d'une règle.

On se place dans le contexte suivant : une règle a été déclenchée à un instant donné. On se trouve maintenant à un point de considération tel que la règle est dans l'état *évaluable*. Le moteur d'exécution a la main. Le processus d'évaluation consiste simplement à rendre temporairement la main au SGBD qui va évaluer la Condition – c'est-à-dire évaluer un prédicat logique ou exécuter une requête du langage de requêtes – et rendre immédiatement la main au moteur de règles.

Remarque : L'unique fait mentionné dans la plupart des travaux, est que l'on suppose que l'évaluation est sans effet de bord, donc qu'elle ne peut pas produire d'événements, ni de ce fait déclencher de règles. On peut cependant s'interroger sur la pertinence de ce postulat, particulièrement dans le cas des systèmes à objets dans lesquels les appels de méthodes sont autorisés dans les Conditions. À notre connaissance, seul NAOS donne un élément de réponse en fournissant un mécanisme qui assure cette propriété : NAOS garantit que l'évaluation ne déclenchera pas de règle en désactivant la détection de tous les événements lors de l'évaluation.

3.1.3 A-exécution

Nous nous intéressons maintenant à l'A-exécution d'une règle, c'est-à-dire à l'exécution de sa partie Action. On se place dans le contexte suivant : une règle a été déclenchée par un événement produit par l'exécution d'une opération d'une transaction ; puis évaluée à vrai. On se trouve maintenant à un point d'exécution, le moteur d'exécution a la main, la règle est dans l'état *A-exécutable*.

3.1.3.1 Substitution de l'opération déclenchante

Les effets de l'exécution d'une règle R déclenchée par un événement produit par une opération élémentaire o – on dit par abus de langage que o est *l'opération déclenchante* de R – *s'additionnent* généralement à ceux de o. Cependant, il peut être intéressant,

dans certains cas, d'exécuter R, plus précisément les opérations qui composent la partie Action de R, à la place de o.

Cela sous-entend que le système est capable :

1. d'annuler l'opération élémentaire o qui a pourtant déjà été exécutée ;
2. ou d'exécuter la règle R avant d'exécuter l'opération élémentaire o et donc de ne pas exécuter o.

Deux systèmes, Postgres et NAOS offrent la possibilité de substituer une règle à son opération déclenchante. Dans les deux cas, c'est la solution 2 donnée ci-dessus qui est mise en œuvre.

L'exécution d'une règle *substituant* ou (exclusif) *s'additionnant à* l'opération déclenchante constitue bien une dimension du modèle d'exécution d'une règle. Toutefois, comme le montre par exemple [CC95c], il est très difficile d'en donner une sémantique précise. En effet, quelle sémantique adopter lorsque plusieurs règles sont déclenchées par un même événement avec parmi elles, une ou plusieurs règles s'exécutant en substituant l'opération déclenchante ? La sémantique adoptée par Postgres, beaucoup plus "propre" sur ce point que celle de NAOS, découle justement d'une politique beaucoup plus simple en ce qui concerne l'exécution d'un ensemble de règles : on est sûr que toutes les règles à considérer ont été déclenchées par le même événement, (cf. section 3.2) et on exécute une seule de ces règles (qu'elle s'exécute en se substituant ou en s'additionnant à l'opération déclenchante).

3.1.3.2 Préemption de l'exécution

Dans la majorité des systèmes – à l'exception de Postgres – l'exécution d'une règle R est susceptible de produire des événements. Soit R' une règle déclenchée par l'exécution de la règle R. On fait toujours l'hypothèse que R' est dans l'état *à considérer*. Deux politiques de considération de R' sont envisageables :

- soit l'exécution de la règle R est suspendue et l'on exécute R' avant de revenir à l'exécution de R, on dit alors que la règle R' a un droit de *préemption* ;
- soit les événements déclenchants de R' sont mémorisés jusqu'à la fin de l'exécution de R ou ils seront alors pris en compte pour l'exécution de R'. Dans ce cas la règle R' n'a pas le droit de préemption.

L'étude de cette dimension soulève les remarques suivantes :

1° elle est implicite (au niveau du langage de règles) dans tous les systèmes que nous avons étudiés ;

2° elle est indispensable car elle ne peut pas être *couverte* (ou simulée) par une combinaison d'autres dimensions ;

3° c'est une dimension de l'exécution d'une règle dont l'influence se fait, en réalité, sentir sur l'exécution d'un ensemble de règles puisqu'elle est susceptible d'induire une exécution récursive des règles.

Discussion Dans [FT95], on trouve une dimension appelée *atomicité de l'exécution d'une règle* qui spécifie si l'exécution d'une règle est *atomique* ou *interruptible*. Les exemples d'utilisation de la dimension *atomicité de l'exécution d'une règle* donnés pour les systèmes étudiés dans [FT95] sont exprimables avec notre dimension *préemption de l'exécution*. Dans A-RDL, Ariel, Chimera, HiPAC, Starburst et Ode, toutes les règles sont *atomiques*, ce qui signifie, pour nous, qu'aucune n'a de droit de préemption. Dans REACH et SAMOS, toutes les règles sont *interruptibles*, ce qui signifie pour nous, qu'elles ont toutes le droit de préemption. Ces deux dimensions ne sont cependant pas réciproques. En effet, dans NAOS, les *règles immédiates* les *règles différées* sont interruptibles uniquement par les *règles immédiates* et non par les *règles différées*. Ceci est exprimable par notre dimension *préemption de l'exécution* mais non par l'*atomicité de l'exécution d'une règle*.

Une solution à ce problème pourrait être de conserver les deux dimensions. Si nous revenons à nos deux règles R et R', nous avons à étudier les quatre cas représentés dans le tableau 3.1.

R \ R'	*préemption*	*non préemption*
atomique	1	2
intérruptible	3	4

TAB. 3.1 – *Atomicité de l'exécution vs. mode de préemption*

Les cas 2 et 4 peuvent être écartés rapidemment : ils conduisent à la non suspension de R par R'. Le cas 3 conduit tout aussi rapidemment à la suspension de R pendant la considération de R'. En ce qui concerne le cas 1 : soit R est suspendue, soit elle n'est pas suspendue ! Dans le premier cas, on donne implicitement une importance plus grande au *mode de préemption* ; dans le second cas, on donne une importance plus grande à *l'atomicité de l'exécution*. Ceci montre, qu'en réalité, ces deux dimensions ne peuvent pas coexister. Pour la raison que nous avons donnée ci-dessus : NAOS ne peut être représenté avec la dimension *atomicité de l'exécution* ; nous choisissons de ne considérer, dans notre taxonomie, que le *mode de préemption*.

3.1.4 Autres dimensions de l'exécution d'une règle

Dans cette section, nous traitons d'un certain nombre de dimensions qui influent, elles aussi, sur l'exécution d'une règle. Nous expliquons pourquoi nous écartons les deux premières qui, à notre avis, ne font pas partie du modèle d'exécution, puis, pourquoi les deux dernières font partie du modèle d'exécution mais sont également fortement dépendantes du modèle de règles.

3.1.4.1 Déclenchement avant ou après l'opération déclenchante

Une règle R, déclenchée par un événement e, généré par une opération élémentaire o est généralement exécutée après l'exécution de o. Cependant, il peut être intéressant, dans certains cas, d'exécuter R avant l'opération déclenchante. Cela implique que l'événement e doit être signalé avant l'exécution de o. Le fait de signaler les événements avant ou après les opérations qui les produisent va influer sur l'ordonnancement de ces événements (cf. 2.2.2). Or, si le processus d'exécution des règles doit prendre en compte, de quelque manière que ce soit, l'ordonnancement des événements, il n'est pas responsable, à notre avis, d'établir cet ordonnancement, ce qui est du ressort du processus de détection des événements. Ceci montre que le déclenchement *avant* ou *après* l'opération déclenchante est une dimension du modèle d'événement et non du modèle d'exécution.

3.1.4.2 Déclenchement implicite et explicite

Certains systèmes, tels que HiPAC ou SAMOS, prônent "la politique du tout objet". Dans ces systèmes, événements et règles sont représentés et peuvent être manipulés comme tous les autres objets. En particulier, il est possible de déclencher explicitement une règle par un simple appel de méthode sur l'objet qui la représente[3]. Le processus d'exécution des règles actives est responsable d'exécuter les règles déclenchées de quelque manière que ce soit. C'est pourquoi, le déclenchement *implicite* ou *explicite* constitue, à notre avis, une dimension du modèle de règle et non du modèle d'exécution d'un SGBDA.

3.1.4.3 Désactivation et (ré-)activation d'une règle

Dans certaines situations, on peut souhaiter *désactiver* une règle afin d'éviter son déclenchement même si un événement déclenchant de cette règle est produit. Cette

3. Il est donc possible de déclencher une règle en l'absence d'événement déclenchant (ce qui ne respecte pas la sémantique la plus générale et la plus généralement admise d'une règle E-C-A !).

fonctionnalité permet de "protéger" une opération ou une séquence d'opérations par rapport à cette règle. La portée de la désactivation peut être :

- *locale* : la règle est désactivée pour la transaction déclenchante (cf. 3.3) mais reste activée dans les autres transactions ;
- *globale* : la règle est désactivée pour toutes les transactions (jusqu'à sa réactivation) ;

Dans EXACT, HiPAC et NAOS, la portée de la désactivation est locale. Dans Starburst et Ode, elle est globale. La désactivation et la réactivation sont des fonctionnalités offertes par le modèle (langage) de règles. Elles ne sont cependant pas étrangères au modèle d'exécution car celui-ci doit spécifier :

1. ce qu'il advient d'une règle qui serait désactivée alors qu'elle était (a) déclenchée, (b) en cours d'évaluation ou (c) en cours d'exécution ;
2. ce qu'il advient d'une *règle virtuellement déclenchée*, c'est-à-dire, réactivée en présence d'événements déclenchants survenus pendant qu'elle était désactivée.

Les solutions envisageable pour le point 1., à chaque étape (a), (b) et (c) sont de stopper l'exécution ou de poursuivre l'exécution, ce qui est fortement dépendant du modèle de transactions (cf. 3.3). Pour le point 2., les solutions sont d'exécuter la règle considérée ou de ne pas l'exécuter.

3.1.4.4 Déclenchement unique et perpétuel

Dans Ode, on trouve deux types de règles :

- les règles à déclenchement *unique* qui sont désactivées après leur exécution et doivent être réactivées explicitement ;
- les règles à déclenchement *perpétuel* qui restent activées après leur exécution.

Cette dimension n'existe que dans Ode dont les règles sont de la forme EC-A. Pour des règles E-C-A "plus classiques", on pourrait imaginer les règles à déclenchement unique désactivées après évaluation ou de manière plus restrictive, après exécution de la partie Action.

3.2 Exécution d'un ensemble de règles

Nous nous intéressons maintenant à l'exécution d'**un ensemble** de règles, c'est-à-dire spécifiquement aux interactions entre les règles d'un ensemble. Nous ne prenons pas le terme *ensemble* dans son sens mathématique : "collection d'objets qui respectent

certains critères" ; mais dans son sens courant qui signifie ici "plusieurs règles"; ou plus précisément l'ensemble des règles définies sur le schéma sur lequel s'exécute l'application qui déclenche les règles.

L'exécution d'un ensemble de règles est schématisée par la figure 3.2 :

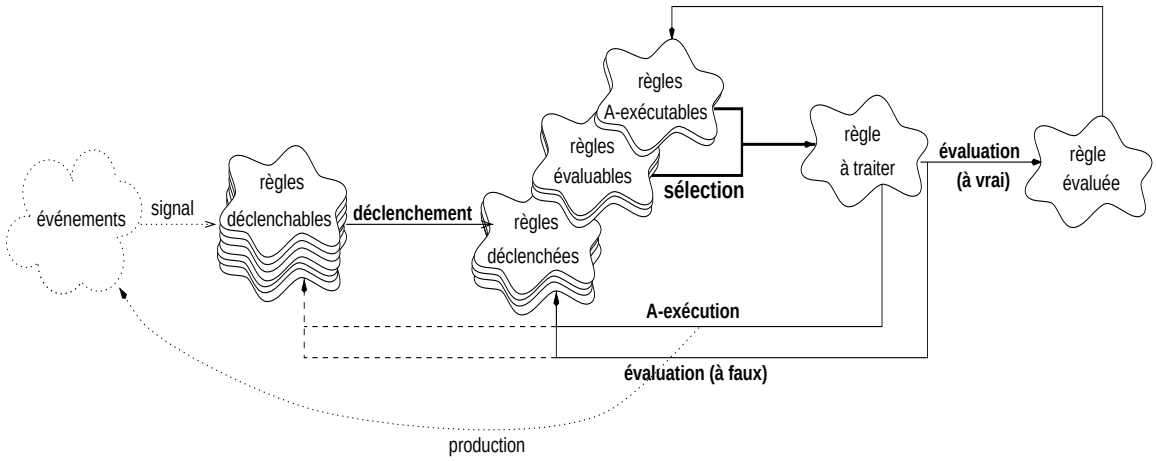

FIG. 3.2 – *Exécution d'un ensemble de règles*

Le déclenchement est un processus identique à celui décrit pour l'exécution d'une règle (cf. section 3.1) à la différence que cette fois, n (n≥1) règles vont recevoir des événements. Un certain nombre m (m≤n) parmi elles vont alors être déclenchées.

À chaque point d'exécution, il s'agit alors de déterminer, par un processus de planification (ou ordonnancement), quand et comment les règles vont être exécutées. Ce processus consiste à examiner l'ensemble des règles *évaluables* ou *A-exécutables* afin de sélectionner la prochaine règle à traiter.

Une fois une règle sélectionnée, elle est évaluée ou A-exécutée :

- si elle est dans l'état *évaluable*, elle est évaluée : si elle est évaluée à faux, elle revient dans l'état *déclenchable* (flèche en pointillé de la figure 3.2), son exécution est terminée ; si elle est évaluée à vrai, elle passe dans l'état *évalué* ;
- si elle est dans l'état *A-exécutable*, elle est A-exécutée et revient à l'état *déclenchable*, (flèche en pointillé de la figure 3.2), son exécution est terminée.

Le processus général de sélection/évaluation/A-exécution est, quant à lui, itéré sur l'ensemble des règles *évaluables* et *A-exécutables*. Les interactions entre les exécutions des règles d'un ensemble sont prises en compte par le processus dont nous venons de parler. Son objectif est d'établir un *plan d'exécution*. Ce plan détermine quand et

comment les règles de cet ensemble vont être exécutées. On distingue généralement deux niveaux de complexité :

1. le premier concerne un ensemble de règles déclenchées par un ensemble d'événements *stable* – on parle alors de *plan d'exécution local* qui vise à résoudre le *problème des règles multiples* ;
2. le second traite d'un ensemble de règles déclenchées par un ensemble d'événements qui peut croître du fait de nouveaux événements produits par l'exécution des règles elles-mêmes (cf. flèche "production" de la figure 3.2) ; on parle alors de *plan d'exécution global* qui vise à résoudre le *problème des cascades de règles*.

3.2.1 Plan d'exécution local (règles multiples)

Lorsqu'un point d'exécution est atteint, le système doit établir un plan d'exécution pour traiter les règles déclenchées jusque là, par un ou plusieurs événements produits à des instants différents ou non. Cinq grandes stratégies sont envisageables pour établir ce plan. Le choix de la stratégie utilisée est important car il a une influence sur le *déterminisme* de l'exécution de l'ensemble de règles considéré. L'étude de ce déterminisme se fonde sur des propriétés statiques des couples de règles telles que la *compatibilité* [Hab93] et la *permutabilité* (ou *commutativité* [AWH92]) :

Définition 3.1 (Compatibilité) *Deux règles* R_1 *et* R_2 *sont compatibles si et seulement si : quelle que soit l'exécution simultanée de* R_1 *et* R_2 *et quel que soit l'état initial* S_i *de la base, l'état final* S_f *de cette exécution est le même que celui de l'exécution séquentielle de* R_1 *suivie de* R_2*, ou de* R_2 *suivie de* R_1*.*

Définition 3.2 (Permutabilité) *Deux régles* R_1 *et* R_2 *sont permutables si et seulement si : quel que soit l'état initial* S_i *de la base, toute exécution de* R_1 *suivie de* R_2 *produit le même résultat final* S_f *que l'exécution de* R_2 *suivie de* R_1 *[AWH92]. Deux règles compatibles sont permutables.*

Exécution séquentielle aléatoire Les règles sont exécutées l'une après l'autre dans un ordre établi de manière aléatoire. L'exécution aléatoire est déterministe si et seulement si toutes les règles de l'ensemble considéré sont permutables deux à deux.

Exécution séquentielle ordonnée Les règles sont exécutées l'une après l'autre. L'ordre d'exécution peut être déterminé de différentes manières :

– Un ordre partiel découle de l'ordre de déclenchement des règles, lui-même basé sur les instants de déclenchement des règles (cf. 3.1.1.1, §instant de déclenche-

ment). Cet ordre n'est que partiel du fait des occurrences simultanées d'événements (cf. section 2.2.2). Dans HiPAC, cette stratégie, appliquée pour l'exécution des actions découplées (cf. section 3.3.1), est appelée "mécanisme de pipelines".

- Un mécanisme de priorités entre règles permet également de spécifier un ordre sur les règles :
  - en associant une priorité numérique aux règles (Postgres);
  - en spécifiant des priorités relatives entre règles ; c'est, de loin, la solution la plus souvent proposée : Starburst, NAOS, Chimera, Peplom^{ad}, etc.;
  - en définissant une fonction de coût associée à chaque règle – cette fonction peut être calculée statiquement : à la compilation des règles ; ou dynamiquement : lors de la considération des règles :EXACT ;
  - en utilisant un langage de contrôle : A-RDL.

On peut imaginer nombre d'autres mécanismes plus ou moins puissants, pratiques, voire exotiques ! Deux faits sont cependant à noter :

1° Ces mécanismes n'assurent la plupart du temps qu'un ordre partiel. C'est pourquoi ils ne servent généralement qu'à surcharger un ordre qui, lui, est total (quoique arbitraire !) comme l'ordre de définition des règles (Starburst, NAOS). C'est cet ordre total qui assure le déterminisme de l'exécution de l'ensemble de règles considéré.

2° Ces mécanismes sont du ressort du modèle de définition de règles et non du modèle d'exécution. En ce qui concerne purement le modèle d'exécution, si le plan d'exécution est déterminé selon la stratégie d'exécution séquentielle ordonnée, il suffit de savoir qu'un ordre total entre les règles existe et peut être pris en compte par le modèle d'exécution.

Exécution parallèle Les règles sont exécutées de manière parallèle. Dans la plupart des cas, elles s'exécutent dans plusieurs sous-transactions concurrentes ou bien dans des transactions séparées, à la condition, bien entendu, que le modèle de transactions du SGBD autorise la création de telles transactions (cf. section 3.3). L'exécution parallèle d'un ensemble de règles est déterministe si et seulement si les règles de l'ensemble sont toutes compatibles deux à deux.

Stratégies mixtes Exécution séquentielle et parallèle peuvent être combinées en une stratégie mixte. Une solution élégante est de permettre l'attribution de priorités

relatives entre règles afin d'établir des ordres partiels et de réaliser une exécution parallèle des règles ayant une même priorité. Ceci fut proposé initialement par HiPAC. Une solution relativement proche est adoptée également dans NAOS [CM95]. Ces deux approches diffèrent toutefois par le fait que NAOS assure une exécution déterministe pour tout ensemble de règles[4] tandis qu'HiPAC prend comme hypothèse la non contradiction des règles de l'ensemble considéré et laisse donc cette responsabilité à la charge du programmeur. Dans SAMOS, on trouve aussi une (autre) stratégie mixte. Certaines règles (les règles *immédiates* et *différées*, cf. 3.3.1) sont exécutées selon un mécanisme de priorités tandis que les autres règles (les règles *séparées-indépendantes*, cf. 3.3.1) sont exécutées en concurrence.

Choix d'une règle On peut aussi, comme Postgres ou EXACT, n'exécuter qu'une seule règle. Le choix de la règle à exécuter peut utiliser tous les mécanismes décrits ci-dessus pour une stratégie d'exécution séquentielle (Postgres utilise un mécanisme de priorités numériques). Le choix d'une règle est déterministe si la fonction de choix l'est. En particulier, le choix d'une règle basé sur une fonction de choix aléatoire ne sera pas déterministe.

3.2.2 Plan d'exécution global (cascades de règles)

L'exécution d'une règle R peut générer un ou plusieurs événements qui, à leur tour, peuvent déclencher une ou plusieurs règles (dont R elle-même). On parle alors de *cascade de règles*. Les cascades de règles influent sur le processus de planification puisque l'ensemble de règles considéré peut croître par adjonction des règles déclenchées par les nouveaux événements. Différentes stratégies d'exécution sont envisageables pour établir un *plan d'exécution global* qui vise à déterminer comment va être exécuté l'ensemble de règles formé de l'union de l'ensemble R_I des règles déclenchées initialement et de l'ensemble R_C des règles déclenchées par l'exécution des règles de R_I.

Pas de cascade La solution la plus simple consiste à ne pas considérer ce problème du tout. La plupart des SGBD commerciaux offrant des fonctionnalités actives ainsi que Postgres n'exécutent ainsi que les règles de R_I et ne considèrent pas les règles de R_C.

4. Dans NAOS, l'exécution parallèle [CM95] de tout ensemble de règles est équivalente à l'exécution séquentielle ordonnée [CCS94b] de ce même ensemble de règles.

Exécution à plat Une seconde solution consiste à traiter uniformément les règles déclenchées initialement et les règles déclenchées par l'exécution de ces règles. Les règles de R_C sont ainsi insérées dans l'ensemble R_I et l'on applique les différentes stratégies présentées dans la section 3.2.1 pour établir le plan d'exécution global. Plan d'exécution local et global sont alors confondus et on parle d'exécution à plat. C'est la solution adoptée notamment par Starburst.

Arbres d'exécution Dans HiPAC, Beeri & Milo, et NAOS, le plan d'exécution global est basé sur les notions d'*arbres de déclenchements*, *cycles d'exécution* et *arbres d'exécution*.

Considérons l'exemple suivant : un événement e_1 déclenche deux règles R_1 et R_2. L'exécution de R_1 et R_2 produit deux événements e_2 et e_4 qui déclenchent, à leur tour, trois règles R_{1a}, R_{1b} et R_{2a}. On considère que ces règles sont définies avec une relation de précédence $(<) : \mathsf{R}_1 < \mathsf{R}_2$ et $\mathsf{R}_{1a} < \mathsf{R}_{2a} < \mathsf{R}_{1b}$ (R_1 est la plus prioritaire, R_{1b} la moins prioritaire). L'arbre de déclenchement correspondant à cet exemple est présenté dans la figure 3.3.

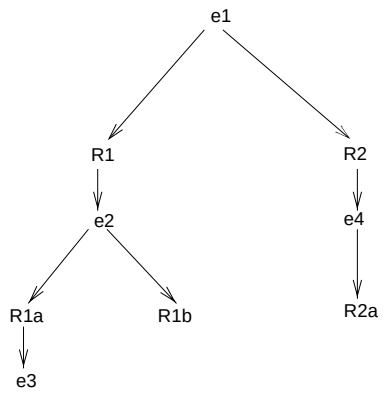

FIG. 3.3 – *Arbre de déclenchement*

Les règles sont exécutées dans des cycles d'exécution. Un cycle d'exécution décrit une séquence d'opérations exécutées dans un programme, dans une transaction ou lors de l'exécution d'une règle. Une règle est toujours exécutée dans un cycle d'exécution distinct de celui dans lequel son(ses) événement(s) déclenchant(s) a(ont) été produit(s). Si plusieurs règles doivent être exécutées dans un même cycle d'exécution (par exemple R_{1a}et R_{1b}, on applique l'une des stratégies d'exécution des règles multiples présentées dans la section 3.2.1.

Dans HiPAC, le plan d'exécution global, c'est-à-dire l'ordre d'exécution de l'ensemble des règles considérées correspond à un parcours *en largeur d'abord* de l'arbre de déclenchement. Dans notre exemple, le plan d'exécution global (cf. figure 3.4) sera donc : $R_1 \mapsto R_2 \mapsto R_{1a} \mapsto R_{2a} \mapsto R_{1b}$.

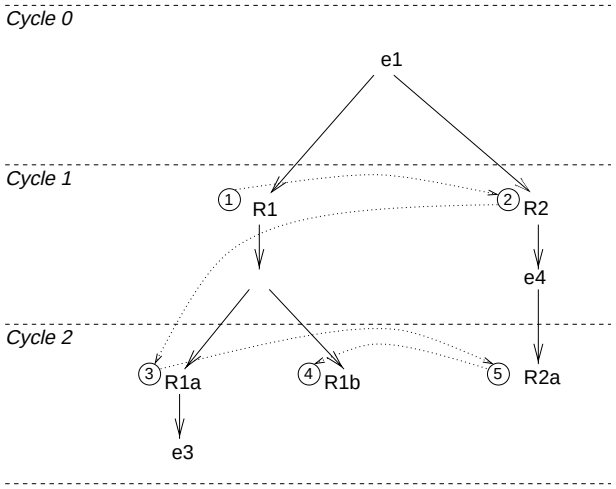

FIG. 3.4 – *Exécution en largeur d'abord*

Le modèle de Beeri & Milo offre un parcours *en profondeur d'abord* de l'arbre de déclenchement. Le plan d'exécution global pour notre exemple est alors (cf. figure 3.5) : $R_1 \mapsto R_{1a} \mapsto R_{1b} \mapsto R_2 \mapsto R_{2a}$.

Dans NAOS, on trouve les deux types de parcours.

Terminaison d'une cascade de règles Les déclenchements de règles en cascade peuvent provoquer une exécution sans fin avec tous les inconvénients que cela comporte. D'une manière générale, le problème de la terminaison de l'exécution d'un ensemble de règles est indécidable.

Dans la plupart des systèmes de règles actives, ce problème est laissé à la charge du programmeur de règles, de la même façon que dans les langages de programmation. Il est alors utile pour ce programmeur de disposer d'outils d'aide à la mise au point des applications tels que DEAR [DJP94] pour EXACT, SimBug [Beh94b] pour ADL [Beh93], [CTZ95] pour Sentinel, ADELA (Animated Debugging and ExpLanation of Active Database Rules) [For95] ou VITAL [BGB95]. Notons que les deux derniers outils cités sont uniquement des simulateurs, ils ne sont pas réellement implantés au-dessus d'un SGBDA.

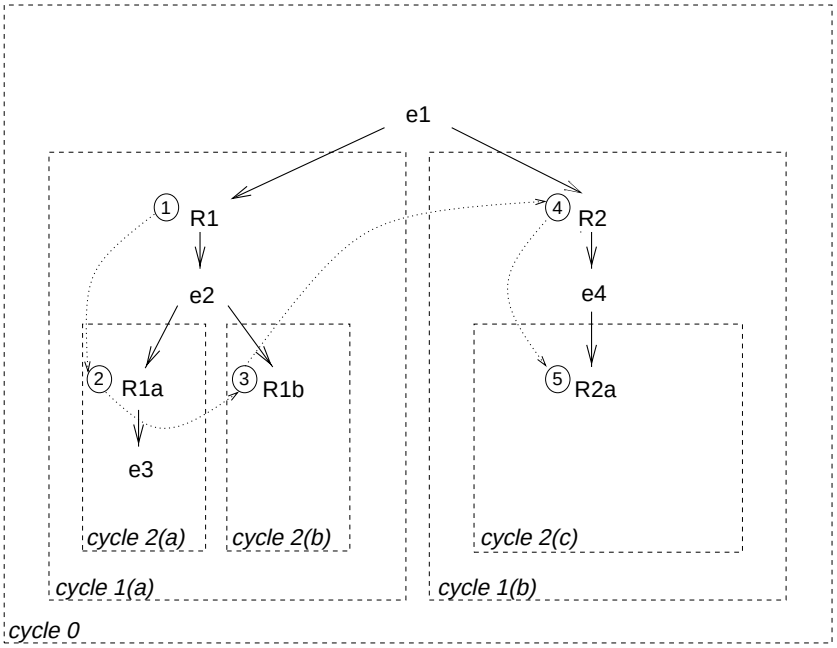

FIG. 3.5 – *Exécution en profondeur d'abord*

Cette solution ne permet cependant pas de garantir qu'une exécution se termine, ce qui est pourtant essentiel pour les concepteurs d'applications. C'est pourquoi certains systèmes imposent des restrictions syntaxiques aux définitions de règles [Ora92] ou effectuent une analyse statique des règles [AWH92, BW94, BCP95] pour limiter le risque de non terminaison. A notre connaissance, les travaux réalisés dans ce sens concernent exclusivement des systèmes relationnels. Dans le contexte des systèmes à objets, ce problème est beaucoup plus délicat à traiter du fait des appels de méthodes et de l'héritage, par exemple.

Des solutions, que l'on peut qualifier d'*empiriques*, telles que *délais de garde*, *nombre maximal de règles dans une cascade* ou *profondeur maximale des cascades* ont été proposées notamment dans Starburst et Oracle v7 [Ora92].

3.3 Règles et transactions

Compte tenu du modèle de comportement d'un SGBD, les opérations appliquées à une base de données, génératrices d'événements déclenchants pour les règles actives, le sont généralement dans le contexte d'une transaction. On considère qu'une règle s'exécute toujours dans une transaction. Une transaction dans laquelle sont produits

des événements est appelée *transaction déclenchante*. Une transaction dans laquelle une règle s'exécute pour traiter tout ou partie de ces événements est appelée *transaction déclenchée*.

3.3.1 Modes de couplage

La notion de *modes de couplage* fut introduite dans HiPAC comme un mécanisme de spécification du modèle d'exécution d'une règle active par rapport au modèle de transactions du SGBD sous-jacent. Un *couplage* [Hab93, CHR96] est caractérisé par un quadruplet : <*mode de déclenchement, mode de transaction, mode de synchronisme, mode de dépendance*> dont les constituants sont détaillés ci-après.

3.3.1.1 Mode de déclenchement

Le *mode de déclenchement* spécifie *quand* la partie déclenchée est exécutée par rapport à la partie déclenchante. Deux valeurs sont généralement considérées :

- *immédiat* : la partie déclenchée est considérée immédiatement à la suite de la partie déclenchante.
  Si le mode de déclenchement pour le couplage E-C est immédiat, alors la condition C est évaluée dès qu'un événement de type E est reconnu – au point de considération qui précède la prochaine opération de la transaction déclenchante.
  Si le mode de déclenchement pour le couplage C-A est immédiat alors l'action A est exécutée immédiatement après l'évaluation de la condition C (si celle-ci est vraie, évidemment).
- *différé* : l'exécution de la partie déclenchée est différée à un point ultérieur de la transaction. Ce point correspond généralement à la fin de la transaction (validation ou abandon). Dans certains systèmes : A-RDL, Starburst, Peplom^{ad}, le déclenchement est différé à un *point de contrôle* donné explicitement dans le code la transaction déclenchante.

Les couplages E-C et C-A d'une même règle peuvent combiner ces deux valeurs possibles du mode de déclenchement : on peut, par exemple, souhaiter évaluer la condition immédiatement après l'événement déclenchant (mode *immédiat*), tout en exécutant l'action ultérieurement (mode *différé*). La signification et l'intérêt de certaines des quatre combinaisons théoriquement possibles sont cependant discutables – c'est le cas notamment de la combinaison *différé-différé*.

3.3.1.2 Mode de transaction

Le *mode de transaction* spécifie la transaction dans laquelle est exécutée la partie déclenchée, relativement à la partie déclenchante :

- la valeur *même transaction* signifie que la partie déclenchée s'exécute dans la même transaction que la partie déclenchante,
- la valeur *séparé* signifie que la partie déclenchée s'exécute dans une nouvelle transaction différente de la transaction déclenchante.

3.3.1.3 Mode de synchronisme

Le *mode de synchronisme* spécifie le type de synchronisation qui prend place entre partie déclenchante et partie déclenchée. L'exécution d'une règle peut se faire de manière *synchrone* ou *asynchrone* :

- *synchrone* : l'exécution de la partie déclenchante est stoppée le temps de l'exécution de la partie déclenchée ;
- *asynchrone* : partie déclenchante et partie déclenchée s'exécutent en parallèle.

Remarque Il existe une certaine confusion, dans la littérature des bases de données actives, entre *parallélisme* et *concurrence*. Les opérations sur une BD étant généralement appliquées dans le contexte d'une transaction, on sous-entend, tout aussi généralement, que le parallélisme ne peut reposer que sur le mécanisme transactionnel, donc sur la concurrence. Dans notre cas, cela signifie que pour exécuter partie déclenchante et partie déclenchée en parallèle, il faut les exécuter dans deux transactions concurrentes. Or, comme le montre [Mac95], il est tout à fait possible de réaliser du parallélisme à l'intérieur d'une même transaction (on parle dans [Mac95] de *parallélisme intra-transaction*). Cette possibilité est utilisée pour la parallélisation des règles actives de NAOS [CM95].

Bref, nous adoptons la définition suivante : le mode *asynchrone*, significatif uniquement si le mode de transaction est *séparé*, spécifie que partie déclenchante et partie déclenchée s'exécutent dans deux transactions différentes – donc concurrentes – et nous ne parlons plus de parallélisme.

3.3.1.4 Mode de dépendance

Le *mode de dépendance*, lui aussi significatif uniquement si le mode de transaction est *séparé*, spécifie la dépendance de validation entre transaction déclenchante et

transaction déclenchée :

- le mode *dépendant* signifie que la transaction déclenchée ne peut être validée que si la transaction déclenchante a elle-même été préalablement validée ;
- le mode *indépendant* signifie que la validation de la transaction déclenchée peut avoir lieu indépendamment de celle de la transaction déclenchante.

3.3.1.5 Discussion

En réalité, les liens entre exécutions de règles actives et transactions sont rarement explicités en fonction des modes de couplage que nous venons de décrire. Cependant, il est généralement possible d'exprimer ces liens en utilisant ce modèle (les couplages) dont la force est, principalement, de permettre de comparer les différents SGBDA sur ce point. Les modes de couplage constituent les dimensions du modèle d'exécution d'un SGBD Actif les plus dépendantes du SGBD sous-jacent puisque dépendantes du modèle de transactions offert par celui-ci.

Pour les SGBD actifs utilisant un modèle de transactions classique, dans lequel il n'est pas possible de créer une (sous)transaction à l'intérieur d'une transaction, les modes de couplage se voient limités au seul mode de déclenchement :

- Ariel et Postgres ne disposent pour les couplages E-C et C-A que du mode *immédiat* ;
- Starburst n'utilise pas la notion de mode de couplage : les modes implicites pour les couplages E-C et C-A sont respectivement *différé* et *immédiat*. Le mode *différé* correspond à *différé à un point de contrôle*. La fin de la transaction déclenchante est considérée comme un point de contrôle par défaut. Il est aussi possible de spécifier des points de contrôle au cours de la transaction déclenchante (dans le code de l'application). Trois types de points de contrôle sont offerts, permettant l'exécution (1) d'une règle particulière, (2) d'un ensemble de règles et (3) de toutes les règles disponibles.
- NAOS, Chimera et A-RDL n'utilisent pas explicitement la notion de mode de couplage. On trouve deux types de règles : les *règles immédiates* dont les couplages E-C et C-A sont respectivement *immédiat* et *immédiat* et les *règles différées* dont les couplages E-C et C-A sont respectivement *différé* et *immédiat*. Dans A-RDL, il est aussi possible de spécifier des points de contrôle au cours de la transaction déclenchante
- Peplom^{ad} n'utilise pas, non plus, la notion de mode de couplage. On trouve trois types de règles : les *règles immédiates*, les *règles différées à la validation* et les *règles différées à point de contrôle*.

Pour les SGBDA utilisant un modèle de transactions évolué, dans lequel il est possible de créer des (sous)transactions emboîtées et découplées (avec et sans dépendance de validation), toutes les combinaisons de modes de couplage sont potentiellement envisageables.

- Dans Ode, le couplage E-C est simulé par des événements transactionnels. Quatre couplages sont proposés pour le couplage C-A : (1) <*immédiat, même transaction, synchrone, dépendant*>, (2) <*différé, même transaction, synchrone, dépendant*>, (3) <*immédiat, séparé, asynchrone, dépendant*> et (4) <*immédiat, séparé, asynchrone, indépendant*>.
- Dans SAMOS et Sentinel, les trois couplages (1), (2) et (4) donnés ci-dessus sont proposés pour les couplages E-C et C-A. Pour SAMOS, seules six combinaisons de ces modes de couplages parmi les neuf potentiellement possibles sont proposées. Dans Sentinel, le couplage (4) a été étudié mais non implanté.
- Dans TriGS [KRRV94a] ainsi que dans le modèle proposé par Beeri et Milo[BM91], les conditions et actions sont toujours exécutées dans des sous-transactions de la transaction déclenchante (le *mode de transaction* est toujours *séparé*) et le *mode d'exécution* est toujours *immédiat*. Dans TriGS, en revanche, le *mode de dépendance* est toujours *indépendant* tandis que dans le modèle de Beeri et Milo, il peut être *dépendant* ou *indépendant*.
- Dans HiPAC, les conditions et actions sont toujours exécutées dans des sous-transactions de la transaction déclenchante. Mise à part cette restriction, toutes les combinaisons de modes de couplages sont possibles. Il faut cependant noter que les travaux sur HiPAC sont principalement académiques.
- REACH propose toutes les combinaisons de HiPAC et ajoute un nouveau mode de dépendance appelé *exclusif* qui spécifie que la règle doit n'être exécutée qu'en cas d'abandon de la transaction déclenchante.

3.3.2 Position par rapport à l'échec

Les modes de couplage n'expriment pas la totalité des rapports entre règles et transactions. Il convient également d'établir la politique adoptée en cas d'échec (1) de la transaction déclenchante TD et (2) d'une transaction TR_i dans laquelle s'exécute une règle R_i. Un échec peut être causé par une erreur[5] ou parce que le code de la

5. Tout comme une erreur peut survenir au cours d'une transaction, une erreur peut survenir lors de l'exécution d'une règle pour un certain nombre de raisons; parce que des données ont été modifiées ou supprimées, parce que des droits d'accès ont été modifiés ou supprimés, à cause d'un interblocage, à cause d'une erreur système, etc.

transaction TD ou d'une règle R_i contient une instruction d'abandon (abort).

Modèle de transactions classique Dans les systèmes utilisant un modèle de transactions classique, les règles s'exécutent dans la transaction déclenchante (TD=TR_i). Un échec de cette transaction, qu'il provienne de l'application ou de l'exécution d'une règle, provoque la fin de l'exécution des règles et l'abandon de la transaction, indépendamment du nombre de règles déjà exécutées ou du nombre de règles déclenchées (et donc en attente d'exécution).

Plus généralement, ces systèmes s'appuyent sur les fonctionnalités du modèle de transaction disponible. Ainsi, si celui-ci relance automatiquement les transactions abandonnées, la transaction déclenchante sera relancée et l'exécution des règles reprendra.

Modèles de transactions évolués Dans les systèmes utilisant un modèle de transactions évolué, la politique en cas d'échec de la transaction déclenchante TD est déterminée par le *mode de transaction* et le *mode de dépendance* des règles R_i :

- l'exécution des règles qui s'exécutent dans la transaction déclenchante (*mode de transaction = même transaction*) est stoppée ;
- l'exécution des règles dont le *mode de transaction* est *séparé* et le *mode de dépendance* est *dépendant* est stoppée du fait de la dépendance de validation entre transaction déclenchante et déclenchée ;
- l'exécution des règles dont le *mode de transaction* est *séparé* et le *mode de dépendance* est *indépendant* continue normalement.

La politique adoptée en cas d'échec d'une règle semble raisonnable pour les systèmes utilisant un modèle de transactions classique. Ce dernier n'offrant que peu d'alternatives faciles à mettre en œuvre[6]. Pour les systèmes utilisant un modèle de transactions plus évolué, l'éventail des possibilités est, en revanche, plus ouvert en ce qui concerne les règles qui ne s'exécutent pas dans la transaction déclenchante :

- Dans TriGS, comme nous l'avons vu, toutes les règles s'exécutent dans des transactions indépendantes, une erreur dans une telle transaction provoque uniquement son abandon. L'exécution des autres règles se poursuit normalement.
- Dans HiPAC, il est possible de choisir, pour chaque règle dont le *mode de dépendance* est *dépendant*, si une erreur au cours de l'exécution de cette règle, doit provoquer (1) l'abandon de la sous-transaction dans laquelle elle s'exécute ou (2) l'abandon de la transaction déclenchante. Pour les règles dont le *mode*

6. C'est sans doute pourquoi il y a si peu de travaux sur ce sujet pour ces systèmes.

de dépendance est *indépendant*, les transactions dans lesquelles elles s'exécutent sont relancées automatiquement en cas d'erreur.

– Dans le modèle de Beeri et Milo, en cas d'erreur dans une règle R_i, il est possible de spécifier si la transaction TR_i dans laquelle elle s'exécute doit être abandonnée ou relancée. Il est même possible de lancer une *transaction de compensation* TC_i.

3.4 Taxonomie proposée, comparaison et discussion

Dans les sections précédentes, nous avons isolé un certain nombre de dimensions qui caractérisent les modèles d'exécution des SGBD actifs. Cela nous permet maintenant de proposer une taxonomie des modèles d'exécution pour SGBD Actifs, présentée dans quatre tableaux (TAB. 3.2 à TAB. 3.5). Nous identifions les dimensions propres au modèle d'exécution d'une règle dans le tableau TAB. 3.2. Nous soulignons la dépendance entre modèle d'exécutions et modèles de transactions dans le tableau TAB. 3.3, puis entre modèles d'exécution et modèles de règles dans le tableau TAB. 3.4. Enfin, nous identifions les dimensions du modèle d'exécution d'un ensemble de règles dans le tableau TAB 3.5.

La taxonomie proposée est constituée, au total, de vingt et une dimensions : les dimensions 1 à 19 caractérisent l'exécution **d'une** règle, les dimensions 20 et 21 caractérisent l'exécution **d'un ensemble** de règles. Pour chaque dimension, nous indiquons les valeurs que peut prendre cette dimension. Ces valeurs sont identifiées par des numéros qui seront utilisés dans la section 3.4.2.

3.4.1 Comparaison avec d'autres travaux

Parmi l'abondante littérature sur les SGBD Actifs (cf. annexe A), on trouve certain travaux qui visent à évaluer et comparer les fonctionnalités de ces SGBDA et en particulier leur modèle d'exécution. Parmi eux, [Dia95] et [FT95] adoptent une approche comparable à la nôtre, et proposent également une taxonomie des modèles d'exécution des SGBDA.

La taxonomie proposée dans [Dia95] comporte huit dimensions qui correspondent, plus ou moins, à nos dimensions 1,5,8,9, 13 et 20. La différence la plus importante entre cette taxonomie et la nôtre est l'existence d'une dimension *mode de consommation des événements* qui correspond, en réalité, au *mode de production des événements composites* de SNOOP et non à notre dimension *mode de consommation des événe-*

	Dimension	Valeurs
1	Mode de traitement des événements	1 Instance
		2 Ensembliste
2	Mode de consommation des événements	1 Consommation
		2 Préservation
3	Portée de la consommation des événements	1 Locale
		2 Globale
4	Instant de la consommation des événements	1 Évaluation
		2 Exécution
5	Effet Net	1 Déclenchement
		2 Environnement
		3 Déclenchement et Environnement
		4 Non
6	Substitution opération déclenchante	1 Oui
		2 Non
7	Mode de préemption	1 Préemption
		2 Non Préemption

TAB. 3.2 – *Dimensions de l'exécution d'une règle*

	Dimension	Valeur
8	Mode de déclenchement	1 Immédiat
		2 Différé
9	Mode de transaction	1 Même transaction
		2 Transaction séparée
10	Mode de synchronisme	1 Synchrone
		2 Asynchrone
11	Mode de dépendance	1 Dépendant
		2 Indépendant
12	Échec Transaction Déclenchante TD	1 Abandon TD
		2 Relancement TD
		3 Lancement Trans. Compensation TC
13	Échec Transaction de Règle TR	1 Abandon TR
		2 Relancement TR
		3 Abandon TR + Relancement TD
		4 Lancement Trans. Compensation TC

TAB. 3.3 – *Dimensions de l'exécution d'une règle qui dépendent du modèle de transactions*

	Dimension	Valeur
14	Désactivation d'une règle déclenchée	1 Arrêt
		2 Poursuite
15	Désactivation d'une règle en cours d'évaluation	1 Arrêt
		2 Poursuite
16	Désactivation d'une règle en cours d'exécution	1 Arrêt
		2 Poursuite
17	Réactivation d'une règle virtuellement déclenchée	1 Exécution
		2 Non exécution
18	Portée du déclenchement	1 Unique
		2 Perpétuel
19	Instant de la désactivation	1 Après déclenchement
		2 Après évaluation
		3 Après exécution

TAB. 3.4 – *Dimensions de l'exécution d'une règle qui dépendent du modèle de règles*

	Dimension	Valeur
20	Plan d'exécution local (règles multiples)	1 Choix d'une règle 2 Exécution séquentielle aléatoire 3 Exécution séquentielle ordonnée 4 Exécution parallèle
21	Plan d'exécution global (cascades de règles)	1 À plat 2 Profondeur d'abord 3 Largeur d'abord

TAB. 3.5 – *Dimensions de l'exécution d'un ensemble de règles*

ments. Cette différence provient du fait que [Dia95] inclut le modèle et le processus de détection des événements dans le modèle d'exécution alors que pour nous, ce sont deux choses distinctes et nous nous focalisons uniquement sur le modèle d'exécution.

La taxonomie proposée dans [FT95] est assez proche de la nôtre. Elle comporte douze dimensions qui correspondent, plus ou moins, à nos dimensions 1, 2, 3, 4, 5, 6, 7, 8, 9, 11 et 20. Nous avons déjà discuté de la dimension 7 dans la section 3.1.3.2. Une différence existe en ce qui concerne la dimension 6 dans laquelle [FT95] inclut le *déclenchement avant ou après l'opération déclenchante* (cf. 3.1.4.1). La raison de cette différence est la même que celle invoquée à propros de [Dia95].

3.4.2 Représentation graphique des modèles d'exécution

Nous avons choisi de ne pas situer les systèmes que nous avons étudiés dans notre taxonomie pour deux raisons :

1° parce que cela est fait dans les deux taxonomies citées qui étudient les mêmes systèmes que nous (à deux ou trois exceptions près) ; le lecteur intéressé est donc invité à consulter directement ces travaux ;

2° parce que cela n'est pas forcément très significatif. Cela permettrait, en effet, de visualiser rapidemment les similitudes et différences entre les systèmes dont le modèle d'exécution pourrait être qualifié de "simple". En revanche, cela perdrait une grande part de sa signification pour des systèmes plus complexes. Pour de tels systèmes, cela reviendrait à les citer pour presque toutes les valeurs de presque toutes les dimensions. Or, toutes les combinaisons de ces valeurs ne sont pas valides dans ces systèmes. La subtilité de leur modèle d'exécution serait donc complètement masquée.

En lieu et place, nous proposons une représentation graphique des modèles d'exécution des SGBDA qui permet de palier à l'inconvénient que nous venons de citer. Les modèles d'exécution sont représentés sous forme de graphes (non orientés) étiquetés dans lesquels :

- les noeuds étiquetés représentent les valeurs des dimensions de notre taxonomie ; par exemple, le noeud 1.1 correspondant à la valeur Instance pour la dimension Mode de traitement des événements ;
- les chaînes (ou configurations) représentent le modèle d'exécution d'une règle et d'un ensemble de règles.

A titre d'exemple, nous donnons les figures 3.6 et 3.7 qui permettent de comparer les modèles d'exécution de Postgres et NAOS. Pour la clarté de la présentation, les

dimensions dépendantes du modèle de règles sont délibéremment ommises. Pour la même raison, seule la dimension *mode d'exécution* (couplage E-C et C-A) est représentée en ce qui concerne les dimensions dépendantes du modèle de transactions.

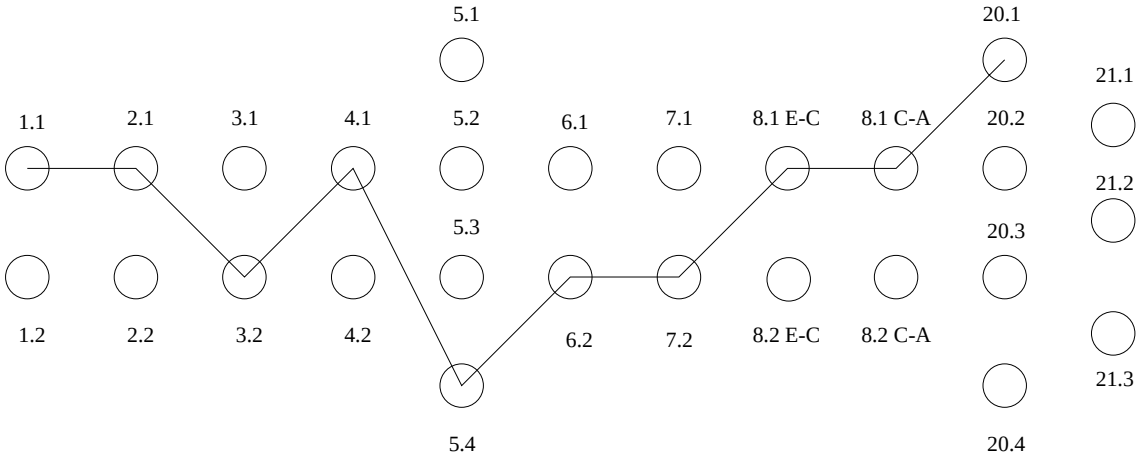

FIG. 3.6 – *Modèle d'exécution de Postgres*

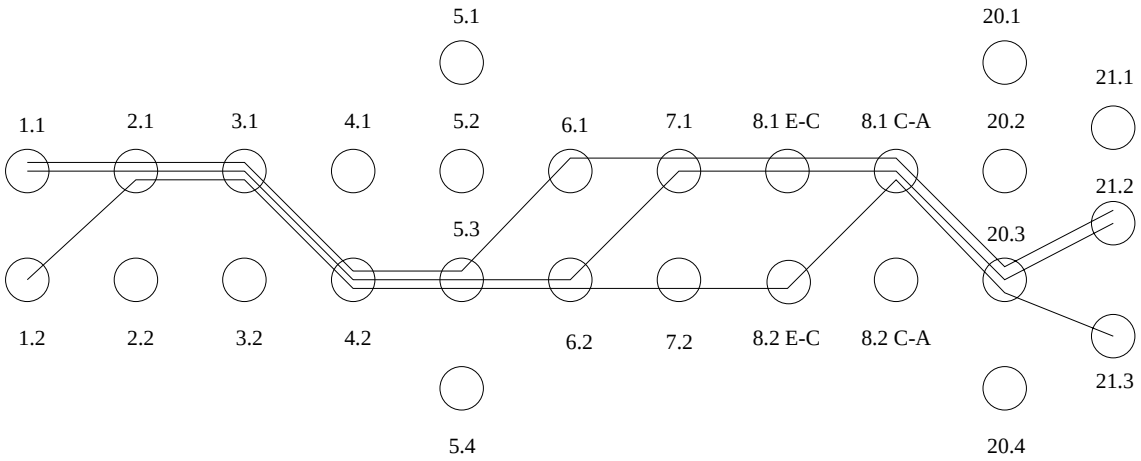

FIG. 3.7 – *Modèle d'exécution de NAOS*

Pour des raisons évidentes de place, nous ne présentons pas la représentation graphique de tous les systèmes que nous avons étudiés. Si l'on fait abstraction d'une ou deux caractéristiques propres à chaque système, ceux-ci peuvent être grossièrement

classés en trois catégories :

- La première catégorie est constituée des systèmes relationnels (non commerciaux) : A-RDL, Ariel, Starburst et Postgres – et d'une certaine manière Ode. Ces systèmes sont très empreints des systèmes déductifs. Leur modèle d'exécution est, peu ou prou, celui des systèmes déductifs. La notion d'*événement* semble effacée par rapport à celle de *condition* : on parle de règle de production et non de règles actives, la partie événement est souvent optionnelle dans l'expression des règles, nombre d'articles traitent de l'optimisation de l'évaluation des conditions, etc.
- La seconde catégorie est constituée de HiPAC et de ses héritiers REACH, Sentinel, Ode, TriGS, SAMOS, Beeri & Milo. Ces systèmes sont "plus actifs". Ils donnent une plus grande importance aux événements et leur modèle d'exécution se différencie nettement de celui des systèmes déductifs. Ces modèles reposent principalement sur des modèles de transactions évolués et s'attachent surtout aux interactions entre règles et transactions.
- La troisième et dernière catégorie est constituée de NAOS, Chimera et Peplom^{ad}. Comme les systèmes du groupe précédent, leur modèle d'exécution est nettement différencié de celui des systèmes déductifs. En revanche, ils considèrent un modèle de transactions classique et si l'on fait abstraction des dimensions dépendantes du modèle de transactions, ils fournissent les modèles d'exécution les plus riches.

3.5 Conclusion

Dans ce chapitre, nous avons étudié les modèles d'exécution des SGBD actifs. Nous avons proposé une taxonomie qui met en exergue les dimensions prises en compte pour l'exécution des règles actives et les valeurs possibles de ces dimensions. Nous avons ensuite introduit une représentation graphique des modèles d'exécution. La taxonomie et la représentation graphique permettent d'évaluer et de comparer les modèles d'exécution des systèmes existants.

Nous retenons deux traits marquants de cette étude : l'*hétérogénéité* et la *rigidité* des modèles d'exécution des systèmes étudiés. En effet, il existe une grande diversité qui se traduit par le nombre imposant de dimensions et de valeurs de ces dimensions qui sont prises en compte dans les différents systèmes. Cependant, lorsque l'on s'intéresse à un système particulier, il n'offre qu'un sous-ensemble de l'ensemble des dimensions et valeurs possibles mais, surtout, il n'offre que certaines combinaisons de valeurs pour chaque dimension. Nous pensons que cette rigidité est rédhibitoire au

vu de l'énorme potentiel que l'on prête au SGBD actifs et qui peut être estimé par le large spectre d'applications dans lesquelles l'utilisation de règles actives pourrait être très bénéfique.

Partant du même constat, certains travaux ont alors mené des investiguations quant à ces utilisations potentielles des règles actives afin de déterminer quelles caractéristiques doivent avoir un modèle d'exécution pour telle utilisation ou tel type d'application, on parle de *typologies des règles actives*. L'objectif est de fournir un modèle d'exécution *ad-hoc*, idéal pour chaque type d'utilisation. À vrai dire, nous sommes peu convaincus par cette approche. Tout d'abord parce que les domaines d'utilisation et les types d'applications dans lesquelles on pourrait utiliser des règles actives sont chaque jour plus nombreuses. Ensuite, parce que, selon nous, tous les domaines d'utilisations mis à jour sont et seront de plus en plus appelés à coexister au sein d'une même application. Dans une application de Génie Logiciel, par exemple, on aura surement besoin d'avoir des règles pour gérer l'intégrité de la base, pour faire de la notification, pour faire coopérer et communiquer différents outils, pour gérer des versions de programmes ou de documentation, etc.

Ces considérations révèlent, selon nous, le besoin de modèles d'exécution *flexibles* qui peuvent s'adapter à différentes utilisations des règles actives dans le cadre d'applications de tout type. Par ailleurs, un point sur lequel nous sommes restés discrets (pour ne pas dire circonspects !) jusqu'à présent concerne l'indépendance (ou la liberté) des dimensions que nous considérons. Certaines combinaisons peuvent se révéler incompatibles ou engendrer des modèles d'exécution incohérents ou incompréhensibles. Il est donc également primordial de veiller à la réalisation d'un choix harmonieux des paramètres et des stratégies d'exécution afin d'obtenir un modèle d'exécution certes puissant et flexible mais également facile à appréhender.

Chapitre 4

Le modèle Fl'are

> ***Carquois*** *n. Étui portable dans lequel l'ancien homme d'état et le juge indigène transportaient leurs plus clairs arguments.*
>
> *"Le chicaneur romain tira de son carquois l'argument qu'il fallait à ce point du litige, et l'adressa en plein dans la réfutation de l'autre querelleur."*
>
> ***Ambrose Bierce - "Le dictionnaire du Diable"***

Nous proposons dans ce chapitre un modèle d'exécution pour SGBD actifs souple, puissant, facile à appréhender et portable.

Comme nous l'avons souligné à la fin du chapitre précédent, les modèles d'exécution existants se caractérisent principalement par leur *rigidité*. À l'inverse, **Fl'are** (**Fl**exible **a**ctive **r**ule **e**xecution), le modèle d'exécution paramétrique que nous proposons est *flexible* et *générique* puisqu'il permet de choisir la sémantique de l'exécution d'ensembles de règles actives que nous appelons *modules*. À la différence des ensembles de règles considérés dans le chapitre précédent, on considére ici que chaque module – destiné à un domaine d'utilisation particulier des règles actives – est spécifié explicitement par le programmeur de règles (au travers d'un langage ou d'un interface).

Le modèle proposé est *facile à appréhender* car la spécification du comportement d'une règle et du module auquel elle appartient consiste simplement à *instancier* des paramètres qui peuvent prendre deux, trois ou quatre valeurs prédéfinies.

Le modèle est *portable*[1] parce qu'en nous focalisant sur l'exécution de règles ac-

1. Ceci peut être vu comme un autre aspect de la généricité proposée.

tives, nous nous plaçons dans un cadre qui se veut le plus général possible pour tous les aspects qui ne concernent pas directement cette exécution. Cela est vrai pour les *aspects classiques* des SGBD, en particulier, le modèle de données et le modèle de transactions. Cela est vrai également pour les apects des SGBD actifs autres que le modèle d'exécution, en particulier, le modèle d'événements et le modèle de règles.

Ce chapitre est organisé de la manière suivante. Dans la section 4.1, nous étudions différentes approches permettant d'offrir la flexibilité des modèles d'exécution puis nous présentons notre propre approche de manière générale, – du point de vue d'un utlisateur, c'est-à-dire d'un programmeur d'applications dans notre cas. Dans les sections 4.2 à 4.4, nous présentons le modèle dans le détail, cette fois du point du vue d'un concepteur de SGBDA. Dans la section 4.5, nous discutons des concepts exclus du modèle. Dans la section 4.6, nous illustrons la puissance du modèle en montrant comment utiliser Fl'are pour couvrir la plupart des fonctions de différents systèmes existants. Enfin, la section 4.7 conclut le chapitre.

4.1 Comment apporter la flexibilité ?

4.1.1 Les approches existantes

4.1.1.1 Heraclitus : une approche par la généricité

L'objectif du projet Heraclitus [HJ91, SGJ93] est le développement d'un Langage de Programmation de Bases de Données (LPBD) et d'autres techniques qui permettent de spécifier et d'implanter des modèles d'exécution de SGBD Relationnels Actifs (SGBDRA) "interchangeables".

Le langage Heraclitus, qui constitue le cœur du projet, est une extension du langage impératif C qui supporte l'algèbre relationnelle ainsi qu'une *algèbre des deltas* qui permet de spécifier la sémantique d'un modèle d'exécution. Le concept central d'Heraclitus est, en effet, le *delta* : une valeur représentable dans le langage qui représente une *mise à jour potentielle* d'une base, c'est-à-dire une *transition* entre deux états virtuels de la base [2].

Les deltas peuvent être utilisés directement ou combinés algébriquement pour créer de nouveaux deltas. Par cette algèbre (et avec un peu d'efforts !), on peut donc implanter, ou tout au moins, simuler certaines dimensions de notre taxonomie (cf. chapitre 3), telles que : mode de traitement des événements, mode de consommation des événements, effet net, mode d'exécution.

2. cf. section 2.2.2 : événements internes dans les systèmes relationnels.

Heraclitus constitue **une approche par la généricité** de la flexibilité des modèles d'exécution : il est possible de définir une infinité de modèles d'exécution ... en codant les fonctions désirées en utilisant le langage Heraclitus. Il est ainsi montré que le modèle d'exécution de Starburst est *couvert* par Heraclitus. Il faut toutefois remarquer qu'Heraclitus est utilisable uniquement dans le cadre des systèmes relationnels dont les fonctions actives sont basées sur la notion de *transitions*.

Si on se place du point de vue de l'utilisateur, donc du programmeur d'applications et de règles dans notre cas, la limitation majeure de cette approche découle paradoxalement de sa puissance. En simplifiant à l'extrême, on pourrait dire qu'"avec Heraclitus, on peut tout faire, mais il faut le faire et on ne sait pas trop comment le faire !" On peut, par exemple, regretter :

- qu'il ne soit pas proposé de cadre général spécifiant ce qu'est un modèle d'exécution avec les dimensions pouvant être prises en compte, et donc que les limitations de l'approche n'apparaissent pas explicitement ;
- qu'il ne soit pas proposé de modèle d'exécution par défaut ou de méthodologie pour créer un nouveau modèle d'exécution. Ce point est d'autant plus restrictif qu'Heraclitus permet de représenter à la fois : l'application (passive) considérée, les règles de cette application, et le modèle d'exécution de ces règles. La spécification du modèle d'exécution se trouve donc éclatée et disséminée dans un programme Heraclitus.

Tempérons toutefois ces critiques en considérant que les concepteurs d'Heraclitus ont une conception du terme "utilisateur" différente de la nôtre. Heraclitus est principalement destiné à des concepteurs de systèmes actifs plutôt qu'à des concepteurs d'applications bases de données actives – qui constituent, en revanche notre cible privilégiée. Heraclitus doit, sans doute, être vu plus comme une plate-forme pour tester la cohérence, la validité ou l'"utilisabilité" des modèles d'exécution que comme un outil pouvant être mis directement à disposition des programmeurs d'applications.

4.1.1.2 EXACT : une approche par l'extensibilité

L'objectif du projet EXACT (EXtensible approach to ACTive object-oriented DBMSs) [DJ94, DP94] est la définition d'un modèle d'exécution de règles actives qui doit présenter les qualités suivantes [DJ94] :

- *hétérogénéité* : différents modules de règles actives peuvent avoir différents modèles d'exécution suivant la sémantique du concept (domaine d'utilisation des règles) qu'ils supportent ;

- *extensibilité* : le *modèle d'exécution standard*, c'est-à-dire celui fourni par le système, doit pouvoir être facilement étendu en redéfinissant ou en ajoutant certaines fonctionnalités ;
- *déclarativité* : le modèle d'exécution désiré doit pouvoir être spécifié au travers d'un ensemble de paramètres plutôt que d'être codé comme cela est fait actuellement (dans Heraclitus par exemple !).

EXACT est implanté au-dessus du SGBD à objets ADAM [Pat89] et prône *la politique du tout-objet*. Les événements, les règles et leur modèle d'exécution sont représentés et manipulés comme tout objet ADAM. L'obtention des trois qualités énoncées ci-dessus est également basée sur le *paradigme objet* et plus particulièrement sur les concepts de *spécialisation* de classes et de *métaclasses* :

- Un exemple de spécialisation est donné par la figure 4.1 qui décrit une hiérarchie de classes ADAM. Le modèle d'exécution standard fournit la classe

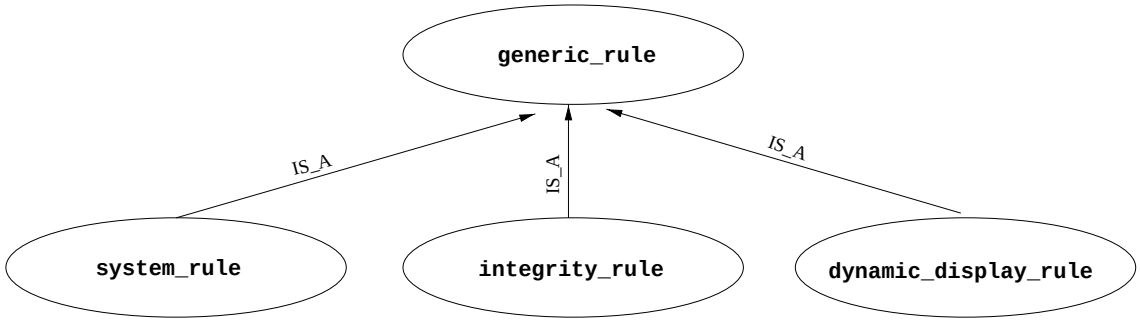

FIG. 4.1 – *Hiérarchie de classes de règles dans EXACT*

`generic_rule` qui est spécialisée pour obtenir les classes `system_rule`, `integrity_rule` et `dynamic_display_rule`. L'*extension* de la classe `integrity_rule`, par exemple, c'est-à-dire l'ensemble des objets instances de `integrity_rule`, représente un ensemble de règles pour une utilisation particulière : le renforcement de l'intégrité d'une BD, avec un modèle d'exécution pour cette utilisation.
- le modèle d'exécution d'un ensemble de règles est fixé par une instance de la métaclasse `rule_manager` (cf. figure 4.2). Il y a donc autant d'instances de `rule_manager` que d'ensembles de règles. La méthode `scheduling` revêt une importance particulière puisque c'est elle qui détermine le plan d'exécution local et le plan d'exécution global d'un ensemble de règles.
- le modèle d'exécution d'une règle est fixé par son appartenance à une classe donnée, c'est-à-dire par les attributs (et leur valeur) de cette classe. Il est pos-

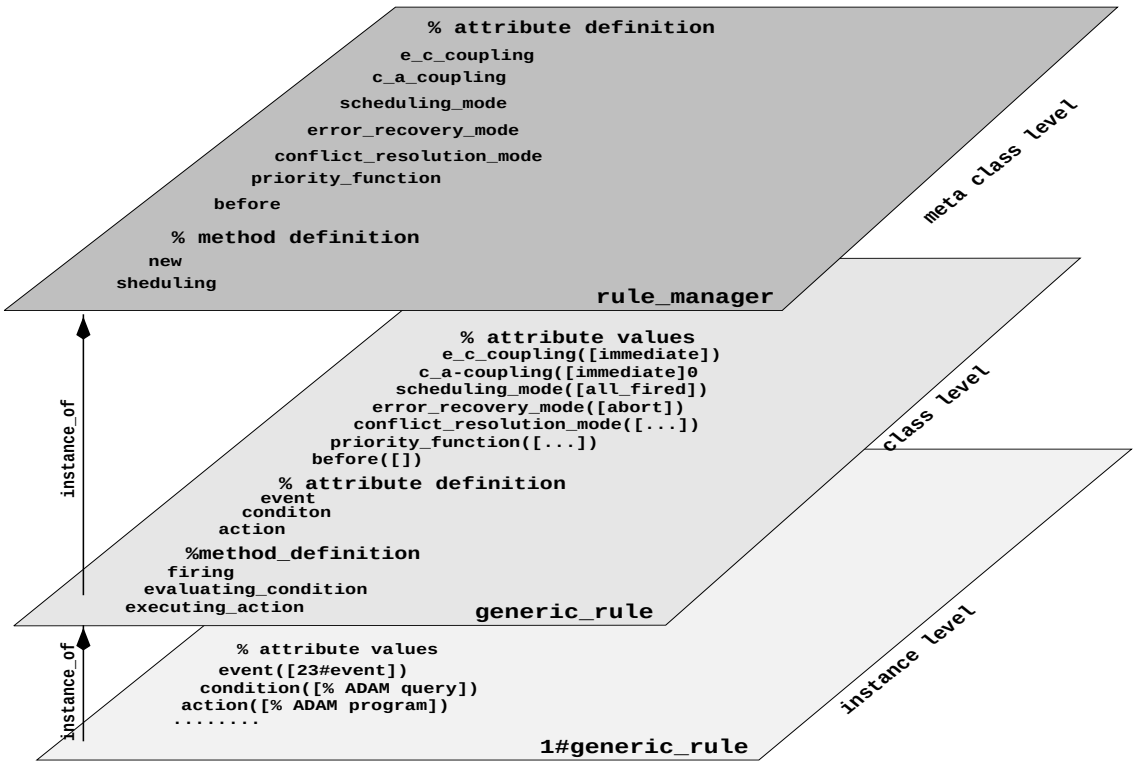

FIG. 4.2 – *Objets, classes et métaclasses de règles dans EXACT*

sible d'ajouter des attributs dans une sous-classe et d'ajouter ou de modifier des méthodes pour modifier le modèle d'exécution d'une règle.

EXACT constitue **une approche par l'extensibilité** de la flexibilité des modèles d'exécution. Cette approche utilise toute la puissance du modèle à objets. Il est possible de construire des ensembles de règles et de donner à chacun une sémantique particulière par instanciation et spécialisation de classes et de métaclasses. Les dimensions du modèle d'exécution prises en compte sont explicitement représentées par les attributs et les méthodes des classes et métaclasses. Il est de plus possible d'ajouter ou de modifier ces attributs et méthodes pour obtenir de nouveaux modèles d'exécution.

Vis-à-vis de l'objectif que nous poursuivons, nous pouvons faire les remarques suivantes :

1° Certaines dimensions de l'exécution d'un ensemble de règles apparaissent dans l'expression des règles elles-mêmes. C'est le cas notamment des attributs `scheduling_mode`, `conflict_resolution_mode`, `priority_function`,

qui servent à établir le plan d'exécution local et que l'on s'attendrait donc plutôt à voir dans la métaclasse `rule_manager`. Ce phénomène peut provoquer des redondances si toutes les règles – c'est-à-dire toutes les instances d'une sous-classe de `generic_rule` – ont les mêmes valeurs pour ces attributs ; des incohérences sinon.

2° Le modèle fourni est un modèle *standard* (ou par défaut) et il est possible de le modifier ou de l'étendre pour obtenir le modèle d'exécution désiré. Dans les faits, le modèle standard est assez simpliste (en particulier en ce qui concerne l'exécution d'un ensemble de règles) et l'utilisateur est rapidemment "invité" à coder le corps des méthodes implantant le modèle d'exécution désiré (en particulier la méthode `scheduling` de la métaclasse `rule_manager`). Ceci constitue, nous semble-t-il, des limitations quant à l'utilité de l'extensibilité et de la déclarativité proposées, pour un utilisateur, c'est-à-dire un programmeur d'applications de règles.[3]

3° C'est l'appartenance d'un objet règle r à une classe de règles R (plus précisement à l'extension de cette classe) – donc une sous-classe de `generic_rule` qui fixe le modèle d'exécution de r. On est alors face à deux éventualités :

(a) soit R hérite simplement, c'est-à-dire sans ajout ou modification d'attributs ou méthodes et la spécialisation perd un peu de son sens ;

(b) soit R définit de nouveaux attributs ou méthodes ou redéfinit des méthodes. Ces nouvelles dimensions de l'exécution ne sont pas prises en compte par le modèle standard et on retombe alors dans le cas 2. ci-dessus.

En réalité, c'est l'utilisation intensive du paradigme objet qui nous parait être la limitation majeure de l'approche comme le montre les deux points développés ci-dessus. Signalons enfin que de la même manière que l'approche proposée par Heraclitus est envisageable uniquement dans le cadre de systèmes relationnels, l'approche proposée par EXACT est envisageable uniquement dans le cadre de systèmes à objets.

4.1.2 Survol de notre approche

4.1.2.1 Vue générale du modèle

Heraclitus et EXACT proposent des modèles flexibles basés respectivement sur la généricité et l'extensibilité. Une seconde caractéristique de ces modèles est qu'ils sont fortement liés à un modèle de données : relationnel pour Heraclitus, objet pour

3. D'un point de vue théorique, le nombre de modèles d'exécution pouvant être générés est infini.

EXACT.

Le modèle Fl'are que nous proposons est quant à lui, flexible et générique, c'est-à-dire (le plus possible) indépendant d'un modèle de connaissances particulier.

Notre approche de la flexibilité est basée sur les deux principes suivants :

1° **la modularité** : les règles sont regroupées dans des **ensembles de règles** ou **modules** – construits par le programmeur de règles. Les modules de règles considérés sont hétérogènes : les règles d'un module peuvent avoir des modèles d'exécution de règles différents. Ceci constitue une différence majeure avec EXACT où les ensembles regroupent des règles de même structure[4].

2° **l'utilisation de paramètres** : les modèles d'exécution des règles et des modules de règles sont spécifiés par l'instantiation de *paramètres* prenant des *valeurs prédéfinies* et non au travers d'un langage. On peut qualifier les modèles d'Heraclitus ou EXACT de *modèles flexibles ouverts* : il est possible de spécifier complétement (à partir de rien) des modèles d'exécution avec Heraclitus ou d'étendre à volonté des modèles existants avec EXACT. Par opposition, le modèle que nous proposons peut être qualifé de *modèle flexible fermé*. C'est un modèle paramétrique qui est basé sur une approche *boîte-à-outils* (*toolkit* ou *plug-and-play* en anglais) : le concepteur d'application a simplement à choisir parmi un ensemble prédéfini les composants (ou "briques") qui permettent de spécifier des modèles d'exécution.

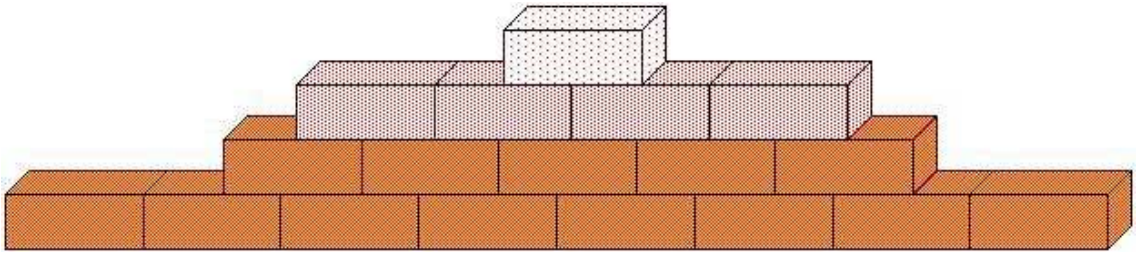

FIG. 4.3 – *Vue générale du modèle (les briques de Fl'are)*

Fl'are a une structure en trois couches (cf. figure 4.3) :

– la *couche I* (en gris foncé) gère l'exécution d'une règle ; elle est composée de huit constantes et de cinq paramètres dont les valeurs effectives spécifient le modèle d'exécution pour une règle ;

4. Pour être plus juste, ce sont les objets qui représentent les règles qui ont la même structure.

- la *couche II* (en gris clair) gère les interactions entre les règles d'un module ou *exécutions intra-module* ; elle est constituée d'un paramètre pouvant prendre quatre valeurs qui représentent quatre stratégies d'exécution possibles pour ce module (plan d'exécution local et global) ;
- la *couche III* (en gris très clair) gère les interactions entre modules de règles ou *exécutions inter-modules* (ou exécutions globales).

4.1.2.2 Méthodologie et motivations

Méthodologie Le modèle que nous proposons se présente sous la forme d'un certain nombre de paramètres et de valeurs possibles pour ces paramètres. Les dimensions candidates pour devenir paramètres proviennent de la taxonomie des modèles d'exécution que nous avons présentée dans le chapitre précédent. Le choix des paramètres est relativement arbitraire. Il est toutefois guidé par (1) le contexte dans lequel nous nous plaçons et (2) les qualités que nous attendons du modèle.

1. Contexte :
   - nous faisons des hypothèses minimales sur le modèle de connaissances – c'est-à-dire tout ce qui n'est pas purement du ressort du modèle d'exécution :
     - le modèle de données (relationnel, relationnel étendu, objet, etc.) nous est indifférent ;
     - en ce qui concerne le modèle et le langage de règles, il nous suffit de savoir que les règles sont de la forme Evénement-Condition-Action ; et qu'il existe un moyen de spécifier les modules de règles, c'est-à-dire un moyen de décrire l'appartenance d'une règle à un module de manière à ce que chaque règle appartienne à *exactement un module* (un et un seul) ;
     - en ce qui concerne le modèle et le langage d'événements, on considère que les événements sont typés mais la nature des événements nous est indifférente ; il nous suffit de connaître les couples (type d'événement,règle) ;
   - le modèle de transactions considéré est classique (cf. 2.3.1). Il n'y a pas de transactions emboîtées ou découplées. La transaction déclenchante est l'unité de production des événements et d'exécution des règles.
2. Qualités attendues :
   - nous visons à offrir le plus grand nombre possible de paramètres tout en préservant l'indépendance de ces paramètres : les paramètres sont indé-

pendants (deux à deux), toutes les combinaisons de valeurs pour tous les paramètres proposés sont valides ;

- les modèles d'exécution créés avec le modèle paramétrique doivent engendrer des exécutions de règles déterministes ;
- les paramètres et valeurs retenus doivent être *pertinents*. Cet aspect est évidemment le plus difficile à estimer car purement qualitatif.

En fonction du contexte et des qualités attendues que nous avons définis, les dimensions mises en évidence dans notre taxonomie sont *transformées* en paramètres avec les valeurs qu'ils peuvent prendre, en constantes (c'est-à-dire en paramètres fixés), ou écartées du modèle.

La couverture des systèmes existants par Fl'are, discuté dans la section 4.6.2, donne une idée de la puissance du modèle et de la pertinence des paramètres retenus.

Motivations Nous visons un modèle d'exécution **puissant** et **facile à appréhender**, c'est-à dire à facile à implanter et facile à utiliser. Puissant grâce à la flexibilité (qui incarne la généricité) fournit par l'utilisation de paramètres, facile à implanter grâce à la portabilité (qui est un autre aspect de la généricité) et facile à utiliser grâce à :

- la modularité : il est plus facile de concevoir et utiliser un module de vingt règles qu'une base de deux cents règles ;
- l'indépendance des paramètres : l'utilisateur n'a pas à mémoriser les combinaisons valides de paramètres – cette simplicité se retrouvera certainement au niveau du(des) langage(s) ou interface(s) offerts pour utiliser notre modèle (on peut imaginer des langages de type *interface de formulaires* par exemple) ; ce point est abordé dans la mise en œuvre de Fl'are présentée dans le chapitre 6 ;
- l'exécution déterministe des règles : celle-ci facilite la prise en main du système par un utilisateur puisqu'elle lui permet de *rejouer* une exécution et donc de *mettre au point* ses modules de règles. A ce sujet, nous avons déjà souligné le besoin d'outils tels que traceurs et déboggeurs pour les systèmes actifs et nous pensons que la conception et l'implantation de ces outils sont grandement simplifiés dans un contexte d'exécution déterministe.

4.2 Couche I : Modèle d'exécution d'une règle

Le modèle d'exécution d'une règle est relativement classique – au sens où nous ne proposons pas de nouvelles caractéristiques. Aussi, nous attachons-nous principale-

ment dans cette partie, à justifier les choix que nous faisons lors de la *transformation* des dimensions en paramètres ou constantes dont nous avons parlé ci-dessus.

Du point de vue du programmeur d'applications, la couche I du modèle se présente sous la forme de huit constantes et cinq paramètres qui prennent des valeurs prédéfinies.

4.2.1 Constantes

Les constantes du modèle d'exécution d'une règle sont celles données par la table 4.1.

	constante	valeur
1	portée de la consommation des événements	locale
2	instant de la consommation des événements	évaluation
3	annulation opération déclenchante	non
4	mode de transaction	même transaction
5	mode de synchronisme	synchrone
6	mode de dépendance	dépendant
7	échec Transaction Déclenchante TD	abandon TD
8	échec transaction de règle	abandon TD

TAB. 4.1 – *Constantes de l'exécution d'une règle*

Constante 1 La dimension portée de la consommation des événements a deux valeurs locale et globale. Si cette dimension était considérée comme un paramètre avec comme valeur possible globale, les paramètres des couches I et II ne seraient plus indépendants puisque la consommation globale des événements par une règle R a une influence sur l'exécution d'un ensemble de règles (les règles déclenchées par les événements qui ont déclenché R).

Constante 2 Dans notre modèle, la consommation des événements a toujours lieu à l'évaluation : une règle déclenchée n'est considérée qu'une seule fois pour un ensemble d'événements donné. En effet, une consommation à l'exécution signifierait qu'une règle dont la condition a été évaluée à `faux` est susceptible d'être considérée à nouveau pour le même ensemble d'événements déclenchants. Le problème est de déterminer quand elle doit être considérée de nouveau. Intuitivement, on serait tenté de considérer à nouveau la règle "lorsque la condition est évaluée à `vrai`". Ceci s'apparente plus à la

sémantique des règles de déduction qu'à la sémantique des règles actives. Un SGBD actif réagit, en effet, à des événements et non à des prédicats sur l'état de la base.

Par ailleurs, une consommation à l'exécution impliquerait également qu'il soit possible que des règles déclenchées ne soient jamais exécutées et restent donc déclenchées à la fin de la transaction déclenchante, causant par là une non terminaison de l'exécution des règles.

Enfin, comme nous l'avons signalé dans la section 3.1.1.2, la consommation des événements à l'exécution a été proposée par un seul système puis supprimée lors de l'implantation de ce système car, précisément, il est extrêmement difficile de définir la sémantique associée à cette consommation et de mettre en œuvre ce type de règle.

Pour toutes ces raisons, nous considérons donc que la possibilité de consommer les événements à l'exécution est *cosmétique*[5], c'est-à-dire relativement peu pertinente !

Constante 3 Nous n'offrons pas la possibilité d'exécuter une règle en substituant l'opération déclenchante (cf 3.1.3.1). En effet, si nous le faisions :

1. notre modèle d'exécution ne serait pas indépendant des modèles de données et d'événements. En effet, la substitution de l'opération déclenchante n'est possible que pour certains types d'événements. Or nous avons choisi de ne pas considérer la nature des événements mais uniquement les associations (type d'événement, règle). Il faudrait ensuite être capable d'annuler, c'est-à-dire de défaire ou de ne pas exécuter du tout l'opération déclenchante, ce qui est fortement lié au modèle de données et plus généralement au SGBD sous-jacent.
2. les paramètres des couches I et II de notre modèle ne seraient pas indépendants puisque la substitution de l'opération déclenchante par une règle peut avoir une influence sur le déclenchement et l'exécution d'autres règles.

Constantes 4 à 8 Les constantes 4 à 8 sont induites par le modèle de transactions classique que nous considérons. Les règles R_i déclenchées dans une transaction déclenchante TD s'exécutent, de manière synchrone dans cette transaction ($TR_i \equiv TD$). Un échec de la transaction déclenchante ou d'une règle provoque l'abandon de la transaction et la fin de l'exécution des règles déclenchées au cours de cette transaction (et non encore exécutées !).

5. Nous empruntons l'adjectif à [Pic95].

4.2.2 Paramètres

Les cinq paramètres du modèle d'exécution d'une règle que nous proposons sont donnés par le tableau 4.2. Nous détaillons uniquement ici les paramètres 3 et 5. Les autres paramètres correspondent directement aux dimensions du chapitre 3.

	paramètre	valeurs
1	mode de traitement des événements	instance
		ensembliste
2	mode de consommation des événements	consommation
		préservation
3	effet net	oui
		non
4	mode de préemption	préemption
		non préemption
5	mode d'exécution	immédiat
		différé
		retardé

TAB. 4.2 – *Paramètres de l'exécution d'une règle*

Paramètre 1 Une règle avec une sémantique d'instance est exécutée pour chaque événement déclenchant tandis qu'une règle ensembliste est exécutée pour un ensemble d'événements déclenchants (produits depuis le début de la transaction déclenchante ou depuis la dernière exécution de la règle dans cette transaction).

Paramètre 2 Si une règle r consomme les événements, chaque événement déclenchant e_i est pris en compte uniquement pour une exécution de r, tandis que si r préserve les événements, chaque événement est pris en compte pour toutes les exécutions de r qui surviennent après occurrence de e_i (et jusqu'à la fin de la transaction déclenchante). Nous rappelons que du fait de l'existence des constantes 1 et 2, dans le mode consommation, cette consommation est locale et a lieu à l'évaluation.

Paramètre 3 Nous indiquons, dans la taxonomie présentée dans le chapitre 3, que l'*effet net* peut être pris en compte pour le déclenchement des règles et/ou la construction des environnements d'exécution des règles ou ne pas être pris en compte du tout.

En ce qui concerne les deux systèmes qui prennent en compte l'effet net uniquement pour le déclenchement ou uniquement pour la construction des environnements

d'exécution, Ode [GJ91] et Chimera [CFPT95a], nous émettons les remarques suivantes :

1° dans Ode, l'effet net est pris en compte uniquement pour le déclenchement des règles mais celles-ci ont une forme particulière (règles EC-A, *événements logiques* et *masques d'événements*). La notion d'environnement est donc , elle aussi, particulière à ce système et induit que l'effet net est implicitement pris en compte pour la construction des environnements d'exécution des règles ;

2° dans Chimera, l'effet net peut être pris en compte uniquement au travers des opérateurs du langage de règles qui permettent l'accès aux historiques d'événements, c'est-à-dire une forme plus générale des environnements d'exécution des règles. On peut donc considérer que, d'une certaine manière, l'effet net n'est pas pris en compte par le modèle d'exécution de Chimera (mais par le modèle de règles).

A la lumière de ces remarques, nous pensons que les quatres valeurs proposées pour la dimension effet net peuvent se ramener à deux. Aussi, le paramètre effet net de notre modèle spécifie si l'effet net est pris en compte ou non pour l'exécution d'une règle. Lorsqu'il est pris en compte, il l'est à la fois pour le déclenchement et pour la construction de l'environnement d'exécution de cette règle.

Paramètre 5 Dans le chapitre 3, le tableau 3.3 met en évidence les dimensions du modèle d'exécution d'une règle dépendantes du modèle de transactions. Nous avons vu dans la section précédente que le fait de considérer un modèle de transactions classique limite les choix quant aux valeurs possibles pour ces dimensions. Seule la dimension mode de déclenchement peut, en fait, offrir plusieurs possibilités. Dans la taxonomie, pour une règle donnée, on doit considérer les modes de déclenchement - Événement-Condition d'une part, - et Condition-Action d'autre part. Dans notre modèle, nous proposons de synthétiser ces deux modes de déclenchement en un seul paramètre, le mode d'exécution (cf. tableau 4.3) qui permet de représenter trois combinaisons parmi les quatre théoriquement possibles. Nous écartons donc la combinaison <différé,différé> à laquelle nous n'avons pu associer de sémantique !

C-A / E-C	immédiat	différé
immédiat	immédiat	retardé
différé	différé	-

TAB. 4.3 – *Mode d'exécution d'une règle*

Le Mode d'exécution permet de définir trois types de règles :
- les règles *immédiates* qui sont évaluées et exécutées immédiatement après leur déclenchement ;
- les règles *différées* dont l'évaluation et l'exécution sont différées à la validation de la transaction déclenchante ;
- les règles *retardées* qui sont évaluées immédiatement après leur événement déclenchement mais dont l'exécution est différée à la validation de la transaction déclenchante.

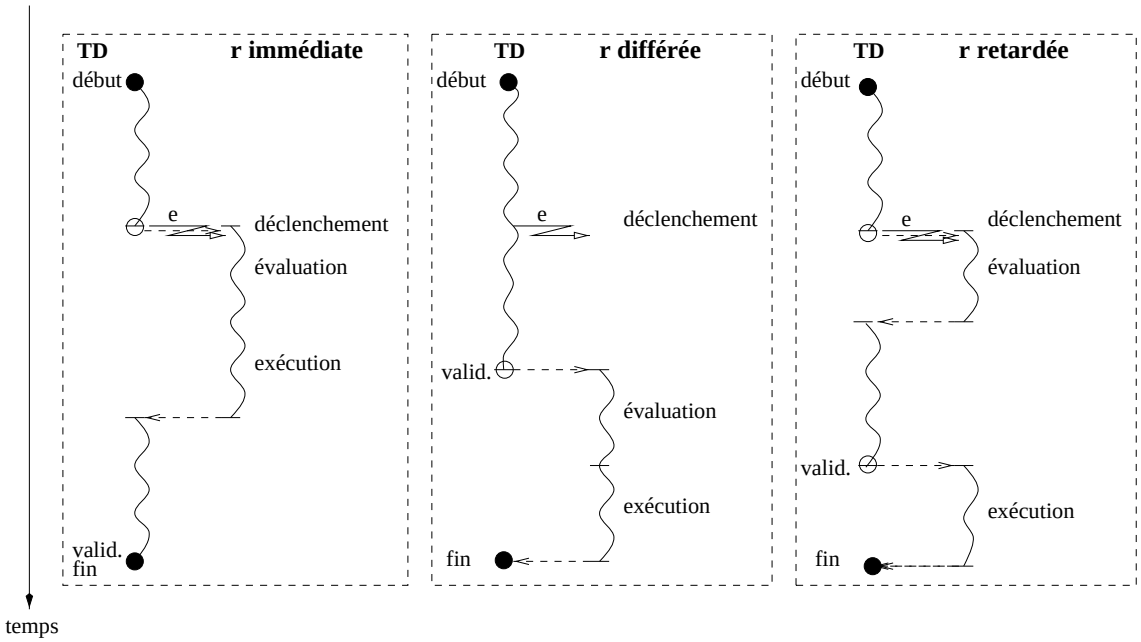

FIG. 4.4 – *Mode d'exécution d'une règle*

Ces trois types de règles sont schématisés dans la figure 4.4 qui présente l'exécution synchrone d'une règle r, déclenchée par un événement e dans une transaction déclenchante TD.

4.3 Couche II : Modèle d'exécution intra-modules de règles

Le modèle d'exécution intra-modules de règles se présente sous la forme d'un paramètre qui permet de choisir une stratégie d'exécution parmi les quatre présentées dans le tableau 4.4.

	paramètre	valeur
6	stratégie d'exécution	priorités (à plat)
		pipelines (FIFO)
		profondeur (LIFO)
		cycles (en largeur)

TAB. 4.4 – *Paramètres de l'exécution intra-modules de règles*

Les quatre stratégies proposées sont issues de la considération de trois sources d'ordonnancement : l'instant de déclenchement des règles, les priorités entre règles et les cycles d'exécution. Ces trois sources fournissent quatre ordres :

- l'*ordre de déclenchement* des règles, lui-même basé sur les instants de déclenchements des règles, est un ordre partiel puisque plusieurs (occurrences de) règles peuvent avoir le même instant de déclenchement (cf. 3.1.1.1) ;
- l'*ordre inverse des déclenchements* ;
- l'*ordre des priorités* : nous considérons qu'il existe un ordre total entre les règles d'un module ; cet ordre peut-être spécifié par le programmeur ou le système ;
- l'*ordre des cycles (d'exécution)* : un cycle est une unité logique d'exécution dans laquelle s'exécutent la transaction déclenchante, une ou plusieurs règles. Une règle déclenchée dans un cycle n sera considérée dans le cycle n+1. L'ordre des cycles est un ordre partiel puisque plusieur règles peuvent être déclenchées dans un même cycle.

Le tableau 4.5 montre comment sont construites les quatre stratégies d'exécution proposées. L'exécution par priorités consiste à considérer (1) l'ordre des priorités entre règles puis (2) l'ordre de déclenchement s'il y a plusieurs occurrences d'une même règle (cf. 3.1.1.1). L'exécution en pipelines (resp. en profondeur) consiste à considérer (1) l'ordre (resp. inverse) de déclenchement des règles puis (2) l'ordre des priorités pour les règles qui ont le même instant de déclenchement. L'exécution par cycles consiste à considérer (1) l'ordre des cycles puis (2) l'ordre des priorités entre règles et enfin (3) l'ordre de déclenchement s'il y a plusieurs occurrences d'une même règle à l'intérieur d'un cycle.

Remarque Comme le suggère le tableau 4.5, il existe d'autres combinaisons des ordres considérés. On pourrait égalemment considérer d'autres ordres comme l'ordre inverse des priorités, l'ordre inverse des cycles d'exécutions, etc. Nous n'avons pas réellement poursuivi l'investigation de ces possibilités.
□

stratégies	ordres			
	priorités	déclenchement	inv. déclenchement	cycles
priorités	1	2		
pipelines	2	1		
profondeur	2		1	
cycles	2	3		1

TAB. 4.5 – *Construction des stratégies d'exécution intra-modules de règles*

Du point de vue d'un concepteur de SGBDA, le modèle d'exécution intra-modules est sans doute la partie la plus complexe du modèle d'exécution car c'est à ce niveau que sont réellement pris en compte les paramètres de la couche I. La couche II est, en quelque sorte, une couche applicative (moteur d'exécution) qui rappelle le moteur d'inférence des systèmes déductifs. Aussi, dans un souci de lisibilité du document, nous décrivons, dans un premier temps, les différentes stratégies, puis dans un deuxième temps, comment les paramètres de la couche I sont pris en compte au niveau de la couche II.

4.3.1 Stratégies d'exécution

Pour décrire dans le détail les quatre stratégies proposées, nous utilisons l'exemple donné par la figure 4.5 qui décrit un graphe de déclenchement.

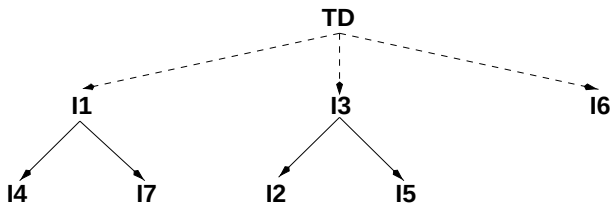

FIG. 4.5 – *Un arbre de déclenchement*

À un point d'exécution de la transaction déclenchante TD, trois règles I1, I3 et I6 sont déclenchées. L'exécution de I1 déclenche à son tour les règles I4 et I7 ; l'exécution de I3 déclenche les règles I2 et I5. On fait, de plus, les hypothèses suivantes :

1. toutes les règles appartiennent à un même module de règles $\mathcal{I}$;
2. les règles I1, I3 et I6 sont déclenchées à un même instant t ;
3. les règles I4 et I7 sont déclenchées à un même instant t' ;
4. les règles I2 et I5 sont déclenchées à un même instant t'' ;

5. $(t' > t$ et $t'' > t)$, $(t' > t''$ ou $t' < t'')$[6] ;
6. les priorités des règles sont données par leur numéro : I1 est la plus prioritaire, I7 la moins prioritaire ;
7. toutes les règles sont immédiates, ont une sémantique d'instance, consomment les événements, et n'ont pas le droit de préemption (cf. 3.1.3.2).

4.3.1.1 Exécution par priorités

Principe On traite les règles dans l'ordre de leur priorité.
Pour ce faire, on utilise l'algorithme suivant :

---

1. construire la liste L des règles à traiter, - les règles sont classées dans cette liste par ordre de priorités décroissantes ;
2. traiter (évaluation et/ou A-exécution) la règle R qui est en tête de la liste L[7] et insérer dans cette liste les éventuelles règles déclenchées par l'exécution de R **selon l'ordre décroissant de leur priorité** ;
3. supprimer R de la liste L ;
4. si la liste L est non vide alors retourner en 2.

Point	État de la liste avant traitement	Traitement d'une règle	État de la liste après traitement[a]
1	[I1,I3,I6]	I1 $\rightsquigarrow$ {I4,I7}	[I3,I4,I6,I7]
2	[I3,I4,I6,I7]	I3 $\rightsquigarrow$ {I2,I5}	[I2,I4,I5,I6,I7]
3	[I2,I4,I5,I6,I7]	I2	[I4,I5,I6,I7]
4	[I4,I5,I6,I7]	I4	[I5,I6,I7]
5	[I5,I6,I7]	I5	[I6,I7]
6	[I6,I7]	I6	[I7]
7	[I7]	I7	[]

FIG. 4.6 – *Une exécution intra-module par priorités*

[a] Après les phases 2 et 3 de l'algorithme.

Exemple La figure 4.6 décrit l'exécution des règles du module $\mathcal{I}$ de notre exemple (cf. figure 4.5) selon la stratégie d'exécution par priorités. Au point d'exécution 1, la liste des règles à traiter est [I1,I3,I6]. La règle I1 est exécutée la première

6. t' et t'' dépendent justement de la stratégie d'exécution de $\mathcal{I}$.
7. Il s'agit toujours de la règle la plus prioritaire.

car elle est la plus prioritaire. Son exécution déclenche les règles I4 et I7 qui sont alors insérées dans la liste selon leur priorités : I4 entre I3 et I6, I7 après I6. Ces règles seront traitées au point d'exécution 2. I1 est, quant à elle, supprimée de la liste.

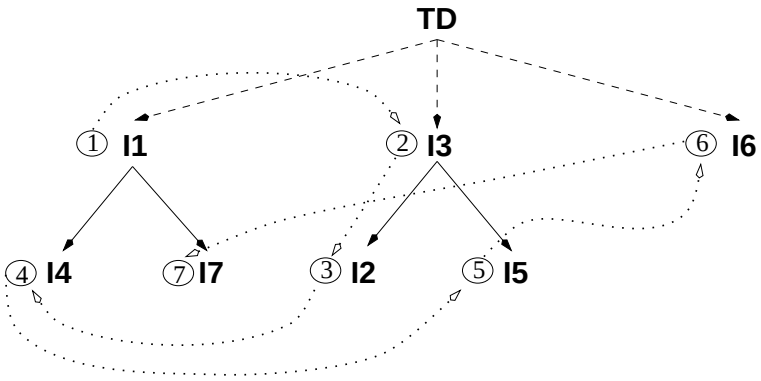

FIG. 4.7 – *Ordre d'exécution de règles par priorités*

L'ordre d'exécution final des sept règles est représenté dans la figure 4.7.

4.3.1.2 Exécution en pipelines

Principe On traite les règles évaluables et A-exécutables dans l'ordre de leur déclenchement. Si plusieurs règles ont le même instant de déclenchement, on les traite dans l'ordre de leur priorité.

Pour ce faire, on utilise l'algorithme suivant :

1. construire la liste L des règles à traiter, - les règles sont classées dans cette liste par ordre de priorités décroissantes[8] ;
2. traiter (évaluation et/ou A-exécution) la règle R qui est en tête de la liste L et insérer les éventuelles règles déclenchées par l'exécution de R **en queue de la liste – dans l'ordre de leur déclenchement, – puis dans l'ordre décroissant de leur priorité pour celles qui ont le même instant de déclenchement** ;
3. supprimer R de la liste L ;
4. si la liste L est non vide alors retourner en 2.

8. Ces règles ont, par définition, le même instant de déclenchement.

Point	État de la liste avant traitement	Traitement d'une règle	État de la liste après traitement
1	[I1,I3,I6]	I1 $\leadsto$ {I4,I7}	[I3,I6,I4,I7]
2	[I3,I6,I4,I7]	I3 $\leadsto$ {I2,I5}	[I6,I4,I7,I2,I5]
3	[I6,I4,I7,I2,I5]	I6	[I4,I7,I2,I5]
4	[I4,I7,I2,I5]	I4	[I7,I2,I5]
5	[I7,I2,I5]	I7	[I2,I5]
6	[I2,I5]	I2	[I5]
7	[I5]	I5	[]

FIG. 4.8 – *Une exécution intra-module par pipelines*

Exemple Au point 2 (cf. figure 4.8), la liste des règles à traiter est [I3,I6,I4,I7]. La règle I3 est sélectionnée. Son exécution déclenche I2 et I5 qui sont insérées en queue de liste, c'est-à-dire après I7, – dans l'ordre de leur priorité, c'est-à-dire I2 puis I5. Au point suivant, la liste des règles à traiter est donc [I6,I4,I7,I2,I5].

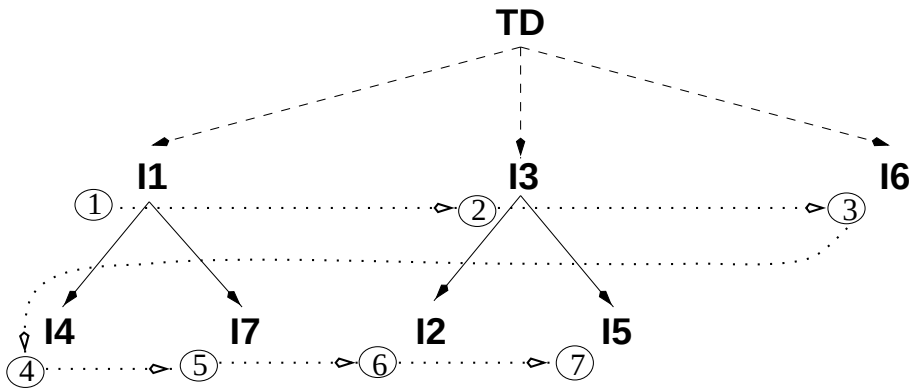

FIG. 4.9 – *Ordre d'exécution de règles par pipelines*

L'ordre d'exécution final des sept règles est représenté dans la figure 4.9.

4.3.1.3 Exécution en profondeur

Principe On traite les règles évaluables et A-exécutables dans l'ordre inverse de leur déclenchement. Si plusieurs règles ont le même instant de déclenchement, on les traite dans l'ordre de leur priorité.

Pour ce faire, on utilise l'algorithme suivant :

1. construire la liste L des règles à traiter, - les règles sont classées dans cette liste par ordre de priorités décroissantes ;
2. traiter (évaluation et/ou A-exécution) la règle R qui est en tête de la liste L et insérer les éventuelles règles déclenchées par l'exécution de R **en tête de la liste L , – dans l'ordre de leur déclenchement, – puis dans l'ordre décroissant de leur priorité pour celles qui ont le même instant de déclenchement** ;
3. supprimer R de la liste L ;
4. si la liste L est non vide alors retourner à l'étape 2.

Point	État de la liste avant traitement	Traitement d'une règle	État de la liste après traitement
1	[I1,I3,I6]	I1 $\leadsto$ {I4,I7}	[I4,I7,I3,I6]
2	[I4,I7,I3,I6]	I4	[I7,I3,I6]
3	[I7,I3,I6]	I7	[I3,I6]
4	[I3,I6]	I3 $\leadsto$ {I2,I5}	[I2,I5,I6]
5	[I2,I5,I6]	I2	[I5,I6]
6	[I5,I6]	I5	[I6]
7	[I6]	I6	[]

FIG. 4.10 – *Une exécution intra-module en profondeur*

Exemple Au point 4 (cf figure 4.10), la liste des règles à traiter est [I3,I6]. La règle I3 est sélectionnée. Son exécution déclenche (au même instant) I2 et I5 qui sont insérées en tête de liste, – dans l'ordre de leur priorité, c'est-à-dire I2 puis I5. Au point suivant, la liste des règles à traiter est donc [I2,I5,I6].

L'ordre d'exécution final des sept règles est représenté dans la figure 4.11.

4.3.1.4 Exécution par cycles

Principe On traite les règles évaluables et A-exécutables par cycles. La transaction déclenchante est referencée comme le cycle 0. Une règle déclenchée dans un cycle n sera traitée dans le cycle n+1. On traite d'abord toutes les règles du cycle n avant de considérer celles du cycle n+1. A l'intérieur d'un cycle, les règles sont traitées dans l'ordre de leur priorité.

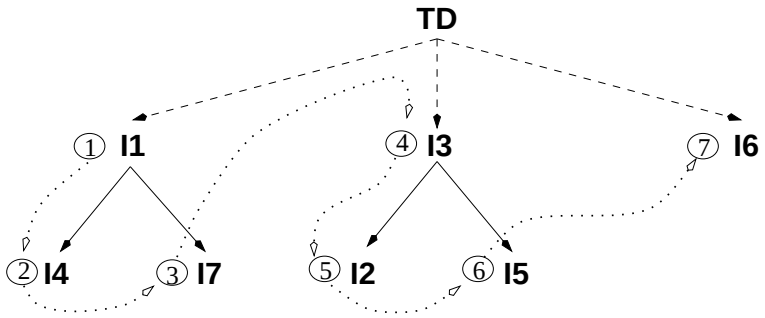

FIG. 4.11 – *Ordre d'exécution de règles en profondeur*

Pour ce faire, on utilise l'algorithme suivant :

1. construire la liste L de listes des règles à traiter, il y a une liste de règles par cycle - les listes de règles sont classées dans L par numéros de cycles croissants, à l'intérieur d'une liste de règles, les règles sont classées par ordre de priorités décroissantes ;
2. traiter (évaluation et/ou A-exécution) la règle R qui est en tête de la liste de règles l_n correspondant à un cycle n qui est la première liste de règles non vide[9] ; et insérer selon l'ordre décroissant de leur priorité les éventuelles règles déclenchées par l'exécution de R dans la liste l_{n+1} correspondant au cycle n+1 (si cette liste n'existe pas, elle est créée) ;
3. supprimer R de la liste de règles l_n ; si l_n est vide alors la supprimer de L ;
4. si la liste de listes de règles L est non vide alors retourner en 2.

Exemple Au point 2 (cf figure 4.12), la liste de listes de règles à traiter est [[I3,I6]^1,[I4,I7]^2] : I3 et I6 sont en attente de traitement dans le cycle 1, I4 et I7 dans le cycle 2. La règle I3 est sélectionnée. Son exécution déclenche I2 et I5 qui sont insérées dans la liste correspondant au cycle 2 (dans laquelle se trouvent déjà I4 et I7).

L'ordre d'exécution final des sept règles ainsi que les cycles sont représentés dans la figure 4.13.

9. Les règles des cycles 0 à n-1 ont déjà été traitées.

Point	État de la liste avant traitement	Traitement d'une règle	État de la liste après traitement
1	[[I1,I3,I6]^1]	I1 $\rightsquigarrow$ {I4,I7}	[[I3,I6]^1,[I4,I7]^2]
2	[[I3,I6]^1,[I4,I7]^2]	I3 $\rightsquigarrow$ {I2,I5}	[[I6]^1,[I2,I4,I5,I7]^2]
3	[[I6]^1,[I2,I4,I5,I7]^2]	I6	[[]^1,[I2,I4,I5,I7]^2]
4	[[]^1,[I2,I4,I5,I7]^2]	I2	[[]^1,[I4,I5,I7]^2]
5	[[]^1,[I4,I5,I7]^2]	I4	[[]^1,[I5,I7]^2]
6	[[]^1,[I5,I7]^2]	I5	[[]^1,[I7]^2]
7	[[]^1,[I7]^2]	I7	[[]^1,[]^2]

FIG. 4.12 – *Une exécution de règles par cycles*

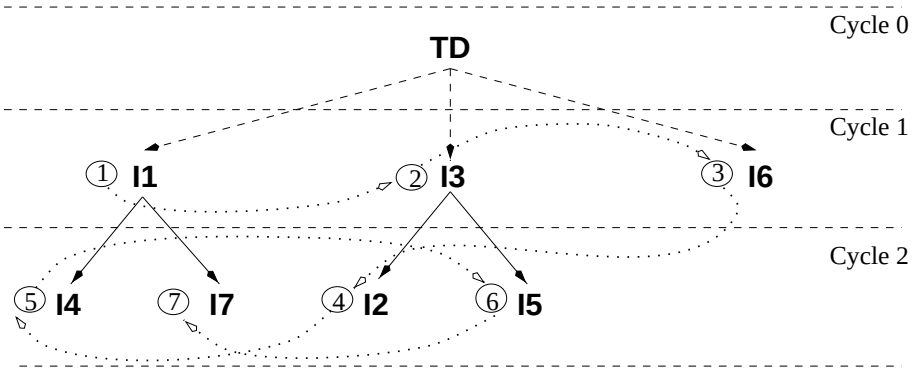

FIG. 4.13 – *Ordre d'exécution intra-module par cycles*

4.3.2 Algorithme générique

Si l'on fait abstraction de la structure de la liste L des règles à traiter, les algorithmes de la section précédente, qui décrivent les différentes stratégies se distinguent principalement par la manière dont sont insérées dans cette liste les règles déclenchées au cours d'un cascade de règles. Ceci nous permet de donner l'algorithme générique suivant qui décrit le processus d'exécution d'un module de règles à un point d'exécution de la transaction déclenchante. Dans cet algorithme, il faut bien garder en mémoire que la procédure **d'insertion** sera différente pour chacune des quatre stratégies.

1. construire la liste L des règles à traiter
2. traiter la règle R qui est en tête de L et *insérer* dans L les règles déclenchées par l'exécution de R

3. supprimer R de L
4. si L est non vide alors retouner en 2.

4.3.3 Interactions entre les couches I et II

Dans la section 4.3.1, nous avons décrit les quatre stratégies d'exécution d'un module de règles et donné pour chacune un algorithme général. Pour cela, nous avons utilisé un exemple illustratif pour lequel nous avons fait un certain nombre d'hypothèses : règles immédiates, sémantique d'instance, consommation des événements, non préemption de l'exécution. Ceci nous a permis ensuite de définir, dans la section 4.3.2 un algorithme générique qui décrit de manière simplifiée le processus d'exécution d'un module de règles commun aux quatre stratégies proposées. Dans cette section, en levant une à une ces hypothèses, nous montrons comment modifier l'algorithme afin de prendre en compte les paramètres de l'exécution d'une règle.

4.3.3.1 Prise en compte du mode d'exécution

La prise en compte du mode d'exécution, c'est-à-dire la présence de règles immédiates, retardées et différées implique deux processus très distincts pour l'exécution d'un module de règles :

1. un premier processus, **au cours de la transaction déclenchante**, pour :
   - évaluer et (A-)exécuter les règles immédiates ;
   - évaluer les règles retardées et les *mémoriser* afin de les exécuter en fin de transaction ;
   - mémoriser les règles différées afin de les évaluer et de les exécuter en fin de transaction ;
2. un second processus, **à la validation de la transaction déclenchante**, pour évaluer et/ou exécuter les règles préalablement déclenchées ou évaluées, ainsi que les règles déclenchées par l'exécution des premières règles.

Pour cela, dans les algorithmes de la section précédente, on manipule non plus une liste de règles mais deux :

- la liste L qui contient les règles à traiter (évaluation et/ou pour A-exécution) à un point d'exécution de la transaction déclenchante ;
- la liste L' qui contient les règles à traiter en fin de transaction.

On redéfinit ensuite l'algorithme générique d'exécution d'un module de règles de la manière suivante :

Si *on est à un point d'exécution de la transaction déclenchante*
alors

1. construire la liste L des règles immédiates et retardées à traiter et insérer dans la liste L' les règles différées à traiter en fin de transaction
2. traiter la règle R qui est en tête de L :
   si R est immédiate alors l'évaluer et l'A-exécuter
   si R est retardée alors l'évaluer et l'insérer dans L'
   si R est différée alors l'insérer dans L'.
   Au cours de cette phase : les règles immédiates et retardées déclenchées par l'exécution de R sont insérées dans L, les règles différées dans L'
3. supprimer R de L
4. si L est non vide alors retouner en 2.

sinon/* *on est en fin de transaction* */
1. insérer les éventuelles règles déclenchées par la fin de transaction dans L'
2. traiter pour évaluation puis A-exécution la règle R qui est en tête de L'
   Au cours de cette phase, les règles déclenchées par l'exécution de R sont insérées dans L'
3. supprimer R de L'
4. si L' est non vide alors retouner en 2 (du sinon).

finsi

4.3.3.2 Prise en compte du mode de traitement des événements

La prise en compte du mode de traitement des événements d'une règle R par les différentes stratégies proposées pour l'exécution d'un module de règles, mérite une attention particulière dans le cas où :

1° R est déclenchée plusieurs fois et doit être exécutée au cours d'une cascade de règles, – on parle de *déclenchement multiple* d'une règle ;

2° la valeur du paramètre mode de traitement des événements de R est ensembliste ;

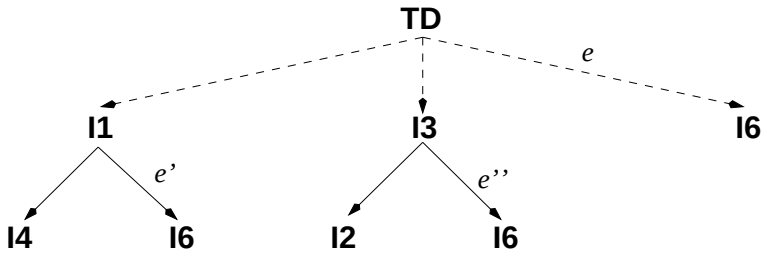

FIG. 4.14 – *Déclenchement multiple d'une règle*

Considérons par exemple, l'arbre de déclenchement de la figure 4.14 et adoptons les même hypothèses que pour l'exemple de la figure 4.5. Supposons, en outre, que la règle I6 est ensembliste. Cette règle I6 est déclenchée une première fois par un ensemble d'événements e produits par la transaction déclenchante TD, puis par un ensemble d'événements e' produits par l'exécution de I1 et par un ensemble d'événements e'' produits par l'exécution de I3.

Les questions auxquelles nous devons répondre maintenant sont : combien de fois la règle va-t-elle être considérée, à quel moment (à quelle place dans le plan d'exécution global), et pour traiter quel ensemble d'événements ?

Exécution par priorités On exécute la règle une seule fois pour l'ensemble des événements déclenchants.
Dans l'algorithme décrivant l'exécution par priorités, cela signifie qu'il peut y avoir au plus une instance d'une règle donnée dans les listes L et L'.
Dans notre exemple, le plan d'exécution global est donc :
$I1 \mapsto I3 \mapsto I2 \mapsto I4 \mapsto I6(e \cup e' \cup e'')$ avec I6 traitant l'ensemble des événements $\{e\} \cup \{e'\} \cup \{e''\}$.

Exécution en pipelines On exécute la règle autant de fois qu'il y a eu de déclenchements. Ceci est justifié par le fait que dans la stratégie d'exécution en pipelines, la priorité est donnée à l'ordre de déclenchement des règles. Pour éviter tout malentendu, signalons que nous "n'écrasons" pas le paramètre mode de traitement des événements. En effet, suite à l'exécution de I3 par exemple, qui génére l'ensemble d'événements e', I6 sera bien exécutée une seule fois pour cet ensemble e' et conserve donc bien la sémantique ensembliste. Dans cette section, nous traitons uniquement des choix que nous faisons lorsque la sémantique la plus générale du *mode de traitement des événements* donnée dans la section 3.1.1.1 est justement trop générale pour rendre compte des situations très particulières qui peuvent se produire lors de l'exécution d'un module de règles.

Dans l'algorithme correspondant à l'exécution en pipelines, il peut donc y avoir plusieurs instances d'une règle donnée (pour traiter des ensembles d'événements différents) dans les listes L et L'.
Dans l'exemple, le plan d'exécution global est :
I1 $\mapsto$ I3 $\mapsto$ I6(e) $\mapsto$ I4 $\mapsto$ I6(e') $\mapsto$ I2 $\mapsto$ I6(e'').

Exécution en profondeur Comme pour l'exécution en pipelines, on exécute la règle autant de fois qu'il y a eu de déclenchements, – pour la même raison.
Dans l'exemple, le plan d'exécution global est donc :
I1 $\mapsto$ I4 $\mapsto$ I6(e') $\mapsto$ I3 $\mapsto$ I2 $\mapsto$ I6(e'') $\mapsto$ I6(e).

Exécution par cycles On exécute la règle une seule fois dans un cycle donné puisque dans cette stratégie, c'est la notion de cycle qui prime.
Dans l'algorithme correspondant à l'exécution par cycles, il peut donc y avoir plusieurs instances d'une règle donnée dans les listes L et L' mais une seule dans chacune des sous-listes correspondant à un cycle.
Dans l'exemple, le plan d'exécution global est:
I1 $\mapsto$ I3 $\mapsto$ I6(e) $\mapsto$ I2 $\mapsto$ I2 $\mapsto$ I6($e' \cup e''$).

4.3.3.3 Prise en compte du mode de consommation des événements

Le mode de consommation des événements spécifie, si lors de l'exécution d'une règle R, le ou les événements déclenchants sont consommés ou préservés pour les exécutions ultérieures de R (cf 3.1.1.2). Ce paramètre est pris en compte à l'étape 3 des algorithmes donnés précedemment :

- si la règle R consomme les événements, elle est supprimée de la liste des règles à considérer après son exécution ;
- si la règle R préserve les événements, elle n'est pas supprimée de la liste mais simplement "marquée" pour indiquer qu'elle n'est plus déclenchée. Ceci permet de préserver les événements déclenchants.

4.3.3.4 Prise en compte du mode de préemption

Le mode de préemption d'une règle R, puisqu'il est susceptible d'entrainer une exécution récursive des règles déclenchées par R influe sur le plan d'exécution global d'un modules de règles – et ce, quelle que soit la stratégie d'exécution adoptée.

Considérons l'arbre de déclenchement de la figure 4.15. Toutes les règles sont immédiates. Les règles sont exécutées selon la stratégie d'exécution par priorités et sans préemption.

Le plan d'exécution global selon cette stratégie est : I2 $\mapsto$ I3 $\mapsto$ I4 $\mapsto$ I5 $\mapsto$ I6.

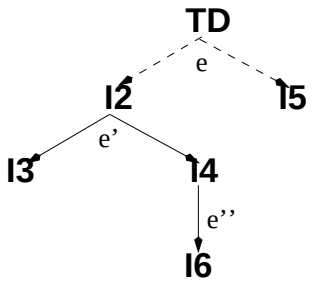

FIG. 4.15 – *Un arbre de déclenchement*

Considérons maintenant que la règle I4 a le droit de préemption, c'est-à-dire que son exécution peut interrompre l'exécution de la règle (ou transaction déclenchante) qui l'a déclenchée. L'exécution du module de règles est alors celle donnée par la figure 4.16. Dans cette figure, les cercles vides représentent à la fois les points d'exécution (il y en a cinq !) et les règles considérées à ces points.

Le plan d'exécution global est alors: $I2 \mapsto I4 \mapsto I3 \mapsto I5 \mapsto I6$.

Dans l'algorithme générique donné dans la section précédente, le mode de préemption est pris en compte dans la phase 2 qui prend maintenant en compte cette considération récursive :

1. construire la liste L des règles à traiter
2. considérer la règle R qui est en tête de L ; insérer dans L et considérer récursivement les règles déclenchées par l'exécution de R qui sont évaluables ou A-exécutable et qui ont le droit de préemption ; insérer dans L les autres règles déclenchées par l'exécution de R
3. supprimer R de L
4. si L est non vide alors retourner en 2.

4.4 Couche III : Modèle d'exécution inter-modules de règles

La couche III du modèle gère l'*exécution inter-modules* de règles, c'est-à-dire l'exécution globale de différents modules de règles. Cette couche n'est pas réellement para-

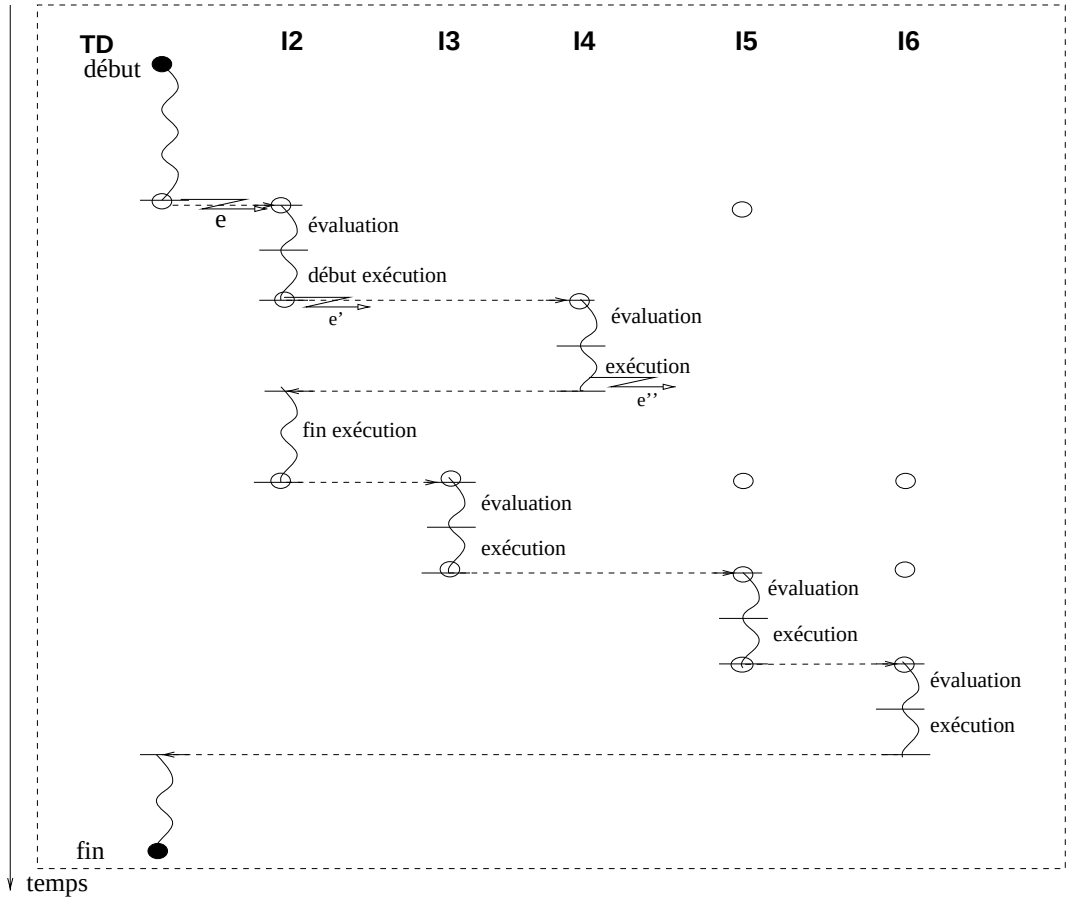

FIG. 4.16 – *Exécution avec préemption d'une règle*

métrique au sens où nous proposons un seul mécanisme[10], très simple, qui est décrit ci-dessous. En réalité l'exécution des modules par priorité est la seule stratégie envisageable. En effet, si l'on considère la construction des quatre stratégies d'exécution de la couche II (cf. 4.3), on constate que les notion d'*instant de déclenchement* et de *cycles d'exécution* ont un sens en ce qui concerne les règles mais non les modules de règles.

4.4.1 Stratégie d'exécution

Principe : Soit un ensemble de modules de règles Ms, $\mathrm{Ms} = \{\mathrm{M}_i\}$ tel que $1 \leq i \leq n$ et dans lequel chaque M_i est un module de règles complètement ordonné par des priorités entre règles et dont les règles sont considérées selon l'une des quatre

10. On peut considérer qu'il s'agit d'une constante, c'est-à-dire un paramètre fixé.

stratégies d'exécution proposées. Ms est également complètement ordonné par des priorités entre modules de règles.

À chaque point d'exécution, on détermine une règle candidate par module de règles $M_i \in$ Ms, – le choix de cette règle est déterminé par la stratégie d'exécution de chacun des modules M_i – on traite alors celle qui appartient au module le plus prioritaire.

La section 4.6.1 donne un exemple d'exécution inter-modules avec deux modules de règles.

4.4.2 Interactions entre les couches II et III

Avec Fl'are, il est possible d'avoir simultanément plusieurs modules de règles dont chacun s'exécute selon sa propre stratégie (parmi les quatre proposées). Les exécutions intra-modules ne sont toutefois pas complètement indépendantes les unes des autres, en particulier lorsque (au moins) l'une d'elle est effectuée par cycles.

Considérons l'exemple suivant (un exemple plus consistant est donné dans la section 4.6.1.). Soit A et B deux modules de règles. A1 et A2 sont des règles de A. B1 est une règle de B. Le module A est exécuté par cycles. Le module B est exécuté selon une stratégie quelconque parmi les quatre stratégies proposées. Supposons maintenant que l'exécution de A1 dans le cycle 1 déclenche B1 dont l'exécution déclenche, à son tour, A2. Celle-ci sera traitée dans le cycle 2 de l'exécution du module A car on considère qu'elle est déclenchée par A1.

Dans une exécution inter-modules de règles, l'exécution d'une règle dans un certain cycle est déterminée en fonction du module auquel appartient la règle et donc *transitivement* par rapport à l'arbre de déclenchement global (inter-modules).

4.5 Concepts exclus du modèle

Dans la section précédente, nous avons expliqué pourquoi et comment les dimensions et valeurs de la taxonomie présentée dans le chapitre 3 étaient transformées en constantes et paramètres avec leurs valeurs possibles. Un lecteur appliqué remarquera cependant que certaines dimensions ou certaines valeurs présentes dans la taxonomie n'apparaissent dans le modèle. Cette section explique pourquoi ces concepts ont été écartés du modèle.

4.5.1 Couche I : exécution différée à un point de contrôle

Dans A-RDL, Starburst et Peplom[ad], la valeur *différé* de la dimension *mode de déclenchement* d'une règle (cf. section 3.3.1.1) – dimension qui correspond au paramètre *mode d'exécution* dans notre modèle (cf. section 4.2.2) – peut prendre, en réalité deux valeurs : *différé à la validation* (DV) et *différé à un point de contrôle* (DPC).

Pour simplifier notre propos, nous considérons uniquement ici un couplage Condition -Action *immédiat* – cas qui correspond à la plupart des systèmes étudiés. Avec ce couplage et dans le cas où la considération d'une règle est DV, une règle déclenchée au cours d'une transaction est évaluée et exécutée à la validation de cette transaction. Dans le cas où la considération est DPC, une règle déclenchée au cours d'une transaction est évaluée et exécutée à un point ultérieur de la transaction appelé *point de contrôle*.

La spécification des points de contrôle d'une part, et la prise en compte de ces points d'autre part, ne sont pas orthogonales. En effet, l'existence de règles DPC pose les questions suivantes :

1° Comment les points de contrôle sont-ils spécifiés ? Sont-ils explicites dans le code des applications et/ou définis par le système ?

2° Peut-on spécifier des points de contrôle dans le code de la partie Action d'une règle ?

3° Existe-t-il différents types de points de contrôle ?

4° Le fait qu'une règle soit DPC doit-il apparaître explicitement dans la spécification de cette règle ?

5° La validation de la transaction déclenchante est-elle toujours considérée comme un point de contrôle ?

Dans les trois systèmes étudiés, les points de contrôle sont explicites dans le code des applications mais on ne peut pas spécifier de point de contrôle dans la partie Action d'une règle. Dans Starburst, il existe trois types de points de contrôle spécifiés par les instructions : `CheckRule` *rulename* qui provoque la considération de la règle *rulename*, `CheckRuleset rulesetspecification` qui provoque la considération des règles de l'ensemble défini par `rulesetspecifica- tion` et `CheckAllRules` qui provoque la considération de toutes les règles définies dans le schéma considéré. Dans Peplom[a,d], le fait qu'une règle soit DPC apparaît explicitement dans la spécification de cette règle, la validation est un PC implicite. Dans Starburst et A-RDL, le fait qu'une règle soit DPC n'apparaît pas explicitement dans la spécification de cette règle, la validation est égalemment un point de contrôle implicite. En clair, dans Peplom[a,d], il est possible de spécifier de *vraies* règles DV, c'est-à-dire des règles considérées uni-

quement à la validation de la transaction déclenchante alors que cela est impossible dans Starburst et A-RDL. Inversement, il n'est possible dans aucun de ces trois systèmes de spécifier une règle qui ne soit considérée qu'aux points de contrôle et non à la validation de la transaction déclenchante.

Les points de contrôle ont été introduits dans ces trois sytèmes principalement (a) comme une solution intermédiaire entre couplages *immédiat* et *différé*, afin de pouvoir différer momentanément le traitement des règles ("l'application est en train de faire quelque chose de plus important") mais sans pour autant attendre la validation de la transaction. Ceci peut être intéressant dans le cadre de *transactions longues*, (b) afin d'éviter des considérations inutiles de règles (par exemple, des règles implantant des contraintes d'intégrité que l'on est certain de ne pas violer).

Les points de contrôle, en particulier les trois types de points de contrôle de Starburst semblent bien correspondre aux trois couches de Fl'are, nous avons cependant choisi de ne pas utiliser les PC pour les raisons suivantes :

1. le fait de spécifier explicitement les points de contrôle **dans les applications** nous semble un *dangereux interventionnisme* des applications dans le modèle d'exécution des règles. Il nous semble important qu'applications et systèmes de règles soient le plus indépendants possibles ; et en tout état de cause, que ce soit le système de règles *qui garde la main* sur l'exécution des règles ;
2. l'utilisation des points de contrôle et des règles DPC nous semblent aléatoire et dangereuse : une règle déclenchée peut ne pas être considérée, une règle peut ne pas être considérée comme celui (ou celle) qui l'a spécifiée l'aurait désiré, etc.
3. le fait de devoir spécifier explicitement les points de contrôle dans les applications va à l'encontre du cadre de notre étude : cadre qui vise à faire les hypothèses minimales sur ce qui n'est pas purement du ressort du modèle d'exécution ;
4. enfin, les modèles d'exécution à un point de contrôle et à la validation sont équivalents, les points de contrôle n'ont donc pas d'intérêt particulier dans notre étude des modèles d'exécution pour SGBD actifs.

4.5.2 Couche II : choix d'une règle

Pour Fl'are, nous proposons quatre stratégies de considération d'un module de règles. Dans notre taxonomie (cf. chap. 3), nous avons mis à jour une autre stratégie : le *choix d'une règle*. Deux systèmes, Postgres et EXACT propose cette stratégie qui consiste à n'exécuter qu'une seule règle parmi un ensemble de règles déclenchées. Le choix de cette règle est effectué selon les priorités entre règles.

Cette stratégie, n'a de sens, à vrai dire, que dans le cas de règles déclenchées

par le même événement ; ce qui est le cas dans Postgres dans lequel il n'y a que des règles immédiates et des événements primitifs (toutes les règles déclenchées à un point d'exécution donné sont effectivement déclenchées par le même événement) ; ce qui n'est pas le cas d'EXACT pour lequel ce point n'est pas abordé (ni le cas des événements simultanés d'ailleurs).

Dans un cadre plus général, comme celui de Fl'are par exemple, du fait des événements simultanés et des différents modes d'exécution (immédiat, retardé et différé), cette stratégie est difficilement applicable. Considérons, par exemple, qu'à un point d'exécution de la transaction, l'ensemble des règles à traiter soit : `[R3,I4,I6]` avec `R3` la règle la plus prioritaire, `I6` la moins prioritaire, `R3` retardée, `I4` et `I6` immédiates. `R3` va être *choisie*, `I4` et `I6` vont être supprimées de l'ensemble sans être traitées. Considérons ensuite que les règles `I1`, `I7` et `D2` aient été déclenchées après le traitement de `R3`. À la validation de la transaction déclenchante, l'ensemble des règles à traiter est donc : `[I1,D2,R3,I7]`. `I1` va être *choisie*, `D2,R3` et `I7` vont être supprimées de l'ensemble sans être exécutées. On remarque que `R3` a été évaluée mais non A-exécutée, ce qui ne correspond plus à la sémantique la plus générale d'une règle active. Si l'on tente de contourner le problème en ajoutant un nouveau principe : "toute règle évaluée (à `vrai`) doit être exécutée, c'est cette fois la stratégie d'exécution *par choix* qui n'est plus respectée.

C'est donc une fois encore pour assurer la cohérence du modèle paramétrique que nous avons choisi de ne pas proposer cette stratégie.

4.6 Instanciation et couverture de Fl'are

Dans cette section, nous montrons comment utiliser notre modèle paramétrique, c'est-à-dire comment *l'instancier* pour simuler le modèle d'exécution du système NAOS puis celui d'autres systèmes existants. Dans notre exemple détaillé de l'utilisation de Fl'are (section 4.6.1), nous avons choisi le système NAOS car celui-ci se prête très bien à cet exercice. En effet, dans NAOS, on trouve deux types de règles dont les modèles d'exécution sont très distincts et on trouve de plus deux stratégies d'exécution intra-modules parmi les quatre que propose Fl'are.

4.6.1 NAOS : une instantiation du modèle paramétrique Fl'are

Considérons deux modules de règles I = { I1, I2, ... , In} et D = { D1, D2, ... , Dm} tels que :

1. toutes les règles de I utilisent un même modèle d'exécution qui est donné par la table 4.6.

	paramètre	valeurs
1	mode de traitement des événements	par instance
2	mode de consommation des événements	consommation
3	effet net	oui
4	mode de préemption	préemption
5	mode d'exécution	immédiat

TAB. 4.6 – *Modèle d'exécution des règles de* I

2. toutes les règles de D utilisent un autre modèle d'exécution qui est donné par la table 4.7.

	paramètre	valeurs
1	mode de traitement des événements	ensembliste
2	mode de consommation des événements	consommation
3	effet net	oui
4	mode de préemption	non préemption
5	mode d'exécution	différé

TAB. 4.7 – *Modèle d'exécution des règles de* D

3. les règles du module I sont exécutées en profondeur ;
4. les règles du module D sont exécutées par cycles ;
5. le module I est prioritaire par rapport au module D.

Appelons E l'ensemble tel que E = I ∪ D.

Dans notre exemple considérons de plus que dans chaque module, les priorités des règles sont représentées par les numéros des règles ; ainsi dans I (resp. D), la règle la plus prioritaire est I1 (resp. D1), la moins prioritaire est In (resp. Dm).

Considérons enfin la situation représentée par l'arbre de déclenchement de la figure 4.17. En fin de transaction, les deux règles D1 et D2 ont été déclenchées. Leur exécution déclenche en cascade les règles I1, I2, D4, I3, D5, I4 et D3.

La figure 4.18 décrit le processus d'exécution de l'ensemble E.

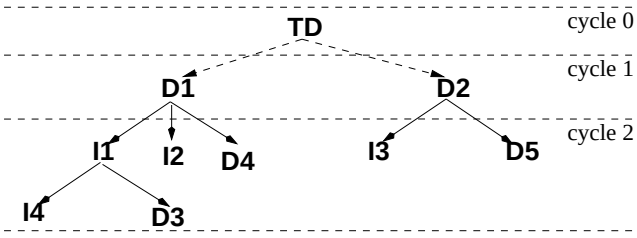

FIG. 4.17 – *Un arbre de déclenchement global de règles NAOS*

Point	Règles		à traiter	Traitement
1.	$[\ []^I$	,	$[[D1,D2]_1]^D\ ]$	D1 $\rightsquigarrow$ {I1,I2,D4}
2.	$[\ [I1,I2]^I$	,	$[[D2]_1,[D4]_2]^D\ ]$	I1 $\rightsquigarrow$ {I4,D3}
3.	$[\ [I4,I2]^I$	,	$[[D2]_1,[D3,D4]_2]^D\ ]$	I4
4.	$[\ [I2]^I$	,	$[[D2]_1,[D3,D4]_2]^D\ ]$	I2
5.	$[\ []^I$	,	$[[D2]_1,[D3,D4]_2]^D\ ]$	D2 $\rightsquigarrow$ {I3,I5}
6.	$[\ [I3]^I$	,	$[[]_1,[D3,D4,D5]_2]^D\ ]$	I3
7.	$[\ []^I$	,	$[[]_1,[D3,D4,D5]_2]^D\ ]$	D3
8.	$[\ []^I$	,	$[[]_1,[D4,D5]_2]^D\ ]$	D4
9.	$[\ []^I$	,	$[[]_1,[D5]_2]^D\ ]$	D5
10.	$[\ []^I$	,	$[[]_1,[]_2]^D\ ]$	

FIG. 4.18 – *Processus d'exécution inter-modules des règles NAOS*

Au point d'exécution 1, qui correspond à la validation de la transaction déclenchante, les deux règles D1 et D2 doivent être traitées. On sélectionne D1 qui est la plus prioritaire. Son exécution déclenche I1, I2 et D4. Au point d'exécution suivant, les règles à traiter sont I1 pour le module I et D2, – qui est traitée avant D4 car les règles de D sont exécutées par cycles – pour le module D. On sélectionne I1 car I est prioritaire sur D.

L'ordre global d'exécution des règles de E est représenté par la figure 4.19. On

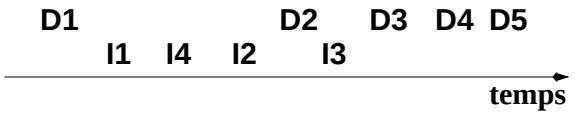

FIG. 4.19 – *Exécution globale des règles NAOS*

note que les règles de I ont bien été exécutées en profondeur tandis que les règles de

D ont été exécutées par cycles.

On peut dire que le modèle d'exécution de E **représente une instanciation du modèle paramétrique qui simule le modèle d'exécution de NAOS**.

Remarque nous avons déjà signalé que, dans un processus global d'exécution de plusieurs modules de règles (cf. 4.18), le traitement d'une règle dans un certain cycle est déterminée en fonction du module auquel appartient la règle et donc *transitivement* par rapport à l'arbre de déclenchement global (cf. 4.17).

Dans notre exemple, les règles D3, D4 et D5 sont exécutées dans le même cycle 2 car toutes trois déclenchées *transitivement* par les les règles D1 et D2 exécutées dans le cycle 1.
□

4.6.2 Couverture des systèmes existants par Fl'are

Dans la section précédente, nous avons montré comment *instancier* Fl'are afin de simuler le modèle d'exécution de NAOS. Il est possible, de manière presque aussi immédiate, d'utiliser Fl'are pour simuler le modèle d'exécution de la plupart des systèmes existants pour la plupart de leurs fonctionnalités – bien que cela ne soit pas sa vocation première !

Starburst Il suffit de créer un seul module de règles dont toutes les règles utilisent le modèle d'exécution donné par la table 4.8. Le module est considéré selon la stratégie

	paramètre	valeurs
1	mode de traitement des événements	ensembliste
2	mode de consommation des événements	consommation
3	effet net	oui
4	mode de préemption	non préemption
5	mode d'exécution	différé

TAB. 4.8 – *Règles de Starburst*

par priorités. Les règles *différées à un point de contrôle* ne sont pas directement supportées par Fl'are mais peuvent être simulées, avec des *événements utilisateurs* par exemple.

Ariel Il suffit de créer un seul module de règles. Les règles *différées*, identiques à celles de Starburst, utilisent le modèle d'exécution donné par la table 4.8. Les règles

immédiates utilisent le modèle d'exécution donné par la table 4.9. Le module est

	paramètre	valeurs
1	mode de traitement des événements	ensembliste
2	mode de consommation des événements	consommation
3	effet net	oui
4	mode de préemption	non préemption
5	mode d'exécution	immédiat

TAB. 4.9 – *Règles immédiates de Ariel*

considéré selon la stratégie *par priorités*.

Chimera Les règles *immédiates* et *différées* de Chimera sont identiques, respectivement, à celles d'Ariel (cf. table 4.9) et Starburst (cf. table 4.8) si ce n'est que l'effet net n'est pas pris systématiquement en compte. Les règles sont exécutées *par priorités*.

A-RDL La notion de *module de règles* existe dans A-RDL de même qu'un ordre sur ces modules. Les règles peuvent être immédiates (cf. table 4.9) ou différées (cf. table 4.8) . Comme pour Starburst, Les règles *différées à un point de contrôle* peuvent facilement être simulées. La consommation globale des événements par une règle, possible dans A-RDL, n'est pas simulable par Fl'are.

Sentinel La notion de module de règles existe dans Sentinel sous le terme de *classe de priorité*, il y a évidemment un ordre total sur les classes de priorités. Les règles ont une sémantique d'instance, consomment les événements[11], ne prennent pas en compte l'effet net, n'ont pas le droit de préemption et sont immédiates ou différées[12]. Le modèle d'exécution des règles immédiates est donné par la table 4.10. Le modèle d'exécution des règles différées est donné par la table 4.11.

Les ensembles de règles sont considérés *par priorités*.

EXACT La notion d'*ensemble de règles* existe également dans EXACT par spécialisation et extension de classes. Les règles ont une sémantique d'instance, consomment les événements, ne prennent pas en compte l'effet net, n'ont pas le droit de préemption et sont immédiates ou différées. Elles sont considérées *par défaut* selon la stratégie

11. SNOOP, le modèle d'événements de Sentinel, est très développé et permet certainement de simuler la consommation/préservation ou le traitement de l'effet net.

12. L'utilisation d'un modèle de transactions non classique est abordé dans les publications relatives à Sentinel mais cet espect n'est pas très développé et n'est pas pris en compte dans l'implantation.

	paramètre	valeurs
1	mode de traitement des événements	instance
2	mode de consommation des événements	consommation
3	effet net	non
4	mode de préemption	non préemption
5	mode d'exécution	immédiat

TAB. 4.10 – *Règles immédiates de Sentinel*

	paramètre	valeurs
1	mode de traitement des événements	instance
2	mode de consommation des événements	consommation
3	effet net	non
4	mode de préemption	non préemption
5	mode d'exécution	différé

TAB. 4.11 – *Règles différées de Sentinel*

par priorités mais peuvent être considérées selon d'autres stratégies. Fl'are, qui n'est pas un modèle extensible, propose trois autres stratégies parmi celles possibles.

HiPAC, Beeri et Milo, Ode, SAMOS, REACH, TriGS Tous ces systèmes s'appuyent sur des modèles de transactions évolués. Les efforts ont surtout portés sur l'utilisation de ces modèles de transactions pour l'exécution des règles actives et non sur le modèle d'exécution des règles lui-même au sens ou nous l'entendons. Fl'are part de l'hypothèse d'un modèle de transactions classique et ne peut évidemment pas simuler ces aspects liés au modèle de transactions.

Toutefois, si l'on met de côté l'utilisation du support transactionnel, ces systèmes sont assez simples sur tous les autres points du modèle d'exécution abordés dans notre taxonomie et peuvent être couverts par Fl'are : les règles ont une sémantique d'instance, elles consomment les événements, Ode prend en compte l'effet net, dans HiPAC, REACH et SAMOS elles peuvent avoir le droit de préemption et peuvent être immédiates ou différées.

Postgres Dans Postgres, toutes les règles ont une sémantique d'instance, consomment les événements, ne prennent pas en compte l'effet net, n'ont pas le droit de préemption et sont immédiates. En fait, Postgres est le système dont le modèle d'exécution d'une règle est le plus simple. Par contre, comme nous l'avons vu dans la section 4.5, les règles sont considérées selon la stratégie *par choix* qui n'est pas proposée par

Fl'are. L'annulation de l'opération déclenchante, comme pour NAOS, n'est pas non plus simulable par Fl'are. Paradoxalement, alors que c'est le système qui a le modèle d'exécution le plus simple, Postgres est cependant le système qui peut être le moins facilement simulé par Fl'are !

4.7 Conclusion

Fl'are est un modèle d'exécution pour SGBD actif, flexible, puissant et facile à appréhender. Il permet d'adapter le comportement de modules de règles destinés chacun à une utilisation particulière des règles au sein d'une même application. Il a une structure en trois couches : la couche I gère l'exécution d'une règle, la couche II gère les interactions entre les règles d'un module et la couche III gère les interactions entre les modules.

Fl'are est facile à appréhender car il est basé sur une approche *boîte-à-outils* : le comportement (modèle d'exécution) des règles et des modules de règles est spécifié simplement par l'*instantiation* de paramètres qui peuvent prendre uniquement des valeurs prédéfinies.

Nous avons illustré la puissance de Fl'are en montrant comment il pouvait être instancié afin de *couvrir* la plupart des fonctions de différents systèmes existants. Nous pouvons tenter de citer quelques chiffres. On trouve un seul type de règle dans Postgres et Starburst, deux (ou trois) dans Chimera et NAOS ... quarante huit dans Fl'are ; tous les paramètres qui constituent la couche I du modèle sont indépendants. On trouve une seule stratégie (mais pas toujours la même) d'exécution d'ensembles de règles dans la plupart des systèmes, deux dans NAOS ... quatre dans Fl'are. Les paramètres des couches I et II sont également indépendants alors que ce n'est pas le cas dans NAOS par exemple.

Nous avons décrit séparément les trois couches, les huit constantes et les six paramètres qui constituent le modèle. Nous avons ensuite évoqué les interactions intra puis inter-couches. Cette explication "à plat" n'est pas complètement satisfaisante. Pour "exhiber" complètement Fl'are, il nous faudrait envisager tous les cas d'exécutions possibles en faisant varier le nombre et la composition des modules, les priorités entre modules, le nombre et le comportement de chaque règle, les priorités entre règles, etc. Ceci n'est évidemment pas raisonnablement envisageable et montre, selon nous, le besoin d'un formalisme pour exprimer clairement la sémantique du modèle. Aussi, dans le chapitre suivant, nous proposons une sémantique dénotationnelle de Fl'are.

Enfin, Fl'are est portable (générique) puisqu'indépendant, autant que faire se

peut, des modèles de données et des modèles de règles. Ceci facilite son implantation dans laquelle Fl'are peut être vu comme un *moteur d'exécution* pour SGBD actif. L'implantation de Fl'are fait l'objet du chapitre 6.

Chapitre 5

Sémantique dénotationnelle de Fl'are

> ***Ésotérique :*** *adj. Parfaitement occulte et particulièrement abscons. Les anciennes philosophies étaient de deux sortes,* - exotériques*, que les philosophes eux-mêmes ne comprenaient qu'à moité, et* ésotériques*, que personne n'a jamais comprises. Ce sont ces dernières qui ont le plus profondément marqué la pensée moderne, et qui jouissent encore de nos jours d'un grand crédit.*
> ***Ambrose Bierce - "Le dictionnaire du Diable"***

Des progrès considérables ont été faits au cours des vingt dernières années dans le domaine des langages de programmation grâce à de nombreux travaux visant à définir un cadre formel qui permet de raisonner sur la sémantique de ces langages.

Ces travaux – [Sto77, Ten91, Gun92, Win93] et bien d'autres – montrent que la sémantique formelle d'un langage est utile, à la fois pour la personne qui va programmer avec ce langage, et pour celle qui le spécifie et qui en implante un compilateur. En effet, la sémantique formelle peut être vue comme un outil de documentation qui fournit **une description complète, précise et non ambigüe** d'un langage. Cette description permet de répondre à des questions subtiles à propos de ce langage, évitant ainsi omissions et contradictions. Elle constitue également une base précise pour l'implantation qui garantit que celle-ci correspond aux vœux de celui (ou celle) qui spécifie le langage et que ce dernier sera implanté de la même manière sur différents

systèmes ou machines. Enfin, la sémantique peut être vue comme un outil de spécification et de conception qui peut **suggérer des solutions élégantes et efficaces pour l'implantation**.

Nous défendons l'idée que le domaine des bases de données actives – et plus particulièrement la spécification des modèles d'exécution des SGBD actifs – peut bénéficier des mêmes apports de la sémantique formelle.

Les SGBD actifs, comme les systèmes à objets, ont été généralement développés de manière *ad hoc*, sans formulation précise et surtout complète de leur modèle d'exécution ; principalement parce que leur sémantique est donnée de façon informelle, en langue naturelle. Une première conséquence de cette expression informelle de la sémantique des modèles d'exécution est qu'il est très difficile – comme nous l'avons vu dans le chapitre 3 – de comprendre, de prévoir et de comparer le comportement de ces systèmes alors même qu'ils partagent bon nombre de fonctions. Un seconde conséquence est que leur implantation – qui n'est déjà pas triviale intrinsèquement – est rendue encore plus ardue par cette formulation incomplète qui implique des "allers et retours" entre spécification et implantation. Certains points délicats que nous avons traités dans le chapitre 4, comme le déclenchement de plusieurs règles par des événements simultanés ou le déclenchement multiple d'une même règle, sont par exemple, rarement abordés dès le début de l'implantation. La prise en compte tardive de ces phénomènes conduit alors, soit à une remise en cause plus ou moins profonde de la spécification du système, soit à des prises de décisions lors de l'implantation qui peuvent conduire à des contradictions et incohérences entre spécification et implantation.

Afin d'éviter ces écueils, nous proposons dans ce chapitre, un cadre formel pour décrire notre modèle d'exécution paramétrique Fl'are. Le formalisme que nous avons retenu est la *sémantique dénotationnelle*. La section 5.1 précise les concepts et notations propres à ce formalisme. Elle donne également une vue générale de notre démarche. Les sections 5.2 et 5.3 exposent ensuite une sémantique dénotationnelle de Fl'are. Dans la section 5.4, nous survolons les principaux travaux de formalisation des modèles d'exécution des SGBD actifs afin de présenter – dans la section 5.4.6 – les raisons qui nous ont poussé à choisir la sémantique dénotationnelle. Enfin, dans la section 5.5, nous clôturons ce chapitre en présentant quelques résultats obtenus par ce travail de formalisation.

5.1 Préliminaires et notations

5.1.1 Vue générale de la sémantique dénotationnelle de Fl'are

La sémantique dénotationnelle [Ten76, Sto77, Sch88] est un formalisme qui permet de donner un sens *mathématique* (*fonctionnel*) à langage ou à un système. Une *sémantique dénotationelle* est constituée d'un ensemble d'*algèbres* et d'un ensemble de *fonctions de valuation* sur ces algèbres. Une algèbre est définie comme un *domaine sémantique* muni d'un ensemble d'*opérations*. Dans la suite, nous utilisons peu d'opérations et celles que nous utilisons sont très simples ; aussi, nous parlons uniquement de *domaines sémantiques*, ou plus simplement de *domaines* et non d'*algèbres*.

On peut établir un parallèle entre les constituants de notre sémantique dénotationnelle : domaines et fonctions et les constituants primaires des SGBD actifs que nous avons mis à jour dans le chapitre 2 : modèle de connaissances et modèle d'exécution.

Les domaines représentent le modèle de connaissances. Ils expriment les informations dont on dispose sur les états de la base, les opérations qui composent les transactions, les événements et les règles. On peut rapprocher la notion de *domaine* de celle de *type* dans les langages de programmation.

Nous rappelons que, dans Fl'are, les règles sont regroupées en modules. Chaque règle a son propre modèle d'exécution qui est spécifié par l'instanciation des paramètres d'exécution d'une règle. Chaque module est exécuté selon une des quatre stratégies d'exécution proposées. À l'intérieur d'un module, les règles sont ordonnées par des priorités. Les modules sont également ordonnés par des priorités.

On peut parler de deux catégories de domaines : les domaines *statiques*, dont les représentants sont inchangés par les exécutions de règles, il s'agit du *domaine des règles* et du *domaine des modules de règles* ; et les domaines *dynamiques* dont l'état évolue en fonction des exécutions de règles. Il s'agit du *domaine des règles à traiter* et du *domaine des modules de règles à traiter*. Une règle à traiter est une occurrence d'une règle à laquelle sont associé son ou ses événements déclenchants, son instant de déclenchement et son état (déclenché, en attente d'évaluation, en attente d'exécution, etc.).

Les fonctions représentent réellement le modèle d'exécution, c'est-à-dire la mécanique de l'exécution. Donner une sémantique dénotationnelle de Fl'are, c'est définir une fonction *ModelExec* qui prend comme arguments : un état de la base, un ensemble de modules de règles et un ensemble d'événements produits par l'exécution d'une opération d'une transaction ; qui construit les règles et les modules de règles à

traiter ; qui décrit les changements d'états des ces règles et modules dus à l'exécution (en cascade) des règles ; et qui, finalement, retourne l'état de la base qui résulte de ces exécutions.

Évidemment, dans un souci de lisibilité et de modularité, *ModelExec* est définie à partir de fonctions plus élémentaires qui réalisent des tâches précises et distinctes : déclenchement des règles, sélection des modules puis des règles à traiter selon la stratégie de chaque module, évaluation et A-exécution des règles. Ces fonctions vont être elles-mêmes décomposées en fonctions plus élémentaires afin de prendre en compte le mode de traitement des événements, le mode de consommation des événements, le mode de préemption des règles, l'effet net ; de calculer les cycles d'exécution pour les modules qui s'exécutent par cycles, etc.

La sémantique dénotationnelle de Fl'are est ainsi constituée de huit domaines (avec leurs opérations) qui sont introduits dans la section 5.2 et de vingt cinq fonctions de valuations introduites dans la section 5.3.

5.1.2 Notations utilisées

Domaines

Un domaine est un *espace de valeurs*. Un domaine peut avoir une structure différente de celle d'un ensemble, mais les ensembles se révèlent adéquats pour toutes les situations rencontrées dans ce travail.

Nous utiliserons les symboles classiques de la théorie des ensembles :

- $\cup$ pour l'union de deux ensembles ;
- $\cap$ pour l'intersection de deux ensembles ;
- $\times$ pour le produit cartésien de deux ensembles ;
- $-$ pour la différence de deux ensembles ;
- $\subset$ pour l'inclusion d'un ensemble dans un autre ;

Nous utiliserons également la notation $\downarrow i$ pour dénoter la projection sur le $i$ème élément d'un produit cartésien.

Enfin, nous utiliserons les symboles $\bot$ (bottom) et $\top$ (top) de la théorie des domaines pour désigner respectivement *l'indéfini* et *l'erroné* ou *l'inconsistant*.

Fonctions

Pour chaque fonction, nous donnons une description informelle et une description formelle. La description formelle utilise le λ-calcul (lambda-calcul) défini par Church [Hin86].

Nous utilisons le λ-calcul car c'est la notation utilisée traditionnellement dans le cadre de la sémantique dénotationnelle. Cependant, pour la simple compréhension de ce document, il n'est pas nécessaire d'entrer dans une description des fondements mathématiques et de l'utilisation du λ-calcul. Aussi, nous nous bornons ici à donner les éléments indispensables à cette compréhension.

- Le λ-calcul peut être vu comme un langage purement fonctionnel : comme en LISP ou ML, on ne manipule que des fonctions qui sont sans effet de bord, il n'existe pas de constructions itératives.
- La notation λ a été introduite pour pouvoir manipuler aisément des fonctions d'ordres supérieurs. Dans notre cas, il suffit de savoir que la fonction définie par $f(x) = x + 1$ sera notée $f = \lambda x.\, x + 1$.
- Les fonctions d'ordre supérieur ont elles-mêmes été introduites afin de traiter les définitions de fonctions récursives. En effet, celles-ci posent un problème lié à l'unicité de leur(s) solution(s). En λ-calcul, les définitions récursives de fonctions sont interdites. Une fonction récursive f est approximée par une autre fonction qui tend vers le *plus petit point fixe* (pppf) de f, c'est-à-dire la solution de $f(x) = x$. En ce qui nous concerne, nous pouvons très bien considérer l'utilisation des plus petits points fixes comme un artifice d'écriture et voir simplement des appels récursifs de fonctions comme dans l'exemple suivant :

$$\begin{array}{ll} & f(x) = \mathit{si}\ x = 0\ \mathit{alors}\ 1\ \mathit{sinon}\ x * f(x-1) \\ \text{soit} & f = \lambda x.\ \mathit{si}\ x = 0\ \mathit{alors}\ 1\ \mathit{sinon}\ x * f(x-1) \\ \text{ou} & f = G(f)\ \text{ avec } G = \lambda h.\lambda x.\ \mathit{si}\ x = 0\ \mathit{alors}\ 1\ \mathit{sinon}\ x * h(x-1) \end{array}$$

- Nous utilisons l'abréviation *let* définie par : $(\mathit{let}\ x\ =\ \mathit{expr}_1\ \mathit{in}\ \mathit{expr}_2)$ $\equiv\ (\lambda x.\, \mathit{expr}_2)\, \mathit{expr}_1$; qui permet d'utiliser "des variables temporaires".
- Nous utilisons également les constructions *si-alors-sinon* et *choix-entre* (`switch` du langage C).
- Enfin, nous utilisons parfois une notation *à la Prolog* dans laquelle le caractère "_" remplace n'importe quelle valeur du domaine considéré.

5.2 Domaines sémantiques

Domaine des états d'une base de données : Soit EBD le domaine des états d'une base de données. Comme nous l'avons dit dans la section 4.1.2.2, le modèle de données du SGBD sous-jacent nous est indifférent et nous ne nous plaçons pas dans

un modèle ou un autre. Aussi, nous ne décrivons pas de manière formelle EBD. Une définition formelle de EBD pour le modèle relationnel peut être trouvée dans [Wid92] et dans [LRV92] pour le modèle à objets du SGBD O_2 [BDK92].

Dans [Wid92], un état *ebd* ($ebd \in EBD$) d'une base de données est un ensemble $ebd = \{t_1, \ldots, t_n\}$ dans lequel chaque t_i ($1 \leq i \leq n$) est un n-uplet avec un identificateur unique et non réutilisable. Pour des raisons de simplicité, il est fait l'hypothèse que les identificateurs de n-uplets identifient également les relations auquelles les n-uplets appartiennent.

De manière informelle, un état d'une base de données dans un modèle à objets est décrit, dans [LRV92], comme un système de types Π et un ensemble consistant d'objets et de valeurs qui représentent des instances des types Π à un instant donné.

Domaine des événements : On appelle E le domaine des événements. Comme nous l'avons dit dans le chapitre précédent, nous ne nous intéressons pas – dans le cadre de Fl'are – à la nature des événements mais uniquement aux associations type d'événement-règle. Soit e un événement de E, e est une occurrence d'un type d'événement – que nous n'avons pas besoin de décrire – produit à un instant t. On ne fait aucune hypothèse sur E. En particulier, les associations type d'événement-règles, c'est-à-dire la faculté pour un événement e de déclencher ou non une règle r, sont représentées dans le domaine R qui décrit les règles (cf. ci-dessous).

Domaine des règles : On appelle R le domaine des règles. Soit r une règle de R :

$$r : EBD \times \mathcal{P}(E) \times \mathbb{N} \rightarrow \mathbb{B} \times \mathcal{P}(E) \times EBD \times \mathcal{P}(E) \times \mathbb{B}$$

r est une fonction qui prend comme arguments un état d'une base de données *ebd*, un ensemble d'événements *es*, et un numéro d'opération *op*. r est de la forme *événement-condition-action*. On suppose que l'évaluation de la partie condition de r est sans effet de bord, c'est-à-dire qu'elle ne modifie pas l'état de la base de données *ebd* et qu'elle ne produit pas d'événement. On modélise la partie action comme une séquence d'opérations $[op_1, op_2, \ldots, op_n]$ dans laquelle chaque op_i avec $1 \leq i \leq n$ peut être elle-même une séquence d'opérations élémentaires.

$r(ebd, es, op)$ retourne :

1. $r(ebd, es, op) \downarrow 1 = vrai$ ssi r est déclenchée par un ou plusieurs événements de es ;
2. $r(ebd, es, op) \downarrow 2$ est l'ensemble des événements déclenchants de r avec $r(ebd, es, op) \downarrow 2 \subset es$;
3. $r(ebd, es, op) \downarrow 3$ est l'état de la base après l'exécution de op ;

4. $r(ebd, es, op) \downarrow 4$ est l'ensemble des événements produits par l'exécution de l'opération op ;
5. $r(ebd, es, op) \downarrow 5 = vrai$ ssi op est la dernière opération de la partie action de r, c'est-à-dire si l'exécution de r s'achève avec l'exécution de op.

Notation Pour améliorer la lisibilité du document, dans la suite, nous utiliserons des *labels* pour référencer les objets retournés par une règle r de R. Ainsi :

1. $r(ebd, es, op) \downarrow 1$ sera noté $r(ebd, es, op) \downarrow EstDéclenchée$;
2. $r(ebd, es, op) \downarrow 2$ sera noté $r(ebd, es, op) \downarrow ÉvtsDéclenchants$;
3. $r(ebd, es, op) \downarrow 3$ sera noté $r(ebd, es, op) \downarrow NouvelÉtatBD$;
4. $r(ebd, es, op) \downarrow 4$ sera noté $r(ebd, es, op) \downarrow EvtsGénérés$;
5. $r(ebd, es, op) \downarrow 5$ sera noté $r(ebd, es, op) \downarrow FinExec$;

Si $r(ebd, es, op) \downarrow Déclenchée = faux$ alors $r(ebd, es, op) \downarrow ÉvtsDéclenchants = \emptyset$, $r(ebd, es, op) \downarrow NouvelÉtatBD = ebd$, $r(ebd, es, op) \downarrow ÉvtsGénérés = \emptyset$ et $r(ebd, es, op) \downarrow FinExec = vrai$

Remarque : Nous ne représentons pas explicitement l'évaluation de la partie condition de r. On suppose qu'implicitement, si la condition est évaluée à `faux` alors $r(ebd, es, op) \downarrow ÉvtsDéclenchants = \emptyset$, $r(ebd, es, op) \downarrow NouvelÉtatBD = ebd$, $r(ebd, es, op) \downarrow ÉvtsGénérés = \emptyset$ et $r(ebd, es, op) \downarrow FinExec = vrai$

Opérations sur R : On définit également sur R les opérations suivantes qui représentent les paramètres de l'exécution d'une règle r de R :

- $ModTraitÉvts : R \rightarrow MTE$ – le mode de traitement des événements de r avec $MTE = \{instance, ensembliste\}$, le domaine des modes de traitements des événements (Couche I, paramètre 1 de Fl'are, cf 4.2) ;
- $ModConsoÉvts : R \rightarrow MCE$ – le mode de consommation des événements de r avec $MCE = \{préservation, consommation\}$, le domaine des modes de consommation des événements (Couche I, paramètre 2 de Fl'are, cf 4.2) ;
- $ModEffetNet : R \rightarrow MNE$ – le mode de prise en compte de l'effet net de r avec $MNE = \{$*avec effet net, sans effet net*$\}$, le domaine des modes de prise en compte de l'effet net (Couche I, paramètre 3 de Fl'are, cf 4.2) ;
- $ModPréemption : R \rightarrow MP$ – le mode de préemption de r avec $MP = \{$*préemption, non préemption*$\}$, le domaine des modes de préemption (Couche I, paramètre 4 de Fl'are, cf 4.2) ;
- $ModExécution : R \rightarrow ME$ – le mode d'exécution de r avec $ME = \{$*immédiat, différé, retardé*$\}$, le domaine des modes d'exécution (Couche I, paramètre 5 de Fl'are, cf 4.2).

Remarque : Nous notons et utilisons les opérations comme des fonctions, c'est-à-dire des équations[1]. De plus, étant donné que les paramètres de l'exécution d'une règle r sont indépendants de son contexte d'exécution, c'est-à dire des arguments de r (état de la base *ebd*, événements produits es, opération génératrice d'événements *op*) ; nous noterons par exemple it ModExécution(r) (au lieu de *ModExécution(r(ebd,es,op))*) pour dénoter le mode d'exécution de r ; ceci afin d'améliorer la lisibilité du document.

Domaine des modules de règles : Soit M le domaine des modules de règles, c'est-à-dire des ensembles de règles définis par le programmeur : $M = \mathcal{P}(R)$.

Opérations sur M : On définit les opérations suivantes sur M :

- *MêmeModule* $M \times M \rightarrow \mathbb{B}$ qui prend comme arguments deux modules et qui rend *vrai* si ces deux modules sont en fait le même (!) ;
- *Stratégie* $M \times STRAT$ – la stratégie d'exécution d'un module avec $STRAT = \{priorits, pipelines, profondeur, cycles\}$ le domaine des stratégies d'exécution d'un module de règles (Couche II, paramètre 6 de Fl'are, cf. 4.4).

Soit m^1 et m^2, deux modules de M : $m^1 = \{r_1^1, r_2^1, ..., r_{n^1}^1\}$ et $m^2 = \{r_1^2, r_2^2, ..., r_{n^2}^2\}$; si *MêmeModule*(m^1, m^2) alors (par définition) *Stratégie*(m^1)=*Stratégie*(m^2).

Domaine des règles traitées : Soit RT le domaine des règles traitées (ou à traiter) : $RT = R \times \mathcal{P}(E) \times \mathbb{N} \times \mathbb{N} \times ETAT$ avec $ETAT = \{déclenché, exécuté\}$. Soit $rt =< r, es, t, c, e >$ une règle de RT, rt est une règle déclenchée en attente d'évaluation/exécution (e=*déclenché*) ou une règle déjà exécutée (e=*exécuté*). es est l'ensemble des événements déclenchants pris en compte lors de l'évaluation et l'exécution de r. t est l'instant de déclenchement de r. c est le numéro du cycle dans lequel sera traitée r. Signalons que le numéro de cycle n'est significatif que si le module auquel appartient la règle est exécuté par cycles.

Opérations sur RT : On définit l'opération *Module :* $RT \rightarrow MT$ qui retourne le module auquel appartient une règle donnée.

Domaine des modules de règles traitées : Soit MT le domaine des modules de règles à traiter : $MT = \mathcal{P}(RT)$.

1. Les opérations peuvent également être définies par des graphes, des tables ou encore des diagrammes.

Soit mt un module de règles à traiter de MT:

$$mt = \{rt_1, rt_2, ..., rt_n\}$$
$$= \{< r_1, es_1, t_1, c_1, e_1 >, < r_2, es_2, t_2, c_2, e_2 >, ..., < r_n, es_n, t_n, c_n, e_n >\}$$

Opérations sur MT : On définit les mêmes opérations *MêmeModule* et *Stratégie* de M sur MT.

Ordre sur les modules : Soit O_M le domaine des ordres totaux sur M. Soit o_M un ordre de O_M, $o_M = \{m \leq_M m'\}$ avec m et m' des modules de M, et $\leq_M$ une relation binaire qui est :

- réflexive : $\forall m \in M, (m \leq_M m)$,
- antisymétrique : $\forall m, m' \in M, ((m \leq_M m' \wedge m' \leq_M m) \Rightarrow m = m')$,
- transitive : $\forall m, m', m'' \in M, ((m \leq_M m' \wedge m' \leq_M m'') \Rightarrow m \leq_M m'')$,
- totale : $\forall m, m' \in M, (m \leq_M m' \vee m' \leq_M m)$.

(M, o_M) **est un ensemble totalement ordonné, c'est-à-dire une chaîne.**

Ordre sur les règles : Soit O_R le domaine des ordres totaux sur R. Soit o_R un ordre de O_R, $o_R = \{r \leq_R r'\}$ avec r et r' des règles de R et $\leq_R$ reflexive, antisymétrique, transitive et totale. (R, o_R) **est également une chaîne.**

5.3 Fonctions de valuation

Schématiquement, la sémantique dénotationnelle de Fl'are est une fonction *ModelExec* qui prend comme arguments un ensemble de modules de règles, un ensemble d'événements produits par l'exécution d'une transaction et un état de la base de données ; et qui retourne le nouvel état de la base de données qui résulte de l'exécution des règles déclenchées par l'ensemble d'événements donné.

Comme nous l'avons vu dans la section 4.3.3, la prise en compte du *mode d'exécution* détermine deux comportements très distincts du modèle d'exécution en cours de transaction et en fin de transaction. Les six sections 5.3.1 à 5.3.6 illustrent ce phénomène en exhibant trois groupes de fonctions qui correspondent aux trois phases de l'exécution d'un ensemble de règles :

1. les fonctions des sections 5.3.1 et 5.3.4 concernent la **construction** des ensembles (où modules) de règles à traiter au cours d'une transaction déclenchante et en fin de transaction déclenchante ;

2. les fonctions des sections 5.3.2 et 5.3.5 concernent la **sélection** d'une règle à traiter au cours d'une transaction déclenchante et en fin de transaction déclenchante ;
3. les fonctions des sections 5.3.3 et 5.3.6 concernent le **traitement** d'une règle au cours d'une transaction déclenchante et en fin de transaction déclenchante ;

ModelExec : La fonction *ModelExec* dénote les interactions entre d'une part, l'application et plus précisément une transaction (déclenchante) et d'autre part le modèle d'exécution des règles actives. On considère que *ModelExec* est appelée :

- à chaque point d'exécution d'une transaction déclenchante TD pour exécuter les règles immédiates et évaluer les règles retardées déclenchées par TD, et traiter les règles déclenchées (en cascade) par l'exécution des règles immédiates ;
- juste avant la validation de TD pour exécuter les règles différées déclenchées au cours de TD et les règles retardées évaluées au cours de TD, et exécuter les règles déclenchées (en cascade) par l'exécution des règles différées et retardées.

ModelExec prend comme arguments (1) un ensemble de modules ms, (2) un ordre sur les modules o_M, (3) un ordre sur les règles o_R, puis (4) un état de la BD *ebd*, (5) un ensemble d'événements es produits à un instant (6) t dans la transaction déclenchante TD, (7) un ensemble de modules $mtsftd$ à traiter à la fin de la transaction TD, (8) un booléen ftd qui indique si *ModelExec* est appelée au cours de la transaction déclenchante ($ftd = faux$) ou en fin de transaction déclenchante ($ftd = vrai$) et (9) un ensemble d'événements *estd* qui est l'ensemble des événements produits depuis le début de la transaction déclenchante hormis ceux de es (c'est-à-dire tous les événements qui ont été produits avant es).

ms, o_M, et o_R représentent la *base de règles*, qui peut être considérée comme *statique*. **Nous faisons de plus l'hypothèse que** $mtsftd$ **et** $estd$ **sont stockés entre deux appels de** *ModelExec* **dans une même transaction.**

Le processus d'exécution des règles peut ne pas se terminer. Dans ce cas, *ModelExec* retourne $\perp$ (bottom). Sinon, *ModelExec* (1) retourne un nouvel état de la BD, (2) l'ensemble des modules de règles à traiter en cours de transaction, l'ensemble des modules de règles à traiter en fin de transaction et un nouvel ensemble d'événements. Les éléments (1), (2), (3) et (4) résultent du traitement (appels de *TraiteModules* et *TraiteModulesFtd*) des règles de ms déclenchées par les événements de es. Les fonctions *Déclenche* et *DéclencheFtd* **construisent** ces ensembles de modules. *ModelExec* est définie comme suit :

$ModelExec : \mathcal{P}(M) \times O_M \times O_R$

$$\rightarrow EBD \times \mathcal{P}(E) \times \mathbb{N} \times \mathcal{P}(MT) \times \mathbb{B} \times \mathcal{P}(E)$$
$$\rightarrow (EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathcal{P}(E)) \vee \{\perp\}$$

$ModelExec[mts, o_M, o_R] = \lambda ebd, es, t, mtsftd, ftd, estd.$
$\quad si\ (\neg ftd)$
$\quad\quad alors\ TraiteModules(ms, o_M, o_R)$
$\quad\quad\quad\quad (ebd, Déclenche(ebd, es, t, mtsftd, estd, \emptyset), t, 1, ftd, estd)$
$\quad\quad sinon\ TraiteModulesFtd(ms, o_M, o_R)$
$\quad\quad\quad\quad (ebd, DéclencheFtd(ebd, es, t, mtsftd, estd, \emptyset), t, 1, ftd, estd)$
$\quad finsi$
□

Remarques :

1° La signature la plus simple et la plus naturelle de la fonction *ModèlExec* est :
$$ModelExec : \mathcal{P}(M) \times O_M \times O_R \times EBD \times \mathcal{P}(E) \times \mathbb{N} \times \mathcal{P}(MT) \times \mathbb{B} \times \mathcal{P}(E)$$
$$\rightarrow (EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathcal{P}(E)) \vee \{\perp\}$$

La signature que nous utilisons est équivalente[2] mais il nous semble "qu'elle est plus parlante" puisqu'elle fait ressortir les domaines statiques et dynamiques que nous avons évoqués dans la section 5.1.

2° La fonction *Déclenche* (resp. *DéclencheFtd*) définie dans le prochain paragraphe admet six paramètres et retourne un couple. Aussi dans l'appel de *TraiteModules* (resp. *TraiteModulesFtd*) ci-dessus, pour être parfaitement "propre", nous devrions écrire :

$ConsidèreModules(ms, o_M, o_R)$
$\quad (ebd,$
$\quad\ Déclenche(ebd, es, t, mtsftd, estd, \emptyset) \downarrow 1,$
$\quad\ Déclenche(ebd, es, t, mtsftd, estd, \emptyset) \downarrow 2,$
$\quad\ t,$
$\quad\ 1,$
$\quad\ ftd,$
$\quad\ estd)$

D'une manière générale, nous nous permettons quelques petites libertés avec les notations lorsque celles-ci améliorent la lisibilité du document.

2. $[[D \times D'] \rightarrow D'']$ et $[D \rightarrow [D' \rightarrow D'']]$ sont isomorphes.

□

5.3.1 Construction des ensembles en cours de transaction

Déclenche : La fonction *Déclenche* prend comme arguments (1) un ensemble de modules ms puis (2) un état de la BD ebd, (3) un ensemble d'événements es produits à un instant (4) t dans la transaction déclenchante TD, (5) un ensemble de modules mts à traiter "immédiatement", c'est-à-dire au cours de TD, (6) un ensemble de modules $mtsftd$ à traiter en fin de TD, (7) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de la transaction déclenchante hormis ceux de es et (8) un module de règles à traiter mt.

Déclenche retourne (1) le nouvel ensemble des modules de règles à traiter au cours de TD qui est la fusion de mts et des règles *immédiates* et *retardées* de ms déclenchées par es et (2) le nouvel ensemble des modules de règles à traiter au cours de TD qui est la fusion de $mtsftd$ et des règles *différées* de ms déclenchées par es.

Le huitième paramètre mt est utilisé pour *le calcul transitif des cycles d'exécution* (cf. 4.4.2). Il représente le module auquel appartient la règle qui a généré les événements es (troisième paramètre de la fonction *Déclenche*). Dans la suite, on appelera *module déclenchant* un tel module. Si les événements es n'ont pas été produits par l'exécution d'une règle mais par la transaction déclenchante, mt sera l'ensemble vide. Le calcul transitif des cycles d'exécution est détaillé dans la fonction *LesDéclenchées* ci-après. *Déclenche* est définie comme suit :

$$\begin{aligned} Déclenche : \mathcal{P}(M) &\rightarrow EBD \times \mathcal{P}(E) \times \mathbb{N} \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathcal{P}(E) \times MT \\ &\rightarrow \mathcal{P}(MT) \times \mathcal{P}(MT) \end{aligned}$$

$$\begin{aligned} &Déclenche[ms] = \lambda ebd, es, t, mts, mtsftd, estd, mt. \\ &\quad < FusionneModules(\\ &\qquad mts, \\ &\qquad (FusionneEnsModules(\\ &\qquad\quad LesDéclenchées(ms)(immédiat, ebd, es, t, estd, mt), \\ &\qquad\quad LesDéclenchées(ms)(retarddé, ebd, es, t, estd, mt))), \\ &\quad (FusionneEnsModules(mts, \\ &\qquad LesDéclenchées(ms)(différé, ebd, es, t, estd, mt)), > \end{aligned}$$

□

LesDéclenchées : La fonction *LesDéclenchées* prend comme arguments (1) un ensemble de modules ms puis (2) un état de la BD ebd, (3) un ensemble d'événements es

produits à un instant (4) t dans la transaction déclenchante TD, (5) un mode d'exécution *modexec*, (7) un ensemble d'événements *estd* qui est l'ensemble des événements produits depuis le début de TD et (8) un module déclenchant mt.

LesDéclenchées **construit** et retourne l'ensemble des modules de règles *immédiates*, *retardées* ou *différées* (suivant la valeur de *modexec*) à traiter.

Chacune des règles de cet ensemble est un n-uplet : $< r_i^I, ed_i^I, t_i^I, c_i^I, état_i^I >$ dans lequel la règle r_i^I est déclenchée ($état_i^I$ =*déclenché*), ed_i^I est l'ensemble des événements déclenchants pris en compte lors du traitement de r_i^I, t_i^I est l'instant de déclenchement de r_i^I, c_i^I est le cycle dans lequel r_i^I sera traitée – si le module auquel appartient r_i^I est exécuté selon la stratégie *par cycles*, évidemment.

C'est ici qu'intervient le **calcul transitif des cycles d'exécution** : si (et seulement si) r_i^I est déclenchée par l'exécution d'une règle qui appartient au même module qu'elle, alors on incrémente son cycle d'exécution de 1 ; si la règle est déclenchée par une règle qui appartient à un autre module ou si la règle est déclenchée par une opération de la transaction déclenchante, son cycle d'exécution reste inchangé. Notons que le calcul des cycles est effectué pour toutes les règles mais qu'il n'est évidemment significatif que pour les règles qui sont exécutées par cycles. *LesDéclenchées* est définie comme suit :

$$LesDéclenchées : \mathcal{P}(M) \rightarrow EBD \times \mathcal{P}(E) \times \mathbb{N} \times ME \times \mathcal{P}(E) \times MT \rightarrow \mathcal{P}(MT)$$

$$\begin{aligned}
&LesDéclenchées[ms] = \lambda ebd, es, t, modexec, estd, mt.\\
&\quad \{\{rt_i^I \ where \ (rt_i^I =< r_i^I, ed_i^I, t_i^I, c_i^I, état_i^I > \ \wedge\\
&\qquad\qquad r_i^I \downarrow ModExécution = modexec \wedge\\
&\qquad\qquad ed_i^I = EstDéclenchée(r_i^I, ebd, es, estd) \wedge\\
&\qquad\qquad ed_i^I \neq \emptyset \wedge\\
&\qquad\qquad t_i^I = t \wedge\\
&\qquad\qquad c_i^I = si \ \ MêmeModule(Module(rt_i^I), mt)\\
&\qquad\qquad\qquad\qquad alors \ PGCcycle(Module((rt_i^I))) + 1\\
&\qquad\qquad\qquad\qquad sinon \ PGCcycle(Module((rt_i^I)))\\
&\qquad\qquad\qquad finsi \wedge\\
&\qquad\qquad état_i^I = déclenché\\
&\quad | \ (\forall \{r_i^I\}, \exists ! \, m \in ms \mid \{r_i^I\} \subset m \wedge \ \ MêmeModule(\{rt_i^I\}, m))\}
\end{aligned}$$

□

Remarque : Le haut niveau d'abstraction de la sémantique dénotationnelle – et plus particulièrement du λ-calcul (description uniquement fonctionnelle) – nous conduit

parfois à utiliser la sémantique dénotationnelle "dans un style opérationnel", notamment dans l'utilisation du symbole "=" (égal) auquel nous donnons parfois une sémantique proche de celle de *l'affectation* que l'on rencontre dans les langages de programmation.

EstDéclenchée : La fonction *EstDéclenchée* prend comme arguments (1) une règle r, (2) un état de la BD *ebd*, (3) un ensemble d'événements es et (4) un ensemble d'événements *estd* qui est l'ensemble des événements produits depuis le début de TD.

***EstDéclenchée* dénote la prise en compte de l'effet net pour le déclenchement d'une règle :**

- Si r ne prend pas en compte l'*effet net* (*ModEffetNet(r)* = *faux*) alors *EstDéclenchée* retourne l'ensemble des événements de es qui sont déclenchants pour r. Si r n'est pas déclenchée – par un ou plusieurs événements de es – *EstDéclenchée* retourne $\emptyset$.
- Si r prend en compte l'*effet net*, (*ModEffetNet(r)* = *vrai*) alors *EstDéclenchée* retourne l'ensemble des événements de $EffetNet(es, estd)$ qui sont déclenchants pour r. *EffetNet* est la fonction qui calcule l'effet net entre (1) les événements correspondant au type d'événement de r et (2) tous les événements de la transaction déclenchante et plus précisément ceux qui sont composables avec les événements de (1).

Le fait de ne pas faire d'hypothèse sur les modèles de données et d'événements constitue une petite limitation de notre approche en ce qui concerne le traitement de l'*effet net*. En effet, nous ne pouvons définir ici la fonction *EffetNet*. Néanmoins une définition formelle de cette fonction peut être trouvée dans [Wid92] qui est basé sur le modèle relationnel et dans [CC95c] qui est basé sur le modèle à objets du SGBD O_2. *EstDéclenchée* est définie comme suit :

$EstDéclenchée : R \times EBD \times \mathcal{P}(E) \times \mathcal{P}(E) \rightarrow \mathcal{P}(E)$

$EstDéclenchée = \lambda r, ebd, es, estd.$
$\quad si$ (*ModEffetNet(r)*$= faux$)
$\quad\quad alors\ si\ (r(ebd, es, \_) \downarrow EstDéclenchée)$
$\quad\quad\quad\quad alors\ r(ebd, es, \_) \downarrow ÉvtsDéclenchants$
$\quad\quad\quad\quad sinon\ \emptyset$
$\quad\quad\quad finsi$
$\quad\quad sinon\ \ si\ (r(ebd, EffetNet(es, estd, \_)) \downarrow EstDéclenchée)$
$\quad\quad\quad\quad alors\ r(ebd, es, \_) \downarrow ÉvtsDéclenchants$

$sinon\ \emptyset$

$finsi$

$finsi$

□

PGCycle : La fonction *PGCycle* prend comme argument un module de règles à traiter (par cycles) et retourne le plus grand cycle associé à une règle de ce module. Ce cycle représente la profondeur maximale des cascades de règles à l'instant ou la fonction est appelée. Un module dénoté par l'ensemble vide représente la transaction déclenchante. *PGCycle* est utilisée pour le calcul transitif des cycles d'exécution ; elle est définie comme suit :

$PGCycle : MT \rightarrow \mathbb{N}$

$$PGCycle = \lambda\{< r_1, es_1, t_1, c_1, e_1 >, \ldots, < r_n, es_n, t_n, c_n, e_n >\}$$
$$let\ mt = \{< r_1, es_1, t_1, c_1, e_1 >, \ldots, < r_n, es_n, t_n, c_n, e_n >\}$$
$$in$$
$$si\ (mt = \emptyset)$$
$$alors\ 1$$
$$sinon\ c_i\ where\ \{< r_j, es_j, t_j, c_j, e_j > \mid i \neq j\ \wedge\ c_j > c_i \wedge 1 \leq i \leq n\ \wedge\ 1 \leq j \leq n\} = \emptyset$$
$$finsi$$

□

FusionneEnsModules : La fonction *FusionneEnsModules* prend comme arguments deux ensembles de modules de règles à traiter mts^1 et mts^2 et retourne un ensemble qui est la fusion de ces deux ensembles. Cet ensemble est l'union :

- des modules qui n'apparaissent que dans mts^1 (resp. mts^2) : $mts^{1,1}$ (resp. $mts^{2,2}$)
- et de la fusion des modules qui apparaissent dans mts^1 et mts^2.

FusionneEnsModules est définie comme suit :

$FusionneEnsModules$: $\mathcal{P}(MT) \times \mathcal{P}(MT) \rightarrow \mathcal{P}(MT)$

$$FusionneEnsModules = \lambda\{mt_1^1, mt_2^1, \ldots, mt_{n^1}^1\}, \{mt_1^2, mt_2^2, \ldots, mt_{n^2}^2\}.$$

$$let\ mts^1 = \{mt_1^1, mt_2^1, \ldots, mt_{n^1}^1\}$$

$$mts^2 = \{mt_1^2, mt_2^2, \ldots, mt_{n^2}^2\}$$
$$mts^{1,1} = \{mt_i^1 \in mts^1 \mid \{mt_j^2 \in mts^2 \mid \mathit{MêmeModule}(mt_i^1, mt_j^2)\} = \emptyset\}$$
$$mts^{2,2} = \{mt_k^2 \in mts^2 \mid \{mt_l^1 \in mts^1 \mid \mathit{MêmeModule}(mt_k^2, mt_l^1)\} = \emptyset\}$$

$$\mathit{in}\ mts^{1,1} \cup mts^{2,2} \cup \{\mathit{FusionneEnsRègles}(mt_o^1, mt_p^2) \mid mt_o^1 \in mts^1 - mts^{1,1} \wedge mt_p^2 \in mts^2 - mts^{2,2} \wedge \mathit{MêmeModule}(mt_o^1, mt_p^2)\}$$

□

FusionneEnsRègles : La fonction *FusionneEnsRègles* prend comme arguments deux modules de règles à traiter mt^1 et mt^2 et retourne un ensemble de règles qui est la fusion de ces deux modules.

***FusionneEnsRègles* dénote la prise en compte du mode de traitement des événements (instance ou ensembliste).** Le mode de traitement des événements d'une règle r a une influence sur le nombre d'occurrences de r dans l'ensemble des règles à traiter en cas de déclenchements multiples de r et sur l'ensemble des événements déclenchants pris en compte lors des exécutions de r (cf. 4.3.3).

L'ensemble de règles retourné est l'union :

- des règles qui n'apparaissent que dans mt^1 (resp. mt^2) : $mt^1, 1$ (resp. $mt^2, 2$) ;
- des règles qui apparaissent dans mt^1 (resp. mt^2) et qui ont un mode de traitement des événements *par instance* : mt^1, ins (resp. mt^1, ins) et
- des règles qui apparaissent à la fois dans mt^1 et mt^2 qui ont un mode de traitement des événements *ensembliste*. Soit r une de ces règles. r apparaitra une seule fois dans l'ensemble retourné et l'ensemble de ses événements déclenchants sera l'union des ensembles des événements de chacune des instances de r dans mt^1 et mt^2.

FusionneEnsRègles est définie comme suit :

$\mathit{FusionneEnsRègles}: MT \times MT \to MT$

$$\mathit{FusionneEnsRègles} = \lambda\{< r_1^1, es_1^1, t_1^1, c_1^1, e_1^1 >, \ldots, < r_{n^1}^1, es_{n^1}^1, t_{n^1}^1, c_{n^1}^1, e_{n^1}^1 >\}, \{< r_2^2, es_2^2, t_2^2, c_2^2, e_2^2 >, \ldots, < r_{n^2}^2, es_{n^2}^2, t_{n^2}^2, c_{n^2}^2, e_{n^2}^2 >\},$$

$$\mathit{let}\ mt^1 = \{< r_1^1, es_1^1, t_1^1, c_1^1, e_1^1 >, \ldots, < r_{n^1}^1, es_{n^1}^1, t_{n^1}^1, c_{n^1}^1, e_{n^1}^1 >\}$$
$$mt^2 = \{< r_2^2, es_2^2, t_2^2, c_2^2, e_2^2 >, \ldots, < r_{n^2}^2, es_{n^2}^2, t_{n^2}^2, c_{n^2}^2, e_{n^2}^2 >\}$$

$$mt^{1,1} = \{< r_i^1, es_i^1, t_i^1, c_i^1, e_i^1 > \in mt^1 \mid \{< r_j^2, es_j^2, t_j^2, c_j^2, e_j^2 > \in mt^2 \mid r_i^1 = r_j^2\} = \emptyset\}$$
$$mt^{2,2} = \{< r_k^2, es_k^2, t_k^2, c_k^2, e_k^2 > \in mt^2 \; ; \mid \{< r_l^1, es_l^1, t_l^1, c_l^1, e_l^1 > \in mt^1 \mid r_k^2 = r_l^1\} = \emptyset\}$$

$$mt^{1,ins} = \{< r_m^1, es_m^1, t_m^1, c_m^1, e_m^1 > \in mt^1 \mid ModTraitÉvts(r_m^1) = instance\}$$
$$mt^{2,ins} = \{< r_o^2, es_o^2, t_o^2, c_o^2, e_o^2 > \in mt^2 \mid ModTraitÉvts(r_o^2) = instance\}$$

$$mt^{1,ens} = \{< r_p^1, es_p^1, t_p^1, c_p^1, e_p^1 > \in ((mt^1 - mt^{1,1}) - mt^{1,ins}) \mid 1 \leq p \leq n^1\}$$
$$mt^{2,ens} = \{< r_q^2, es_q^2, t_q^2, c_q^2, e_q^2 > \in ((mt^2 - mt^{2,2}) - mt^{2,ins}) \mid 1 \leq q \leq n^2\}$$

$$in\ (mt^{1,1} \cup mt^{2,2}) \cup (mt^{1,ins} \cup mt^{2,ins}) \cup$$
$$\{< r_p^1, es, t_p^1, c_p^1, e_p^1 > \mid < r_p^1, es_p^1, t_p^1, c_p^1, e_p^1 > \in mt^{1,set} \wedge$$
$$< r_q^2, es_q^2, t_q^2, c_q^2, e_q^2 > \in mt^{2,set} \wedge$$
$$r_p^1 = r_q^2 \wedge$$
$$es = (\bigcup_{p=1}^{|mt^{1,ens}|} es_p^1 \cup \bigcup_{q=1}^{|mt^{2,ens}|} es_q^2)$$

□

5.3.2 Sélection d'une règle en cours de transaction

TraiteModules : La fonction *TraiteModules* prend comme arguments (1) un ensemble de modules ms, (2) un ordre sur les modules o_M et (3) un ordre sur les règles o_R.

TraiteModules retourne le *plus petit point fixe* (*pppf*)[3] d'une fonction $\mathcal{CM}$. $\mathcal{CM}$ prend (1) un état de la BD *ebd*, (2) un ensemble de modules mts à traiter "immédiatement", c'est-à-dire au cours de TD, (3) un ensemble de modules $mtsftd$ à traiter en fin de TD, (4) un entier tc qui est l'estampille (temporelle) courante, et (5) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de TD.

Si aucune règle n'est (plus) à traiter dans mts et $mtsftd$ alors $\mathcal{CM}$ retourne $< ebd, \emptyset, mtsftd, \emptyset >$; sinon $\mathcal{CM}$ sélectionne une règle rt à traiter parmi le module le plus prioritaire puis s'applique elle-même sur (1) le nouvel état de la BD, (2) le nouvel ensemble de modules à traiter au cours de *TD*, (3) le nouvel ensemble de modules à traiter en fin de TD et (4) le nouvel ensemble des événements produits depuis le début de TD. (1), (2), (3) et (4) résultent de l'exécution de rt sur *ebd*, mts, $mtsftd$

3. La fonction *TraiteModules* est en réalité une fonction récursive. Nous rappelons que nous utilisons la notion de *plus petit point fixe* comme un simple artifice d'écriture.

et *tc*. *TraiteModules* est définie comme suit :

$$\begin{aligned} TraiteModules :& \mathcal{P}(M) \times O_M \times O_R \\ & \rightarrow EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathbb{N} \times \mathcal{P}(E) \\ & \rightarrow (EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathcal{P}(E)) \vee \{\bot\} \end{aligned}$$

$$\begin{aligned} &TraiteModules[ms, o_M, o_R] = \underline{pppf}(\lambda \mathcal{CM}. \\ &\quad \lambda < ebd, mts, mtsftd, tc, estd, ftd > . \\ &\quad si\ (SontDéclenchées(ms, mts) = \emptyset) \\ &\quad\quad alors\ < ebd, \emptyset, mtsftd, \emptyset > \\ &\quad\quad sinon\ let\ rt = SelectRègle(o_R)(SelectModule(o_M)(mts)) \\ &\quad\quad\quad\quad in\ \mathcal{CM}(TraiteRègle(rt, ebd, mts, mtsftd, tc, estd))) \\ &\quad finsi \end{aligned}$$

□

Remarque La preuve de l'existence d'un plus petit point fixe est complexe et dépasse notre propos. On pourrait montrer que les domaines que l'on manipule, *augmentés* de $\bot$ (l'indéfini) et $\top$ (l'erroné ou l'inconsistant) sont des treillis complets, continus et dénombrables. Les fonctions définies sur ces domaines sont alors continues donc monotones. Le théorème de Tarski assure ensuite que pour des fonctions du type $f : D \rightarrow D'$ avec f monotone et D treillis complet, D' est un treillis de points fixes et il existe un plus petit point fixe.

Pour la problématique qui nous occupe – la formalisation de Fl'are – on peut remarquer que ce problème est très similaire à celui que l'on rencontre lorsque l'on veut définir la sémantique d'une instruction `while` dans les langages impératifs : il est très possible qu'une telle instruction "ne finisse jamais" [4]. Néanmoins il est possible de définir formellement la sémantique d'une instruction `while` et de telles constructions sont utilisées quotidiennement partout dans le monde ! Ce point est discuté dans [Ten76, Sto77].

□

SontDéclenchées : La fonction *SontDéclenchées* prend comme argument (1) un ensemble de modules de règles *ms* puis (2) un ensemble de modules de règles à traiter *mts* et retourne un booléen qui est *vrai* s'il reste des règles à traiter dans *mts*. Une règle *r* reste à traiter si elle n'a pas encore été exécutée (*état=déclenché*) ou si elle a déjà été exécutée (*état=exécuté*) mais elle n'a pas été supprimée de *mts* – car

4. Que le programmeur à qui cela n'est jamais arrivé me jette la première pierre !

elle préserve les événements (*mode de consommation des événements = préservation*) – afin de pouvoir prendre en compte l'ensemble des événements déclenchants de r survenus depuis le début de la transaction déclenchante à chaque traitement de r.

***SontDéclenchées* résulte de la prise en compte, par la fonction *Reflète* (cf. ci-après), du mode de consommation des événements d'une règle.** *SontDéclenchées* est définie comme suit :

$SontDéclenchées : \mathcal{P}(M) \rightarrow \mathcal{P}(MT) \rightarrow \mathbb{B}$
$SontDéclenchées[ms] = \lambda\{\{< r_i^I, es_i^I, t_i^I, c_i^I, e_i^I >\}\}.$
$\quad let\ mts = \{\{< r_i^I, es_i^I, t_i^I, c_i^I, e_i^I >\}\}\ in$
$\quad\quad si\ (mts = \emptyset) \vee \{\{r_i^I\} \mid es_i^I = déclenché\} = \emptyset$
$\quad\quad\quad alors\ faux$
$\quad\quad\quad sinon\ vrai$
$\quad\quad finsi$
□

SelectModule : La fonction *SelectModule* prend comme arguments (1) un ordre o_M sur les modules puis (2) un ensemble de modules de règles à traiter $\{mt^1, mt^2, \ldots, mt^n\}$. Elle retourne le module le plus prioritaire parmi l'ensemble donné. Ce module existe toujours puisque $(M, \leq_M)$ est une chaîne.

$SelectModule : O_M \rightarrow \mathcal{P}(MT) \rightarrow MT$

$SelectModule\ [o_M] = \lambda\{mt^1, mt^2, \ldots, mt^n\}.$
$\quad mt^i\ where\ 1 \leq i \leq n\ \wedge\ \{mt^j \mid 1 \leq \jmath \leq n\ \wedge\ mt^i \leq_M mt^j\} = \emptyset\}$
□

SelectRègle : La fonction *SelectRègle* prend comme argument (1) un ordre o_R sur les règles puis (2) un module de règles à traiter mt et retourne la règle qui sera la prochaîne règle de mt traitée. **Le choix de cette règle, donc la fonction *SelectRègle*, dénote la stratégie d'exécution du module *mt*** (*Stratégie(mt)*). *SelectRègle* est définie comme suit :

$SelectRègle : O_R \rightarrow MT \rightarrow RT$

$SelectRègle[o_R] = \lambda mt.$
$\quad cas\ (mt \downarrow 2)\ entre$
$\quad\quad priorités\quad : APlat(PremièresDéclenchées(Prioritaire(o_R)(mt)))$

$pipelines$: $APlat(Prioritaire(o_R)(PremièresDéclenchées(mt))))$
$profondeur$: $APlat(Prioritaire(o_R)(DernièresDéclenchées(mt)))$
$cycles$: $APlat(PremièresDéclenchées(Prioritaire(o_R)(PPCycle(mt))))$
fincas
□

APlat : La fonction *APlat* prend comme argument un ensemble de règles à traiter qui sera toujours en réalité un singleton et retourne la règle que contient ce singleton. Nous ne donnons pas la définition de cette fonction qui est triviale. La signature de la fonction *APlat* est la suivante :

$APlat : \mathcal{P}(RT) \rightarrow RT$

Prioritaire : La fonction *Prioritaire* prend comme arguments (1) un ordre o_R sur les règles puis (2) un module de règles à traiter. Elle retourne l'ensemble des règles les plus prioritaires parmi l'ensemble donné.

Notons que soit l'ensemble retourné est un singleton ; soit il contient plusieurs occurrences d'une même règle[5] (qui a donc forcément un mode de traitement des événements ensembliste) puisque $(R, \leq_R)$ est un chaîne. *Prioritaire* est définie comme suit :

$Prioritaire : O_R \rightarrow MT \rightarrow \mathcal{P}(RT)$

$Prioritaire[o_R] = \lambda\{< r_1^I, es_1^I, t_1^I, c_1^I, e_1^I >, \ldots, < r_{n^I}^I, es_{n^I}^I, t_{n^I}^I, c_{n^I}^I, e_{n^I}^I >\}.$
$\{< r_i^I, es_i^I, t_i^I, c_i^I, e_i^I > \mid$
$1 \leq i \leq n \wedge$
$\{< r_j^I, es_j^I, t_j^I, c_j^I, e_j^I > \mid 1 \leq j \leq n \wedge j \neq i \wedge r_i \leq_R r_j\} = \emptyset\}$
□

PremièresDéclenchées La fonction *PemièresDéclenchées* prend comme argument un module de règles à traiter. Elle retourne l'ensemble des règles *déclenchées les premières*, ce qui revient à dire qu'il n'existe pas de règles déclenchées avant les règles de cet ensemble, parmi l'ensemble donné. Plusieurs règles pouvant partager le même instant de déclenchement, on ne peut présumer en rien sur la structure de l'ensemble

5. Ces occurrences sont néanmoins des entités distinctes puisque ce sont des n-uplets de la forme $< r, es, t, c, e >$ dans lesquels r est le même mais es, t, c et e sont différents.

retourné. *PremièresDéclenchées* est définie comme suit :

$PremièresDéclenchées{:}MT \rightarrow \mathcal{P}(RT)$

$$PremièresDéclenchées = \lambda\{< r_1^I, es_1^I, t_1^I, c_1^I, e_1^I >, \ldots, < r_{n^I}^I, es_{n^I}^I, t_{n^I}^I, c_{n^I}^I, e_{n^I}^I >\}.$$
$$\{< r_i^I, es_i^I, t_i^I, c_i^I, e_i^I > \mid$$
$$1 \leq i \leq n \wedge$$
$$\{< r_j^I, es_j^I, t_j^I, c_j^I, e_j^I > \mid 1 \leq j \leq n \wedge j \neq i \wedge t_j \leq t_i\} = \emptyset\}$$

□

DernièresDéclenchées La fonction *DernièresDéclenchées* prend comme argument un module de règles à traiter. Elle retourne l'ensemble des règles *déclenchées les dernières* parmi l'ensemble donné. Comme pour la fonction *PremièreDéclenchée*, on ne peut présumer de rien sur la structure de l'ensemble retourné. *DernièresDéclenchées* est définie comme suit :

$DernièresDéclenchées{:}MT \rightarrow \mathcal{P}(RT)$

$$DernièresDéclenchées = \lambda\{< r_1^I, es_1^I, t_1^I, c_1^I, e_1^I >, \ldots, < r_{n^I}^I, es_{n^I}^I, t_{n^I}^I, c_{n^I}^I, e_{n^I}^I >\}.$$
$$\{< r_i^I, es_i^I, t_i^I, c_i^I, e_i^I > \mid$$
$$1 \leq i \leq n \wedge$$
$$\{< r_j^I, es_j^I, t_j^I, c_j^I, e_j^I > \mid 1 \leq j \leq n \wedge j \neq i \wedge t_j \geq t_i\} = \emptyset\}$$

□

PPCycle La fonction *PPCycle* prend comme argument un module de règles à traiter. Elle retourne l'ensemble des règles devant être traitées dans le plus petit cycle parmi l'ensemble donné. Plusieurs règles pouvant partager le même cycle, on ne peut présumer en rien sur la structure de l'ensemble retourné. *PPCycle* est définie comme suit :

$PPCycle{:}MT \rightarrow \mathcal{P}(RT)$

$$PPCycle = \lambda\{< r_1^I, es_1^I, t_1^I, c_1^I, e_1^I >, \ldots, < r_{n^I}^I, es_{n^I}^I, t_{n^I}^I, c_{n^I}^I, e_{n^I}^I >\}.$$
$$\{< r_i^I, es_i^I, t_i^I, c_i^I, e_i^I > \mid$$
$$1 \leq i \leq n \wedge$$
$$\{< r_j^I, es_j^I, t_j^I, c_j^I, e_j^I > \mid 1 \leq j \leq n \wedge j \neq i \wedge c_j \leq c_i\} = \emptyset\}$$

□

5.3.3 Traitement d'une règle en cours de transaction

TraiteRègle La fonction *TraiteRègle* prend comme arguments (1) un ensemble ms de modules de règles puis (2) une règle à traiter rt, (3) un état ebd de la base, (4) un ensemble mts de modules de règles à traiter en cours de transaction déclenchante TD, (5) un ensemble $mtsftd$ de modules de règles à traiter en fin de TD, (6) un entier tc qui est l'estampille courante et (7) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de TD.

Si la règle r est *immédiate* ($r(\_,\_,\_) \downarrow ModExécution{=}immédiat$), elle est évaluée et exécutée. Si r est *retardée*, elle est simplement *évaluée*. *TraiteRègle* n'est jamais appelée avec une règle différée comme argument puisqu'on ne fait aucun traitement sur les règles différées en cours de transaction. C'est pour cette raison que le cas *différé* n'est pas traité dans la fonction. *TraiteRègle* est définie comme suit :

$$\begin{aligned}
&TraiteRègle : \mathcal{P}(M) \rightarrow RT \times EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathbb{N} \times \mathcal{P}(E) \\
&\qquad\qquad \rightarrow EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathcal{P}(E) \\
&TraiteRègle[ms] \;=\; \lambda < r, es, t, c, e >, ebd, mts, mtsftd, tc, estd. \\
&\quad \text{cas } (r(ebd, es, _) \downarrow 5)\ entre \\
&\qquad immédiat :\ ExecRègle(ms)(< r, es, t, c, e >, 1, ebd, mts, mtsftd, tc, cc, estd) \\
&\qquad retardé \quad :\ EvalRègle(ms)(< r, es, t, c, e >, mts, mtsftd) \\
&\quad fincas
\end{aligned}$$

□

ExecRègle La fonction *ExecRègle* prend comme argument un ensemble de modules de règles ms. *ExecRègle* retourne le *plus petit point fixe* (*pppf*) d'une fonction $\mathcal{ER}$. $\mathcal{ER}$ prend comme arguments (1) une règle à traiter $rt =< r, es, t, c, e >$, (2) l'opération de la partie action de r à exécuter, (3) un état de la BD ebd, (4) un ensemble de modules mts à traiter "immédiatement", c'est-à-dire au cours de TD, (5) un ensemble de modules $mtsftd$ à traiter en fin de TD, (6) un entier tc qui est l'estampille courante et (7) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de TD.

Si l'exécution de r est terminée (op est la dernière opération de r), ($r(ebd, es, op) \downarrow FinExec = vrai$), ER retourne un quadruplet formé de :

1. $r(ebd, es, op) \downarrow NouvelÉtatBD$: l'état de la base de données résultant de l'exécution de l'opération op sur la base dans l'état ebd ;

2. $Reflète(< r, es, t, c, e >, nv\_mts) et Reflète(< r, es, t, c, e >, nv\_mtsftd)$: les ensembles de modules de règles à traiter qui sont construits (1) en insérant dans

mts et $mtsftd$ les règles déclenchées, à l'instant tc+1, par les événements générés par l'exécution de l'opération $op : r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s$. Ces règles seront traitées en prenant en compte, par la fonction *Reflète*, le *mode de consommation des événements* de r.

3. nv_estd : l'ensemble de tous les événements de la transaction déclenchante : $estd \cup r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s$ avec $r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s$ les événements générés par l'exécution de op.

Si l'exécution de r n'est pas terminée :

1° $\mathcal{ER}$ traite les règles ayant un droit de préemption déclenchées par l'exécution de l'opération op à l'instant tc (fonction *Préemptibles*). Ces règles seront traitées à l'instant $tc + 1$;

2° après traitement de ces règles, $\mathcal{ER}$ s'applique elle-même pour exécuter, à l'instant $tc+1$, l'opération suivante de la partie action de $r : op+1$ sur l'état de la base de données résultant de l'exécution de $op : r(ebd, es, op) \downarrow NouvelÉtatBD$, les ensembles de modules de règles qui contiennent les règles déclenchées par les événements générés par l'exécution de $op : nv\_mts$ et nv_mtsftd. Les événements générés par l'exécution de op sont insérés dans l'ensemble contenant tous les événements de la transaction déclenchante ($estd \cup r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s$).

Notons que l'estampille prise en compte comme instant de déclenchement des règles déclenchées par l'exécution de r est tc qui est l'estampille courante associée à la dernière opération effectuée et non t qui est l'instant de déclenchement de r.

ExecRègle est définie comme suit :

$$\begin{aligned} ExecRègle : \mathcal{P}(M) &\rightarrow RT \times \mathbb{N} \times EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathbb{N} \times \mathcal{P}(E) \\ &\rightarrow EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathcal{P}(E) \end{aligned}$$

$$\begin{aligned} &ExecRègle\,[ms] = \underline{pppf}(\lambda\,\mathcal{ER}. \\ &\quad \lambda < r, es, t, c, e >, op, ebd, mts, mtsftd, tc, estd. \\ &\quad let\ nv\_estd = estd \cup r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s \end{aligned}$$

$$\begin{aligned} nv\_mts = &Déclenche(ms)(r(ebd, es, op) \downarrow NouvelÉtatBD, \\ &r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s, \\ &tc + 1, \\ &mts, \\ &mtsftd, \end{aligned}$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad nv\_estd \\
\qquad\qquad\qquad\qquad\qquad\qquad Module(< r, es, t, c, e >)) \downarrow 1 \\
nv\_mtsftd = Déclenche(ms)(r(ebd, es, op) \downarrow NouvelÉtatBD, \\
\qquad\qquad\qquad\qquad\qquad\qquad r(ebd, es, op) \downarrow ÉvtsGénérés, \\
\qquad\qquad\qquad\qquad\qquad\qquad tc + 1, \\
\qquad\qquad\qquad\qquad\qquad\qquad mts, \\
\qquad\qquad\qquad\qquad\qquad\qquad mtsftd, \\
\qquad\qquad\qquad\qquad\qquad\qquad nv\_estd \\
\qquad\qquad\qquad\qquad\qquad\qquad Module(< r, es, r, c, e >)) \downarrow 2
\end{array}
$$

$$
\begin{array}{l}
in\ si\ (r(ebd, es, op) \downarrow FinExec) \\
\quad alors\ < r(ebd, es, op) \downarrow NouvelÉtatBD, \\
\qquad\quad Reflète(< r, es, t, c, e >, nv\_mts), \\
\qquad\quad Reflète(< r, es, t, c, e >, nv\_mtsftd), \\
\qquad\quad nv\_estd > \\
\quad sinon\ si\ (Préemptibles(nv\_cms) \neq \emptyset) \\
\qquad\quad alors\ TraiteModules(ms, o_M, o_R) \\
\qquad\qquad\qquad\quad (r(ebd, es, op) \downarrow NouvelÉtatBD, \\
\qquad\qquad\qquad\quad Préemptibles(nv\_mts), \\
\qquad\qquad\qquad\quad nv\_mtsftd, \\
\qquad\qquad\qquad\quad tc + 1, \\
\qquad\qquad\qquad\quad nv\_estd) \\
\qquad\quad sinon\ \mathcal{ER}(< r, es, t, c, e >, \\
\qquad\qquad\qquad\quad op + 1, \\
\qquad\qquad\qquad\quad r(ebd, es, op) \downarrow NouvelÉtatBD, \\
\qquad\qquad\qquad\quad nv\_mts, \\
\qquad\qquad\qquad\quad nv\_mtsftd, \\
\qquad\qquad\qquad\quad tc + 1, \\
\qquad\qquad\qquad\quad nv\_estd) \\
\qquad\quad finsi \\
\quad finsi
\end{array}
$$

□

Reflète La fonction *Reflète* prend comme arguments (1) une règle à traiter $rt =< r, es, t, c, e >$ et (2) un ensemble mts de modules de règles à traiter. Elle retourne un

nouvel ensemble de modules de règles à traiter tel que :

- si r *consomme* les événements (*ModConsoÉvts(r)=consommation*), elle est supprimée ($\ominus$) de l'ensemble des modules de règles à traiter après son exécution ;
- si r *préserve* les événements, elle n'est pas supprimée de l'ensemble des modules de règles à traiter après son exécution mais simplement *marquée* (*e=exécuté*) afin de dénoter qu'elle a déjà été exécutée. Ceci permet de conserver les événements déclenchants de r entre deux traitements de r.

***Reflète* dénote donc la prise en compte du mode de consommation des événements d'une règle, par le modèle d'exécution, à la fin de l'exécution de cette règle.**

L'opérateur $\oplus$ permet d'insérer un élément dans un ensemble. L'opérateur $\ominus$ permet de supprimer un élément d'un ensemble. Nous ne donnons pas les définitions formelles de ces opérateurs qui sont triviales. *Reflète* est définie comme suit :

$Reflète : RT \times \mathcal{P}(MT) \rightarrow \mathcal{P}(MT)$

$Refète = \lambda rt, mts.$
$\quad let\ rt =< r, es, t, c, e >$
$\quad in\ si\ (ModConsoÉvts(r) = consommation)$
$\qquad alors\ mts \ominus rt$
$\qquad sinon\ let\ nv\_rt =< r, es, t, c, exécuté >$
$\qquad\qquad in\ (mts \ominus rt) \oplus nv\_rt$
$\qquad finsi$
□

Préemptives La fonction *Préemptives* prend comme argument un ensemble de modules de règles à traiter et retourne le sous-ensemble de cet ensemble qui contient les règles qui ont le *droit de préemption*. *Préemptives* est définie comme suit :

$Préemptives : \mathcal{P}(MT) \rightarrow \mathcal{P}(MT)$

$Préemptives[ms] = \lambda\{\{< r_i^I, es_i^I, t_i^I, c_i^I, e_i^I >\}\}.$
$\quad \{\{< r_j^I, es_j^I, t_j^I, c_j^I, e_j^I > \mid \{r_j^I\} \subset \{r_i^I\} \wedge ModPréemption(r_j^I) = premption\}$
□

EvalRègle Comme nous l'avons dit précédemment, nous considérons que l'évaluation des conditions ne modifie pas l'état de la base de données et ne produit pas

d'événements. L'évaluation d'une règle (retardée) en cours de transaction consiste donc simplement à la *transférer* de l'ensemble des modules de règles à traiter en cours de transaction vers l'ensemble des modules de règles à traiter en fin de transaction. L'évaluation d'une règle est dénotée par la fonction *EvalRègle* qui est définie comme suit :

$$EvalRègle : RT \times \mathcal{P}(MT) \times \mathcal{P}(MT) \to \mathcal{P}(MT) \times \mathcal{P}(MT)$$

$$EvalRègle = \lambda rt, mts, mtsftd.$$
$$< mts \ominus rt,\ mtsftd \oplus rt >$$

□

5.3.4 Construction des ensembles en fin de transaction

DéclencheFtd La fonction *DéclencheFtd* prend comme arguments (1) un ensemble de modules ms puis (2) un état de la BD ebd, (3) un ensemble d'événements es[6] produits à un instant (4) t dans la transaction déclenchante TD, (5) un ensemble de modules $mtsftd$ à traiter en fin de TD, (6) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de la transaction déclenchante hormis ceux de es et (7) un module déclenchant mt.

DéclencheFtd retourne (1) le nouvel ensemble des modules de règles à traiter au cours de TD qui sera toujours vide puisque l'on est à la fin de TD et (2) le nouvel ensemble des modules de règles à traiter à la fin de TD qui est la fusion de (a) $mtftd$ et (b) des règles *immédiates, retardées* et *différées* de ms déclenchées par es, (c) des règles *retardées* évaluées au cours de TD mais non exécutées et (d) des règles *différées* déclenchées au cours de TD. *DéclencheFtd* est définie comme suit :

$$DéclencheFtd : \mathcal{P}(M) \to EBD \times \mathcal{P}(E) \times \mathbb{N} \times \mathcal{P}(MT) \times \mathcal{P}(E) \times MT \to \mathcal{P}(MT) \times \mathcal{P}(MT)$$

$$DéclencheFtd[ms] = \lambda ebd, es, t, mtsftd, estd, mt.$$
$$let\ LesDeclImm = LesDéclenchées(ms)(immédiat, ebd, es, t, estd, mt)$$
$$LesDeclRet = LesDéclenchéees(ms)(retardé, ebd, es, t, estd, mt)$$
$$LesDeclDif = LesDéclenchées(ms)(différé, ebd, es, t, estd, mt)$$
$$in\ < \emptyset, FusionneModules($$
$$mtsftd,$$
$$FusionneModules($$

6. Notons que cet ensemble sera généralement vide hormis l'événement fin de transaction lui-même !

$LesDeclDif,$
$FusionneModules($
$LesDeclImm,$
$LesDeclRet)))$

□

5.3.5 Sélection d'une règle en fin de transaction

TraiteModulesFtd La fonction *TraiteModulesFtd* prend comme arguments (1) un ensemble de modules ms, (2) un ordre sur les modules $o_M$et (3) un ordre sur les règles o_R.

$$\begin{aligned} TraiteModulesFtd :& \mathcal{P}(M) \times O_M \times O_R \\ & \rightarrow EBD \times \mathcal{P}(MT) \times \mathbb{N} \times \times \mathbb{B} \times \mathcal{P}(E) \\ & \rightarrow (EBD \times \mathcal{P}(MT) \times \mathcal{P}(MT) \times \mathcal{P}(E)) \vee \{\perp\} \end{aligned}$$

$TraiteModulesFtd[ms, o_M, o_R] = \underline{pppf}(\lambda \mathcal{TMF}.$
$\lambda < ebd, mtsftd, tc, estd, ftd > .$
$si\ (SontDéclenchées(ms, mtsftd) = \emptyset)$
$alors\ < ebd, \emptyset, \emptyset, \emptyset >$
$sinon\ let\ rt = SelectRègle(o_R)(SelectModule(o_M)(mts))$
$in\ \mathcal{TMF}(TraiteRègleFtd(rt, ebd, mtsftd, tc, estd)))$
$finsi$

□

TraiteModulesFtd retourne le *plus petit point fixe* ($\underline{pppf}$) d'une fonction $\mathcal{TMF}$. $\mathcal{TMF}$ prend (1) un état de la BD ebd, (2) un ensemble de modules $mtsftd$ à traiter, (3) un entier tc qui est l'estampille courante et (4) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de TD.

Si aucune règle n'est (plus) à traiter dans $mtsftd$ alors $\mathcal{TMF}$ retourne $< ebd, \emptyset, mtsftd, \emptyset >$; sinon $\mathcal{TMF}$ sélectionne une règle rt à traiter parmi le module le plus prioritaire puis s'applique elle-même sur (1) le nouvel état de la BD, (2) le nouvel ensemble de modules à traiter en fin de TD et (3) le nouvel ensemble d'événements produits depuis le début de TD. (1), (2) et (3) résultent de l'exécution de rt sur ebd, $mtsftd$ et tc.

5.3.6 Traitement d'une règle en fin de transaction

TraiteRègleFtd La fonction *TraiteRègleFtd* prend comme arguments (1) un ensemble ms de modules de règles puis (2) une règle à traiter rt, (3) un état ebd de la BD, (4) un ensemble $mtsftd$ de modules de règles à traiter en fin de TD, (5) un entier tc qui est l'estampille courante et (6) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de TD. Elle retourne (1) un nouvel état de la base de données, (2) un nouvel ensemble de modules de règles à traiter et (3) un ensemble d'événements. (1), (2) et (3) résultent de l'exécution de la règle r.

Notons qu'en fin de transaction, le *mode d'exécution* des règles n'est plus réellement pris en compte. On peut considérer qu'en fin de transaction, toutes les règles sont *immédiates*. *TraiteRègle* est définie comme suit :

$$\begin{aligned} \mathit{TraiteRègleFtd}: \mathcal{P}(M) &\rightarrow RT \times EBD \times \mathcal{P}(MT) \times \mathbb{N} \times \mathcal{P}(E) \\ &\rightarrow EBD \times \mathcal{P}(MT) \times \mathcal{P}(E) \end{aligned}$$

$$\begin{aligned} &\mathit{TraiteRègleFtd}[ms] \;=\; \lambda < r, es, t, c, e >, ebd, mtsftd, estd. \\ &\quad \mathit{ExecRègleFtd}(ms)(rt, 1, ebd, mtsftd, t, estd) \end{aligned}$$

□

ExecRègleFtd La fonction *ExecRègleFtd* prend comme arguments un ensemble de modules de règles ms. *ExecRègleFtd* retourne le *plus petit point fixe* (*pppf*) d'une fonction $\mathcal{ERF}$. $\mathcal{ERF}$ prend comme arguments (1) une règle à traiter $rt =< r, es, t, c, e >$, (2) le numéro de l'opération de la partie action de r à exécuter, (3) un état de la BD ebd, (4) un ensemble de modules $mtsftd$ à traiter en fin de TD, (5) un entier tc qui est l'estampille courante et (6) un ensemble d'événements $estd$ qui est l'ensemble des événements produits depuis le début de TD.

Si l'exécution de r est terminée (op est la dernière opération de r), ($r(ebd, es, op) \downarrow FinExec = vrai$), $\mathcal{ERF}$ retourne un triplet formé de :

1. $r(ebd, es, op) \downarrow \mathit{NouvelÉtatBD}$: l'état de la base de données résultant de l'exécution de l'opération op sur la base dans l'état ebd ;
2. $\mathit{Reflète}(< r, es, t, c, e >, nv\_mtsftd)$: les ensembles de modules de règles à traiter qui sont construits (1) en insérant dans $mtsftd$ les règles déclenchées, à l'instant $tc + 1$, par les événements produits par l'exécution de l'opération op : $r(ebd, es, op) \downarrow \mathit{ÉvtsGénérés}$. Ces règles seront traitées en prenant en compte, par la fonction *Reflète*, le *mode de consommation des événements* de r.
3. nv_estd : l'ensemble de tous les évenements de la transaction déclenchante :

$estd \cup r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s$ avec $r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s$ les événements générés par l'exécution de *op*.

Si l'exécution de r n'est pas terminée :

1° $\mathcal{ERF}$ traite les règles ayant un droit de préemption déclenchées par l'exécution de l'opération *op* à l'instant *tc* (fonction *Préemptibles*). Ces règles seront traitées à l'instant $tc + 1$;

2° après traitement de ces règles, $\mathcal{ERF}$ s'applique à elle-même pour exécuter, à l'instant $tc+1$, l'opération suivante de la partie action de $r : op+1$ sur l'état de la base de données résultant de l'exécution de $op : r(ebd, es, op) \downarrow NouvelÉtatBD$, l'ensemble de modules de règles qui contiennent les règles déclenchées par les événements générés par l'exécution de *op* : nv_mtsftd. Les événements produits par l'exécution de *op* sont insérés dans l'ensemble contenant tous les événements de la transaction déclenchante ($estd \cup r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s$).

$$\begin{aligned} ExecRègleFtd : \mathcal{P}(M) &\rightarrow RT \times \mathbb{N} \times EBD \times \mathcal{P}(MT) \times \mathbb{N} \times \mathcal{P}(E) \\ &\rightarrow EBD \times \mathcal{P}(MT) \times \mathcal{P}(E) \end{aligned}$$

$$\begin{aligned}
&ExecRègle\,[ms] = \underline{pppf}(\lambda\, \mathcal{ERF}. \\
&\quad \lambda < r, es, t, c, e >, op, ebd, mtsftd, tc, estd. \\
&\quad let\ nv_estd = estd \cup r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s \\
&\qquad nv_mtsftd = D\acute{e}clencheFtd(ms)(r(ebd, es, op) \downarrow NouvelÉtatBD, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad r(ebd, es, op) \downarrow \acute{E}vtsG\acute{e}n\acute{e}r\acute{e}s, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad tc + 1, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad mtsftd, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad nv_estd, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad Module(< r, es, r, c, e >)) \downarrow 2 \\
&\quad in\ si\ (r(ebd, es, op) \downarrow FinExec) \\
&\qquad alors\ < r(ebd, es, op) \downarrow NouvelÉtatBD, \\
&\qquad\qquad\quad Reflète(< r, es, t, c, e >, nv_mtsftd), \\
&\qquad\qquad\quad nv_estd > \\
&\qquad sinon\ si\ (Préemptibles(nv_cmsftd) \neq \emptyset) \\
&\qquad\qquad\quad alors\ TraiteModulesFtd(ms, o_M, o_R) \\
&\qquad\qquad\qquad\qquad\quad (r(ebd, es, op) \downarrow NouvelÉtatBD, \\
&\qquad\qquad\qquad\qquad\quad\ Préemptibles(nv_mtsftd), \\
&\qquad\qquad\qquad\qquad\quad\ tc + 1, \\
&\qquad\qquad\qquad\qquad\quad\ nv_estd)
\end{aligned}$$

$sinon\ \mathcal{ERF}(< r, es, t, c, e >,$
$op + 1,$
$r(ebd, es, op) \downarrow NouvelÉtatBD,$
$nv\_mtsftd,$
$tc + 1,$
$nv\_estd)$
$finsi$
$finsi$

□

5.4 Comparaison avec d'autres approches formelles

Dans cette section, nous présentons les principaux travaux de formalisation de la sémantique des SGBD actifs, et plus particulièrement de leur modèle d'exécution. Notre objectif est d'évaluer et de comparer les *formalismes* utilisés dans ces différentes approches.

Dans les sections 5.4.1 à 5.4.5, nous cherchons uniquement à donner "la saveur" des différentes approches. Leur avantages et inconvénients respectifs sont discutés dans la section 5.4.6.

5.4.1 Sémantique opérationnelle et "evolving algebras"

[Beh94a] donne une sémantique opérationnelle d'ADL [Beh93] (Activity Description Language) ou plus précisement de la partie active d'ADL – qui est appelée A_{ADL} (Active ADL). Cette sémantique utilise les *evolving algebras* introduites par Gurevich [Gur91].

Une *evolving algebra* (EA) peut être vue comme une *machine abstraite* dont l'évolution dans le temps d'une *algèbre* – c'est-à-dire le passage d'un état de cette algèbre à un autre état – est décrite par un ensemble de *règles de transition* (Gurevich parle de *mises à jour gardées*). Une algèbre est définie par un ensemble d'*univers* et un ensemble de *fonctions* sur ces univers.

Les règles de A_{ADL} sont de la forme (EC)-A. La partie action (A) est décrite dans un langage proche de Modula-2. L'EA de A_{ADL} est donc basée sur l'EA de Modula-2 donnée dans [GM87].

[Beh94a] ne décrit pas le système complet mais se focalise sur le processus de détection des événements et le processus d'exécution des règles. Le processus d'exécution – qui seul nous intéresse – est décrit comme une machine abstraite qui sélectionne

et exécute les règles déclenchées par les événements en provenance du processus de détection.

Univers Les principaux univers utilisés sont :

- $EVENTQUEUE$: l'univers des queues d'événements – un événement est un élément de l'univers $OCCEVENT$;
- $RULE$: l'univers des règles ;
- $ACTION$: l'univers des actions (partie action des règles) ;
- $ACTIONSTACK$: l'univers des piles d'actions ;

Fonctions $EVENTQUEUE$ est munie des fonctions d'insertion et de suppression ; $ACTIONSTACK$ des fonctions d'empilement et de dépilement. $RULE$ est munie d'une fonction $action\_part : RULE \rightarrow ACTION$ qui exécute l'action d'une règle. On définit de plus la fonction $eventtriggerrule : OCCEVENT \rightarrow \mathcal{P}(RULE)$ qui retourne l'ensemble des règles déclenchées par un événement.

Un élément eq de $EVENTQUEUE$ contient les événements – primitifs et composites – en provenance de l'application, classés par ordre d'apparition. Un autre élément $etrq$ (*event-triggered-rule-queue*) de $EVENTQUEUE$ contient les références des règles déclenchées par chacun des événements de eq.

États La machine abstraite peut être dans les trois états :

- *eventprocessing* : elle est en train de reconnaître des événements ;
- *ruleselection* : elle est en train de choisir une règle ;
- et *actionexecution* : elle est en train d'exécuter une règle – ou une cascade de règles.

Transitions Les transitions d'états de la machine abstraite depuis l'état *eventprocessing* sont dictées par le processus suivant :

1. la règle de transition *Select-Leaf-Rule* sélectionne le premier événement e de $etrq$; si $etrq$ est vide, la machine reste dans l'état *eventprocessing*, sinon elle passe dans l'état *ruleselection* ;
2. la règle de transition *Select-Rule-Rule* sélectionne les règles déclenchées par l'événement e considéré (par la fonction *eventtriggerrule*) ; celles-ci sont stockées dans une queue aq de $ACTIONQUEUE$ selon leur priorité ; e est retiré de la queue $etrq$;

3. la première règle r de aq est empilée dans une pile as de $ACTIONSTACK$ qui modélise les déclenchements de règles en cascade ; la machine passe dans l'état *actionexecution* et l'exécution de r débute (fonction *action_part*) ;
4. si des règles sont déclenchées par l'exécution de r, elles sont empilées et exécutées récursivement ; lorsque as est vide, c'est-à-dire lorsque toutes les règles cascadées ont été exécutées, la machine repasse dans l'état *ruleselection* et l'on retourne à la phase 3 ; lorsque aq est vide, la machine repasse dans l'état *eventprocessing* et l'on retourne à la phase 1.

5.4.2 Sémantique opérationnelle et machines relationnelles

[Pic95] et [PV95] proposent un cadre formel afin d'étudier la puissance d'expression et la consistance sémantique des bases de données **relationnelles** actives. Deux modèles formels basés sur les *machines relationnelles* sont proposés :

- le *modèle abstrait* qui se focalise sur la sémantique du déclenchement qui relie le SGBD actif et une application (transaction) ;
- le *modèle générique* qui se concentre sur le modèle d'exécution d'un SGBDA lui-même.

Les machines relationnelles ont été introduites dans [AV91] et [AV95] pour représenter les langages du type C+SQL dans lesquels un langage de requêtes est *embarqué* dans un langage de programmation conventionnel (impératif). Une machine relationnelle est une *machine de Turing* augmentée d'un stockage relationnel.

Modèle abstrait Une machine relationnelle représente un *système de triggers*[7]. Les programmes externes sont aussi représentés comme des machines relationnelles avec des *états de déclenchement* qui transfèrent le contrôle vers le système de triggers. Pour ce faire, les langages utilisateurs courants (*FO, fixpoint, while, Datalog*, etc.) sont étendus pour permettre de spécifier explicitement ces états de déclenchements où le système de triggers prend le contrôle et réagit.

La sémantique (opérationnelle) d'un système de triggers est alors définie comme la transformation qui prend en entrée un programme externe et qui retourne une *mise-à-jour de la BD* qui résulte de l'exécution du programme externe *en conjonction* avec le système de triggers.

7. Les auteurs utilisent indifféremment – semble-t-il – les termes *trigger* et *règle active*. En l'absence d'informations à ce sujet, nous essayons ici de rester aussi proches que possible de [Pic95] et [PV95].

Modèle générique Les états de déclenchements ne sont plus réservés aux programmes externes mais sont également utilisés par les systèmes de triggers afin de prendre en compte les déclenchements en cascade. Pour le modèle d'exécution d'une règle, d'autres caractéristiques comme les *delta relations*, le *mode de couplage (immédiat, différé)* sont prises en compte.

Un programme de triggers[8] P est un quintuplet : $< Rules, cpl, ev, pri, Delta\text{-}type >$ dans lequel :

- $Rules$ est un ensemble de règles sur un schéma $sch(P)$;
- cpl est une fonction de $Rules$ vers $\{imm, def\}$ qui désigne le mode de couplage de chaque règle ;
- ev est une fonction de $Rules$ vers $\{R^+, R^-, R^{+-} | R \in sch(P)\}$ qui indique si les événements qui déclenchent les règles de P sont des insertions, des suppressions ou les deux ;
- pri est une fonction de $Rules$ dans $\{1, \ldots, |Rules|\}$[9] qui spécifie un ordre total sur les règles de P.

On manipule de plus deux queues q_{imm} et q_{def} pour les règles immédiates et différées respectivement. Lorsqu'une règle est déclenchée, elle est placée dans l'une ou l'autre de ces deux queues et traitée par une fonction *exec-program* qui utilise deux fonctions *exec-imm* et *exec-def*. Il y a des appels mutuellement récursifs entre *exec-program* et *exec-imm*. Les fonctions de manipulation des queues *add*, $merge_{imm}$ et $merge_{def}$ peuvent être adaptées pour prendre en compte différentes stratégies d'exécution.

Les modèles abstrait et générique sont utilisés pour décrire la sémantique de Postgres, Sybase, Starburst, ARDL et une *version relationnelle* de HiPAC [CBB+89] **moyennant des simplifications de ces systèmes et des hypothèses plus ou moins drastiques**. On considère, par exemple, uniquement le *mode d'exécution* au niveau du modèle d'exécution des règles (pas de mode de traitement des événements, de mode de consommation des événements, de mode de préemption, etc.).

5.4.3 Sémantique dénotationnelle

Comme nous l'avons vu, une *sémantique dénotationelle* [Ten76, Sto77, Sch88] est constituée d'un ensemble d'*algèbres* et d'un ensemble de *fonctions de valuation* sur ces algèbres. Une algèbre est définie comme un *domaine sémantique* et munie d'un ensemble d'*opérations*.

8. Programme de triggers≡ modèle d'exécution.
9. $|Rules|$ est le cardinal de $Rules$.

5.4.3.1 Sémantique dénotationnelle de Starburst

[Wid92] donne une sémantique dénotationnelle de Starburst en tant que langage de règles. Le concept central de [Wid92] est la notion d'*ensemble de mises à jour*. Une *mise à jour* représente à la fois les effets de l'exécution des opérations d'une transaction (d'une application) et les effets de l'exécution d'une règle. Une mise à jour est une insertion ou une suppression d'un n-uplet dans une relation[10]. Le point d'entrée de la sémantique dénotationnelle de Starburst est une fonction *sens* $\mathcal{M}$ (*meaning* en anglais) qui prend comme arguments un ensemble de règles et retourne la fonction qui transforme un *ensemble de modifications* et un *état de la base de données* en un nouvel état de la base qui résulte de l'exécution de ces règles.

Domaines Les domaines utilisés sont au nombre de cinq :

1. $\mathcal{S}$: le domaine des états de la base de données ;
2. Δ : le domaine des mises à jour ;
3. $\mathcal{R}$: le domaine des règles ; une règle r de $\mathcal{R}$ est une fonction :
   $r \ : \ \Delta \times \mathcal{S} \rightarrow \{true, false\} \times \Delta \times \mathcal{S}$
   dans lequel le booléen retourné indique si r est déclenchée par les éléments de l'ensemble de mises à jour donné en argument ;
4. $\mathcal{O}$: le domaine des ordres sur $\mathcal{R}$; $\mathcal{O}$ représente les priorités entre règles ;
5. $\mathcal{RT}$: le domaine des ensembles de couples *(règle, ensemble de mises à jour)* – $\mathcal{RT} \ \subset \ \mathcal{P}(\mathcal{R} \times \Delta)$.

Fonctions de valuation Les fonctions intermédiaires suivantes sont également utilisées afin de définir la fonction $\mathcal{M}$:

- *Distrib* qui *distribue* un ensemble de mises à jour sur un ensemble de règles ;
- *Elligible*, *Choose-Triggered* et *Select* qui permettent de calculer l'ensemble des règles déclenchées par un ensemble de mises à jour puis de sélectionner la règle la plus prioritaire dans cet ensemble ;
- *Net-Effect* et *Add-Changes* qui permettent de représenter les effets de l'exécution d'une règle – par la fonction *Run-Rule* – sur l'exécution des autres règles en tenant compte de l'*effet net* des opérations.

5.4.3.2 Sémantique dénotationnelle de NAOS

Nous donnons dans [CC95c] une sémantique dénotationnelle du SGBD actif NAOS. Dans ce travail, nous nous sommes volontairement inspirés de [Wid92] décrit ci-dessus

10. En ce qui concerne les "vraies" mises à jour de n-uplets, [Wid92] considère qu'elles peuvent être modélisées par une combinaison d'insertion et de suppression.

afin de montrer, d'une part qu'il était également[11] possible de définir la sémantique formelle d'un système de règles actives **pour SGBD à objets** ; et d'autre part, afin de souligner les similitudes et les différences induites par le modèle de données – relationnel pour Starburst, à objets pour NAOS – sur l'expression dénotationnelle des modèles d'exécution des SGBD actifs.

Le point d'entrée de la sémantique dénotationnelle de NAOS est une fonction $\mathcal{M}$ qui prend comme argument un ensemble de règles et retourne une fonction qui prend comme arguments un état de la base de donnée, un ensemble d'événements (produits par l'exécution d'une transaction ou d'une règle) ; et qui retourne le nouvel état de la base qui résulte de l'exécution (en cascade) des règles déclenchées par l'ensemble d'événements donné.

Domaines Les domaines utilisés sont les suivants :

1. DBS : le domaine des états d'une base de données ;
2. E le domaine des événements. $E = OPT \times EID \times P \times V \times \Delta$. Soit e un événement de E, $e = (opt, eid, p, v, \delta)$ dans lequel :
   - $opt \in OPT$ est un type d'opération (création, destruction, lecture, mise-à-jour, appel de méthode, etc.) ;
   - $eid \in EID$ est un identificateur d'entité ;
   - $p \in P$ est un chemin dans l'entité identifiée par eid ;
   - $v \in V$ est la valeur de l'entité identifiée par eid ($p = \perp$) ou la sous-entité qui peut être accédée par p ($p \neq \perp$) ;
   - $\delta \in \Delta$ est la valeur de la *mise-à-jour* de l'entité identifiée par eid ou de la sous-entité accédée par p.
3. S le domaine des ensembles d'événements : $S = \mathcal{P}(E)$;
4. O_E : le domaine des ordres sur les événements, O_E permet d'ordonner les événements dans le temps ;
5. R : le domaine des règles. $R = IR \uplus DR$ avec IR le domaine des règles immédiates et DR le domaine des règles différées. Par exemple, r une règle immédiate de IR est une fonction : $r : DBS \times E \rightarrow \mathbb{B} \times DBS \times S$ dans lequel le booléen indique si r est déclenchée par l'événement donné en argument. Si la règle est déclenchée par cet événement, r retourne le nouvel état de la base et un nouvel ensemble d'événements qui résulte de l'exécution de r ;
6. O_R : le domaine des ordres sur les règles, O_R représente les priorités entre les règles ;

11. Un des objectifs de [Wid92] était de prouver qu'il était possible de définir *proprement*, c'est-à-dire de manière formelle, la sémantique d'un SGBD actif.

7. *RSC* : le domaine des triplets *règles-événements-cycle*. Soit rsc un triplet de RSC, $rsc =< r, s, c >$ dans lequel s est l'ensemble d'événements considéré pour l'exécution de r dans le cycle d'exécution c.

Fonctions de valuation $\mathcal{M}$ est définie à partir des fonctions intermédiaires suivantes :

- *Distrib* qui distribue un ensemble d'événements sur un ensemble de règles ;
- *Eligible-I, Eligible-D, Choose-Triggered, Select-I, Select-D, Greatest-Cycle* et *First-Event-Rule* qui permettent de calculer l'ensemble des règles immédiates et différées déclenchées par un ensemble d'événements puis de sélectionner la règles la plus prioritaire dans cet ensemble ;
- *Net-Effect*, *Compose*, *Compose-Events*, *Reflect* et *Reflect-e* qui permettent de représenter les effets de l'exécution d'une règle – par la fonction *Execute* – sur l'exécution des autres règles en tenant compte de l'effet net de opérations.

L'influence du modèle de données – relationnel pour Starburst, à objets pour NAOS – se traduit principalement au niveau des domaines des états d'une base de données et des domaines qui décrivent les situations qui peuvent déclencher les règles. Dans les deux travaux, les domaines des états d'une base – S pour Starburst, DBS pour NAOS – ne sont pas primordiaux en ce sens que les deux travaux s'attachent uniquement à décrire les modèles d'exécution des règles et non les SGBD actifs en totalité. En revanche, la description des situations qui peuvent déclencher des règles est plus importante. Il s'agit de Δ le domaine des mises-à-jours pour Starburst et S le domaine des ensembles d'événements pour NAOS. Les éléments de S sont plus complexes que les éléments de Δ car les types d'opérations sur les entités susceptibles de générer des événements sont plus nombreux dans le modèle d'événements de NAOS que dans celui de Starburst. En effet, dans Starburst, les manipulations de données se limitent à des insertions et suppressions dans des relations, alors que NAOS prend en compte les créations, les destructions, les accès (en lecture) et les mises-à-jours d'entités ou parties d'entités, les insertions et suppressions dans les collections, les changements d'état de persistence des entités et les appels de méthodes sur les objets.

Le modèle de données considéré a également une influence sur le calcul de l'effet net. Ce calcul est plus complexe et moins intuitif pour NAOS dans lequel – encore une fois – les manipulations de données ne se limitent pas à des insertions et suppressions dans des relations. Signalons d'ailleurs, qu'à notre connaissance, le travail que nous avons fait dans [CC95c] est la première formalisation de l'effet net dans les SGBD à objets actifs.

En ce qui concerne les modèles d'exécutions et les fonctions de valuation qui les représentent, on trouve dans les deux travaux les trois même ensembles de fonctions :

- celles qui s'intéressent au déclenchement des règles : distribution des événements sur les règles ;
- celles qui permettent de sélectionner une règle parmi l'ensemble des règles déclenchées ;
- et celles qui s'intéressent à l'exécution proprement dite d'une règle sélectionnée et à l'influence de cette exécution sur l'exécution des autres règles.

Le modèle d'exécution plus complexe de NAOS, qui considère deux types de règles avec des sémantiques très différentes (les règles immédiates et les règles différées), conduit à une sorte de "duplication" d'un certain nombre de fonctions : *Select-I* et *Select-D* pour sélectionner la prochaine règle immédiate ou différée à exécuter, par exemple.

5.4.4 Spécification formelle en Object-Z

[CPW95a] et [CPW95b] proposent un cadre formel pour spécifier la sémantique de différents SGBD actifs avec l'objectif de comparer précisément ces systèmes. À ce jour, ce cadre formel a été utilisé pour définir la sémantique de Starburst, Postgres et Ariel.

Le cadre proposé est basé sur le formalisme **Object-Z** [DKRS91, Ros92] qui est lui-même une *extension objet* de **Z** [Spi92].

Z est une notation basée sur la théorie des ensembles et sur la logique du premier ordre avec des notations pour la logique des prédicats, les ensembles, les relations, les fonctions, les opérations, etc. Une notation appelée *schéma* a été introduite par C. Morgan pour permettre la réutilisation du texte mathématique par un mécanisme d'*inclusion* d'un schéma dans un autre. Un schéma **Z** est une notation graphique à deux dimensions : les *variables* et les *contraintes*. Un *schéma* contient la déclaration de *variables d'intérêt*, avec leur *type*. Ces variables sont *mutuellement contraintes*. On utilise les schémas pour spécifier des *états* et des *transitions* entre ces états On définit un *calcul des états* qui permet de composer des états par inclusion, union, intersection et négation. Ce calcul des états est utilisé pour spécifier des *opérations*. Une opération est définie par un ensemble de transitions entre états.

Les spécifications **Object-Z** sont construites à partir d'un certain nombre de *classes*. Une classe est un schéma **Z** particulier qui encapsule un schéma et un ensemble d'opérations sur ce schéma. Les opérations d'une classe correspondent aux

méthodes des langages à objets. Les classes peuvent bien sûr être reliées par des liens de *composition* et d'*héritage*.

Dans l'utilisation d'**Object-Z** pour la spécification de la sémantique des SGBD actifs, une description abstraite d'un SGBD actif générique est *raffinée* progressivement par héritage (ou sous-typage) afin de converger vers les fonctionnalités d'un système particulier. À titre d'exemple, on trouve dans le SGBD générique les classes :

- *Object* et *ObjectBase* qui modélisent les états d'une base de données,
- *EventClass*, *SEventClass*, *Log*, *SLog*, etc. qui modélisent les événements, ensembles d'événements, etc.
- *Rule*, *RuleBase*, *PartialOrder* qui modélisent les règles, les ensembles de règles et les priorités entre règles ;
- *Activedatabase* qui modélise le modèle d'exécution d'un SGBD actif avec des opérations comme *Triggered*, *Choose*, *PickandRun*, *RunRule* qui sélectionnent et exécutent les règles.

5.4.5 Sémantique semi-formelle et sémantique opérationnelle

Comme [CPW95a] et [CPW95b] que nous venons d'évoquer, l'objectif de [FT95], [CFPT95c] et [CFPT95b] est de proposer une approche structurée pour la définition de la sémantique des SGBD actifs avec l'objectif de comparer précisément ces systèmes. Les auteurs proposent dans [FT95] tout d'abord une taxonomie des modèles d'exécution pour SGBD actifs dont nous avons déjà parlée dans la section 3.4.1. Cette taxonomie est ensuite utilisée pour décrire la sémantique des systèmes étudiés (SQL3, Starburst, Postgres et Chimera) par :

1. une sémantique informelle en langage naturel dans laquelle les systèmes sont situés par rapport à la taxonomie proposée ;
2. une sémantique *semi-formelle* appelée *format Événement-Condition-Action Étendu (EECA en anglais)* qui est une syntaxe permettant de représenter tous les paramètres du modèle d'exécution d'une règle que propose la taxonomie de [FT95] (même s'il ne sont pas tous explicitement pris en compte par tous les systèmes) ;
   Exemple :

```
define granularity S/O
       EC deferred
       interruptable False
       comsumption-scope local
       consumption-time execution
rule Règle-Starburst
```

```
event: ...
condition: ...
action: ...
```

Cet exemple montre comment une règle Starburst est représentée dans le format EECA. Dans la clause `define`, on définit le modèle d'exécution de la règle. Cette règle – comme toutes les règles dans Starburst – a un mode de traitement ensembliste des événements (`granularity S/O`), un mode d'exécution différé (`EC deferred`), n'est pas interruptible (`interruptable False`), consomme les événements localement (`comsumption-scope local`) à l'exécution (`consumption -time exécution`) ;

3. une sémantique formelle (opérationnelle) du modèle d'exécution des règles. Règles qui sont représentées dans un format appelé *core* (noyau).

5.4.6 Discussion

[PCFW95] est une étude comparative des formalismes utilisés pour spécifier les SGBD actifs. Elle recoupe approximativement la nôtre : elle ne prend pas en compte les différentes sémantiques opérationnelles dont nous parlons (evolving algebras, machines relationnelles) mais considère des modélisations par la théorie des graphes ou les réseaux de Petri. Elle présente les forces et les faiblesses de chacune des approches pour représenter trois aspects d'un SGBD actif (cf. chap. 2) : le modèle de règles, le modèle d'exécution et la gestion des règles actives. Cette étude présente des conclusions intéressantes :

1. les différentes propositions présentent rarement la spécification formelle d'un SGBDA complet (cf. chap. 2) ; principalement parce que plusieurs formalismes semblent nécessaires pour cette tâche (aucun formalisme ne semble pouvoir couvrir tous les aspects d'un SGBD actif) ;
2. l'implantation des systèmes découlent rarement de leur spécification formelle alors que la spécification formelle découle parfois de l'implantation comme c'est le cas pour Starburst et NAOS. Les potentialités de la sémantique formelle sont donc, en réalité, rarement utilisées.

En ce qui concerne les modèles d'exécution, [PCFW95] donne des indices sur la *correction* et la *maniabilité*[12] (ou *utilisabilité*) des différents formalismes pour décrire

12. Dans le modèle qualimétrique Facteurs-Critères-Métriques (FCM) de McCall, la correction et la maniabilité d'un système sont des facteurs de qualité influencés respectivement par les critères de qualité : cohérence, complétude, simplicité, concision, clarté et précision. Voir [CRW77] et [For] pour une description plus détaillée du modèle FCM.

quatre dimensions de ces modèles : le *mode de traitement des événements*, le *mode d'exécution*, la *prise en compte de l'effet net* et la *stratégie d'exécution* d'un ensemble de règles : la sémantique dénotationnelle et le formalisme **Object-Z** sont équivalents sur ces quatre aspects tandis que la sémantique opérationnelle permet plus difficilement d'exprimer la prise en compte de l'effet net et la stratégie d'exécution d'un ensemble de règles.

De notre propre expérience, il ressort que la sémantique opérationnelle – sous ses différentes formes – semble plus naturelle et bien adaptée à la description des modèles d'exécution, c'est-à-dire à la *mécanique* de l'exécution. Toutefois, comme nous l'avons signalé dans les sections précédentes, les bons résultats obtenus par la sémantique opérationnelle l'ont été : soit du fait de la simplicité des modèles eux-mêmes, dans le cas d'ADL, Starburst ou Postgres ; soit au prix de simplications plus ou moins importantes du modèle d'exécution des systèmes considérés, c'est la cas notamment de la description du modèle d'exécution de HiPAC par les machines relationnelles dans [Pic95]. Dans le cas de modèles d'exécution plus *complexes* – comme HiPAC, Sentinel, REACH, EXACT, Chimera, Peplom^{ad} ou NAOS – il y a en quelque sorte une *explosion* du nombre d'états et de transitions qui rend la lecture d'une sémantique opérationnelle très ardue.

Le formalisme **Object-Z**, grâce à l'héritage, qui permet la réutilisation et les *raffinements successifs*, semble être bien adapté pour prendre en compte des modèles d'exécution complexes. Il permet en outre de spécifier des modèles d'exécution très facilement extensibles. Toutefois, **Object-Z** n'introduit aucune notation pour décrire des algorithmes. Il est donc le formalisme qui nécessite le plus d'effort pour aller vers une implantation.

La sémantique dénotationnelle est la plus abstraite, et à priori, la plus difficile à appréhender. Cependant, elle fournit des descriptions compactes qui permettent d'appréhender des modèles d'exécution complexes. Le faible nombre de concepts manipulés – ensembles et fonctions – permet, en outre, d'isoler la complexité des modèles d'exécution décrits de la compléxité de la description elle-même. Les trois modèles d'exécution décrit par la sémantique dénotationnelle : Starburst, NAOS et Fl'are illsutrent la capacité de ce formalisme à décrire des modèles de plus en plus complexe. Par exemple, dans la description de Fl'are, on trouve une prise en compte du *synchronisme*, c'est-à-dire la prise en compte d'une certaine forme du temps, entre l'exécutions des règles – dont certaines sont interrompues par l'exécution d'autres règles – et l'exécution des transactions. Ce point n'est pas abordé dans les descriptions de Starburst et NAOS.

En résumé, nous pensons comme [PCFW95] qu'un seul formalisme particulier ne peut permettre à lui seul de décrire facilement un SGBD actif complet. Nous allons plus loin en pensant que même pour un aspect donné d'un SGBD actif, comme le modèle d'exécution, un seul formalisme ne peut être utilisé pour décrire tous les modèles d'exécution actuels ou à venir, et que le choix d'un formalisme dépend de la complexité des modèles d'exécution considérés. Pour un modèle "complexe", nous pensons que la sémantique dénotationnelle est particulièrement bien adaptée. La sémantique dénotationnelle, du fait de son haut niveau d'abstraction, est un outil utile aussi bien pour un utilisateur (programmeur) que pour un concepteur de SGBD actif. Pour ce dernier, une sémantique dénotationnelle d'un modèle d'exécution peut même, éventuellement, donner des pistes pour une implantation de ce modèle, c'est-à-dire être considérée comme une *spécification implantable*. C'est pour ces raisons que nous avons choisi d'utiliser la sémantique dénotationelle pour décrire de manière formelle le modèle d'exécution paramétrique Fl'are.

5.5 Conclusion

Dans ce chapitre, nous avons donné une sémantique formelle du modèle d'exécution paramétrique Fl'are qui **accompagne** la sémantique informelle qui a fait l'objet du chapitre précédent. Sémantique formelle et informelle sont indissociables et nécessaires, conjointement, à la définition de Fl'are qui est un modèle d'exécution complexe.

Les apports de la sémantique formelle dans notre travail sont les suivants :

1° la sémantique formelle fournit une description complète, précise et sans ambiguïté de Fl'are que n'offre pas une description informelle ;

2° elle suggère des solutions en vue d'une implantation de Fl'are;

3° l'approfondissement necessité par ce travail nous a fourni "un retour" (feedback) sur le modèle lui-même : découverte d'incohérences dans le cas de la stratégie d'exécution *par choix* qui faisait partie initialement du modèle et que nous avons ensuite écartée (cf 4.5.2) ; ou mise en évidence des sources d'ordonnancement pour l'établissement des stratégies d'exécution intra-module – point qui peut d'ailleurs donner lieu à de futurs développements.

Les caractéristiques de la sémantique dénotationnelle qui nous ont poussé à choisir ce formalisme sont les suivants :

1° la sémantique dénotationnelle est relativement ardue au premier abord mais devient rapidemment *naturelle* et facile à appréhender du fait du faible nombre de concepts manipulés : ensembles et fonctions ;

2° elle fournit une description compacte et concise ; ceci est un avantage sur les sémantiques opérationnelles, par exemple, dans lesquelles on est vite débordé par le nombre et *l'éclatement* des états et des transitions ;

3° elle fournit une description *élégante*, par exemple, dans la description des différentes stratégies d'exécution proposées (fonction *SelectRègle*) ou la description des interactions entre l'exécution d'une application et l'exécution des règles (récursion mutuelle entre les fonctions *Considère-Module* et *Exec-Règle*) ;

4° enfin, elle fournit un algorithme ou programme directement implantable dans un langage fonctionnel (LISP, ML, etc.), et facilement implantable dans un langage impératif. Elle constitue une donc une spécification implantable de Fl'are, c'est-à-dire une base pour son implantation ; implantation qui fait l'objet du prochain chapitre.

Chapitre 6

Expérimentation

> ***Inventeur*** *n. Personne qui fait un ingénieux arrangement de roues, de leviers et de ressorts, et qui croit que c'est la civilisation.*
> ***Ambrose Bierce - "Le dictionnaire du Diable"***

Notre travail est centré sur l'exécution de règles actives. Il s'est concrétisé par la proposition d'un modèle d'exécution paramétrique. La sémantique formelle de Fl'are, présentée dans le chapitre précédent, constitue une première validation, que l'on peut qualifier de "théorique".

Pour valider, ou du moins évaluer, notre proposition de manière plus expérimentale, il nous faut disposer de mécanismes permettant de définir des règles et de détecter les événements qui déclenchent ces règles. Plusieurs solutions s'offrent à nous. Nous pouvons évidemment spécifier et implanter ces mécanismes pour aboutir à un SGBD actif complet. Nous pouvons également simuler ces mécanismes qui ne concernent pas directement le processus d'exécution des règles. Enfin, nous pouvons nous placer dans un contexte qui offrent ces mécanismes et rester ainsi focalisés sur l'exécution des règles. C'est cette dernière solution que nous avons retenue.

Pour cela, nous utilisons NAOS (Native Active Object System), un système de règles actives pour le SGBD O_2 développé depuis 1992 à l'université de Grenoble. NAOS a différents composants qui permettent de définir des règles, de détecter des événements (internes, utilisateurs, temporels et composites), et finalement d'exécuter les règles déclenchées par ces événements. L'expérimentation que nous présentons ici consiste à utiliser NAOS comme support pour le développement de Fl'are. Nous proposons de remplacer le moteur d'exécution de NAOS par Fl'are.

La section 6.1 introduit le système NAOS, tandis que la section 6.2 détaille son architecture et son implantation. La section 6.3 décrit ensuite la mise en œuvre de Fl'are *à l'intérieur* de NAOS. Enfin, la section 6.4 conclut ce chapitre.

6.1 Contexte de l'expérimentation

Le système NAOS permet de définir des règles actives, puis d'exécuter ces règles dans le cadre d'applications O_2. Il a été développé en considérant le SGBD O_2 comme noyau de gestion des données et d'exécution des applications. Il s'intègre dans l'architecture modulaire d'O_2 et constitue un nouvel outil les applications. NAOS est développé au sein de l'équipe STORM du laboratoire LSR-IMAG à l'Université de Grenoble depuis septembre 1992. Il a été en grande partie financé par le projet Européen GOODSTEP [GOO94] (Projet Esprit III - no. 6115) [CHCA94]. Les aspects caractéristiques de NAOS concernent son intégration dans le modèle à objets d'O_2 ; le modèle et le processus de détection des événements [CC96a, CC96b] qui est intégré dans le moteur d'O_2 (et non au-dessus d'O_2) ; et son modèle d'exécution [CCS94a, CCS94b, CC95c], intégré dans l'architecture client-serveur d'O_2. Le prototype NAOS a fait l'objet de plusieurs démonstrations [CC95a, CC95b, CCR96].

Nous présentons maintenant l'intégration du modèle de règles NAOS dans le modèle de données O_2, le modèle et le langage de règles puis, de manière informelle, le modèle d'exécution de NAOS. Cette présentation est issue en partie de [Col96].

6.1.1 Règles et modèle de données

Nous montrons ici de quelle manière les règles se comportent vis-à-vis des concepts O_2 tels que schémas, classes, persistance, encapsulation et réutilisabilité.

6.1.1.1 Caractéristiques d'O2

O_2 offre les concepts orienté-objets de l'ODMG (Object Dayabase Managment Group [CAD+94]. Les deux principales originalités de ce modèle sont d'une part la formalisation qui en a été donnée [LRV92], et d'autre part la totale orthogonalité des constructeurs utilisés pour définir la structure des objets et des valeurs (ils peuvent être combinés à tous les niveaux).

Dans la suite de cette section, nous donnons les principales caractéristiques du modèle de données O_2. Une description plus complète du système O_2 est donnée dans [BDK92].

Schémas et bases La définition d'une application O_2 s'effectue en plusieurs étapes : (i) la définition des classes, (ii) la définition des objets et valeurs nommés, (iii) la définition du corps des méthodes, (iv) la définition des variables globales et des programmes qui constituent l'application.

Les composants d'une application appartiennent à un schéma et manipulent les données stockées dans les bases associées à ce schéma.

Un schéma O_2 est un ensemble de définitions : définition d'un ensemble de classes inter-reliées par des liens d'héritage ou de composition, définition des types, fonctions, programmes et applications. Un schéma définit la structure et le comportement des données (objets et valeurs) qui seront stockées dans la ou les bases qui lui sont associées.

Objets et valeurs Une originalité du système O_2 est la distinction entre objets et valeurs : un objet est une instance d'une classe, une valeur est une instance d'un type.

Un objet a un identificateur unique, une valeur et un comportement défini par ses méthodes. Un objet peut être partagé (référencé) par plusieurs entités (objets ou valeurs).

Un type est défini récursivement en utilisant les types atomiques (integer, boolean, char, string, real et bits), les classes et des trois constructeurs (tuple, list et set).

Une classe décrit la structure et le comportement d'un ensemble d'objets. La partie structurelle d'une classe est définie par son type et la partie comportementale par les méthodes. Une classe n'est pas un objet comme dans d'autres systèmes ou langages objets (Smalltalk, Objective-C) du fait de l'absence de méta-niveau.

Racines de persistance D'une manière générale, les objets et valeurs créés durant l'exécution d'un programme ne sont pas persistants. Pour devenir persistants, c'est-à-dire être stockés dans une base, ces objets et valeurs doivent être attachés directement ou indirectement à une (ou plusieurs) racine de persistance. Ces racines sont décrites par des noms (des poignées) auxquels sont attachés des objets et des valeurs. Les définitions de racines de persistance appartiennent à un schéma.

Héritage Le mécanisme d'héritage est basé sur une relation de sous-typage multiple (substitution). Les classes d'un schéma sont organisées en hiérarchie : une classe peut posséder plusieurs sous-classes ou plusieurs super-classes (héritage multiple). L'héritage est à la fois structurel et comportemental. Une sous-classe hérite de la structure de ses super-classes qu'elle peut enrichir ou modifier, et de leurs méthodes qu'elle peut redéfinir (surcharge sémantique).

Programmes et transactions Une application O_2 appartient à un schéma. Son exécution nécessite la mise en place du schéma auquel elle appartient et la référence à une ou plusieurs bases associées à ce schéma. Les programmes et transactions peuvent accéder aux données des bases. Ils peuvent appliquer des méthodes sur des objets de l'application, consulter et modifier des valeurs. Pour assurer la cohérence des bases, les objets persistants ne peuvent être mis à jour qu'à l'intérieur d'une transaction.

Le modèle de transactions du système O_2 est un modèle classique. Les commandes transaction,validate, commit, abort permettent respectivement d'entrer en mode transactionnel, de valider les mises à jour effectuées pendant la transaction courante et de continuer l'exécution du code dans lequel la transaction a débutée, de valider les mises à jour effectuées pendant la transaction courante et de stopper l'exécution du code dans lequel la transaction a débutée, et d'annuler toutes les mises à jour effectuées sur les objets, valeurs et variables globales pendant la transaction courante.

Encapsulation et réutilisabilité L'encapsulation est définie à plusieurs niveaux. Tout d'abord, les propriétés (attributs et méthodes) sont, par défaut, privées dans leur classe, c'est-à-dire visibles uniquement dans la classe (et ses sous-classes) dans laquelle elles sont définies. De la même manière, les programmes et transactions sont privés dans leur application. Les éléments d'un schéma sont privés dans ce schéma et ne peuvent pas être utilisés dans un autre schéma.

Cependant, les propriétés peuvent être rendue publiques, et par souci de réutilisabilité, O_2 fournit un mécanisme d'import/export : un schéma peut exporter des classes, des objet et valeurs nommés et autoriser ainsi d'autres schémas à les utiliser en les important.

6.1.1.2 Règles NAOS

Règles et schémas Comme le montre la figure 6.1, une règle appartient à un schéma. Elle est définie au même niveau que les classes ou les applications. L'exécution d'une règle se déroule dans le contexte de l'exécution d'une application qui interagit avec une ou plusieurs bases. On parle de règles externes, par opposition aux règles internes qui sont définies comme des propriétés d'une classe (cf. chapitre 2).

Les règles permettent de structurer une application. Elles peuvent être considérées à un plus haut niveau que les programmes, les fonctions et les méthodes. C'est pourquoi les règles sont isolées des programmes et des méthodes : une règle peut être manipulée uniquement par une autre règle. Par exemple, la désactivation et la réactivation d'une règle ne peuvent pas être effectuées dans le corps d'un programme ou

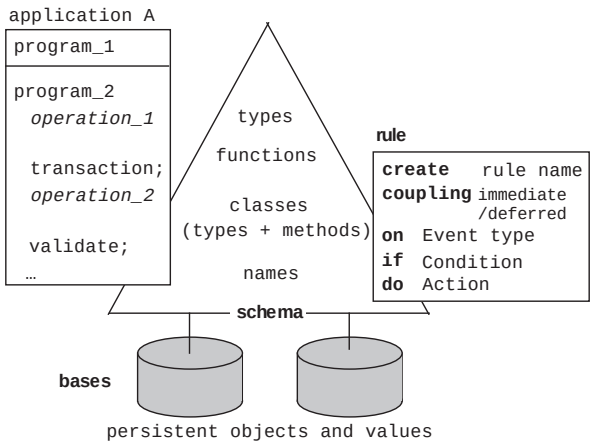

FIG. 6.1 – *Intégration des règles NAOS dans le SGBD O_2*

d'une méthode mais uniquement dans la partie Action d'une règle.

Règles et persistance Dans le modèle de données O_2, la persistance est transparente pour le programmeur d'applications. Ce principe est respecté par le mécanisme de règles : un événement peut être produit par une manipulation d'entités, que ces entités soient persistantes ou non.

Règles et encapsulation Le mécanisme de règles respecte l'encapsulation : seules les opérations autorisées sur des objets ou des valeurs peuvent produire des événements valides. De telles opérations sont les accès (dont mises à jour) à des attributs publics d'objet, les appels de méthodes publiques sur des objets.

6.1.2 Définition d'une règle

La structure générale de la définition d'une règle est la suivante:

[create] rule < *Identificateur de règle* >
[precedes < *Liste de règles* >]
[coupling < *Mode d'exécution* >]
on < *Spécification de type d'événement* >
[with < *Identificateur de delta-structure* >]
[if < *Condition* >]
do [instead] < *Action* >

Un exemple de règle est donné dans la figure **??**. Il est utilisé pour expliquer les divers aspects d'une définition de règle. Considérons le schéma O_2 de la figure 6.2.

```
class Personne public type tuple (
        nom : tuple(nom_famille : string, prenom : string),
        conjoint : Personne,
        age: integer
        enfants : set(tuple(prenom : string, age : integer)))
end;

class Employe inherits Personne
public type tuple (
        profil : list(string),
        salaire : integer)
end;

name Le_chef : Employe;
```

FIG. 6.2 – *Un schéma O_2*

La classe Personne représente des personnes ayant un nom, un age, un conjoint et des enfants. La classe Employe est une sous-classe de Personne et représente les employés d'une entreprise qui ont un profil et un salaire. L'objet nommé Le_chef représente un employé particulier.

La règle Maj_Emp_Sal de la figure 6.3 implante une contrainte d'intégrité dynamique : *"le salaire d'un employé ne peut pas diminuer"*. La règle Maj_Emp_Sal est exécutée avant chaque mise à jour de l'attribut salaire d'un objet de la classe Employe. Si la nouvelle valeur du salaire est inférieure à la valeur initiale – ce qui est testé dans la partie Condition de la règle – un message d'erreur est affiché et l'opération de mise à jour n'est pas effectuée (clause instead). La clause on décrit le type d'événement (before, update Employe->salaire). Un événement de ce type est produit lorsque l'attribut salaire d'un objet de la classe Employe est mis à jour. Cet objet, qui représente l'environnement de l'événement déclenchant, est accessible via la delta structure e. Les opérateurs current et new permettent d'accéder respectivement à la valeur courante (au moment de l'exécution de la règle) et à la nouvelle valeur de p (si l'opération de mise-à-jour a bel et bien lieu).

```
rule Maj_Emp_Sal
on before update Employe -> salaire with e
if new(e) -> salaire < current(e) -> salaire
do instead {
            display( "Mise à jour du salaire de l'employé "
                    + e -> nom + "refusée." );
}
```

FIG. 6.3 – *Une règle active NAOS*

6.1.2.1 Spécification de type d'événement

Un type d'événement décrit une situation particulière qui peut être reconnue par le système. Nous avons identifié des types d'événements primitifs qui sont principalement associés aux manipulation d'entités (objets et valeurs) et des types d'événements composites qui caractérisent des compositions d'événements.

Un *intervalle de validité* est associé à chaque type d'événement. Il permet de déterminer l'intervalle pendant lequel un événement du type peut se produire et déclencher une règle. Dans NAOS, les intervalles de validité sont implicites, i.e. déterminés en fonction de l'unité de production des événements. L'unité de production des événements et d'exécution des règles est la transaction (déclenchante).

Événements primitifs : Il existe quatre catégories d'événements primitifs :

- les *événements de manipulation d'entités* qui sont associés aux manipulations d'entités ou parties d'entités (attributs) : créations et destructions d'objets, modifications d'objets et de valeurs, insertions et suppressions dans des collections, appels de méthodes ;
- les *événements transactionnels* qui sont associés au début et à la fin de l'exécution des transactions ;
- les *événements applicatifs* qui sont produits par l'exécution d'une application O_2 (exécution de programmes et d'applications, et événements utilisateur) ;
- les *événements système* qui sont produits par l'exécution d'applications ou d'outils extérieurs à O_2, qui interagissent avec une application O_2 via le système d'exploitation. On trouve dans cette catégorie les événements temporels.

Dans un type d'événements caractérisant des opérations sur les données on précise via les modificateurs before ou after quand l'événement est produit par rapport à

l'opération qui génére un événement de ce type.

Le type d'événement (after, update(Personne->age)) de la règle Modification_age caractérise une modification de l'attribut age d'un objet de la classe Personne. Un événement de ce type est signalé après chaque mise à jour de l'age d'une personne.

Événements composites : Ils sont construits récursivement à partir d'événements (primitifs et composites) connectés par des *opérateurs* : conjonction, disjonction, négation, séquence, séquence stricte, itération et itération stricte. Nous ne détaillons pas ici les événements composites (cf. [CC96a, CC96b, CCR96].

6.1.2.2 Identificateur de delta structure

Une *delta structure* contient des données associées à l'événement ou aux événements déclenchants. Les informations contenues dans ces delta structures peuvent être utilisées au moment de l'exécution de la règle et tout particulièrement dans la Condition et l'Action. Une delta structure a un type qui dépend du type de l'événement et de certaines dimensions de l'exécution d'une règle : le mode d'exécution, le mode de traitement des événements et le mode de consommation des événements. De manière à offrir une vision homogène des informations et un mécanisme de manipulation général, NAOS offre d'une part la possibilité de nommer ces structures et d'autre part de construire des vues qui donne une vision simplifiée des informations.

Pour la règle Modification_age, p dénote la delta-structure associée à la règle au moment de son exécution. p va donc permettre d'accéder aux informations véhiculées par l'événement déclenchant, c'est-à-dire l'objet concerné par l'opération et l'ancienne valeur de l'attribut age. Lorsque la règle Modification_salaire est exécutée, la delta-structure e qui lui est associée contient l'employé concerné ainsi que la nouvelle valeur de son salaire.

6.1.2.3 Condition

La partie Condition d'une règle NAOS est une formule composée de requêtes OQL (O_2SQL) sur les objets et les valeurs. Un prédicat est vrai si le résultat de la requête correspondante n'est pas vide. Un prédicat peut faire référence aux delta structures qui contiennent des informations relatives aux événements. Le résultat d'une requête peut être passé à la partie Action de la règle.

Si au cours d'une application utilisant le schéma exemple, la valeur de l'attribut age de l'instance Le_chef est modifiée alors la règle Modification_age est déclenchée[1]

1. Cette règle est déclenchée car Le_chef est également une instance de la classe Personne.

et exécutée de suite car il s'agit d'une règle immédiate. p dénote alors une delta-structure contenant l'objet nommé Le_chef et l'ancienne valeur de l'attribut age ; current(p) permet de retrouver l'état courant de Le_chef et old(p) retourne son ancien état. L'expression current(p)->age - old(p)->age != 1 permet donc de vérifier que l'age a été incrémenté de 1.

6.1.2.4 Action

La partie Action est définie par du code O2C. La forme la plus simple pour une action est une méthode appliquée à l'objet concerné par l'événement déclenchant. Cet objet est accessible à travers la delta structure associée à la règle lors de son traitement. L'Action peut éventuellement provoquer l'abandon de la transaction courante dans laquelle l'événement déclenchant s'est produit. Dans la règle Modification_age si la nouvelle valeur pour l'attribut age ne satisfait pas la contrainte (la condition est vraie) alors l'action de la règle est exécutée : un message est affiché et la transaction courante est annulée.

Quand une règle possèdant une clause do instead est exécutée, les opérations de cette clause sont substituées à l'opération déclenchante. Cette substitution n'a de sens que si l'exécution de la règle est immédiate et si l'événement est détecté avant que l'opération qui le produit ne soit traitée.

6.1.3 Exécution d'une règle

Nous décrivons ici rapidemment le modèle d'exécution de NAOS. Celui-ci à été abordé dans le chapitre 3, puis dans le chapitre 4 lorsque nous avons montré la couverture de NAOS par Fl'are. Ce modèle est décrit de manière informelle dans [CCS94b], et de manière formelle dans [CC95c].

Dans NAOS, on trouve deux types de règles : les *règles immédiates* et les *règles différées*. Les règles immédiates sont exécutées immédiatement après occurrence de leur(s) événement(s) déclenchant(s). Les *règles différées* "accumulent" les événements déclenchants au cours de la transaction déclenchante, leur exécution est différée à la validation de cette transaction. Les règles immédiates ont un mode de traitement des événement par instance. Les règles différées, ont un mode de traitement ensembliste des événements. Toutes les règles consomment les événements et prennent en compte l'effet net à la fois pour le déclenchement et pour la construction des environnements d'exécution (delta structures).

L'exécution d'un ensemble de règles est basée sur la notion de *cycle d'exécution* (cf. refsec:cascades,sec:stratcyles). Un cycle d'exécution est une séquence d'opérations

exécutées dans la transaction déclenchante ou pendant l'exécution de la partie Action d'une règle. Les règles sont toujours exécutées dans un cycle différent de celui dans lequel le ou les événements qui les ont déclenchées ont été produits. Si plusieurs règles doivent être exécutées dans un cycle, elles les sont suivant leurs *priorités*. Des priorités par défaut sont assignées par le système. Elles sont basées sur l'ordre de définition des règles mais peuvent être surchargées en spécifiant des relations de précédence entre des couples de règles. Les règles immédiates sont exécutées *en profondeur d'abord*. Les règles différées sont exécutées *en largeur d'abord*. Les règles immédiates ont une priorité implicite sur les règles différées. Enfin, les règles immédiates et différées sont *interruptibles* par les règles immédiates (toutes les règles immédiates ont le droit de préemption, toutes les règles différées en sont privées).

6.2 Mise en œuvre de NAOS

NAOS est un exemple d'*architecture intégrée*. Il n'est pas implanté *au-dessus* du SGBD O_2 – on parlerait alors d'une *architecture en couches* – mais *dans* O_2. En termes d'utilisation, NAOS fonctionne comme une librairie dynamique d'un processus client O_2. En terme d'implantation, NAOS est un ensemble de fonctions C et de classes C++, représentant environ 31000 lignes de code dans sa dernière version, la version 2.1.2 de juillet 1996.

6.2.1 Architecture générale

La figure 6.4 montre les principaux composants de NAOS. Le traitement des définitions de règles est réalisé par l'Analyseur de règles et le Constructeur de règles. La détection des événements est réalisée par le Détecteur d'événements et le Gestionnaire d'événements. L'exécution proprement dite est réalisée par l'Exécutif.

Définition des règles

NAOS offre deux interfaces pour définir les règles [Cou93] : (i) un langage de règles pour des manipulations aisées par les utilisateurs finaux, (ii) une interface de programmation des règles qui permet un accès direct aux primitives de manipulation des règles. Cette seconde interface de bas niveau est semblable à l'interface O2API. Elle peut être utilisée par les applications générant automatiquement les règles (e.g. mécanismes d'intégrité) ou par les applications voulant implanter leur propre langage de règles (e.g. environnements de Génie Logiciel).

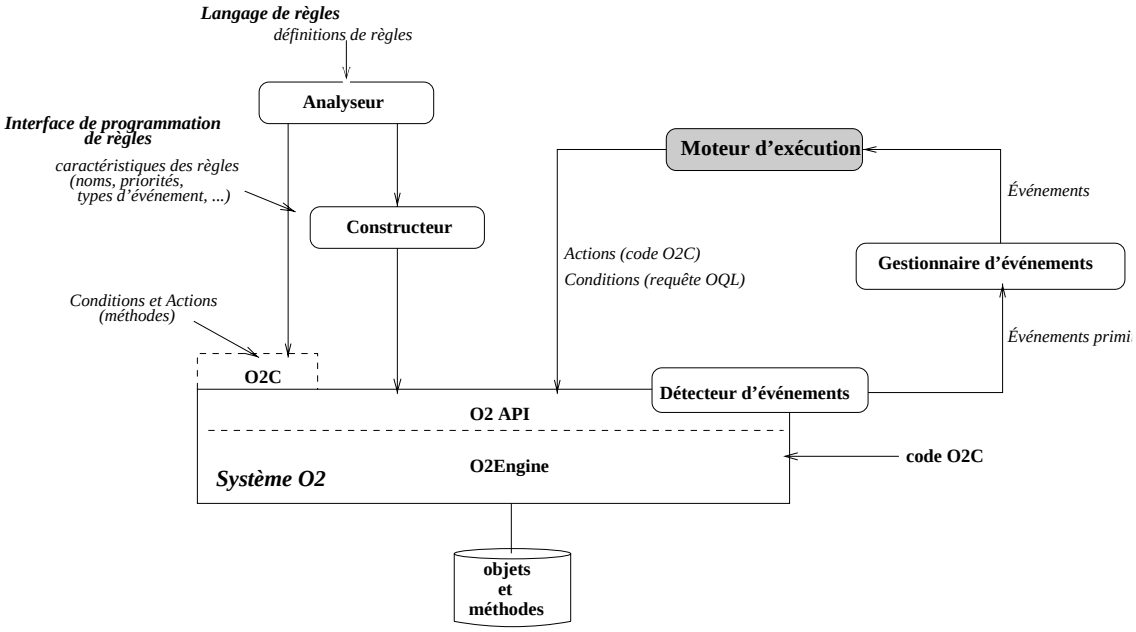

FIG. 6.4 – *Architecture de NAOS*

Les caractéristiques des règles sont stockées sous forme d'objets persistants (dans des bases O_2), les Conditions et les Actions sont stockées sous forme de méthodes O2C compilées.

L'Analyseur communique avec les programmeurs qui veulent facilement définir et manipuler des règles, en utilisant le langage de règles. Il a pour fonctions principales l'analyse (compilation) d'une définition de règle et la production : (i) d'une représentation intermédiaire de règle envoyée au Constructeur, et (ii) de méthodes O_2C implantant la Condition et l'Action d'une règle. Ces méthodes appartiennent aux deux classes pré-définies Conditions et Actions.

Le Constructeur de règles est un ensemble de fonctions C-O2API grâce auquel les caractéristiques d'une règle (nom, priorité, mode d'exécution, type d'événement) peuvent être définies. Ces fonctions construisent les objets O_2 persistants représentant les règles. A ce stade, les Conditions et Actions des règles sont définies directement (par le programmeur) par des méthodes dans les classes Conditions et Actions.

Détection des événements

La détection des événements est réalisée par le Détecteur d'événements et le Gestionnaire d'événements.

Le Détecteur d'événements est un composant vital du système car il ne peut y avoir d'exécution de règles sans détection d'événements. C'est également la partie la plus susceptible de dégrader les performances globales du système en temps d'exécution. Pour assurer des performances convenables, la détection des événements primitifs présente deux caractéristiques :

1° le Détecteur est intégré dans le noyau du système O_2 – ce qui minimise et accélère les communications entre le Détecteur et O_2 ;

2° la détection est basée sur un principe d'*abonnements* : le Détecteur signale au Gestionnaire uniquement les événements dont le type a été préalablement *abonné*.

Lorsqu'un schéma devient actif, un abonnement est réalisé pour chaque type d'événement primitif concerné. Pour chaque abonnement, l'adresse d'une fonction de manipulation (handling function) est fournie, réalisant par ce biais un mécanisme d'abonnement dynamique qui peut également être utilisé par des applications autres que NAOS. Cette fonction doit nécessairement être liée aux types d'événement et aux delta-structures. Le gestionnaire d'objets O_2 considère toutes les opérations sur les objets comme de possibles occurrences de types d'événements abonnés.

Le Gestionnaire d'événements reçoit les événements primitifs en provenance du Détecteur, reconnait les événements composites grâce à un *graphe de détection* et signale tous les événements et leurs environnements à l'Exécutif.

Avant la détection proprement dite, une des tâches du Gestionnaire et de construire les graphes de détection et d'abonner les types d'événements primitifs auprès du Détecteur.

Exécution des règles

L'Exécutif ou moteur d'exécution est un autre composant essentiel de NAOS. Il implante le modèle d'exécution de NAOS : il sélectionne les règles à déclencher, évalue les Conditions et exécute les Actions en tenant compte du mode d'exécution, du traitement et de la consommation des événements, du calcul de l'effet net, des déclenchements multiples de règles et des cascades de règles.

6.2.2 Le moteur d'exécution

Nous nous intéressons maintenant à l'implantation et au fonctionnement du moteur d'exécution de NAOS. Celui-ci est représenté dans la figure 6.5 qui détaille le composant Moteur d'exécution de la figure 6.4.

Lorsqu'un schéma O_2 est choisi par un utilisateur ou une application, des abonnements sont envoyés au Détecteur d'Événements pour les types d'événements associés aux règles de ce schéma.

Une fois la phase d'abonnement terminée, le détecteur d'événements – au sens large, c'est-à-dire Détecteur et Gestionnaire d'événements – commence sa surveillance des opérations sur la base de données.

C'est seulement suite à la reconnaissance d'un événement, dont le type est abonné, que les prochaines actions seront effectuées : l'événement et son environnement sont envoyés à la fonction fournie au moment de l'abonnement. Cette fonction réveille le moteur d'exécution qui stocke l'événement dans un *journal* (log) et exécute les règles déclenchées, soit immédiatement après la détection, soit à la fin de la transaction déclenchante.

NAOS se présente sous la forme d'une librairie de fonctions C++ appelées dynamiquement par un client O_2. La section 6.2.2.1 décrit les classes C++ qui implantent le moteur d'exécution. La section 6.2.2.2 décrit les opérations préparatoires à la détection des événements et à l'exécution des règles. La section 6.2.2.3 décrit ensuite le fonctionnement du moteur d'exécution en montrant comment interagissent dynamiquement les différents objets C++, instances des classes de la section 6.2.2.1.

6.2.2.1 Structures de données

Meta_Event_type : La classe Meta_Event_type est la racine d'une hiérarchie de classes qui représentent tous les types d'événements de NAOS et qui permettent de traiter les occurrences simultanées d'événements. À un instant donné, les objets de ces classes représentent les types d'événements qui peuvent être détectés dans le cadre de l'application qui est en train de s'exécuter. Ces objets sont principalement utilisés pour le déclenchement des règles.

Caractéristiques générales : Meta_Event_type est spécialisée en deux classes : Sim_event_type pour traiter les événements simultanés et Event_type pour les autres événements. Dans la suite, nous nous focalisons sur les événements qui ne sont pas simultanés. La classe Event_type est elle-même spécialisée en Primitive_event_type pour

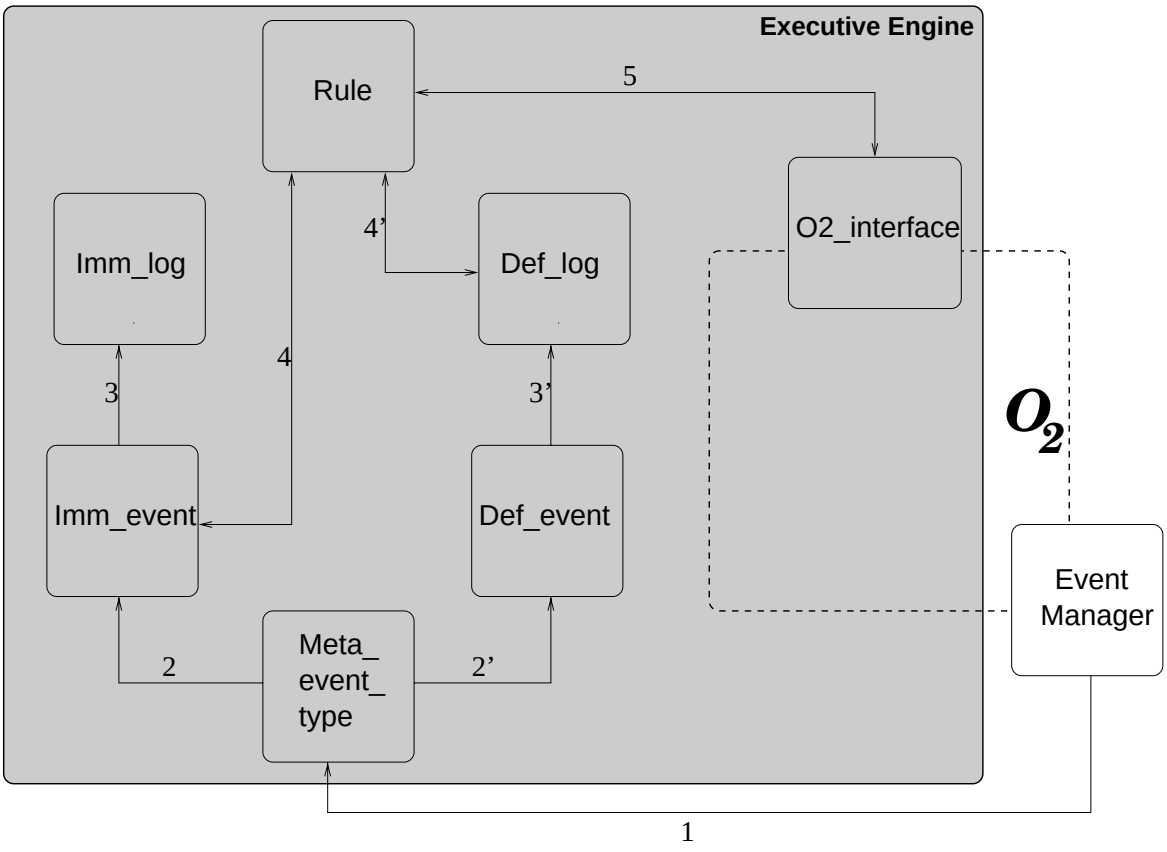

FIG. 6.5 – *Architecture du moteur d'exécution de NAOS*

les types d'événements primitifs et Composite_event_type pour les types d'événements composites.

Les attributs d'un objet de Meta_Event_type permettent d'accéder à une description complète du type d'événement (type d'opération, type d'entité, chemin dans l'entité, instant de production) ; à la liste des règles déclenchées par ce type et à la liste des événements de ce type déjà survenus (depuis le début de la transaction déclenchante).

La méthode principale de la classe Meta_Event_type est la méthode signal qui est appelée lorsqu'un événement du type correspondant est détecté. À la réception d'un événement, la méthode crée un ou plusieurs objets Event (instances des classes Imm_event et/ou Def_event) qui doit calculer l'effet net et déclencher les règles associées au type de l'événement. À l'objet (ou aux objets) créé(s) est envoyé le type complet de l'événement, la liste des règles déclenchées et l'environnement de l'événe-

ment (delta-structure).

Caractéristiques spécifiques: La classe Primitive_event_type a pour descendants : Internal_event_type et User_event_ type qui représentent les types d'événements internes et utilisateurs. Internal_event_type a elle-même pour descendants Entity_event_type, Transaction_event_type et Schema_ event_type représentant les types d'événements associés aux manipulation d'entités, de transactions et de schémas. Les deux dernières classes réalisent des tâches administratives (du modèle d'exécution) relativement importantes lorsqu'elles reçoivent des occurrences d'événements[2].

Une instance de la classe Schema_event_type représente un type d'événement O2_SCHEMA_EVENT. Un événement de ce type est important dans le système de règles car il indique l'initialisation d'un nouvel ensemble de règles (faisant suite à un changement de schéma), et souvent la suppression de l'ensemble de règles précédent.

Les instances de la classe Transaction_event_type représentent les types d'événements liés aux transactions. Si un événement dont le type est 02_TRANS_COMMIT_EVENT ou 02_TRANS_VALID_EVENT est réceptionné, la fin du cycle d'exécution 0 (fin de la transaction) est signalée. Un nouveau cycle débute alors, celui-ci commence par l'exécution des règles différées.

Imm_event et Def_event : La classe Event est une classe virtuelle, racine d'une hiérarchie de classes qui prennent en charge la composition des événements (pour le calcul de l'effet net) et l'exécution des règles. Cette classe a deux descendantes directes : Imm_event et Def_event. La hiérarchie de classes décrivant les événements est comparable à celle décrivant les types d'événements. Tandis qu'un objet d'une classe de la hiérarchie des types d'événements représente l'abonnement à un type d'événement particulier ; un objet de la hiérarchie des événements représente un événement particulier, c'est-à-dire une occurrence d'un type d'événement produit à un instant donné.

Les objets Imm_event sont associés à l'exécution des règles immédiates. Ils calculent l'effet net des événements en coopération avec Imm_log.

Les objets Def_event sont associés à l'exécution des règles différées. Ils calculent l'effet net en coopération avec Def_log. La classe Def_event en collaboration avec la classe Def_log permet également de construire les delta collections pour l'exécution des règles différées.

2. Le type d'événement associé aux changements de schémas n'est d'ailleurs pas disponible pour le programmeur de règles. Il est utilisé uniquement de manière interne au moteur d'exécution.

Imm_log et Def_log : Les objets des classes Imm_log et Def_log contrôlent les objets Event qui sont construits et détruits continuellement. Ce sont des listes de Imm_event et Def_event, respectivement.

L'objet Def_log qui s'occupe des événements associés aux règles différées est le plus actif puisqu'il doit manipuler deux listes internes (les événements sur le point d'être traités et ceux qui sont accumulés pour le cycle suivant) et déplacent des objets de l'une à l'autre à la fin de chaque cycle différé.

L'objet Imm_log va surtout être un outil de stockage pour les objets représentant des événements déclenchant les règles immédiates.

Imm_log et Def_log reçoivent tous les deux des événements et calculent l'effet net. En simplifiant, cela est réalisé en composant le nouvel événement avec chacun des événements déjà présents dans la ou les liste(s).

Rule : La classe Rule représente les règles. Les instances de cette classe sont des images dynamiques (en mémoire vive) des représentations persistantes des règles. Ces objets C++ sont créées (une et une seule fois) au moment d'un changement de schéma. Ceci permet d'améliorer les performances du système.

Les attributs de cette classe représentent les caractéristiques statiques des règles : nom, priorité, mode d'exécution (immédiat ou différé), substitution de l'opération déclenchante et type d'événement. Le seul attribut dont la valeur peut, en réalité, changer dans l'un de ces objets, au cours de l'exécution d'une application, est l'attribut qui représente l'état (activé ou désactivé) de la règle.

Les principales méthodes de la classe Rule sont la méthode fire qui permet de provoquer l'évaluation et l'exécution d'une règle et les méthodes permettant de désactiver et de réactiver une règle.

O2_interface : O2_interface est un objet (c'est l'unique instance de la classe O2_interface) qui a pour tâche de cacher les particularités d'O_2 aux autres objets de NAOS – rendant ainsi NAOS plus indépendant du SGBD et facilitant les modifications futures dues aux nouvelles versions de l'interface de programmation d'application (O_2API) du système O_2.

Toute les communications entre O_2 et NAOS, les entrées comme les sorties, passent par O2_interface. En fait, si cette partie de NAOS était remplacée, le système de règles pourrait être facilement intégré à un autre système de base de données à objets fournissant un mécanisme de base pour la détection d'événements.

6.2.2.2 Opérations préparatoires

Initialisation de l'exécution des règles L'initialisation de l'exécution des règles est réalisée au lancement d'un client O_2. Le noyau O_2 – par un appel à une fonction rule_executive() – réveille le moteur d'exécution qui crée :

- un objet de la classe Event_type_list qui est une liste de Meta_Event_type et qui contiendra les types d'événements qui devront être détectés ;
- un objet de chacune des classes Imm_log et Def_log ;
- une instance de Schema_event_type abonnant le type d'événement 02_SCHEMA_EVENT auprès du détecteur d'événements – afin d'indiquer que le choix d'un schéma devra être signalé comme un événement.

Changement de schéma Lorsqu'un nouveau schéma est choisi, un nouvel ensemble de règles est initialisé (s'il existe des règles associées à ce schéma, bien entendu). Les définitions des règles sont lues à partir des objets O_2 dans lesquels elles sont stockées ; les objets C++ correspondant sont alors créés. De plus, les types d'événements primitifs décrits dans ces règles sont abonnés auprès du Détecteur d'événements. Le graphe de détection des événements composites est alors construit. Toutes ces actions sont effectuées par l'objet instance de Schema_event_type répondant au signal d'un événement du type O2_SCHEMA_EVENT.

Plus précisément :

1. L'objet Schema_event_type commence par désactiver la détection des événements.
2. Ensuite, il parcourt la liste des types d'événements. Chaque type d'événement est enlevé de la liste et la liste d'instances de Rule qui lui est attachée est supprimée.
3. Puis, la nouvelle base de règles est construite en parcourant les règles définies dans l'objet persistant O_2 gRdefs. Il s'agit d'une liste présente dans chaque schéma dans lequel sont définies des règles : pour un schéma donné, elle contient donc la définition de toutes les règles associées à ce schéma. Pour chacune des règles de gRdefs, l'objet Schema_event_type :
   (a) crée un objet C++ instance de la classe Rule ;
   (b) recherche le type d'événement de la règle parmi les objets instances de Meta_event_type déjà créés dans la liste Event_type_list ; s'il est trouvé, l'objet Rule est inséré dans la liste des règles associée à ce type d'événement ;
   (c) si ce type n'est pas trouvé, une nouvelle instance de la classe du type d'événement approprié est créée, envoyant la nouvelle règle comme paramètre

du constructeur C++ du type d'événement ; après avoir donné des valeurs aux attributs de l'objet, le constructeur réalise les tâches suivantes :

i. il abonne son propre type d'événement ;

ii. il crée de nouveaux types d'événements (non abonnés) pour tous les événements qui pourraient être composés avec des événements du type courant. Ensuite, tous ces objets sont ajoutés à Event_type_list ;

4. L'objet Schema_event_type ré-active la détection des événements et rend le contrôle au système O_2 par le biais d'O2_interface.

6.2.2.3 Réception des événements et exécution des règles

Réception d'un événement : Un objet de la hiérarchie de racine Meta_event_type reçoit un événement en provenance du gestionnaire d'événement (flèche 1 de la figure 4.17) : le type de l'objet en question est déterminé par le type de l'événement. Ce sera, par exemple, un objet de la classe Method_event_type (sous-classe de entity_event_type) pour un événement produit par un appel de méthode.

Cet objet possède une liste de règles à déclencher et sait si ces règles sont de type immédiat ou différé. Selon cette information, il crée (flèche 2 et 2' de la figure 6.5) une instance de Imm_event ou une instance de Def_event ou des deux (des règles immédiates et différées peuvent être associées à un même type d'événement).

Lorsque ces instances sont créées, le type de l'événement, son environnement (pour construire la delta-structure) et les objets Rule correspondant aux règles déclenchées sont passés en paramètres au constructeur de Imm_event et/ou Def_event.

Événements déclenchant des règles immédiates : Le constructeur d'un objet Imm_event commence par s'insérer dans Imm_log (flèche 3 de la figure 6.5) :

- s'il n'est pas le premier événement de Imm_log et qu'un événement produit antérieurement est sur le point de provoquer l'exécution d'une règle de type "before", un message d'erreur est affiché et aucune règle n'est exécutée ;
- dans le cas contraire (le plus courant), la première règle est exécutée, c'est-à-dire que sa Condition est évaluée et son Action exécutée (flèche 4 et 5 de la figure 6.5 respectivement).

La fin de l'exécution de la règle est ensuite signalée à l'objet Imm_event qui a déclenché cette exécution (flèche 5 et 4 de la figure 6.5 de O2_interface vers Imm_event).

Il est possible que la règle s'exécute en se substituant à l'opération déclenchante (règle avec clause instead). Si c'est le cas, aucune autre exécution de règle n'a lieu pour cet événement.

Durant l'exécution d'une règle R, d'autres événements déclenchant des règles immédiates peuvent arriver. Par conséquent, avant chaque exécution, notre premier événement doit vérifier si cela est arrivé afin de calculer l'effet net pour le déclenchement des règles (les *nouvelles* règles et les règles déjà déclenchées mais non encore exécutées). Pour cela, l'objet Imm_log garde la trace de tous les événements arrivés depuis le dernier déclenchement de la règle R. L'événement va essayer de se composer avec tous les nouveaux événements, en commençant par le plus ancien. Chaque nouvel événement peut annuler l'événement considéré, modifier son environnement ou n'avoir tout simplement aucun effet. Les événements seront détruits après la composition si et seulement si l'événement est le premier dans Imm_log.

Lorsqu'il n'y a plus de règles à exécuter, l'objet se retire lui-même de Imm_log.

Événements déclenchant des règles différées : Le constructeur de Def_event commence par parcourir la liste de règles qu'il a eu en paramètre, en demandant à chaque règle si elle est différée et non désactivée à cet instant. Seules les règles répondant positivement à ces deux critères seront ajoutées à la liste des règles (déclenchées par cet événement particulier) à exécuter.

Déclenchement des règles, calcul des cycles d'exécutions et de l'effet net : L'objet Def_event met à jour un attribut booléen indiquant que l'environnement de Def_event ne doit pas être modifié lors de compositions, seul l'environnement de l'événement avec lequel il est composé peut être modifié. Les nouveaux événements qui sont liés à des règles différées tenteront d'abord de se composer avec les événements qui ont des règles sur le point d'être exécutées. Considérons que nous sommes au milieu du cycle d'exécution 1, c'est-à-dire le premier cycle différé, et qu'un événement *différé* (un événement associé à une règle différée) arrive. Il faut tout d'abord vérifier que cet événement n'a aucun rapport avec les événement différés survenus dans le cycle 0. Lors de la composition avec ces événements, le nouvel événement ne doit pas être affecté. Par contre, il peut être modifié par une composition avec les événements appartenant au cycle 1.

Par suite, la mise à jour de cette variable, Def_event s'insère à Def_log (flèche 3' de la figure 6.5) qui s'occupe à son tour de le composer avec les autres événements de Def_log. Lorsqu'il a terminé ce processus pour les événements du cycle courant, il indique à Def_event qu'il est autorisé à modifier son environnement puis le compose avec les événements étant déjà arrivés pour le cycle suivant.

Exécution des règles : Quand un événement du type O2_TRANS_COMMIT_EVENT ou O2_TRANS_VALIDATE_EVENT est reçu, il est envoyé à une instance de Transaction_event_type. Cette instance traite l'événement en indiquant à l'objet Def_log qu'il doit exécuter les règles différées (flèches 4' et 5 de la figure 6.5).

À l'intérieur de Def_log, une nouvelle liste est préparée pour recevoir les événements différés arrivant durant le cycle 1. Puis chacun des événements arrivés durant le cycle 0 voit ses règles prises en considération. En les recevant, Def_log les trie par ordre de priorités. Après cela, il commencera par exécuter celle qui a la plus grande priorité. Puis, il continuera en parcourant la liste par ordre de priorité décroissant jusqu'à ce que toutes les règles soient exécutées. Pendant le parcours de la liste, un événement peut être supprimé en raison de l'arrivée d'autres événements avec lesquels il s'annule par effet net. Dans ce cas l'événement alertera Def_log, ainsi les règles correspondantes seront supprimées de la liste.

Évaluation des Conditions L'évaluation de la partie Condition d'une règle est effectuée par l'objet Rule représentant cette règle (flèche 5 de la figure 6.5). Celui-ci envoie l'environnement ou la liste d'environnements à O2_interface qui se charge de construire le delta élément ou la delta collection selon le type de règle. Le résultat de l'opération est retourné à notre objet Rule (flèche 5 de la figure 6.5 de O2_interface vers Rule) qui décide alors d'exécuter ou non la partie Action – suivant le résultat de l'évaluation.

Exécution des Actions Les Actions sont également exécutées par les objet Rule, d'une manière très similaire à l'évaluation des Conditions. O2_interface, une fois de plus, crée les objets O_2 nécessaires – notamment pour représenter les environnements d'exécution. Au retour de l'exécution de la règle, l'objet Rule va signaler si la clause instead était présente, ce qui implique une suppression de l'événement déclenchant (et des règles déclenchées par cet événements en attente d'exécution).

6.3 Mise en œuvre de Fl'are

Nous venons de décrire le système NAOS, en détaillant son architecture, et son implantation dans le SGBD O_2. Nous allons maintenant décrire l'expérimentation que nous menons dans ce contexte afin d'implanter Fl'are. Fl'are étant uniquement un moteur d'exécution, l'expérimentation que nous décrivons ici consiste à intégrer Fl'are dans un environnement qui permette la définition des règles et la détection des événements afin d'obtenir un SGBD actif complet et opérationnel (cf. chap. 2).

L'architecture de NAOS exhibe six composants destinés à trois tâches : la définition des règles, la détection des événements et l'exécution des règles. De part son architecture modulaire, nous pensons que NAOS constitue un support adéquat pour la mise en œuvre de Fl'are. Schématiquement, la mise ne œuvre que nous proposons consiste à remplacer un composant spécifique de NAOS : son moteur d'exécution, par Fl'are.

Fl'are peut être vu comme une généralisation du modèle d'exécution de NAOS. Il est, en tout état de cause, *plus sophistiqué*. La prise en compte de certaines de ses spécificités – comme par exemple le fait de regrouper les règles en modules – nous conduit également à apporter quelques modifications mineures dans les autres composants de NAOS – comme le composant Définition de règles pour prendre en compte cette notion de module – ou dans les interfaces entre les composants. La section 6.3 décrit l'implantation de Fl'are. La section 6.3.2 décrit les modifications à apporter dans les autres composants de NAOS afin qu'ils puissent communiquer avec Fl'are et autoriser la définition et l'exécution de règles selon les principes de Fl'are.

6.3.1 Le moteur d'exécution Fl'are

Le support de développement de Fl'are est le moteur d'exécution de NAOS. Cette section montre que bon nombre des composants de NAOS décrits dans la section 6.2 peuvent être *ré-utilisés*, moyennant quelques modifications.

6.3.1.1 Structures de données

Les classes C++ du moteur d'exécution de NAOS (cf. 6.2.2.1) peuvent être classés en trois catégories :

1. la hiérarchie de classes dont la racine est Meta_event_type qui est utilisée pour la reception des événements ;
2. les classes Event et Event_log qui sont utilisées pour le déclenchement et l'exécution des règles ;
3. les classes Rule et O2_interface qui sont utilisées pour l'évaluation des Conditions et l'exécution des Actions.

Par ailleurs, le modèle d'exécution de NAOS présente deux comportements très distincts : le premier en cours de transaction pour l'exécution des règles immédiates, et le second en fin de transaction pour l'exécution des règles différées (cf. sections 4.6.1 et 6.1.3). Toutes les règles immédiates s'exécutent de la même manière (mode d'exécution immédiat, traitement par instance des événements, consommation des

événements, effet net, péemption de l'exécution, etc.). Toutes les règles différées s'exécutent également de la même manière(mode d'exécution différé, traitement ensembliste des événements, consommation des événements, effet net, non préemption de l'exécution,etc.). Cette forte dichotomie se retrouve dans l'architecture du moteur d'exécution de NAOS[3], en ce qui concerne le déclenchement et l'exécution des règles : on trouve d'un côté les classes lmm_event et lmm_log qui gérent l'exécution des règles immédiates ; et d'un autre côté les classes Def_event et Def_log qui gérent l'exécution des règles différées.

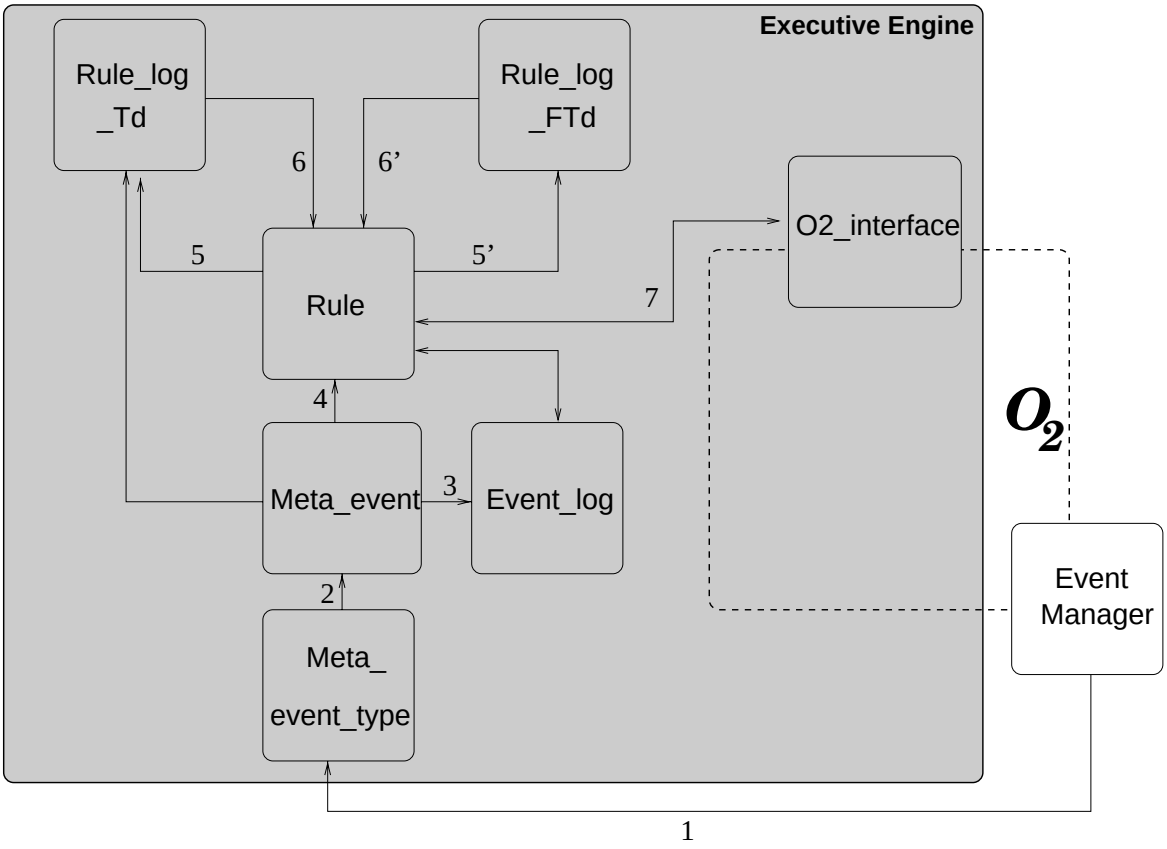

FIG. 6.6 – *Architecture du moteur d'exécution Fl'are*

Dans le cadre de Fl'are, ce schéma n'est plus applicable : on a bi en deux comportement distincts en cours et en fin de transaction, mais à chaque point d'exécution, les règles peuvent s'exécuter de manière complétement différente : traitement par instance ou ensembliste des événements, consommation ou préservation des événements,

3. Dire que cette dichotomie a induit l'architecture serait même plus exact.

prise en compte ou non de l'effet net, avec préemption ou non. Aussi, dans l'architecture de Fl'are que nous proposons (cf. figure 6.6), on retrouve bien les trois catégories évoquées ci-dessus. En revanche, la dichotomie entre modèle d'exécution en cours de transaction et modèle d'exécution en fin de transaction n'est plus représentée par la dualité des couples (Imm_event, Imm_log) et (Def_event, Def_log) mais par la dualité de deux nouvelles classes : Rule_log_Td et Rule_log_Ftd.

Meta_event_type : La hiérarchie de classes dont la racine est Meta_event_type est quasi inchangée. Le rôle des objets de ces classes est simplement de créer un objet d'une classe de la hiérarchie dont la racine est Meta_event suivant le type des événements qui proviennent du gestionnaire d'événement (Event_manager).

Meta_event : La hiérarchie de classes dont la racine est Meta_event est simplifiée puisque l'on supprime la dualité représentée par les classes Imm_event et Def_event. Les classes de cette hiérarchie ont un rôle moins prédominant que dans NAOS : un objet d'une de ces classes représente simplement une occurrence d'événement mais ne contrôle plus l'exécution des règles comme dans NAOS.

Event_log : Les classes Imm_log et Def_log ne sont plus utilisées. La classe Event_log est simplifiée car, elle non plus, n'assure plus aucune fonction administrative, pour l'exécution des règles, comme c'est le cas dans NAOS.

Elle est utilisée comme un simple lieu de stockage des événements. Au cours de l'exécution d'une transaction, on a un seul objet Event_log qui garde trace de tous les événements survenus au cours de la transaction. Event_log est utilisé conjointement par Meta_event et Rule pour le calcul de l'effet net – pour les règles qui prennent en compte l'effet net.

Rule : La classe Rule est quasiment inchangée dans son comportement. La méthode fire est toujours chargée d'évaluer les Conditions et d'exécuter les Actions mais l'exécution de l'Action ne succède pas forcément immédiatement à l'évaluation de la Condition (règles retardées).

En revanche, sa structure est étoffée pour prendre en compte le modèle d'exécution d'une règle qui est plus riche dans Fl'are que dans NAOS.

Les principaux attributs d'un objet Rule sont donc : son nom, sa priorité, son mode de traitement des événements, son mode de consommation des événements, son mode de prise en compte de l'effet net, son mode de préemption, son mode d'exécution. Ces *attributs statiques* – qui sont stockés dans la représentation persistante des règles

– sont complétés par des *attributs dynamiques* – qui sont utiles uniquement pour l'exécution des règles – qui sont : son état (évaluable, exécutable, exécutée, etc.), son instant de déclenchement, le cycle dans lequel elle sera considérée [4], et son (ses) événement(s) déclenchant(s) (et son(ses) environnement(s)). En réalité, la structure de la classe Rule correspond au domaine RT des règles à traiter de la sémantique dénotationelle de Fl'are donnée dans le chapitre précédant.

O2_interface : Cette classe est également quasiment inchangée si ce n'est que la méthode get_rule qui accéde à la représentation persistante des règles doit prendre en compte la nouvelle structure des règles.

Rule_log : Les classes Rule_log_Td et Rule_log_FTd, descendantes d'une classe virtuelle Rule_log, sont véritablement nouvelles. Ce sont principalement elles qui incarnent le modèle d'exécution Fl'are. Elles sont, en effet, responsables, en collaboration avec la classe Rule, du déclenchement, de la sélection et de l'exécution (des modules) de règles.

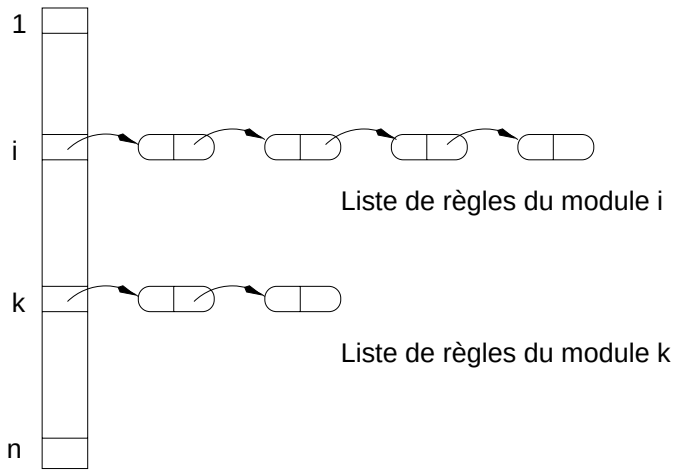

FIG. 6.7 – *Structure d'un objet* Rule_log

Il y a un seul objet instance de chacune de ces deux classes pour l'exécution d'une transaction. Leur structure est représentée par la figure 6.7. Ce sont des tables de listes de règles. Les entrées dans les tables correspondent aux modules de règles. Le nombre d'entrées dans chacune des tables est le nombre de modules de règles définis dans un schéma. Afin d'optimiser les performances du système, les modules sont placés par

4. Les cycles ne sont valides que si la règle appartient à un module qui est exécuté par cycles.

ordre de priorités décroissantes. Les listes du second niveau représentent les règles d'un module. Les règles sont perpétuellement insérées et supprimées des objets Rule_log_Td et Rule_log_FTd au cours d'une transaction. L'insertion est *placée* suivant la stratégie d'exécution de chaque module : en queue pour exécution en pipelines, en tête pour exécution en profondeur, etc. (cf. chapitre 4).

La structure de la classe Rule_log est inspirée du domaine des modules de règles à traiter MT de la sémantique dénotationnelle de Fl'are : les ensembles de la sémantique formelle sont implantés par des tables ou des listes dans un souci de performances.

6.3.1.2 Opérations préparatoires

Les opérations préparatoires sont exactement les mêmes que celles décrites dans la section 6.2.2.2 pour l'initialisation des exécutions NAOS.

6.3.1.3 Réception des événements et exécutions des règles

À la réception des événements en provenance du gestionnaire d'événements (flèche 1 de la figure 6.6), les objets Meta_event_type créent les objets Meta_event correspondants (flèche 2 de la figure 6.6). Ces objets contiennent les informations suivantes : types d'événement, instant d'occurrence et environnement. Ces informations sont utilisées pour affecter l'instant de déclenchement des règles et calculer l'effet net.

Les événements sont alors insérés dans l'objet Event_log (flèche 3 de la figure 6.6). Celui-ci est utilisé pour calculer l'effet net – pour les règles qui prennent en compte l'effet net – en coopération avec les objets Rule et Meta_event.

Ensuite débute la *phase de construction des ensembles de règles* (flèches 4, 5 et 5' de la figure 6.6). Chaque événement, qui connait les règles qu'il déclenche, s'adresse à chacun des objets Rule correspondant s'ils existent, ou bien les crée – suivant le mode de traitement et le mode de consommation de chacune des règles. Les règles sont alors insérées (pour celles qui viennent d'être créées) ou *actualisées* (prise en compte des nouveaux événements pour celles qui étaient déjà présentes) dans Rule_log_Td ou Rule_log_FTd – suivant leur mode d'exécution et le type du point de considération courant (en cours de transaction ou en fin de transaction). Cette phase est décrite formellement dans le chapitre précédent, dans les sections 5.2.2.1 pour le processus en cours de transaction et 5.2.2.4 pour le processus en fin de transaction. Ce processus est spécifié par les fonctions *Déclenche* et *DéclencheFtd* respectivement. On peut espérer de bonnes performances de cette phase de déclenchement qui est coûteuse en temps

d'exécution, du fait de la correspondance directe – on peut parler d'*indexation par modules* – qui existe entre la structure des listes de listes de règles associées à chaque événement, et les tables de listes de règles qui constituent les Rule_log dans lesquelles les règles sont insérées.

Après quoi débute la *phase de sélection d'une règle* (sections 5.2.2.2 et 5.2.2.4 du chapitre précédant). À chaque itération (flèches 5 et 6 ou 5' et 6' de la figure 6.6), les modules de règles sont examinés afin de sélectionner la prochaine règle à traiter. Ce processus est spécifié par les fonctions *TraiteModules* et *TraiteModulesFtd* de la sémantique formelle de Fl'are.

Chaque règle sélectionnée est alors traitée (sections 5.2.2.3 et 5.2.2.6 du chapitre précédent) pour évaluation et/ou A-exécution : flèche 7 de la figure 6.6 ; fonctions *TraiteRègle* et *TraiteRègleFtd* de la sémantique formelle.

6.3.2 Fl'are et les autres composants de NAOS

Détection des événements et exécution des règles : Le protocole de communication entre le moteur d'exécution Fl'are, le Détecteur d'événements, le Gestionnaire d'événements et le SGBD O_2, pour la détection des événements et l'exécution des règles est exactement celui utilisé dans NAOS : Fl'are remplace simplement le moteur d'exécution de NAOS.

Définition et représentation des règles : Dans NAOS, les règles sont représentées comme des objets O_2 persistants [Cou93]. Douze classes O_2 représentent les types d'événements, les Conditions, les Actions et les informations qui circulent entre les trois parties d'une règle : environnements d'évaluation et d'exécution. La structure de ces environnements dépend du type d'événement des règles et de leur mode d'exécution (immédiat ou différé).

Le point d'entrée de ce graphe de classes est la classe Rdef. Ses principaux attributs, pour chaque règle, représentent : son nom, sa priorité, son mode d'exécution et son mode de substitution de l'opération déclenchante (oui/non). Les règles sont stockées dans une liste O_2 persistante de nom gRdefs dans laquelle elles sont classées par priorités décroissantes.

Le modèle d'exécution d'une règle est plus sophistiqué dans Fl'are que dans NAOS : il n'y a pas de dépendance entre le mode d'exécution d'une règle et ses autres dimensions. Celles-ci doivent donc être représentées explicitement pour chaque règle. Ainsi, Rdef est redéfinie. De nouveaux attributs sont introduits pour représenter le

mode de traitement des événements, le mode de consommation des événements, le mode de prise en compte de l'effet net et le mode de préemption. La structure de la classe Rdef correspond au domaine des règles R de la sémantique dénotationnelle de Fl'are donnée dans le chapitre précédent.

Pour représenter l'appartenance d'une règle à un module, on peut soit ajouter un attribut dans Rdef et conserver ainsi la structure de gRdefs ; soit changer la structure de gRdefs. Nous choisissons la seconde solution qui présente l'avantage d'être plus explicite. Elle pourra également être plus adéquate pour des outils d'analyse statique de règles, des *explorateurs* ou *navigateurs* de règles, etc. gRdefs n'est donc plus une liste (plate) de règles mais une liste de listes de règles. Chacune des listes du premier niveau correspondant à un module.

Cette représentation est créée par le Constructeur de règles, aussi, ce module est modifié pour construire cette nouvelle représentation.

Enfin, ces changements dans la représentation des règles dans O_2 doivent évidemment être prise en compte dans les fonctions d'accès à cette représentation pour l'exécution des règles : il s'agit principalement de la foncion get_rule du module O2_interface qui appartient au moteur d'exécution.

6.4 Conclusion

Nous avons décrit l'expérimentation que nous menons afin de remplacer le moteur d'exécution du système NAOS par notre propre moteur d'exécution Fl'are et obtenir ainsi un SGBD actif – avec un modèle d'exécution paramétrique ! Nous avons détaillé l'architecture et l'implantation de NAOS afin de montrer que les composants de NAOS et les interfaces entre ces composants sont ré-utilisables pour implanter Fl'are, moyennant quelques modifications mineures.

En ce qui concerne l'implantation du moteur d'exécution Fl'are, nous avons montré qu'ici encore, une part non négligeable du moteur d'exécution de NAOS est réutilisable. Nous avons également montré que l'architecture proposée isole, beaucoup plus que dans NAOS, les processus propre au modèle d'exécution (ordonnancement, sélection) dans quelques composants (Rule et Rule_log) ; et que finalement, l'effort d'implantation à partir de la sémantique formelle de Fl'are, s'en trouve grandement facilité.

Signalons enfin que dans l'optique de fournir un système réellement utilisable basé sur le moteur d'exécution Fl'are, il convient de porposer un langage ou un interface de définition des règles qui prenne en compte les caractéristiques du modèle paramé-

trique. La définition d'un *interface d'accès* à Fl'are n'est pas une tâche triviale car l'expression des Conditions et des Actions n'est pas complètement indépendante du modèle d'exécution des règles : la construction des environnements des événements dépend du mode de traitement et du mode de consommation des événements, par exemple. Des travaux doivent être menés dans cette direction. Une solution pour masquer, autant que faire se peut la complexité de la définition des *règles Fl'are*, serait peut-être un interface de type *langage de formulaires* qui permettrait d'assister le programmeur dans l'écriture des règles.

Chapitre 7

Conclusion

> ***Péroraison*** *n. Explosion d'une fusée oratoire. Elle éblouit, mais pour un observateur ne manquant pas de nez, sa plus évidente caractéristique est l'odeur des différentes sortes de poudre utilisées dans sa préparation.*
> ***Ambrose Bierce - "Le dictionnaire du Diable"***

7.1 Résumé du travail effectué

Un SGBD actif est capable d'entreprendre automatiquement certaines actions en réponse à des événements spécifiques. La capacité de réaction d'un SGBD actif repose sur un mécanisme de règles actives dont le formalisme le plus unanimement adopté est le formalisme Événement-Condition-Action (E-C-A).

Le modèle de connaissances d'un SGBD actif décrit comment sont représentées et manipulées les données et les règles, et en particulier les Événements, les Conditions et les Actions. Le modèle d'exécution spécifie, quant à lui, quand et comment sont exécutées des règles actives déclenchées par des événements produits par l'exécution d'une transaction.

La simplicité de la sémantique la plus générale des règles actives – "lorsque l'événement E se produit, si la condition C est vérifiée alors exécuter l'action A" – n'est qu'apparente : les modèles d'exécution sont variés et le plus souvent complexes.

Les SGBD actifs se distinguent, à notre avis, principalement par leur modèle

d'exécution. Il est cependant très difficile, sans une méthodologie adaptée, d'évaluer et de comparer les nombreuses propositions qui ont été faites. Ceci permettrai pourtant de déterminer les dimensions essentielles de l'exécution des règles, de celles plus anecdotiques, et également de déterminer les interactions entre ces dimensions.

Notre étude spécifique des modèles d'exécution a mis en évidence plus de vingt dimensions qui caractérisent l'exécution d'une règle de manière isolée puis d'un ensemble de règles. L'exécution d'une règle dépend de certaines dimensions qui influent sur les trois phases de l'exécution globale de cette règle : le déclenchement, l'évaluation de la partie Condition et l'exécution de la partie Action. L'exécution d'un ensemble de règles dépend bien évidemment du modèle d'exécution de chacune des règles de cet ensemble, mais également du choix d'une stratégie d'exécution. Plusieurs stratégies ont été proposées afin de gérer de multiples règles déclenchées à un même point d'exécution et les exécutions de règles en cascade. Nous nous sommes également intéressé à l'exécution des règles au cours d'une application, c'est-à-dire plus spécifiquement aux interactions entre règles et transactions. Enfin, nous avons isolé un certain nombre de dimensions qui sont liées à l'exécution des règles dans de nombreux travaux mais que nous considérons, pour notre part, comme cosmétiques, ou pour le moins, fortement dépendantes des modèles de règles et non uniquement du ressort des modèles d'exécution. Cet aspect concerne principalement la désactivation et la ré-activation dynamiques des règles.

Nous avons proposé une **taxonomie** des modèles d'exécution des SGBD actifs et une **représentation graphique** – basée sur la taxonomie proposée – de ces modèles d'exécution. Cette représentation graphique permet d'avoir une perception immédiate d'un système particulier. Elle permet également de visualiser directement les analogies et les différences entre différents systèmes.

Nous retenons deux traits marquants de notre étude : l'*hétérogénéité* et la *rigidité* des modèles d'exécution des systèmes étudiés. En effet, il existe une grande diversité de modèles d'exécution. Cette diversité se traduit par le nombre imposant de dimensions et de valeurs de ces dimensions qui sont prises en compte dans les différents systèmes. Cependant, lorsque l'on s'intéresse à un système particulier, il n'offre qu'un sous-ensemble de l'ensemble des dimensions et valeurs possibles mais, surtout, il n'offre que certaines combinaisons de valeurs pour chaque dimension. Nous pensons que cette rigidité est rédhibitoire au vu du potentiel que l'on prête aux SGBD actifs et qui peut être estimé par le large spectre d'applications dans lesquelles l'utilisation de règles actives pourrait être bénéfique.

Partant du même constat, certains travaux ont alors mené des investigations quant

aux utilisations potentielles des règles actives afin de déterminer quelles caractéristiques doit avoir un modèle d'exécution pour telle utilisation ou tel type d'application – on parle de *typologies des règles actives*. L'objectif de ces travaux est de fournir un modèle d'exécution *ad-hoc*, idéal pour chaque type d'utilisation. Nous ne croyons guère à cette approche. Tout d'abord parce que les domaines d'utilisation et les types d'applications dans lesquelles on pourrait utiliser des règles actives sont chaque jour plus nombreuses. Ensuite, parce que, selon nous, tous les domaines d'utilisations mis à jour sont et seront de plus en plus appelés à coexister au sein d'une même application. Dans une application de Génie Logiciel, par exemple, on a besoin d'avoir des règles pour gérer l'intégrité de la base, pour faire de la notification, pour faire coopérer et communiquer différents outils, pour gérer des versions de programmes ou de documentation, etc.

Fl'are, le **modèle d'exécution paramétrique** pour SGBD actif que nous proposons, est quant à lui flexible, puissant et facile à appréhender.

Il est flexible car il permet de spécifier le comportement de modules de règles destinés chacun à une utilisation particulière, et ce, au sein d'une même application. Il a une structure en trois couches. La couche I gère l'exécution d'une règle. La couche II gère l'exécution d'un module de règles, c'est-à-dire les interactions entre les règles de ce module. La couche III gère les interactions entre les modules.

Il est facile à appréhender car il est basé sur une approche boîte-à-outils : le comportement des règles et des modules est spécifié simplement en choisissant la valeur des paramètres proposés parmi un ensemble prédéfini. De plus, tous les paramètres proposés sont indépendants : toutes les combinaisons de valeurs pour tous les paramètres sont autorisées et conduisent à des exécutions déterministes.

Nous avons illuster la facilité d'utilisation et la puissance de Fl'are en montrant comment l'utiliser pour *couvrir* une grande part des fonctions de différents systèmes existants.

La **formalisation des modèles d'exécution** des SGBD actifs "non purement relationnels" constitue une autre contribution de notre travail.

Nous avons, d'une part, présenté une sémantique dénotationnelle de Fl'are qui offre une description complète, précise et sans ambiguïté du modèle, que ne permet pas une description en langue naturelle.

Nous avons, d'autre part, comparé notre approche à d'autres travaux qui utilisent également la sémantique dénotationnelle, la sémantique opérationnelle ou d'autres formalismes afin d'évaluer et de comparer ces différents formalismes. Nous pensons que la sémantique dénotationnelle est bien adaptée à la description des modèles d'exécution

des SGBD actifs : elle en donne des descriptions concises, élégantes et relativement naturelles.

Enfin, une originalité de la sémantique dénotationnelle de Fl'are vis-à-vis de la plupart des approches formelles proposées – et en particulier vis-à-vis de la sémantique dénotationnelle du modèle d'exécution de NAOS [CC95c] que nous avions défini – concerne la **prise en compte du synchronisme** entre l'exécution des règles interrompues par l'exécution d'autres règles mais surtout entre l'exécution des règles et la transaction déclenchante.

Nous avons ensuite décrit la mise en œuvre de Fl'are qui est en cours. NAOS, le système de règles actives pour le SGBD O_2 développé dans l'équipe STORM, constitue le contexte de notre expérimentation. C'est un mécanisme de règles actives qui permet de définir des règles puis de détecter des événements et d'exécuter les règles correspondantes.

L'expérimentation que nous proposons consiste à remplacer le moteur d'exécution de NAOS par Fl'are. Nous avons montré, d'une part, que l'architecture modulaire de NAOS se prête très bien à cet *échange standard* et que bon nombre de constituants de son moteur d'exécution peuvent être ré-utilisés, moyennant des modifications mineures. Nous avons montré, d'autre part, que la sémantique dénotationnelle de Fl'are pouvait être utilisée comme une spécification implantable.

7.2 Principales contributions

Notre travail est centré sur les modèles d'exécution des SGBD actifs. Dans ce domaine, certains points que nous avons approfondi dans cette thèse et que nous rappelons ci-dessous nous paraissent être des contributions importantes :

1° taxonomie et représentation graphique des modèles d'exécution,
2° modèle d'exécution paramétrique,
3° formalisation de la sémantique du modèle proposé.

Ces points constituent, nous semble-t-il, un travail original par rapport aux précédents travaux dans ce domaine. Les premiers travaux sur les modèles d'exécution des SGBD actifs, représentés par HiPAC et "ses héritiers", se sont principalement intéressés aux rapports entre règles et transactions. Ils considèrent des modèles de transactions évolués, le plus souvent des transactions emboîtées. Les règles sont alors vues comme des transactions particulières. Ces travaux s'attachent principalement à décrire le synchronisme, la concurrence et la politique en cas d'échec des ces transactions. Plus récemment, d'autres travaux ([CFPT95c, PV95, CPW95a] se sont in-

téressés à la formalisation des modèles d'exécution dans le but d'offrir des *modèles génériques* qui seraient capable d'exprimer la sémantique des modèles d'exécution des SGBD actifs existants. Le but de ces travaux n'est (pour l'instant) pas de fournir des systèmes opérationnels mais uniquement de comparer les propositions existantes. Seul ACTO, "un système de règles à la sémantique paramétrable" [MFLS96], représente une approche comparable à la nôtre, et peut-être annonciatrice de nouveaux travaux sur le chemin vers des modèles d'exécution flexibles que nous avons tenté de débroussailler ?

7.3 Perspectives

La sémantique dénotationnelle de Fl'are constitue, en quelque sorte, une validation théorique mais il nous semble important de valider expérimentalement nos propositions en poursuivant l'implantation de notre modèle d'exécution paramétrique afin de tester sur des cas concrets les différentes possibilités d'exécution offertes.

Une autre approche que l'on peut qualifier d'*implantation légère* est envisageable. En effet, notre travail ne concerne que les modèles d'exécution et l'un des points les plus originaux concerne les différentes stratégies d'exécution possibles pour une même module de règles. Dans ce contexte, il n'est peut-être pas nécessaire de disposer de tous les services qu'offre un SGBD comme la gestion de la mémoire, la gestion des droits accès, la reprise après panne ; ni de tous les services qu'offre un mécanisme des règles actives, notamment aux niveaux des modèles d'événements.

Une telle implantation serait en outre certainement plus adaptée à notre perspective de construire des outils d'analyse du comportement d'ensembles de règles actives. Le besoin d'outils de **visualisation** et de **mise au point** d'applications bases de données actives a été maintes fois mis en avant : "un dessin vaut mieux que dix mille mots" dit le proverbe chinois !

Les travaux que nous avons cité [DJP94, Beh94b, CTZ95, For95, BGB95], constituent un premier pas dans cette direction. Cependant, s'ils nous paraissent correspondre à une *vocation pédagogique*, ils nous semblent mal adaptés à des bases de règles volumineuses utilisant des modèles d'exécution complexes tel que Fl'are.

Durant l'exécution d'une application bases de données active, de nombreuses informations doivent être visualisées : le déroulement de l'application, les occurrences d'événements (avec leur type, les opérations qui les ont produits, leur environnement, etc.), puis l'exécution des règles déclenchées par ces événements (changement d'état des règles, ordre d'exécution, déclenchements en cascade, synchronisation, etc.). La

visualisation est intrinsèquement dynamique puisqu'elle "suit" le déroulement de l'application, mais il peut être également important de conserver une *vision historique* qui visualiserait une exécution complète.

Dans une optique de visualisation – qui constituerait une première étape dans cet axe de recherche – l'utilisateur ne peut pas intervenir sur l'exécution. Dans une optique de mise au point – qui constituerait une seconde étape beaucoup plus complexe – l'utilisateur peut agir sur l'exécution en modifiant la valeurs de certains paramètres de l'exécution.

Cette perspective se concrétiserait selon nous par un *banc de montage d'applications Fl'are* qui utiliserait des techniques de visualisation tri-dimentionnelles (parce qu'il y a beaucoup d'informations à visualiser) dans un environnement multi-vues (afin de fournir différentes visions d'une même information : vision dynamique et historique, liens entre règles et transactions, reconnaissance des événements, exécutions cascadées, etc.)

Une des hypothèses fondatrices de nos travaux est la prise en compte d'un modèle de transactions classique. Pour carricaturer, nous avons fait cette hypothèse pour montrer que la complexité des modèles d'exécutions des SGBD actifs ne se limite pas "à la spécification des modes de couplage, à la mise en correspondance des règles et des transactions, et à la structure et la synchronisation des transactions" [Buc94]. Nous espérons avoir démontré, dans les chapitres 3, 4 et 5, qu'un modèle d'exécution pouvait déjà être complexe même dans le cadre d'un modèle de transactions bien connu et simple !

Notre objectif de construire un modèle d'exécution conduisant à des exécutions déterministes nous a conduit à imposer un ordre total, d'une part, entre les modules, et d'autre part, entre les règles d'un module. L'ordre total entre les modules (couche III ou exécution inter-modules) représente cependant une limitation de notre modèle. En effet, il est fort probable qu'un programmeur veuille spécifier que tel module est plus prioritaire que tous les autres (un module regroupant des règles vérifiant des contraintes d'intégrité par exemple). Cependant, il est également probable, qu'il ne veuille pas spécifier de priorités entre tel ou tel autre module ou qu'il veuille, finalement, spécifier des *classes de priorités*. D'après nos recherches, il s'avère que ceci est difficilement réalisable avec un modèle de transactions classique. Notre hypothèse d'un modèle de transactions classique est sans doute trop restrictive dans ce contexte. Une troisième perspective de nos travaux est donc de prendre en compte des modèles de transactions plus évolués, tel le *modèle de transactions multi-niveaux* qui est, en outre, le modèle adopté par l'ODMG [OMG92] – sans pour autant refaire le travail

qui a été fait dans le cadre de HiPAC et de ses héritiers.

Par ailleurs, nous pensons qu'il serait vraiment temps de s'intéresser aux utilisations des règles actives et plus généralement aux applications bases de données actives afin d'évaluer les nombreuses propositions (dont les nôtres !) qui ont été faites en ce qui concerne les modèles d'exécution des SGBD actifs. Il nous semble important d'offir des méthodes d'analyse et de conception, bref, une méthodologie pour assister le programmeur dans le développement d'applications actives.

Enfin, dans cette optique "applications", il serait certainement intéressant d'étudier l'utilisation des règles actives, dans les SGBD distribués, répartis et temps-réel. Il nous semble, au premier abord, que ces types de systèmes pourraient nécessiter une puissance et une flexibilité dans l'exécution des règles qui va dans le sens de nos travaux.

Nous pensons plus particulièrement au domaine des SGBD temps-réel qui nous semblent liés aux SGBD actifs sur de nombreux points [BH95]: prise en compte du temps, problèmes d'ordonnancement, d'interruption, de terminaison, de performances, etc. Systèmes temps-réel et systèmes actifs ont également en commun des solutions d'architecture et d'implantation (implantation en mémoire principale, contrôle de concurrence allégé, etc.), et surtout des domaines d'application : contrôle de procédés industriels, conception et fabrication assistés par ordinateurs, contrôle et gestion (réservation, billeterie) de trafic aérien, maritime, ferroviaires, applications bancaires et boursières, etc.

Bibliographie

[ABD+92] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik, *The object-oriented database system manisfesto*, Building an Object-Oriented Database System - The Story of O2 (F. Bancilhon, C. Delobel, and P. Kanellakis, eds.), Morgan Kaufmann, 1992, pp. 3–20.

[AC93] M. Adiba and C. Collet, *Objets et bases de données - le SGBD o_2*, Hermes, 1993.

[ACC+93] M. Adiba, C. Collet, T. Coupaye, P. Habraken, J. Machado, H. Martin, and C. Roncancio, *Trigger Systems: Different Approaches*, Technical Report Aristote – SUR007, LGI-IMAG, France, June 1993.

[ACL91] R. Agrawal, R.J. Cochrane, and B.G. Lindsay, *On Maintening Priorities in a Production Rule System*, Proc. of the 17th Int. Conf. on Very Large Data Bases (Barcelona, Spain), September 1991, pp. 479–487.

[Adi93] M. Adiba, *Temps et versions dans les bases de données: passé, présent et avenir*, Notes de cours postgrade, Ecole Polytechnique Fédérale de Lausanne, Mai 1993.

[AE92] D. Agrawal and A. El Abbadi, *Transaction management in database systems*, Database Transactions Models for Advanced Applications (A.K. Elmagarmid, ed.), Morgan Kaufmann, 1992, pp. 1–31.

[AG89] R. Agrawal and N. Gehani, *Ode (Object Database and Environment): The Language and the Data Model*, Proc. of the ACM SIGMOD Int. Conf. on Managment of Data (Portland, USA) (J. Clifford (B. Lindsay) and D. Maier, eds.), June 1989.

[AV91] S. Abiteboul and V. Vianu, *Generic computation and its complexity*, Proc. of ACM SIGACT Symp. on the Theory of Computing, 1991, pp. 209–219.

[AV95] S. Abiteboul and V. Vianu, *Computing with first-order logic*, Journal of computer and System Sciences (1995).

[AWH92] A. Aiken, J. Widom, and J. M. Hellerstein, *Behavior of database production rules: termination, confluence, and observable determinism*, Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data (San Diego), ACM Press, May 1992, pp. 59–68.

[BaJS91] P. Butterworth and A. Otis ans J. Stein, *The GemStone Object Database Management System*, Communications of the ACM **34** (1991), no. 10, 64–77.

[BB95] H. Branding and A. Buchmann, *On providing soft and hard real-time capabilities in an active DBMS*, Proc. of the Int. Workshop on Active and Real-Time Database Systems (Skovde - Sweden), Workshops in Computing, Springer Verlag, June 1995.

[BBKZ92] A. P. Buchman, H. Branding, T. Kudrass, and J. Zimmermann, *REACH: A REal-time, ACtive and Heterogeneous mediator system*, IEEE Bulletin of the Technical Committee on Data Engineering **15** (1992), no. 1-4.

[BBKZ93] A. P. Buchman, H. Branding, T. Kudrass, and J. Zimmermann, *Rules in an open system: The REACH rule system*, Proc. of the 1st Workshop on Rules in Databases Systems, (RIDS'93) (Edinburgh, UK) (N. Paton et M. Williams, ed.), Workshops in Computing, Springer Verlag, September 1993, pp. 111 – 126.

[BCP95] E. Baralis, S. Ceri, and S. Paraboschi, *Improved Rule Analysis by Means of Triggering and Activation Graphs*, Proc. of the 2nd Workshop on Rules in Databases Systems, RIDS'95 (Athens, Greece) (T. Sellis, ed.), Springer Verlag, September 1995, pp. 165–181.

[BDK92] F. Bancilhon, C. Delobel, and P. Kanellakis, *Building an Object Oriented Database - The story of O_2*, Morgan Kaufmann, 1992.

[Beh93] H. Behrends, *ADL-Activity Description Language*, Technical Report 3-93, Rostock university, 1993.

[Beh94a] H. Behrends, *An Operational Semantics for the Activity Description Language ADL*, Technical Report 4-94, Oldenburg university, 1994, To be published.

[Beh94b] H. Behrends, *Simulation-based debugging of active databases*, Proc. of the 4th International Workshop on Research Issues in Data Engineering: Active Database Systems (Houston, Texas), 1994.

[BGB95] E. Benazet, H. Guehl, and M. Bouzeghoub, *VITAL: a Visual Tool for Analysis of Rules Behavior in Active Databases*, Proc. of the 2nd Workshop on Rules in Databases Systems, RIDS'95 (Athens, Greece) (T. Sellis, ed.), Springer Verlag, September 1995, pp. 182–196.

[BH95] M. Berntsson and Jörgen Hansson (eds.), *Proc. of the Int. Workshop on Active and Real-Time Database Systems*, Lecture Notes in Computer Science, Skövde, Sweden, Springer Verlag, September 1995.

[Bid92] N. Bidoit, *Bases de Données Déductives*, Armand Colin, Paris, France, 1992.

[BK91] N.S. Barghouti and G.E. Kaiser, *Concurrency control in advanced database applications*, ACM Computing Surveys **23** (1991), no. 3, 269–317.

[BM91] C. Beeri and T. Milo, *A model for active object oriented database*, Proc. of the 17th Int. Conf. on Very Large Data Bases (Barcelona - SP), September 1991, pp. 337–349.

[Buc94] A.P. Buchmann, *Active Object Systems*, Advances in Object-Oriented Database Systems (A. Dogac, M.T. Özsu, A. Biliris, and T. Sellis, eds.), NATO ASI Series, Springer Verlag, 1994, pp. 201–224.

[BW94] E. Baralis and J. Widom, *An Algebraic Approach to Rule Analysis in Expert Database Systems*, Proc. of the 20th Int. Conf. on Very Large Data Bases (Santiago, Chile) (J. Bocca and C. Zaniolo, eds.), September 1994, pp. 463–474.

[BZBW95] A. P. Buchman, J. Zimmermann, J. A. Blakeley, and D. L. Wells, *Building an integrated active OODBMS: Requirements, architecture and*

design decisions, Proc. of the 11thInt. Conf. on Data Engineering (Taipei, Taiwan), IEEE, IEEE Computer Society Press, September 1995, pp. 292–308.

[CAD+94] R. Cattel, T. Atwood, J. Dubl, G. Ferran, M. Loomis, and D. Wade, *The object database standart: Odmg-93*, Morgan Kaufmann, 1994.

[CAM93] S. Chakravarthy, E. Anwar, and L. Maugis, *Design and Implementation of Active Capability for an Object-Oriented Database*, Tech. Report UF-CIS-TR-93-001, University of Florida, Gainesville, January 1993.

[CBB+89] S. Chakravarthy, B. Blaustein, A. Buchmann, M. Carey, U. Dayal, D. Goldhirsh, M. Hsu, R. Jauhari, R. Ladin, M. Livny, D. Mc.Carthy, R. McKee, and A. Rosenthal, *Hipac: a research project in active time-constrained database management. technical report.*, Tech. report, Xerox Advanced Information Technology, 1989.

[CC95a] C. Collet and T. Coupaye, *NAOS Native Active Object-Oriented Database System (Démonstration de Prototypes)*, Actes des 11èmes Journées Bases de Données Avancées (Nancy - France), August 30 - September 1 1995.

[CC95b] C. Collet and T. Coupaye, *The NAOS System*, Proc. of the 1995ACM-SIGMOD Symp. on the Management of Data (Exhibit program) (San Jose - California), May 23-25 1995.

[CC95c] T. Coupaye and C. Collet, *Denotationnal semantics for an active rule execution model*, Proc. of the 2nd Int. Workshop on Rules in Database Systems, RIDS'95 Lecture Notes in Computer Science 985 (Athens, Greece) (T. Sellis, ed.), Springer Verlag, Athens, Greece, September 1995, pp. 36–50.

[CC96a] C. Collet and T. Coupaye, *Composite Events in NAOS*, Proc. of the 7th Int. Conf. on Database and Expert Systems Applications(DEXA'96) (Zurich - Switzerland), September 9-13 1996.

[CC96b] C. Collet and T. Coupaye, *Primitive and Composite Events in NAOS*, Actes des 12ièmes Journées Bases de Données Avancées (Cassis - France), September 1996.

[CCR96] C. Collet, T. Coupaye, and C. Roncancio, *NAOS 2.1: Dealing with Composite Events*, Proc. of the 5th Int. Conf. on Extending Database Technology - EDBT'96 (Exhibit program) (Avignon, France), May 1996.

[CCS94a] C. Collet, T. Coupaye, and T. Svensen, *Efficient and modular reactive capabilities in an object-oriented database system*, Actes des 10èmes Journées Bases de Données Avancées (Clermont-Ferrand, France), September 1994.

[CCS94b] C. Collet, T. Coupaye, and T. Svensen, *NAOS efficient and modular reactive capabilities in an object-oriented database system*, Proc. of the 20th Int. Conf. on Very Large Data Bases (Santiago, Chile), September 1994, pp. 132–143.

[CF95] S. Ceri and P. Fraternali, *Draft of the IDEA Methodology*, Technical Report IDEA.DD.22P.001.02, Politecnico di Milano, march 1995, Draft.

[CFPT95a] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca, *Active Rule Management in Chimera*, Active Database Systems (San Mateo, CA, USA) (Morgan Kaufmann, ed.), S. Ceri and J. Widom, San Mateo, CA, USA, August 1995.

[CFPT95b] S. Comai, P. Fraternali, G. Psaila, and L. Tanca, *A Customizable Model for the Semantics of Active Databases*, Proc. of the 6th IFIP Tc-2 Working Conference on Data Semantics, DS-6 (Stone Mountain, Atlanta, Georgia, USA) (R. Meersman and L. Mark, eds.), Chapman and Hall, June 1995.

[CFPT95c] S. Comai, P. Fraternali, G. Psaila, and L. Tanca, *A Uniform Model to Express the Behavior of Rules with Different Semantics*, Proc. of the 2nd Workshop on Active and Real-Time Database Systems, ARTDB'95 (Skovde, Sweden) (M. Berndtsson and J. Hansson, eds.), Springer Verlag, September 1995, pp. 190–208.

[Cha89] S. Chakravarthy, *Rule Management and Evaluation: An Active DBMS Perspective*, SIGMOD Record **18** (1989), no. 3, 20–28.

[Cha93] S. Chakravarthy, *A Comparative Evaluation of Active Relational Databases*, Tech. Report UF-CIS-TR-93-002, University of Florida, Gainsville, USA, January 1993.

[CHCA94] C. Collet, P. Habraken, T. Coupaye, and M. Adiba, *Active rules for the GOODSTEP software engineering platform*, Proc. of the 2nd Int. Workshop on Database and Software Engineering (Sorrento, Italy), May 1994.

[CHR96] C. Collet, P. Habraken, and C. Roncancio, *Règles actives dans les SGBD*, Ingénierie des systèmes d'information **4** (1996), no. 3.

[CM93] S. Chakravarthy and D. Mishra, *Snoop: An Expressive Event Specification Language For Active Databases*, Tech. Report UF-CIS-TR-93-007, University of Florida, Gainesville, March 1993.

[CM95] C. Collet and J. Machado, *Optimization of Active Rules with Parallelism*, Proc. of the Int. Workshop on Active and Real-Time Database Systems (Skovde - Sweden), June 1995.

[Cod70] E. F. Codd, *A relationnal model of data for large shared data banks*, Communications of ACM **13** (1970), no. 6.

[Col96] C. Collet, *Bases de Données Actives : des systèmes relationnels aux systèmes à objets*, Diplome D'Habilitation à Diriger les Recherches (DHDR), LSR-IMAG, Université Joseph Fourier, October 1996.

[Cou93] T. Coupaye, *Langage et Compilateur de Règles Actives pour le SGBD O_2*, Rapport Aristote MEM017, LGI-IMAG, Université de Grenoble, September 1993.

[CPW95a] J. Campin, N. Paton, and M. H. Williams, *Specifying Active Database Systems in an Object-Oriented Framework*, Technical report, Heriot-Watt University, Riccarton, Edimburgh, UK, June 1995.

[CPW95b] J. Campin, N. Paton, and M. H. Williams, *A structured specification of an active database system*, Information and Software Technology **37** (1995), no. 1, 47–61.

[CRW77] J. A. Mac Call, P. K. Richards, and G. F. Walters, *Factors in Software Quality*, Tech. report, US Rome Air development Center Volumes I, II, and III, Report NTIS ADA-049 014,015,055, Nationnal Technical Information Service, US Department, 1977.

[CTZ95] S. Chakravarthy, Z. Tamizuddin, and J. Zhou, *A Vizualisation and Explanation Tool for Debugging ECA Rules in Active Databases*, Proc. of the 2nd Workshop on Rules in Databases Systems, RIDS'95 (Athens, Greece) (T. Sellis, ed.), Springer Verlag, September 1995, pp. 197–209.

[DA82] C. Delobel and M. Adiba, *Bases de données et systèmes relationnels*, Dunod, 1982.

[Da86] K. Dittrich and al., *An event/trigger mechanism to enforce complex consistency constraints in design databases*, SIGMOD Record **15** (1986), no. 3.

[DA94] P. Dechamboux and M. Adiba, *Peplom: Experimenting a Database Programming Language*, Ingénierie des systèmes d'information (1994).

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, and S. Sarin, *The HIPAC Project: Combining Active Databases and Timing Constraints*, SIGMOD Record **17** (1988), no. 1, 51–69.

[DBC96] U. Dayal, A. Buchman, and C. Chakravarthy, The HiPAC Project, ch. 7, Morgan Kaufman Publishers Inc., 1996.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy, *Rules are objects too: a knowledge model for an active, object-oriented database system*, Proc. of the 2nd Int. Workshop on Object-Oriented Database Systems, September 1988, pp. 129–143.

[DGG95] K. R. Dittrich, S. Gatziu, and A. Geppert, *The Active Database Management System Manifesto: A Rulebase of ADBMS Features*, Proc. of the 2nd Int. Workshop on Rules in Database Systems, RIDS'95 Lecture Notes in Computer Science 985 (Athens, Greece) (T. Sellis, ed.), Springer Verlag, Athens, Greece, September 1995, pp. 3–17.

[DHW95] U. Dayal, E. Hanson, and J. Widom, *Active Database Systems*, Modern Database Systems (ACM-PRESS Won Kim, ed.), Addison-Wesley Publishing Comp., 1995, pp. 435–456.

[Dia95] O. Diaz, *Dimensions of Active Database Systems*, Actes des 10èmes Journées Bases de Données Avancées (Nancy, France), September 1995, pp. 3–22.

[DJ94] O. Díaz and A. Jaime, *EXACT: an EXtensible approach to ACTive object-oriented databases*, Tech. report, Workshop on Active DBMS, Dagsthul - Allemagne, March 1994.

[DJP94] O. Diaz, A. Jaime, and N. Paton, *DEAR: a DEbugger for Active Rules in an object-oriented context*, Proc. of the 1stWorkshop on Rules in Database Systems (Edimburgh, Scotland) (N. W. Paton and M. H. Wiliams, eds.), Springer Verlag, 1994.

[DKRS91] R. Duke, P. King, G. A. Rose, and G. Smith, *The Object-Z Specification Language: Version 1*, Technical Report 91-1, Department of Computer Science, University of Queensland, St. Lucia 4072, Australia, April 1991.

[DLR91] C. Delobel, C. Lécluse, and P. Richard, *Bases de données: des systèmes relationnels aux systèmes à objets*, InterEditions, 1991.

[dMS88] C. de Maindreville and E. Simon, *Modelling non-deterministic queries and updates in a deductive database*, Proc. of the 14th Int. Conf. on Very Large Data Bases, August 1988.

[DP94] O. Diaz and N. Paton, *Making Object-Oriented Databases Extensible through Metaclasses: an experience*, IEEE Software (94).

[DR94a] P. Dechamboux and C. Roncancio, *Integrating Deductive Capabilities into an Object-Oriented Database Programming Language*, Proc. of the 10ièmes Journées Bases de Données Avancées (Clermont-Ferrand, France), September 1994, pp. 171–185.

[DR94b] P. Dechamboux and C. Roncancio, *peplomd: an Object-Oriented Database Programming Language Extended with Deductive Capabilities*, Int. Conf. on Database and Expert Systems Applications, Lecture Notes in Computer Science, no. 856, September 1994, pp. 2–14.

[For] T. Forse, *Qualimétrie des systèmes complexes*, Les Éditions d'Organisation.

[For95] T. Fors, *Vizualization of Rule Behavior in Active Databases*, Proc. of the 3rd Working Conference on Visual Database Systems, VDB-3 (Lauzanne, Switzerland) (Working Group 2.6 IFIP, ed.), March 1995, pp. 191–209.

[FT95] P. Fraternali and L. Tanca, *A Structured Approach for the Definition of the Semantics of Active Databases*, ACM Transactions on Database Systems **20** (1995), no. 4, 414–471.

[GD94] S. Gatziu and K. Dittrich, *Detecting composite events in active database systems using petri nets*, Proc. of the 4th Int. Workshop on Research Issues in Data Engineering: Active Database Systems (Houston), IEEE, Houston, February 1994.

[GGD91] S. Gatziu, A. Geppert, and K.R. Dittrich, *Integrating Active Concepts into an Object-Oriented Database System*, Proc. of the 3rd Int. Workshop on Database Programming Languages: Bulk Types & Persistent Data (Nafplion, Greece), Morgan Kaufmann, 1991, pp. 399–415.

[GGD93] S. Gatziu, A. Geppert, and K.R. Dittrich, *Events in an Active Object-Oriented Database System*, Proc. of the 1stWorkshop on Rules in Database Systems (Edinburg), August 1993.

[GJ91] N. Gehani and H.V. Jagadish, *Ode as an Active Database: Constraints and Triggers*, Proc. of the 17th Int. Conf. on Very Large Data Bases (Barcelona, Spain), September 1991, pp. 327–336.

[GJS92] N. Gehani, H.V. Jagadish, and O. Shmueli, *Event Specification in an Active Object-Oriented Database*, Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data (San Diego, California), 1992, pp. 81–90.

[GM87] Y. Gurevich and J. M. Morros, *Algebrais Operationnal Semantics and Modula-2*, Proc. of CSL'97 1st Workshop on computer Science Logic (Karlsruhe, Germany) (E. Börger, H. Kleine Büning, and M.M. Richter, eds.), Springer Verlag LNCS 329, 1987.

[GOO94] GOODSTEP Team, *The GOODSTEP Project: General Object-Oriented Database for SofTware Engineering Processes*, Proc. of the Asia-Pacific Software Engineering Conference (Tokyo, Japan), IEEE Computer Society Press, 1994, pp. 10–19.

[GR93] J. Gray and A. Reuter, *Transaction processing: Concepts and techniques*, Morgan Kaufmann, 1993.

[Gun92] C. A. Gunter, *Semantics of Programming Languages, Structures and Techniques*, The MIT Press, 1992.

[Gur91] Y. Gurevich, *Evolving Algebras: An Attempt To Discover Semantics*, Bull. EATCS (1991), no. 43, 264–284.

[GV91] G. Gardarin and P. Valduriez, *SGBD Avancés - Bases de Données Objets, Déductives, Réparties*, Eyrolles, 1991.

[Hab93] P. Habraken, *Un système de règles actives pour le sgbd o_2*, Tech. report, Mémoire CNAM, Université Joseph Fourier, Grenoble, 1993.

[Han89] E. Hanson, *An initial report on the design of Ariel a DBMS with an integrated production rule system*, SIGMOD Record **18** (1989), no. 3, 12–19.

[Han92] E. Hanson, *Rule condition testing and action execution in Ariel*, Proc. of the 1992 ACM SIGMOD Int. Conf. on Management of Data (San Diego, California), June 1992, pp. 281–290.

[HB91] E. N. Hanson and W. R. Baker, *The design and implemantation of the Ariel active database rule system*, Rapport technique, Univ. of FLorida, Gainsville, October 1991.

[Hin86] J. R. Hindley, *Introduction to Combinators and λ-Calculus*, Cambridge University Press, Cambridge, England, 1986.

[HJ91] R. Hull and D. Jacobs, *Language Constructs for Programming Active Databases*, Proc. of the 17th International Conference on Very Large Data Bases (Barcelona, Spain), 1991, pp. 455–467.

[HLM88] M. Hsu, R. Ladin, and D. McCarthy, *An execution model for active database management systems*, Proc. of the 3rd Int. Conf. on Data and Knowledge Bases, June 1988, pp. 171–179.

[HW92] E. N. Hanson and J. Widom, *An Overview of Production Rules in Database Systems*, Tech. Report CIS-TR-92-031, University of Florida, Gainsville, USA, october 1992.

[IEE88] *Special issue on temporal databases*, Data Engineering Bulletin **11** (1988), no. 4.

[Ing91] Ingres, *Ingres/sql reference manual*, 1991.

[JF95a] U. Jaeger and J. C. Freytag, *An Annotated Bibliography on Active Databases*, Tech. report, Humboldt-Universität zu Berlin, Germany, june 1995, Version longue de [JF95s].

[JF95b] U. Jaeger and J. C. Freytag, *An Annotated Bibliography on Active Databases (Short Versionn)*, SIGMOD-RECORD **25** (1995), no. 1.

[KDM88] A. Kotz, K. Dittrich, and J. Mulle, *Supporting semantic rules by a generalized event/trigger mechanism*, Proc. of the 1988 Int. Conf on Extending Database Technology (Venice - Italy), March 1988.

[Kim90] W. Kim, *Introduction to object oriented databases*, MIT Press, 1990.

[Kim95] W. Kim, *Modern database systems*, Addison-Wesley Publishing Comp., 1995.

[KMS90] J. Kiernan, C. Mandreville, and E. Simon, *Making deductive database a practical technology: a step forward*, Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data (Atlantic City), May 1990.

[KRRV94a] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and S. Vieweg, *The TriGS Execution Model, Implementing Active Concepts on Top of Commercial ooDBMS*, Tech. report, Univ. of Viena - Univ. of Linz, Austria, 1994.

[KRRV94b] G. Kappel, S. Rausch-Schott, W. Retschitzegger, and S. Vieweg, *TriGS Making a Passive Object-Oriented Database System Active*, JOOP (1994).

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb, *The ObjectStore Database System*, Communications of the ACM **34** (1991), no. 10.

[LLPS91] G. Lohman, B. Lindsay, H. Pirahesh, and K. Schiefer, *Extensions to Starburst: Objects, Types, Functions, and Rules*, Communication of the ACM **34** (1991), no. 10, 94–109.

[LRV92] C. Lecluse, P. Richard, and F. Velez, *O_2, an Object-Oriented Data Model*, Building an Object-Oriented Database - The story of O_2, Morgan Kaufmann, 1992, pp. 77–97.

[Mac86] E. MacKenzie, *Bibliography : Temporal databases*, ACM SIGMOD RECORD **15** (1986), no. 4.

[Mac95] J. C. Machado, *Parallélisme et Transactions dans les Bases de Données à Objets*, Ph.D. thesis, Université Joseph Fourier, Grenoble, October 1995.

[MaSHHG92] F. Manola, M. L. Brodie ans S. Heiler, M. Hornick, and D. Georgakopoulos, *Distributed Object Management*, Journal of Intelligent and Cooperative Information Systems (1992).

[MD89] D. McCarthy and U. Dayal, *The Architecture of an Active Data Base Management System*, Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of Data (Portland, Oregan), ACM Press, May 1989, pp. 215–223.

[MFLS96] M. Matulovic, F. Fabret, F. Llirbat, and E. Simon, *Architecture d'un Système de Règles à la Sémantique Paramétrable*, Actes des 12ièmes Journées Bases de Données Avancées (Cassis - France), September 1996, À paraitre.

[Mos85] J.E.B. Moss, *Nested transactions: An approach to reliable distributed computing*, The Mit Press, Cambridge, Ma, 1985.

[MPC96] R. Meo, G. Psaila, and S. Ceri, *Composite Events in Chimera*, Proc. of the 5th International Conference on Extending Database Technology, EDBT'96Lecture Notes in Computer Science 1057 (Avignon, France) (P. Apers, M. Bouzeghoub, and G. Gardarin, eds.), Springer Verlag, Avignon, France, march 1996, pp. 56–78.

[NG92] O. Shmueli N.H. Gehani, H.V. Jagadish, *Composite event specification in active databases: model & implementation*, Proc. of the 18th Int. Conf. on Very Large Data Bases, September 1992.

[ODV94] M.T. Özsu, U. Dayal, and P. Valduriez, *Distributed object management*, Morgan Kaufmann, 1994.

[OMG92] OMG, *Object Management Architecture Guide*, Morgan Kaufmann, Second Edition, Object Management Group, September 1992.

[Ora92] Oracle, *Oracle 7 reference manual*, Oracle corporation, 1992.

[OV91] M.T. Özsu and P. Valduriez, *Principles of distributed database systems*, Prentice-Hall, 1991.

[Pap86] C.H. Papadimitriou, *The theory of database concurrency control*, Computer Science Press, 1986.

[Pat89] N. Paton, *Adam: An Object-Oriented Database System implemented in Prolog*, Proc. of the British National Conference on Databases (M. H. Wiliams, ed.), Cambridge University Press, 1989, pp. 147–161.

[PCFW95] N.W. Paton, J. Campin, A.A.A. Fernandes, and M.H. Williams, *Formal Specification of Active Database Functionality: A Survey*, Proc. of the 2nd Int. Workshop on Rules in Database Systems, RIDS'95 Lecture Notes in Computer Science 985 (Athens, Greece) (T. Sellis, ed.), Springer Verlag, Athens, Greece, September 1995, pp. 21–35.

[Pic95] P. Picouet, *Puissance d'expression et consistance sémantique des bases de données actives*, Ph.D. thesis, École Nationale Supérieure des Télécommunications, Paris, December 1995.

[PV95] P. Picouet and V. Vianu, *Semantics and Expressinveness Issues in Active Databases*, Proc. of the 14th ACM Symposium on Principles of Database Systems (San Jose, CA, USA), May 1995, pp. 126–138.

[Ron94] C. Roncancio, *Règles actives et règles déductives dans les bases de données à objets*, Ph.D. thesis, Université Joseph Fourier, Grenoble I, LGI-IMAG, December 1994.

[Ros92] G. A. Rose, Object-Z, ch. Object-Orientation in Z, pp. 59–77, Springer Verlag, 1992, pp. 59–77.

[RS87] L. Rowe and M. Stonebraker, *The Postgres data model*, Proc. of the 13th Int. Conf. on Very Large Data Bases (Brighton, England), September 1987.

[Sch88] D. A. Schmidt, *Denotational Semantics*, Wm. C. Brown Publishers, Dubuque, Iowa, 1988.

[SGJ93] R. Hull S. Ghandeharizadeh and D. Jacobs, *On Implementing a Language for Specifying Active Database Execution Models*, Proc. of the 19th International Conference on Very Large Data Bases (Dublin, Ireland), 1993, pp. 441–454.

[SHH87] M. Stonebraker, E. Hanson, and C. H. Hong, *The design of the Postgres rule system*, Proc. of the 3rdInt. IEEE Conf. on Data Engineering, 1987.

[SHP89] M. Stonebraker, M. Hearst, and S. Potomanios, *A commentary on the POSTGRES rules system*, SIGMOD Record **18** (1989), no. 3, 5–11.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, *On rules, procedures, caching and views in data base systems*, Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data (Atlantic City), ACM Press, May 1990.

[SK95] E. Simon and J. Kiernan, *The A-RDL System*, Morgan-Kaufmann publisher, San Francisco, California, Editor: J. Widom and S. Ceri and U. Dayal, June 1995.

[SKD95] E. Simon and A. Kotz-Dittrich, *Promisies and realities of active database systems*, Proc. of the 21th Int. Conf. on Very Large Data Bases, 1995.

[SKM92] E. Simon, J. Kierman, and C. Maindreville, *Supporting Deductive and Active Rules on Top of a Relational DBMS*, Rr1580, INRIA - Rocquencourt (France), January 1992.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan, *An architecture for transforming a passive DBMS into an active DBMS*, Proc. of the 17th Int. Conf. on Very Large Data Bases (Barcelona), September 1991, pp. 469–478.

[Spi92] J. M. Spivey, *The Z notation: a reference manual*, Prentice Hall, 1992, (2nd edition).

[SR86] M. Stonebraker and L. Rowe, *The design of Postgres*, Proc. of the 1986 ACM SIGMOD Int. Conf. on Management of Data (Wahington D.C., USA), ACM Press, June 1986.

[Sto77] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, Massachussets, 1977.

[Sto92] M. Stonebraker, *The integration of rules systems and DB systems*, IEEE Transactions on Knowledge and Data Engineering **4** (1992), no. 5, 415–423.

[Syb92] Sybase, *Transact-sql user's guide*, Sybase Inc., 1992.

[TC92] A. Tchounikine and C. Chrisment, *A behaviour rule based approach for active object oriented dbms design*, DEXA92 (Valence, Spain), September 1992.

[TCG+93] A. U. Tansel, J. Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (eds.), *Temporal databases*, The Benjamins/Cummings Publishing Company, 1993.

[Ten76] R. D. Tennent, *The Denotational Semantics of Programming Languages*, Communications of the ACM **19** (1976), no. 8, 437–453.

[Ten91] R. D. Tennent, *Semantics of Programming Languages*, Prentice Hall International Series in Computer Science, 1991.

[Ull88] J. D. Ullman, *Principles of database and knowledge-base systems*, Computer Science Press, 1988.

[vdVK91] M.H. van der Voort and M.L. Kersten, *Facets of database triggers*, Tech. Report PCS-R9122, Center for Mathematics and Computer Science, Amsterdam, April 1991.

[WCL91] J. Widom, R.J. Cochrane, and B.G. Lindsay, *Implementing Set-Oriented Production Rules as an Extension to Starburst*, Proc. of the 17th Int. Conf. on Very Large Data Bases (Barcelona, Spain), September 1991, pp. 275–285.

[WF90] J. Widom and S. J. Finkelstein, *Set Oriented Production Rules in Relationnal Database Systems*, Proc. of the 1990 ACM SIGMOD Int. Conf. on Management of Data, SIGMOD Record, ACM Press, May 1990, pp. 259–270.

[Wid92] J. Widom, *A Denotational Semantics for the Starburst Production Rule Language*, SIGMOD Record **21** (1992), no. 3, 4–9.

[Wid93] J. Widom, *Deductive and Active Database: Two Paradigms or Ends of a Spectrum?*, Proc. of the 1st Workshop on Rules in Databases Systems, (RIDS'93) (Edinburgh, UK) (N. Paton et M. Williams, ed.), Workshops in Computing, Springer Verlag, September 1993, pp. 306 – 315.

[Wid94] J. Widom, *Research Issues in Active Database Systems: Report from the Closing Panel at RIDE-ADS' 94*, SIGMOD RECORD **23** (1994), no. 3, 41–43.

[Win93] G. Winskel, *The Formal Semantics of Programming Languages, An Introduction*, The MIT Press, 1993.

Annexe A

Principaux travaux utilisés

Nous introduisons ici les principaux systèmes de règles actives que nous avons étudiés et qui sont cités dans ce document. Nous ne prétendons pas être exhaustif. À l'instar de l'ensemble de ce document, nous nous focalisons sur les travaux qui sont significatifs vis-à-vis des modèles d'exécutions. Nous mentionnons égalemment les principaux travaux de synthèse que l'on trouve parmi l'abondante littérature sur les bases de données actives.

A.1 Systèmes et prototypes

A.1.1 Systèmes relationnels

A-RDL RDL-1 [dMS88, KMS90] est un environnement de programmation basé sur le formalisme des règles déductives (règles de production), proposé comme processeur frontal d'un SGBD relationnel. Le langage RDL-1 est une extension du langage Datalog [Bid92] développée à l'INRIA-Rocquencourt sur le système SABRINA. Il supporte entre autres la négation et la définition de nouveaux types par l'utilisateur. A-RDL est une extension du langage RDL-1 qui incorpore des règles actives [SKM92, SK95].

Ariel Ariel [Han89] est un projet du "Wright Research and Development Center" de la "Wright State University" puis de l'université de Floride, Gainsville. Un prototype d'Ariel [HB91] est implanté au-dessus du SGBD extensible EXODUS. L'objectif d'Ariel est d'étendre les systèmes relationnels avec des règles de production. Comme A-RDL, Ariel s'apparente nettement aux systèmes déductifs : l'effort le plus important concerne l'évaluation des conditions [Han92]. Les notions d'événements et de modèle d'exécution ne sont pas prédominantes.

Postgres L'objectif du projet Postgres [SR86, RS87] – mené à l'Université de Californie, Berkeley – était de construire un SGBD relationnel étendu, avec pour objectif principal, la gestion des objets et des connaissances. La gestion des objets implique le stockage et la manipulation de types de données non conventionnels tels que texte et image. La gestion des connaissances implique la capacité d'enregistrer et d'exécuter un ensemble de règles [SHH87, SHP89, SJGP90, Sto92] qui constitue une partie de la sémantique de l'application. Le langage utilisé dans Postgres est POSTQUEL, un langage de requête ensembliste qui est une extension du langage de requête relationnel QUEL. Postgres est l'un des SGBD actifs les plus simples tant au niveau du langage de règles que du modèle d'exécution. Une version commerciale de Postgres, appelée ILUSTRA, est disponible depuis 1994.

Starburst L'objectif du projet Starburst [LLPS91] – mené depuis 1985 au centre de recherches IBM d'Almaden, San Jose, Californie – était le développement d'un SGBD extensible à tous les niveaux : méthodes de stockage, stratégies d'exécution, analyse de requêtes, optimisation, etc. Les extensions doivent être réalisées par des concepteurs de SGBD relationnels étendus. Le système Starburst est basé sur le modèle relationnel et des extensions du langage de requêtes SQL. Les extensions développées fournissent le support de hiérarchies de types et de fonctions définies par l'utilisateur, de la gestion d'objets complexes et de règles. Deux systèmes différents ont été developpés : le système Alert [SPAM91] et les règles de production [WF90, WCL91]. Dans la suite du document, Starburst désigne le prototype de système de règles de production de Starburst et non Starburst dans sa totalité. Un important travail a été mené dans le domaine de l'analyse (statique) de bases de règles [BW94] et de du comportement d'une base de règles [ACL91, AWH92].

A.1.2 Systèmes à objets

HiPAC Le projet HiPAC (High Performance ACtive database system) [DBB+88, MD89, CBB+89, Cha89, DBC96] a été lancé en 1987 par la Defense Advanced Research Projects Agency et le Rome Air Development Center. Il a été developpé au CCA (Advanced Information Technology Division of Computer Corporation of America) puis dans les centres de recherche de Xerox (Xerox Advanced Information Technology Center) puis DEC. Le prototype est une extension du SGBD extensible PROBE mais il n'implante pas toutes les fonctionnalités du modèle HiPAC.

Notons que HiPAC est l'un des premiers SGBD a considérer réellement des règles actives (règles ECA) (et non plus de règles de production) ; et ce dans un contexte

à objets [DBM88]. Notons égalemment que HiPAC reste, à ce jour, "la" référence en ce qui concerne les modéles d'exécution des SGBD actifs [HLM88]. Il a été le premier à considérer un modèle de transactions avec des transactions emboîtées et séparées comme principal support du modèle d'exécution. À ce titre, Sentinel, REACH, Ode, SAMOS, TriGS et les travaux de Beeri et Milo – que nous décrivons ci-après – peuvent être considérés comme des héritiers de HiPAC.

Sentinel Sentinel est un projet mené au Database Research and Development Center à l'Université de Floride, Gainsville, USA. Il découle directement des résultats obtenus dans le projet HiPAC. Un prototype [CAM93], est implanté au-dessus de Open OODB Toolkit de Texas [CTZ95] Instrument qui est un SGBD extensible. Les plus importants travaux, dans ce projet, concernent les modèles d'événements et le processus de détection des événements avec le langage Snoop [CM93] ; et les outils (tracer, debbuger) pour systèmes de règles actives

REACH REACH(REal-time, ACtive and Heterogeneous mediator system) [BBKZ92, BBKZ93, BB95] est un projet de l'université de Darmstadt, Allemagne, influencé directement par HiPAC et DOM. DOM (Distributed Object Management) [MaSHHG92] est un modèle de transactions évolué qui fournit des *closed nested transactions* (transactions emboîtées de Moss [Mos85]) et des *open nested transactions*. Il est égalemment possible de combiner les deux types de transactions. Trois prototypes, l'un au dessus de O_2 et l'autre au-dessus de ObjectStore, et le dernier au-dessus de Open OODB [BZBW95], illustrent en particulier "l'orientation objet-distribué" de REACH. Le coeur du système, appelé REACT, implante un langage d'événements proche de Snoop.

Ode Ode (Object Database and Environment) [GJ91] est un SGBDO – developpé aux Laboratiores AT&T Bell – qui dispose d'un système de règles. Ode [AG89] propose un langage de programmation persistant O++, dérivé de C++. La définition des classes O++ a été étendue avec la définition de règles actives. Un aspect original et particulier à Ode est le fait que les règles soient de la forme Événement-Action : la partie Condition et complétement intégrée dans la partie Événement. La partie la plus importante de ce travail concerne le modèle d'événements [GJS92, NG92].

SAMOS SAMOS (Swiss Active Mechanism-Based Object Oriented Database System) [KDM88, Da86, GGD91] est un projet de l'université de Zurich, Suisse. Un prototype a été implanté au-dessus de Exodus puis ObjectStore [LLOW91]. Le mo-

dèle d'exécution est directement inspiré de celui de HiPAC. Un modèle de transactions emboîtées est évoqué mais le modèle d'exécution de SAMOS est en réalité assez peu décrit. La partie la plus importante de ce travail concerne le modèle et le processus de détection des événements composites [GD94, GGD93].

Beeri & Milo Le travail, de Catriel Beeri et Tova Milo (Université de Jerusalem, Israel) dans [BM91], ne vise pas à définir un SGBD actif complet, mais propose uniquement un modèle d'exécution pour SGBD actif. Ce modèle, relativement proche de celui de HiPAC, est basé sur le paradigme objet et le concept de transactions emboîtées. Il décrit particulièrement les interactions entre *transactions de règles* et *transactions d'application* : concurrence sur les objets, échec et abandon de transactions, reprise après échec, etc. Toutefois Beeri & Milo n'exploitent pas la notion de *modes de couplage* introduite dans HiPAC : ce sont les types d'événements qui spécifie quand et comment doivent être exécutées les règles.

TriGS TriGS (Triggers for GemStone) [KRRV94b] est un projet commun aux universités de Vienne et Linz (Autriche), mené dans le cadre du projet européen "Design, Development and Implementation of a Knowledge-Based Leistand" (projet ESPRIT III - no. 5161). C'est un système de règles actives pour le SGBD à objets GemStone [BaJS91] . Le modèle de connaissances vise à intégrer des règles actives dans un modèle de données à objets. On trouve ainsi des règles locales à une classe et globales à un schéma. En ce qui concerne le modèle d'exécution [KRRV94a], l'approche est basée sur celle d'HiPAC. Une différence notoire et que l'exécution des règles est spécifié dans les types d'événements comme dans le modèle de Beeri & Milo. Dans le prototype, les transactions emboîtées et séparées, qui n'existent pas dans le modèle de transactions classique de GemStone, sont simulées par des sessions concurrentes de GemStone (comme SAMOS avec ObjectStore).

Chimera Chimera est le produit d'une collaboration entre l'insitut Politecnico di Milano (Italie), l'Université de Bonn (Allemagne) et le centre de recherches ECRC de Munich (Allemagne) dans le cadre du projet Européen IDEA (Intelligent Database Environment for Advanced applications) [CF95] (Projet Esprit III - no. 6333). Chimera est un langage de programmation de bases de données qui intègre un modèle de données à objets, un langage de requête déclaratif basé sur des règles déductives et un langage de règles actives. La partie active de Chimera est réalisée par Politecnico di Milano [CFPT95a]. Les aspects plus particulièrement étudiés sont le modèle d'événement [MPC96] ; le modèle d'exécution – basé sur un modèle de transactions

classique – qui se rapproche de ceux de Peplom^{ad} et NAOS ; et les outils de conception et d'analyse statique et dynamique des règles [BCP95].

Peplom^{ad} Peplom^{ad} est le produit d'une collaboration entre le Centre de Recherches Bull (Bull-IMAG, Grenoble) et l'université de Grenoble (LGI-IMAG, projet Aristote). Peplom^{ad} est un langage de programmation pour bases de données (LPBD) orientés objet qui offre des fonctions déductives et actives [Ron94, DR94a, DR94b, DA94]. Le langage propose les notions de *classe* et *module*. Les modules offrent un niveau d'abstraction supérieur aux classes et permettent la définition de données et d'opérations. Les règles peuvent être attachées aux classes ou aux modules. Le modèle d'exécution, bien que reposant sur un modèle de transaction classique, est riche, puissant et particulièrement bien adapté à la cohabitation des règles déductives et actives.

NAOS NAOS (Native Active Object System) est un projet mené au sein de l'équipe STORM (LSR-IMAG) à l'Université de Grenoble depuis septembre 1992. Il a été en grande partie financé par le projet Européen GOODSTEP [GOO94] (Projet Esprit III - no. 6115) [CHCA94]. NAOS [Hab93, Cou93] est un système de règles actives pour le SGBD O_2 [BDK92]. Les aspects caractéristiques de NAOS concernent son intégration dans le modèle à objets d'O_2 ; le modèle et le processus de détection des événements [CC96a, CC96b] qui est intégré dans le moteur d'O_2 (et non au-dessus d'O_2) ; et son modèle d'exécution [CCS94a, CCS94b, CC95c] qui comme Chimera et Peplom^{ad} repose sur un modèle de transactions classique. NAOS est décrit abondamment dans le chapitre 6 dans lequel une expérimentation de nos travaux autours de NAOS est esquissée.

EXACT EXACT (EXtensible approach to ACTive object-oriented DBMSs) [DJ94, DP94] – developpé à l'université du pays basque, San Sebastian, Espagne – est un système de règles actives implanté au-dessus du SGBD à objets ADAM [Pat89]. EXACT se démarque nettement des travaux évoqués ci-dessus puiqu'il s'agit d'un système *extensible* à tous les niveaux : modèle de connaissances et modèle d'exécution. L'extensibilité est basée sur un modèle à objets et une spécialisation par classes et métaclasses. EXACT est décrit plus longuement dans le chapitre 4, section 4.1.1.2, puisqu'il vise, comme nous, à fournir un modèle d'exécution flexible.

A.2 Synthèses et études comparatives

L'activité intense qui anime le domaine des SGBD actifs est égalemment quantifiable par le nombre important de synthèses (surveys) et autres études comparatives.

Le manifeste des SGBD actifs [DGG95] présente un certain nombre de caractéristiques qui permettent de déterminer si un système est un SGBD actif ou non.[1]

Viennent ensuite, par ordre chronologique : [vdVK91, HW92, Cha93, ACC+93, Buc94, DHW95, CHR96] qui sont des études générales et qui sont autant de "points d'entrée" dans le domaine. Mentionnons également [Dia95] et [FT95] qui sont des travaux plus rigoureux et qui proposent (chacun) une taxonomie des SGBD actifs.

Longtemps délaissés et ignorés, les fondements formels des SGBD actifs constituent aujourd'hui – et ce, depuis un ou deux ans – un domaine de recherche en constante progression. Des travaux visant à fournir des sémantiques formelles de certains SGBD actifs ont vus le jour ; ainsi que des travaux visant à comparer ces approches. Ces travaux sont présentés dans le chapitre 5.

Enfin, mentionnons [JF95b] et [JF95a] qui présentent une bibliographie annotée – fort utile mais hélas incomplête (!) – des SGBD actifs existants et [Wid94] qui présente des perspectives de recherche dans le domaine.

1. Ce manifeste émane d'une institution particulière et non d'un groupe de standardisation. Il n'engage donc que ses auteurs !

Résumé : Un *système de bases de données actif* est capable d'exécuter automatiquement des actions prédéfinies en réponse à des événements spécifiques lorsque certaines conditions sont satisfaites. Les *règles actives*, de la forme Événement-Condition-Action, sont au cœur de cette approche. Dans cette thèse, nous nous intéressons aux *modèles d'exécution* des systèmes actifs. Le modèle d'exécution d'un système actif décrit quand et comment (ordonnancement, synchronisation) sont exécutées, au cours d'une application, les règles déclenchées lors de l'exécution d'une transaction. Nous proposons tout d'abord une taxonomie et une représentation graphique des modèles d'exécution des systèmes actifs. Nous exposons ensuite **un modèle d'exécution paramétrique** nommé Fl'are (**Fl**exible **a**ctive **r**ule **e**xecution). Une caractéristique essentielle de ce modèle est de considérer des *modules de règles* – chaque module étant destiné à une utilisation particulière des règles. On peut spécifier, d'une part, le comportement de chaque règle d'un module, et d'autre part, la stratégie d'exécution de chaque module. Il suffit, pour cela, de choisir une valeur pour chacun des paramètres proposés parmi un ensemble de valeurs prédéfinies. Nous donnons également une sémantique dénotationnelle (ou fonctionnelle) du modèle. Nous montrons que ce formalisme fournit une spécification implantable que nous utilisons dans le cadre de l'expérimentation que nous menons autours de NAOS – un mécanisme de règles actives pour le SGBD à objets O_2 – afin de remplacer son moteur d'exécution par Fl'are.

Mots clés : bases de données actives, systèmes de règles actives, modèles d'exécution, sémantique dénotationnelle.

Abstract : An *active database system* is able to execute automatically some predefined actions in response to specific events when some conditions are satisfied. *Active rule*, of the form Event-Condition-Action, are the core of this approach. This thesis is concerned with active systems *execution models*. The execution model of an active rule system describes when and how (scheduling, synchronization) rules triggered during the execution of a transaction are executed during an application. First, we propose a taxonomy and a graphic representation of active systems execution models. Then we set out **a parametric execution model** named Fl'are (**Fl**exible **a**ctive **r**ule **e**xecution). An essential characteristic of the model is to consider *rule modules* – each module being intended to a particular use of rules. The behaviour of each rule of a module can be specified, and then, the execution strategy of each module. In order to do that, one just has to choose a value for each proposed parameter among a set of predefined values. We also give a denotational (or functionnal) semantics of the model. We show that this formalism provides an implementable specification that we use within the framework of the experiment we are making around NAOS – an active rule system for the O_2 object-oriented DBMS – in order to replace its execution engine by Fl'are.

Keywords : active databases, active rule systems, execution models, denotational semantics.