Exceptions dans les langages à objets

Serge Lacourte

THESE

Présentée par

Serge Lacourte

pour obtenir le titre de

Docteur de l'Université
Joseph Fourier - Grenoble 1

(arrêté ministériel du 23 novembre 1988)

Spécialité : INFORMATIQUE

Exceptions dans les langages à objets

Thèse soutenue devant la commission d'examen le :
11 Juillet 1991

Jacques Mossière	Président
Claude Bétourné	Rapporteur
Jean Bézivin	Rapporteur
Sacha Krakowiak	Directeur de thèse
Roland Balter	Directeur du laboratoire

Thèse préparée au sein du Laboratoire Unité Mixte Bull-IMAG

Ce travail n'aurait pas pu aboutir sans le soutien d'un grand nombre de personnes, c'est pourquoi je veux remercier ici

Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble et Directeur de l'ENSIMAG. Il m'a fait l'honneur d'accepter de présider le jury de soutenance.

Claude Bétourné, Professeur à l'Université Paul Sabatier de Toulouse, et Jean Bézivin, Professeur à l'Université de Nantes. Ils ont bien voulu consacrer de leur temps à rapporter sur mon travail, et je les remercie particulièrement pour leur appréciation.

Sacha Krakowiak, Professeur à l'Université Joseph Fourier de Grenoble, qui m'a accueilli au sein de l'équipe Guide qu'il dirige. Je lui suis très reconnaissant pour la confiance qu'il m'a accordée et je lui dois la réussite de ce travail.

Roland Balter, Docteur es Sciences et Directeur de l'Unité Mixte Bull Imag. Ses qualités humaines ont grandement facilité mon intégration dans le laboratoire. Il m'a permis de réaliser ce travail dans des conditions matérielles et psychologiques de premier ordre.

Michel Riveill, Docteur de l'Institut National Polytechnique de Grenoble, qui m'a encadré pendant toute la durée de ce travail. Je le remercie également pour m'avoir rendu possible l'accès à la maquette Guide via le compilateur, et aussi pour les soirées passées à relire ce document.

Enfin je voudrais remercier l'ensemble des membres du projet pour l'accueil qu'il m'ont réservé, avec une pensée particulière pour Cécile Roisin, pour les premières discussions de conception, pour Xavier Rousset, relecteur et témoin d'une présoutenance mémorable, et pour André Freyssinet, qui a su m'accompagner jusqu'au bout sur la longue route du thésard.

Chapitre I

Introduction

On trouve dans les dictionnaires Robert et Grand Larousse Universel, trois acceptions intéressantes du mot exception.

"personne, chose qui échappe à la règle générale, qui est unique en son genre"

On reconnaît là l'exception comme objet exclu de son groupe normal de définition, sur un critère particulier qui la rend unique. Ainsi Samson et Cyrano de Bergerac, l'un de par sa force surhumaine, l'autre de par la démesure de son panache, sont-ils des exceptions. Traduite en termes informatiques, et plus précisément dans l'environnement orienté-objets, cette définition marque la possibilité pour un objet de ne pas satisfaire toutes les contraintes que sa classe impose à ses champs. Cela permet en particulier l'insertion dans une base de connaissance PSN [44] d'une "exception individuelle statique" "Capt'n-Kidd", instance de la classe "Humain", mais dont le champ "première_jambe" ne satisfait pas la contrainte d'être une "Jambe_humaine" imposée par la classe puisque c'est une jambe de bois.

Le concept est adouci dans une deuxième définition.

"ce qui est en dehors du général, du courant ; anomalie, particularité, singularité"

L'exception perd la propriété d'unicité, mais conserve un caractère de rareté plus ou moins prononcé. Cela traduit le fait qu'une exception dans le langage courant est toujours beaucoup moins fréquente que le cas général, mais qu'elle peut se répéter. Ainsi la taille moyenne d'un homme adulte est d'un mètre soixante-dix, excepté pour un nain où elle descend à quatre-vingt centimètres. Les exceptions qui se ressemblent donnent alors naissance à un groupe exceptionnel, en termes PSN une "exception générique statique". On peut également voir ce type d'exceptions sous un autre angle, en appliquant la première définition sur les ensembles ou classes d'objets. Cela permet la définition d'une classe qui ne respecte pas les contraintes imposées par sa super-classe, comme la classe "nain" qui redéfinit les valeurs moyennes de la classe "humain".

On trouve enfin une définition dynamique, qui s'applique à un processus plus qu'à un état.

"faire une exception pour quelqu'un : ne pas le traiter comme l'ensemble des autres"

Elle est illustrée dans la phrase suivante : "Je demande toujours l'avis d'un mécanicien quand j'achète une voiture, mais comme tu es mon frère je ferai une exception". On retrouve la première définition de l'exception, mais appliquée au processus d'achat d'une voiture. Traduite en termes informatiques, elle montre l'occurrence de situations particulières où l'algorithme principal ne convient pas, et qui nécessitent l'exécution de traitements particuliers. Le caractère de rareté est toujours présent ; l'exécution du traitement exceptionnel survient beaucoup moins fréquemment que l'exécution de l'algorithme principal.

A partir de ces trois points se sont développés deux mécanismes fortement distincts, qui ne sont toujours pas réunifiés à l'heure actuelle. La recherche en bases de données et bases de connaissances s'est concentrée sur l'aspect statique des exceptions, c'est-à-dire les objets et classes exceptionnelles, qui recouvrent les deux premiers points évoqués. Elle touche également l'aspect dynamique, mais essentiellement pour ce qui est de la manipulation de ces données exceptionnelles. La recherche en langages de programmation a beaucoup plus développé le troisième point, à savoir le traitement des situations exceptionnelles, l'aspect dynamique des exceptions. Le travail présenté dans cette thèse se place dans le deuxième cadre, en regardant plus précisément les langages orientés-objets.

I.1 But du travail

Dans ce contexte algorithmique, l'exception apparaît comme conséquence des limites qu'une mise en œuvre introduit par rapport à un modèle idéal. Le traitement de ces cas limites par les structures de contrôle traditionnelles n'est pas satisfaisant, et c'est pourquoi pratiquement tout langage offre des instructions de déroutement à longue portée, c'est-à-dire qui brisent l'unité offerte par la procédure. C'est dans un souci de structuration de ces instructions qu'ont été conçus les systèmes de gestion des exceptions (SGE).

De tels systèmes sont encore peu répandus. Les bases en ont été jetées par Goodenough en 1975, et les deux langages Ada [36] et CLU [46] en proposent un depuis 1979. Mais c'est seulement depuis peu qu'un mécanisme, encore "expérimental", est décrit dans le langage C++ [39], et qu'une proposition existe pour normaliser un autre mécanisme pour Common Lisp [61]. Il existe par ailleurs d'autres travaux récents sur des langages orientés-objets de diffusion moindre.

Un des objectifs de cette thèse est d'analyser les contraintes, mais aussi les avantages, qu'un langage orienté-objets apporte dans la conception d'un SGE. Un point caractéristique d'un tel environnement est le principe d'encapsulation des objets, manipulables uniquement à travers une interface. Cette contrainte d'encapsulation découle directement des langages modulaires, mais est singulièrement renforcée puisqu'à une interface peut correspondre différentes mises en œuvre (polymorphisme). En outre la résolution de la liaison entre interface et code effectif peut parfois n'être réalisée qu'à l'exécution, ce qui est le cas pour le langage Guide. Ce principe d'encapsulation amène différentes contraintes, tant sur la nature d'une exception que sur sa portée, ou sur la diversité des traitements applicables. Mais un environnement à objets apporte également des possibilités nouvelles, comme l'utilisation de l'héritage ou la représentation des exceptions sous forme d'objets.

L'autre objectif de ce travail est de concevoir et de réaliser un SGE pour le langage Guide. Comme nous allons le voir, Guide est un langage orienté-objets, ce qui nous permet bien évidemment de reprendre les conclusions de la première partie de la thèse. Le projet Guide est né il y a cinq ans, avec pour objectif l'étude d'un système réparti pour le support d'applications coopératives. Il s'est développé au sein d'une équipe mixte regroupant des chercheurs du CNRS, de l'IMAG, et du centre de recherche Bull, équipe qui s'est depuis transformée en une Unité Mixte CNRS de Recherche Bull-IMAG n^{o} 122. Très rapidement le besoin s'est fait sentir de la définition d'un nouveau langage, capable de traduire les

nouveaux concepts de la machine développée. Une première phase s'est terminée l'année passée, avec la réalisation d'un prototype du système Guide au dessus d'Unix et d'un compilateur du langage Guide pour ce système. Aussi bien système que langage ont été conçus avec l'objet comme principe constructeur. L'objet Guide est l'unité de stockage, de liaison, de partage et de programmation. La conception d'un SGE pour le système Guide, qui comme nous venons de le rappeler offre persistance et partage, nous a amené à nous poser le problème de la cohérence des objets, et à en proposer une solution. D'autres aspects spécifiques comme l'instruction de parallélisme offerte par le langage ont également été abordés. Nous avons effectué une réalisation qui est disponible sur le prototype de Guide mis en œuvre sur Unix.

I.2 Plan

Le plan de la thèse découle naturellement de notre volonté d'étudier les SGE pour des langages orientés-objets, et d'en réaliser un pour un langage orienté-objets particulier. Il se découpe en cinq chapitres et une annexe.

Exceptions : définitions et objectifs

Nous établissons tout d'abord le cadre du travail en définissant les exceptions et les situations où elles peuvent survenir, et en présentant les mécanismes prévus pour les traiter. Nous montrons l'insuffisance des instructions de déroutement local, et le besoin de mécanismes spécifiques qui structurent les déroutements à longue portée, les SGE. Ces mécanismes distinguent une entité signalante qui détecte l'exception et une entité traitante, liée dynamiquement, qui en propose un traitement. Conçus initialement pour traiter les situations d'exception, ces mécanismes ont naturellement été utilisés en tant que simples structures de contrôle additionnelles du langage.

Les exceptions dans le modèle objet

Nous étudions ensuite les contraintes imposées par un environnement à objets pour en déduire la structure d'un mécanisme idéal de gestion des exceptions. La contrainte forte est, comme nous l'avons déjà noté, le principe d'encapsulation des objets. Nous montrons comment cette contrainte influe sur le signalement d'une exception, ainsi que sur la déclaration d'un traitant. Elle joue également un rôle important dans le choix d'offrir ou non la "politique de reprise" comme traitement possible. Nous regardons les possibilités nouvelles que nous offre l'environnement à objets, comme l'utilisation de l'héritage pour factoriser des traitants, ou la représentation des exceptions sous forme d'objets. Ce chapitre se termine par une synthèse des conclusions auxquelles nous avons abouti sur chacun des points abordés.

Propositions existantes

Le mécanisme idéal défini dans le chapitre précédent nous permet d'analyser une large palette de propositions existantes. On peut distinguer deux grands groupes de SGE, et un ensemble d'autres propositions plus ou moins atypiques. La catégorie des mécanismes avec terminaison dérive du modèle des langages modulaires, que nous décrivons en détail. Ces mécanismes ne dissocient pas complètement code exceptionnel et algorithme principal, et de manière générale ils ne fournissent pas de mécanisme spécifique pour maintenir la cohérence

des objets. Une deuxième catégorie de langages choisit de représenter les exceptions sous forme d'objets. Elle illustre une discussion qui a lieu dans le second chapitre. Nous présentons ensuite la proposition développée en [79], qui décrit un mécanisme où code exceptionnel et algorithme principal sont réellement dissociés. Nous terminons par l'étude des mécanismes définis pour C++ et Eiffel, qui illustrent respectivement une utilisation intermédiaire de la représentation objet et une réponse spécifique au problème de cohérence.

Proposition pour Guide

Nous entamons ensuite la seconde partie de la thèse en décrivant très succintement les caractéristiques du langage Guide, avant de décrire dans le détail le mécanisme proposé. Nous rappellons les choix qui dérivent des conclusions du chapitre III, puis nous développons les points particuliers au langage. Le cas particulier du traitement des exceptions à l'intérieur d'un traitant d'exception est analysé, avec le danger de bouclage qui lui est afférent. Les exceptions système apparaissent à l'utilisateur de manière homogène aux autres exceptions grâce à la représentation d'un appel système comme un appel de méthode à un objet de type *system.* Une solution au problème de la cohérence est décrite, qui garantit l'exécution de code de restauration même en cas de signalement d'exception. Nous définissons la prise en compte des exceptions dans la règle de conformité. Nous montrons l'utilisation de l'héritage dans la définition de traitants généraux et la factorisation du code de restauration de classe. Nous abordons enfin le problème du parallélisme, en modifiant la sémantique de la terminaison d'un bloc parallèle.

Développements futurs

Nous terminons par un chapitre qui nous tient particulièrement à cœur. Nous tirons les enseignements qualitatifs de la première réalisation pour affiner ou enrichir les solutions adoptées. Nous abordons en particulier le problème de la nature relativement pauvre des exceptions, et des complications que cela amène ; nous proposons une structure intermédiaire entre la chaîne de caractères et l'objet C++. Nous restructurons le mécanisme de restauration, de manière à prendre en compte la concurrence d'accès aux objets. Nous décrivons ensuite une mise en œuvre possible sur le micro-noyau Mach permettant de traiter les exceptions hardware, en les signalant au programmeur sous forme d'exceptions systèmes. Nous livrons enfin au lecteur quelques idées qui se trouvent dans un état encore embryonnaire.

Mise en œuvre

Nous donnons enfin en annexe la description de la mise en œuvre du mécanisme sur la maquette Unix de Guide. Cette description est relativement détaillée, afin d'être utile à celui qui voudra comprendre ou retoucher le code généré. Elle demande par contre une certaine connaissance de la mise en œuvre de la machine et du compilateur Guide, dont le résumé donné au début risque d'être insuffisant. Nous refermons cette description par une évaluation essentiellement quantitative des mécanismes.

Chapitre II

Exceptions : définitions et objectifs

Il est bon, pour comprendre ce qui va suivre, de présenter tout d'abord les principaux concepts qui touchent aux exceptions, ainsi que les mécanismes spécifiquement créés pour les traiter. La première partie démontre que les exceptions existent et qu'elles ne peuvent pas être, ou sont mal traitées par les structures de contrôle habituelles. La seconde décrit les traits principaux des mécanismes de gestion d'exceptions, et introduit vocabulaire et notations. La dernière classe les exceptions en catégories correspondant aux types de problèmes qu'elles sont censées traiter.

II.1 Introduction

Pour définir les exceptions nous partirons de la formalisation de la programmation qui débute avec la logique de Hoare [32].

II.1.1 Hoare

Le but recherché est de prouver formellement la correction d'un programme. Cela nécessite deux choses. D'une part il faut pouvoir exprimer ses spécifications, et d'autre part il faut fournir un ensemble d'axiomes et de théorèmes qui traduisent les effets de chaque instruction du langage et de leur composition.

Hoare propose de formaliser les spécifications d'un programme *Q* par deux assertions, des préconditions *P* et des postconditions *R*. Si *P* est vraie avant l'exécution de *Q*, alors *R* sera vraie après. Cela se note :

$$P \ \{Q\} \ R \qquad (1)$$

Il fournit en outre un certain nombre d'axiomes décrivant l'addition, l'affectation, la boucle **while**, et surtout il donne la règle formalisant la composition de deux instructions Q_1 et Q_2. Il note en particulier que les préconditions de Q_2 doivent être assurées par les postconditions de Q_1 :

$$\left.\begin{array}{l} P \ \{Q_1\} \ R_1 \\ R_2 \ \{Q_2\} \ R \\ R_1 \Rightarrow R_2 \end{array}\right\} \Rightarrow P \ \{Q_1; Q_2\} \ R \qquad (2)$$

Il découle de ce formalisme une programmation idéale où les instructions de l'algorithme principal réalisant les spécifications du programme s'enchaînent suivant la règle (2). Seulement voilà, nous sommes loin de cet idéal, et la réalité nous impose toutes sortes de limites qui se traduisent par des contraintes sur les préconditions ou les postconditions de ces

instructions. Ces contraintes font que dans certains cas particuliers le troisième prémisse de (2) n'est pas vérifié. Ces cas particuliers demandent un traitement hors de la norme ; on les appelle exceptions.

Le traitement des exceptions peut s'insérer dans la logique de Hoare de deux manières différentes. Nous allons le voir sur un exemple simple, en essayant d'exprimer les spécifications de l'addition ($a + b$) qui, comme chacun sait, pose problème dès que la somme dépasse *maxint*, le plus grand entier représentable en machine.

Dans le premier cas, la contrainte fait partie des préconditions, qui peuvent s'exprimer par

$$b <= (maxint - a) \tag{3}$$

alors que les postconditions sont simplement

$$retour = a +_N b \tag{4}$$

où $+_N$ représente l'addition arithmétique dans les entiers. Le programmeur doit être sûr que (3) est remplie avant d'effectuer l'addition, ou alors il fait précéder celle-ci d'un test, et fournit le traitement exceptionnel dans l'autre cas (5).

if b <= (maxint - a) **then** a + b **else** traitement d'exception **end** (5)

Cette formulation paraît être la plus propre car elle conserve aux postconditions leur sémantique de "but atteint". Mais elle n'est pas toujours souhaitable, comme c'est le cas ici puisqu'on exécute un calcul supplémentaire qui double le temps d'exécution utile. Elle est même parfois impossible, comme on peut s'en rendre compte avec l'exemple d'un fournisseur de ressources partagées. Un tel exemple est le système Unix, fournisseur de mémoire partagée à travers la primitive *shmget()* pour l'ensemble des processus qu'il gère. Dans ce cas le programmeur d'un processus pourrait tester s'il reste une ressource de libre avant de la demander, mais elle pourrait être prise par un autre processus entre temps. Il doit donc accepter d'exécuter l'appel sans savoir à l'avance s'il va réussir, la sémantique de *shmget()* n'incluant pas l'attente de la libération possible d'un segment. Si plus aucune ressource n'est disponible, alors l'exception survient ; on dit qu'elle est signalée. Elle apparaît dans les postconditions (6).

$$(retour = a +_N b) \text{ ou exception} \tag{6}$$

Si l'exception est rapportée par les structures habituelles du langage, comme par exemple à travers une variable globale *exception*, alors le programmeur doit tester sa possible occurrence après l'addition .

a + b; **if** exception **then** traitement d'exception **end** (7)

Nous allons voir maintenant les limites de cette solution.

II.1.2 Insuffisance des codes retour

Que ce soit dans (5) ou (7), l'exception est banalisée. Au niveau syntaxique, le traitement de l'exception côtoie l'instruction signalante. L'algorithme principal se trouve alors noyé sous le code de traitement des exceptions, et devient difficilement lisible. Un premier objectif d'un mécanisme de gestion d'exceptions est donc de dissocier syntaxiquement le code de l'algorithme principal du code exceptionnel. Il faut d'ailleurs noter qu'ils sont pensés

séparément par le programmeur, qui conçoit d'abord le premier avant de réfléchir au second.

Une autre conséquence du traitement de l'exception près du point de détection est de risquer de dupliquer le code de traitement en chaque point où le risque d'exception se répète. Un SGE doit permettre de factoriser le code de traitement sur plusieurs instructions semblables.

Ce traitement coûte non seulement lorsque l'exception est effectivement signalée, mais aussi dans le cas normal : le test est toujours exécuté. Cela peut être considéré comme bénin dans certains cas, mais ne l'est certainement pas dans le cas de l'addition, pour laquelle le coût relatif du test est singulièrement élevé. Un SGE ne doit si possible rien coûter à l'exécution lorsqu'aucune exception n'est signalée.

Un mélange plus grave s'opère au niveau sémantique, où deux effets s'ajoutent. D'une part le programmeur risque très fortement d'oublier de traiter les cas exceptionnels, et c'est d'ailleurs tout-à-fait normal. Qui pense, à chaque fois qu'il additionne deux entiers, que le résultat peut déborder ? L'exception devient alors erreur à l'exécution, et peut faire quelques dommages avant d'être à nouveau détectée. L'objectif d'un SGE est ici de garantir qu'une exception, une fois détectée, ne peut pas être involontairement oubliée.

D'autre part, comme le signalement de l'exception dans (6) utilise des mécanismes aussi classiques et variés qu'une valeur de retour égale à *-1*, ou différente de *0*, ou encore une variable globale qui joue le rôle de valeur de retour implicite, alors le programmeur, aussi bien qu'un encore hypothétique vérificateur formel, ont toutes les chances de ne pas savoir distinguer l'exception du cas normal. Un SGE doit fournir un outil spécifique de signalement et de détection d'exception.

Toutes ces remarques sont parfaitement illustrées par le langage C, et, d'après les messages diffusés dans le newsgroup comp.lang.c, cela ne plait pas vraiment aux programmeurs. Cela montre que le test, instruction de déroutement de base de tout langage, n'est pas la réponse adéquate au problème de la détection et du traitement des exceptions. Il est difficile de lui trouver un remplaçant dans le cas (5). Par contre d'autres mécanismes existent ou ont été créés pour gérer le cas (7), et nous allons maintenant les regarder.

II.1.3 Mécanismes de déroutement

La principale caractéristique de ces mécanismes est qu'ils permettent de s'affranchir du cadre rigide imposé par la structure des langages procéduraux. Elles sont beaucoup plus proches de notions plus internes comme le pointeur de sommet de pile ou le compteur ordinal. Le problème est de pouvoir les formaliser utilement, c'est-à-dire en relation avec la formalisation des autres structures du langage. Ce serait un non-sens de donner une formalisation des procédures, si on autorise par ailleurs la manipulation sans contrôle du compteur ordinal. C'est d'ailleurs le principal problème du **goto**, qui est restreint aussi bien en pascal qu'en C à un déroutement vers une étiquette locale à la procédure où il est utilisé.

L'existence de variables référençant du code, qui apparaissent sous diverses formes dans de nombreux langages, n'est pas suffisante. Cela permet un branchement direct sur un code

de traitement de l'exception, code éventuellement lié dynamiquement dans le cas des signaux Unix, mais ce branchement conserve la sémantique d'un appel de procédure.

Les instructions qui permettent un réel déroutement non local à une procédure capturent un environnement en même temps qu'une adresse de code. Ces informations sont dans un premier temps capturées par le **setjmp** de C ou le **catch** de Lisp, puis utilisées par la primitive de déroutement, respectivement **longjmp** ou **throw**. Ces primitives de déroutement provoquent la restauration de l'environnement sauvegardé et le branchement à l'adresse associée. Elles imposent toutefois que le déroutement soit effectué alors que la procédure pendant laquelle la sauvegarde a été faite n'a pas encore terminé. Les continuations de Scheme généralisent ce mécanisme en supprimant les contraintes sur le déroutement.

En fait ces primitives restent de bas niveau, et le traitement invoqué est toujours passé en paramètre, même variable, de l'instruction de déroutement. Elles servent plutôt de base à la mise en œuvre de mécanismes plus structurés, dont nous allons maintenant regarder les principales caractéristiques.

II.2 Structure d'un SGE

Le principal apport des mécanismes spécifiques est la liaison dynamique du code de traitement (ou traitant) d'une exception donnée. Viennent ensuite les différentes techniques d'association et de recherche du traitant, et les politiques de traitement possibles.

II.2.1 Liaison dynamique

La caractéristique de base d'un SGE est donc de complètement séparer la détection de l'exception de son traitement. Il offre au programmeur deux primitives. Dans un premier temps les différentes entités du programme déclarent des traitants grâce au mot clef **except**. Dans un second temps la primitive **raise** permet au programmeur de signaler à qui veut bien l'entendre qu'il a rencontré une situation exceptionnelle. Le système choisit alors parmi les traitants proposés celui qui traite l'exception.

Cette séparation est réelle puisque pour une même instruction **raise** le traitant peut varier. Ce n'est donc pas un simple appel de procédure déguisé. Cette variation ne peut provenir, d'une manière directe ou indirecte, que de la pile des appelants du programme signalant.

Cette notion d'appelant et celle d'appelé sont naturelles. Le programme (instruction, opération) qui essaie de remplir ses postconditions est en général appelé par un autre programme, l'appelant, qui l'utilise pour réaliser lui-même ses spécifications. On trouve ainsi plusieurs niveaux de procédures imbriquées pour un processus.

Quand l'appelé signale une exception, il annonce que le traitement normal qu'il devait réaliser n'a pas pu aboutir. L'algorithme de l'appelant est alors interrompu et il est logique que celui-ci participe au traitement de l'exception. Dans la majeure partie des SGE existants toutes les entités impliquées dans la pile d'appels peuvent proposer un traitant pour l'exception.

Dans l'exemple deux exécutions de la méthode *M* sur l'objet *x0* sont représentées. On voit que le signalement de l'exception *erreur*, exécuté dans les deux cas, ne conduit pas au même résultat. Le traitant exécuté en réponse dépend de la pile des appelants de la méthode.

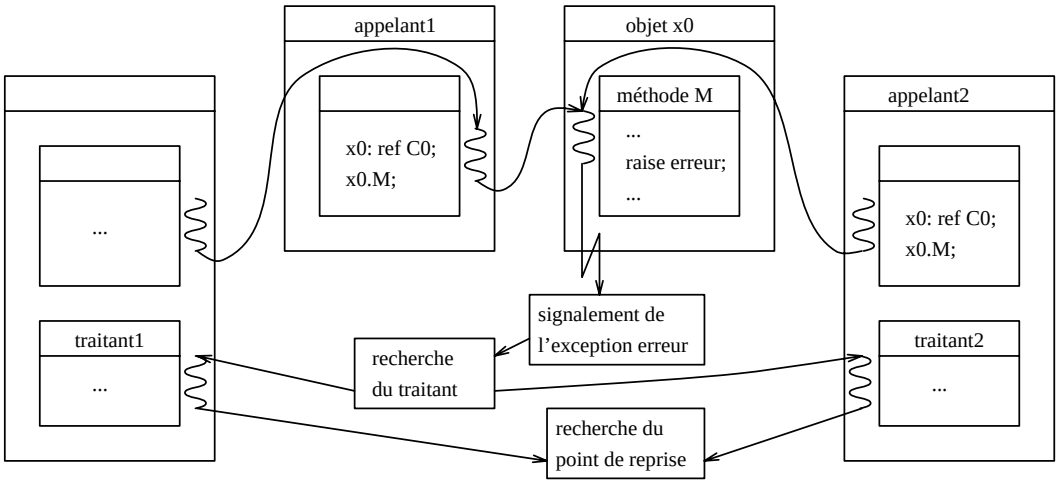

Fig. 2.1 : variation du traitant suivant l'appelant

Certains SGE autorisent l'entité signalante à fournir au traitant quelques paramètres supplémentaires qui le renseignent plus précisément sur l'exception.

II.2.2 Association et choix du traitant

Le lien entre signalement et traitement se fait grâce à l'exception passée en paramètre de l'instruction **raise**. La nature de l'exception est variable, mais la plus répandue est celle de la chaîne de caractères qui l'identifie. Le traitant doit alors être associé à la même chaîne pour être éligible en cas de signalement. Les paramètres supplémentaires passés lors du signalement peuvent également entrer en ligne de compte lors de la recherche du traitant.

Il faut en outre que l'exception soit signalée dans la portée du traitant. Celui-ci est syntaxiquement associé à un élément de programme (instruction, bloc d'instructions, procédure, programme entier), mais sa portée est plus étendue. Elle est dite dynamique car relative à l'exécution de l'élément de programme associé : le traitant est armé au début de cette exécution et désarmé en fin. La portée dynamique du traitant est l'autre manière de voir la liaison dynamique décrite dans la section précédente.

Il peut exister plusieurs traitants armés et associés à une même exception lors du signalement de celle-ci. Dans ce cas c'est le traitant associé au bloc le plus englobé, donc le plus "près" de l'exception, qui est exécuté.

Ces points relatent le fonctionnement simple d'un SGE. Plusieurs techniques visent à factoriser les traitants, compliquant le mécanisme. Il s'agit de regrouper plusieurs exceptions

dans un ensemble, autrement dit de leur donner une structure hiérarchique. Un traitant peut alors être associé à un noeud de cet arbre, indiquant par là qu'il est prêt à traiter toutes les exceptions filles de ce noeud. Il se pose alors un problème de priorité entre un traitant plus interne qu'un autre, mais associé à une exception englobant hiérarchiquement l'exception à laquelle l'autre traitant est associé.

D'autre part la portée du traitant peut être restreinte dans deux directions. Il peut n'être armé qu'à partir d'un niveau d'imbrication de procédure, auquel cas il ne peut pas traiter une exception signalée dans la procédure où il est défini. Il peut également n'être armé qu'au plus à un niveau d'imbrication de procédure, auquel cas il ne peut pas traiter une exception signalée deux niveaux plus bas. Ces deux points sont illustrés par le schéma suivant, où le traitant n'est concerné que par le seul signalement depuis l'objet x1.

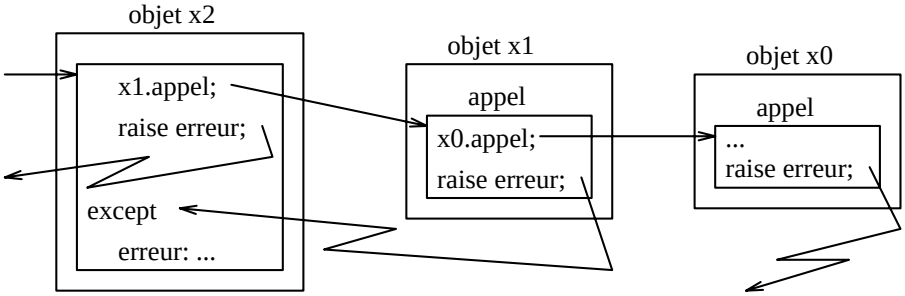

Fig. 2.2 : portée restreinte

II.2.3 Exécution du traitant

Le traitant est habituellement exécuté dans le contexte d'association. En effet le traitant peut être syntaxiquement inclu dans une procédure, et ainsi avoir accès aux variables de travail de cette procédure. La partie de la pile comprise entre point de signalement et point de traitement est alors temporairement écartée. Il semble qu'un dialogue s'instaure entre entité signalante et entité traitante. L'appelant s'interrompt après avoir demandé un service, mais il reçoit comme réponse un appel à l'aide. Il essaie alors de faire quelque chose : il exécute le traitant, dans son espace.

Cet environnement d'exécution explique le traitement d'une exception signalée dans un traitant. Dans ce cas le traitant de la nouvelle exception est recherché chez les appelants de l'entité traitante. On ne tient absolument pas compte de la première entité signalante interrompue. Cela confirme le fait que la combinaison signalement/traitement n'est pas un appel de procédure, sinon le traitant de la seconde exception serait recherché également chez le premier signalant. Ce point est illustré par le schéma suivant. Encore une fois nous représentons côte-à-côte deux exécutions de méthode sur l'objet *x0.* Dans la partie gauche la méthode appelée, *Normal*, effectue un appel de la méthode *Méthode1* sur l'objet *x1* appelant. Dans la partie droite la méthode appelée, *Exception*, signale l'exception *erreur* et

provoque l'exécution du traitant *traitant2* fourni par l'objet *x2* appelant. La différence que nous voulons montrer entre appel de méthode et exécution de traitant apparaît dans la recherche d'un traitant pour les exceptions *erreur1* et *erreur2* respectivement signalées par *Méthode1* et *traitant2* . Dans le premier cas le traitant est recherché dans *x0*, alors que dans le second cas seul *x3* est pris en compte.

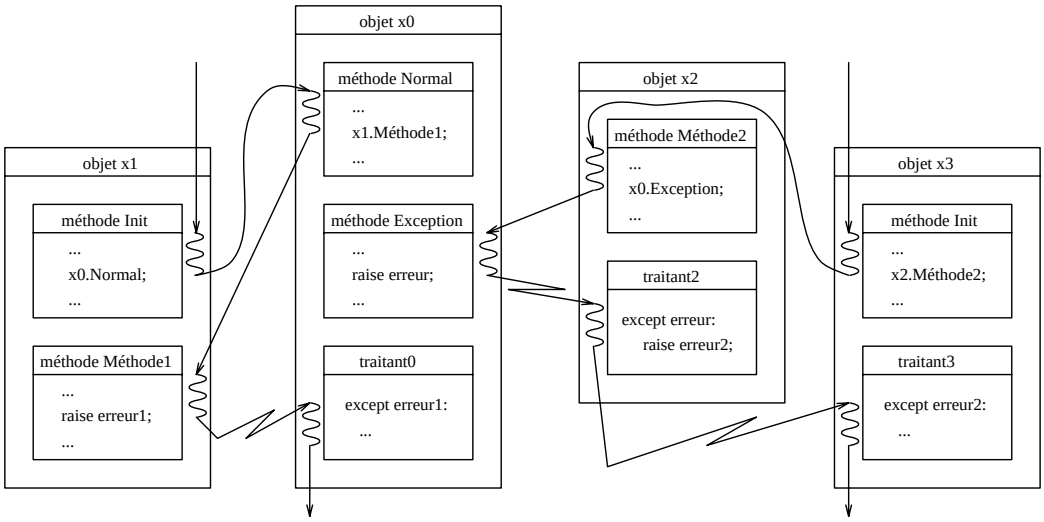

Fig. 2.3 : le signalement n'est pas un appel de procédure

De manière générale le traitement des exceptions dans un traitant se déduit en considérant le traitant comme l'entité de programme à laquelle il est associé. Quels traitants protègent les instructions d'un traitant ? Ce sont ceux qui protègent les instructions de l'entité associée. Quels traitants peuvent rattraper une exception directement signalée dans un traitant ? Ce sont ceux qui peuvent rattraper une exception directement signalée dans l'entité associée.

II.2.4 Traitements possibles

Le grand intérêt des traitants est qu'ils disposent d'instructions privilégiées qui leur permettent de contrôler le flot d'exécution. Mais ils n'agissent pas directement sur celui-ci, comme c'était le cas avec les instructions décrites en section II.1.3. Les points de reprise sont bien identifiés, correspondant à un nombre restreint de politiques de traitement exprimées par des mots clefs. Nous illustrons les différents cas dans le schéma commun Fig. 2.4.

La première politique, souvent appliquée par défaut, est la terminaison. Le point de reprise est situé au niveau de l'association du traitant, en général juste après l'élément de programme auquel il est associé. C'est *traitant1* qui demande de reprendre à l'appel suivant *x0.Do*. Cette politique peut être complétée d'une instruction de déroutement local, par exemple un **return** qui sort de la procédure où le traitant est déclaré. C'est le cas de traitant3 qui demande de reprendre après *x1.Exec*.

La deuxième politique est le réessai, demandé par le mot clef **retry**. Cela amène en général à une nouvelle exécution de l'opération signalante, et même de l'élément de programme tout entier auquel le traitant est associé. La question se pose alors de savoir si cette exécution est ou non en tous points semblable à la première, en particulier au niveau des traitants qui la protègent. C'est *traitant2* qui demande à réexécuter *x0.Do*.

La troisième politique est la reprise, demandée par le mot clef **resume**. Le point de reprise est l'instruction suivant immédiatement le point de signalement, donc dans la procédure appelée. C'est *traitant5* qui demande à reprendre après le signalement de *erreur*. La partie de la pile écartée lors de l'exécution du traitant (voir section II.2.3) est alors en quelque sorte restaurée. Cela fait ressembler l'exécution d'un tel traitant à un appel de procédure, aux différences près liées à l'environnement d'exécution que nous avons développées dans les sections II.2.1 et II.2.3.

Le traitant peut enfin se déclarer incompétent et propager la même ou une autre exception. C'est *traitant4* qui propage l'exception *erreur1*. Cette technique est nécessaire en particulier si la portée des traitants est restreinte au seul niveau d'imbrication strictement inférieur. Le langage peut réutiliser le mot clef **raise**. Pour permettre la propagation de la même exception, le programmeur peut utiliser **raise** sans paramètre.

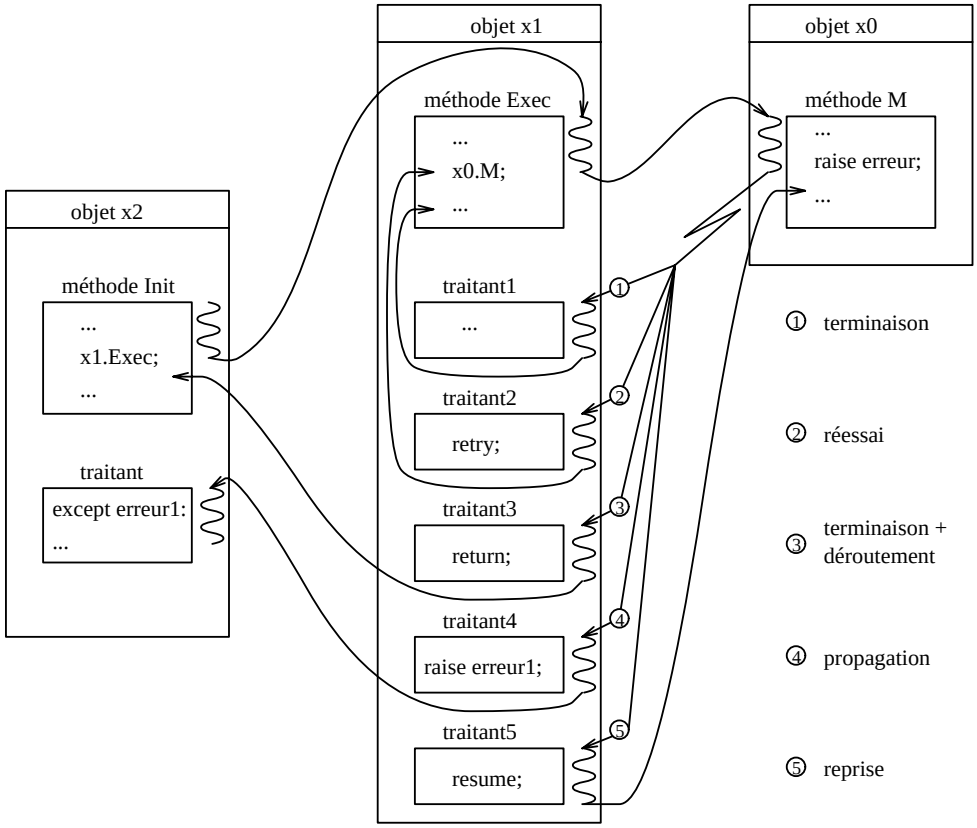

Fig. 2.4 : traitements possibles

Le système fournit enfin un traitant par défaut qui s'occupe des exceptions qui n'ont pas trouvé d'autre traitant plus précis. En général ce traitant propage une exception qui peut être différente, ou même stoppe l'exécution de l'activité.

II.3 Utilisations

Ces mécanismes spécifiques sont donc utilisés pour traiter les exceptions comme nous les avons définies en section II.1.1, et que nous appellerons exceptions vraies. Mais, comme tout nouvel outil, leur utilisation s'est étendue et ils traitent maintenant des exceptions programmées. D'autre part ils se révèlent utiles pour détecter les erreurs.

II.3.1 Exceptions vraies

L'exception vraie est la rencontre par un élément de programme en cours d'exécution d'une circontance qui l'empêche de satisfaire la partie normale de ses postconditions. Elle peut être due aux limitations de l'arithmétique informatique, ou au caractère fini d'une ressource partagée, comme nous l'avons vu en section II.1.1. Récursivement la terminaison en exception d'un fournisseur de service (procédure appelée) est également un motif d'exception vraie.

Les traitements appliquables sont au nombre de quatre. Tout d'abord l'appelant peut fournir un traitement alternatif, qui atteint, de son point de vue, les mêmes objectifs que ceux attendus de l'appel signalant, puis il choisit la politique de terminaison. Il a en quelque sorte remplacé une séquence de code par une autre, mises à part les instructions exécutées durant la première tentative qui a échoué.

Le deuxième choix dont il dispose est de corriger les causes de l'exception, par exemple demander la libération de ressources inutilisées dans le cas du fournisseur de ressources, puis réexécuter la demande. Là encore il importe de contrôler la cohérence du second appel vis-à-vis de l'ensemble d'instructions exécutées durant le premier essai.

Le programmeur peut aussi demander la reprise au point de signalement après avoir corrigé les causes de l'exception. Il faut ici contrôler l'influence des instructions de traitement sur la validité de la partie de la pile d'exécution comprise entre point de signalement et point de reprise, et qui est récupérée telle quelle lors de la reprise. Les caractéristiques d'un tel traitement dans un environnement à objets sont examinées plus en détail en section III.4, mais on peut dire dès maintenant qu'elles correspondent à un fonctionnement de mise au point, ou de correction depuis un environnement interactif. Il y a intervention directe de l'utilisateur.

Enfin l'exception peut toujours être propagée. C'est d'ailleurs le traitement par défaut généralement fourni par le système, et qui garantit qu'une exception ne peut pas être involontairement oubliée.

II.3.2 Exceptions programmées

Comme le mécanisme de gestion d'exceptions offre des primitives de contrôle du flot d'exécution, il est naturellement exploité hors de son objectif originel, dans les trois circonstances suivantes.

L'appelant peut réaliser un appel de programme en sachant à l'avance qu'il va, ou peut, provoquer le signalement d'une exception. Lorsque l'exception est signalée, il peut ainsi obtenir des renseignements sur les paramètres passés en entrée à la procédure, sans avoir à effectuer le test qui le garantirait contre le signalement. Cela est particulièrement sensible lorsque l'appel de procédure signalant fait partie d'une boucle ; le signalement de l'exception peut être un moyen de détecter la fin de boucle. Dans l'exemple suivant on détecte la fin de l'algorithme de recherche du plus grand commun diviseur (pgcd) des deux nombres *a* et *b* par le signalement de l'exception système *division_par_0* générée lors d'un appel (x **mod** 0).

```
procedure pgcd(in a, b: Integer): Integer;
    r: Integer;
begin
    if a < b then
        return pgcd(b, a);
    end;
    while true do
        r := a mod b;
        a := b;
        b := r;
    end;
except
    division_par_0: return a;
end;
```

Notons que ce test, que l'on veut éviter au niveau de l'appelant, est souvent réalisé au niveau de l'appelé, et même parfois de manière plus efficace comme dans l'exemple précédent où il est effectué par le matériel. Le traitement correspondant à ce type d'exception est la terminaison, suivie d'une instruction de déroutement local.

Deux autres types d'utilisation exploitent la politique de reprise. Tout d'abord le programmeur peut choisir d'étendre le "domaine d'appel normal" (voir [79]) d'une procédure, i.e. le domaine où la procédure remplit son objectif principal. Dans l'exemple suivant, la procédure *printTab* imprime les caractères correspondant aux codes ASCII contenus dans un tableau d'entiers et signale l'exception *non_imprimable* lorsqu'un entier ne convient pas. Cette fonction est étendue par le traitant pour imprimer un '?' à la place de chaque entier non imprimable.

```
method printTab(in tab: array of Integer; long: Integer);
    i: Integer;
begin
    for i := 0 to long-1 do
        if ! estImprimable(tab[i]) then
            raise non_imprimable(i);
        end;
        output.WriteChar(ascii[tab[i]]);
    end;
end;
```

```
method appelant;
    tab: array of Integer;
begin
    ...
    printTab(tab, 10);
    ...
except
    non_imprimable(i): begin
        tab[i] := 63;   // '?'
        resume;
        end;
end;
```

Cette utilisation s'explique dans un contexte de récupération de code, qu'on ne peut modifier et qu'on ne veut pas entièrement réécrire. Elle correspond en fait à la définition d'une nouvelle procédure, avec ici des spécifications transformées qualitativement puisque le comportement en face d'un entier non imprimable est modifié. Mais elle suppose des conditions très particulières. Il faut d'abord que l'appelé fournisse suffisamment de paramètres complémentaires lors du signalement pour préciser le point d'interruption. Il faut ensuite que le traitant ait les moyens d'agir sur les variables de l'appelé, en l'occurrence que *tab* soit passé par variable. Il faut enfin que l'instruction de signalement soit suffisamment bien placée pour que la reprise ait un sens. Toutes ces contraintes réduisent à bien peu les chances de réutilisation par ce procédé.

Le programme précédent suppose que l'appelé ne se préoccupe pas de l'appelant. Il peut être vu d'une manière très semblable, mais pourtant différente, en supposant cette fois que l'appelé prévoit la possible reprise. L'appelé fait alors un appel explicite à l'appelant traitant de l'exception pour demander une valeur, ou une indication de comportement, qui lui permettra de continuer. L'exemple s'écrit de la manière suivante :

```
method printTab(in tab: array of Integer; long: Integer);
    i: Integer;
    c: Char;
begin
    for i := 0 to long-1 do
        if estImprimable(tab[i]) then
            c := ascii[tab[i]];
        else
            c := raise non_imprimable(i);
        end;
        output.WriteChar(c);
    end;
end;
method appelant;
    ...
except
    non_imprimable(i): begin
        resume '?';
        end;
end;
```

On est ici très proche de l'appel d'une procédure passée en paramètre. D'ailleurs, est-ce vraiment autre chose ?

II.3.3 Erreurs

Voyons maintenant les liens qui existent entre exceptions et erreurs. Une erreur est un écart existant entre les spécifications d'une procédure et leur réalisation [53]. L'erreur devrait bien sûr être détectée par un vérificateur formel. En l'absence d'un tel outil, l'erreur peut apparaître en deux endroits types. Soit l'enchaînement des instructions est incorrect, i.e. il ne respecte pas dans tous les cas de figure le troisième prémisse de l'équation (2), soit les postconditions ne sont pas forcément remplies après la dernière instruction du programme, i.e. l'équation globale (1) n'est pas vérifiée dans tous les cas. Notons que cette deuxième équation doit être vérifiée aussi bien pour les procédures écrites par le programmeur, que pour les fonctions proposées par le système.

L'erreur de programmation peut demander des conditions spéciales pour provoquer une erreur à l'exécution, c'est-à-dire des postconditions non vérifiées après un appel, ou des préconditions non vérifiées avant. Elle peut rester longtemps sans effet, ce qui n'est pas gênant. Le problème est d'être capable de détecter l'erreur à l'exécution quand elle survient. La vérification systématique par une procédure de ses préconditions et de ses postconditions permettrait bien sûr de le faire, mais c'est bien trop coûteux. C'est là où les exceptions entrent en jeu. L'utilisation d'un mécanisme spécifique amène à répondre par une exception à tout appel ne satisfaisant pas les préconditions, phénomène renforcé par l'utilisation du SGE pour les exceptions programmées (voir section II.3.2). C'est cette vérification plus ou moins systématique des préconditions qui va servir à détecter des erreurs.

Mise à part la propagation, on peut traiter une exception dûe à une erreur par la provision d'un traitement alternatif, utilisant donc un autre code, suivi de la politique de terminaison. Le problème reste toujours d'évaluer les conséquences des instructions effectuées durant la première tentative. On peut également envisager une politique de correction puis de reprise dans le cadre d'un metteur au point.

II.3.4 Exceptions de contrôle

Pour mémoire nous citons les exceptions de contrôle qui apparaîssent dans la classification de Goodenough. Ces exceptions servent pour l'appelé à redonner de temps en temps le contrôle à l'appelant, pour lui rendre des résultats intermédiaires et lui permettre de prendre éventuellement la décision d'arrêter le calcul. L'exemple en est une illustration extrême. Il met en scène un objet *Sac* qui conserve un ensemble d'*Elément*s dans une structure complexe. On suppose que la navigation dans cette structure est trop complexe pour que soit efficace une réalisation d'une interface d'accès aux éléments similaire à celle d'un tableau (accès indexé) ou d'une liste (accès chaîné). En conséquence l'objet offre une primitive *Parcours* qui, en un appel, rend un à un tous les éléments conservés. Les éléments sont rendus par l'intermédiaire d'un paramètre lors du signalement de l'exception *trouvé*. L'appelant utilise ce paramètre dans le traitant de l'exception, avant de commander l'élément suivant en demandant à l'appelé de reprendre au point de signalement.

```
// définition de la classe Sac et de sa méthode Parcours
class Sac is
    method Parcours;
        x: ref Elément;
    begin
        while (<<encore un élément>>) do
            x := <<calcul de l'élément suivant>>;
            raise trouvé(x);
        end;
    end Parcours;
end Sac.
// utilisation de la méthode Parcours
    sac: ref Sac;
    begin
        sac.Parcours;
    except
        trouvé(x: ref Elément): begin
            <<traitement de l'élément x>>
            resume;
            end;
    end;
```

Ces exceptions ne sont plus vraiment d'actualité dans un environnement parallèle, où l'appelé peut être contrôlé depuis une autre activité, ou dans un environnement à objets qui offre d'autres solutions de mise en œuvre.

Chapitre III

Les exceptions dans le modèle objet

Nous avons présenté dans le chapitre précédent les caractéristiques principales des systèmes de gestion d'exceptions, à savoir la séparation du code exceptionnel de l'algorithme principal, les techniques d'association, de recherche et d'exécution du traitant, les traitements possibles et les types d'exceptions traitées. Nous allons maintenant plonger ces données dans l'environnement orienté-objets, regarder les contraintes qu'il impose et les opportunités qu'il offre, pour en déduire la forme d'un mécanisme adapté.

III.1 Caractéristiques d'un langage orienté-objets

Les langages orientés-objets sont l'évolution naturelle des langages modulaires. Outre la modularité, leurs principales caractéristiques sont l'héritage, qui permet une définition incrémentale des classes, et l'instanciation, qui fait de la classe un générateur d'objets. Cette évolution va de pair avec une plus grande formalisation des relations entre appelant et appelé, introduisant la notion de polymorphisme contrôlée par les règles de conformité.

III.1.1 Modularité

Un module regroupe des données et le code qui peut les manipuler. Il exporte une interface qui est le seul moyen pour un autre module d'accéder et de modifier l'état du premier. La notion de classe dérive de celle de module. Elle définit une structure de données et le code qui peut la manipuler ; elle exporte dans son interface une liste de méthodes, et éventuellement une liste de variables membres si celles-ci sont différenciées.

La classe cache ses données derrière ses méthodes. Ainsi, dans le schéma Fig. 3.1, *index* et le tableau *valeurs*, données de l'objet *pile_d'entiers* , ne sont pas manipulables directement. Un utilisateur de l'objet ne peut invoquer que les trois méthodes *empiler*, *dépiler* et *taille* , définies dans l'interface et qui peuvent, elles, manipuler les variables.

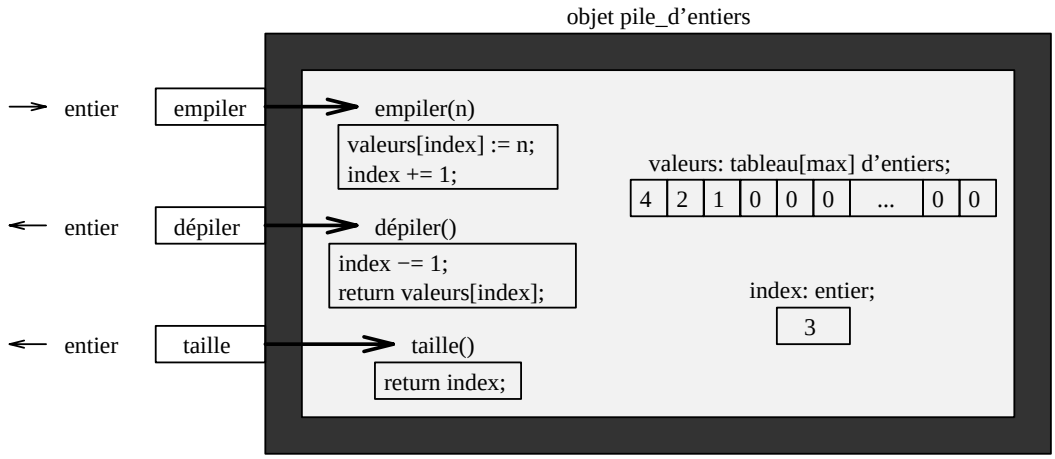

Fig. 3.1 : modularité

III.1.2 Objet et appel de méthode

La classe est un générateur d'objets. Chaque objet possède en propre un exemplaire du modèle de données défini dans sa classe. Il ne peut être manipulé que par le code également défini dans sa classe.

L'appel de méthode sur un objet est l'opération élémentaire dans la programmation objet. Dans un modèle "pur" il n'existe pas de procédure ou de fonction autonome ; le seul code disponible est le code des méthodes. L'objet appelé définit l'environnement sur lequel la méthode appelée doit s'exécuter.

En général un objet est manipulé à travers une variable typée. La variable définit donc une interface d'accès à l'objet qu'elle référence. Le compilateur peut alors contrôler statiquement la validité des appels effectués. Il existe bien sûr des exceptions à cette règle, dont la plus connue est le langage Smalltalk. Dans ce cas l'existence de la méthode appelée dans la classe de l'objet appelé est contrôlée dynamiquement à l'exécution.

III.1.3 Polymorphisme et conformité

Un des apports majeurs des langages orientés-objets est l'encapsulation du code, caché derrière une interface. Pour expliquer cette notion nous allons repartir de la modularité et du schéma Fig. 3.1.

Nous avons vu que la modularité interdit l'accès direct aux données d'un objet. Un utilisateur ne peut pas lire directement le champ *index* d'un objet *pile_d'entiers* , il est obligé d'exécuter la fonction *taille* qui est exportée dans l'interface. L'encapsulation du code veut dire que l'utilisateur ne peut pas exécuter lui-même la fonction *taille* , car le code de cette fonction ne lui est pas visible. Il ne peut qu'envoyer un message à l'objet *pile_d'entiers* , lui demandant d'exécuter la fonction *taille* qu'il a déclarée dans son

interface. La sémantique d'un appel de méthode sur un objet est celle d'un message envoyé à et interprété par l'objet.

Cette notion n'a véritablement de sens que si plusieurs mises en œuvre existent pour un même point d'entrée défini dans une interface. En d'autres termes, la même instruction

```
pile: ref pile_d'entiers;
n: entier;
n := pile.taille;
```

peut amener l'exécution de l'une ou l'autre des fonctions *taille* définies dans Fig. 3.1 et Fig. 3.2, suivant la nature de l'objet *pile_d'entiers* référencé par *pile*. Cette fonction s'appelle polymorphisme.

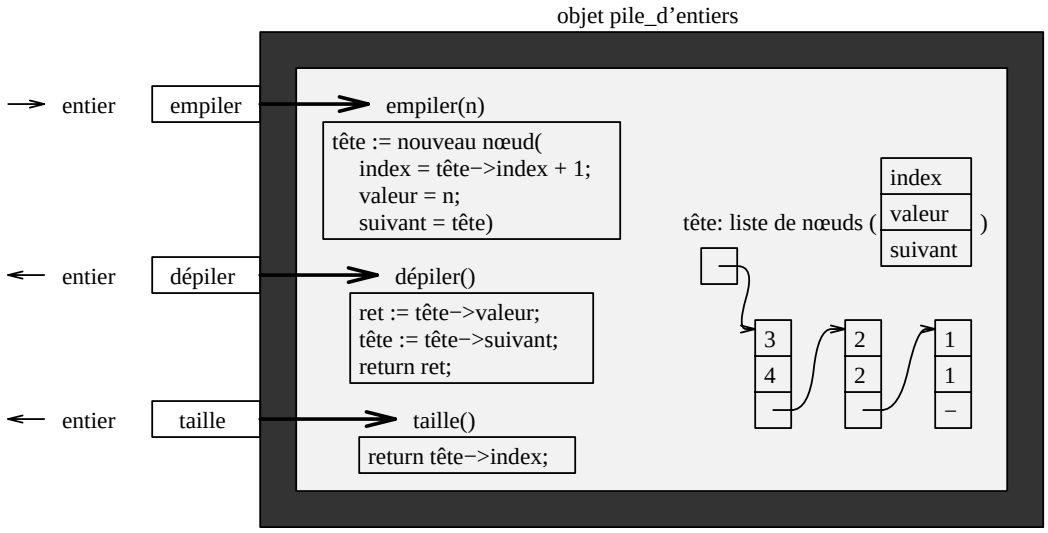

Fig. 3.2 : polymorphisme

L'interface, ou type abstrait, peut être déduite implicitement de la définition d'une mise en œuvre particulière, appelée classe. Mais il s'agit bien de deux concepts différents, et les langages qui séparent les définitions des types et des classes assurent une réelle encapsulation du code qui est un principe majeur de l'orientation objet.

Le polymorphisme complique les éventuelles vérifications statiquement faites par le compilateur à partir du type de la variable qui référence un objet. En effet le type effectif de l'objet peut être différent du type de la variable. On peut toutefois définir une règle, dite règle de conformité entre les types, qui assure que ce qui est vrai pour un type est vrai pour tous les types qui lui sont conformes. Le compilateur vérifie alors la validité des appels de méthode à partir du type de la variable, et contrôle lors de l'affectation de celle-ci que le type de l'objet affecté est conforme au type de la variable. La relation de conformité est décrite et justifiée dans <Bibliographie> et [57].

III.1.4 Héritage

L'héritage est la possibilité pour une classe, appelée sous-classe, de dire qu'une partie de son code ou de ses données reprend ceux définis dans une autre classe, appelée sa super-classe. Ces éléments repris sont dits hérités. Les relations entre une classe et sa super-classe ne sont pas uniformément définies dans tous les langages. En particulier, l'accès aux données définies dans la super-classe peut être ou non autorisé depuis la sous-classe. Néanmoins c'est une fonction puissante des langages à objets, qui a deux utilisations reconnues très intéressantes.

L'héritage permet la réutilisation de code. Dans ce cas la super-classe a été écrite avant la sous-classe, et éventuellement par un autre programmeur. L'héritage peut être aussi utilisé comme un outil structurant la programmation. Il permet la factorisation de code entre différentes classes écrites en même temps pour réaliser une application.

La propriété d'héritage fait que l'on retrouve dans une classe toutes les propriétés de sa super-classe, ce qui rend a priori la classe conforme à sa super-classe ; mais ce n'est pas toujours le cas. En effet l'héritage se complique du mécanisme de surcharge qui autorise la sous-classe à redéfinir un point d'entrée de sa super-classe, et les relations entre surcharge et conformité ne sont pas uniformément définies. Tous les langages sont généralement d'accord pour autoriser la manipulation d'un objet d'une classe à travers une variable typée par une de ses super-classes. Pour ce faire la plupart imposent la surcharge conforme. Eiffel par contre veut récupérer les avantages du polymorphisme sur les sous-classes, sans payer le prix de la règle de conformité parfois contraignante ; il en résulte de possibles erreurs à l'exécution.

III.2 Mécanisme bâti autour de l'appel de méthode

L'appel de méthode sur un objet est l'élément fondamental de la programmation dans un langage à objets ; il est donc logique de faire apparaître les exceptions à ce niveau. Nous regardons maintenant les conséquences des principes d'encapsulation des données et du code sur le signalement des exceptions, la recherche du traitant et son exécution.

III.2.1 Signalement

Lorsque la méthode appelée signale une exception, elle avertit l'appelant que le traitement normal qu'elle devait réaliser n'a pas pu aboutir. C'est une réponse au message initial d'invocation qui apparaît comme un paramètre de retour particulier. Ce retour doit être décrit dans les spécifications de l'interface de la méthode, au même titre que le retour normal. La méthode doit donc déclarer dans sa signature le type des exceptions qu'elle est susceptible de signaler. La déclaration peut prendre la forme d'une simple énumération, ou bien une autre forme suivant la nature des exceptions. Une méthode ne peut pas signaler une exception qu'elle n'a pas d'abord déclarée dans sa signature.

Si le signalement de l'exception accepte des paramètres, alors les types respectifs de ceux-ci doivent également être déclarés dans la signature de la méthode. A ce sujet il faut remarquer que les paramètres de signalement doivent avoir un sens au niveau des spécifications de la méthode. Il est donc incorrect de passer à l'appelant le maximum de variables lors du signalement, sous prétexte de lui donner de plus grandes possibilités de

traitement, car cela trahit la mise en œuvre de l'appelé et va à l'encontre du principe d'encapsulation du code.

Comme l'interface de la méthode entre en jeu dans la définition des règles de conformité, celles-ci doivent être complétées pour prendre en compte les exceptions. Comme l'exception est une réponse, elle doit suivre la règle des paramètres de retour, c'est-à-dire que le type de l'exception signalée par la surcharge de la méthode doit être conforme au type de l'exception signalée par la méthode de base. La vérification de cette règle peut prendre des formes différentes suivant la nature des exceptions dans le langage.

résumé

Le signalement d'une exception marque la terminaison d'une méthode qui n'a pas pu réaliser le traitement normalement attendu d'elle. Le type des exceptions qu'une méthode peut signaler doit apparaître dans sa signature. Les règles de conformité doivent être complétées pour les prendre en compte.

III.2.2 Portée de l'exception

L'occurrence d'une exception est liée à l'exécution d'une méthode sur un objet. L'entité qui traite l'exception doit donc avoir connaissance de l'appel de méthode. Comme le principe d'encapsulation du code interdit à l'appelant de savoir comment la méthode qu'il invoque est mise en œuvre, alors la portée de l'exception ne peut pas dépasser un niveau d'imbrication. Dans l'exemple qui suit, un objet de classe C_2 ne peut pas traiter l'exception E_0 signalée par l'invocation de M_0 sur o_0 ; il ne connaît que le type T_1 et non sa mise en œuvre, et n'a donc pas connaissance de l'appel $o_0.M_0$.

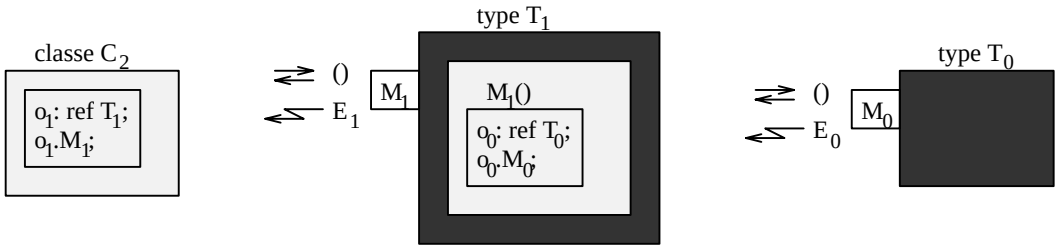

Fig. 3.3 : encapsulation et portée de l'exception

Dans l'autre sens, on peut se poser la question de savoir si l'objet signalant peut traiter lui-même l'exception. Mais ce cas n'est pas conforme à la sémantique de l'exception que nous avons donnée plus haut.

résumé

La portée de l'exception, dynamique, se réduit au strict appelant de la méthode qui la signale.

III.2.3 Traitement

Nous avons vu dans la section II.2.4les différents traitements généralement permis par un SGE. Nous discuterons la reprise en section III.4, et nous nous contentons de regarder dans cette section les politiques de terminaison, de réessai et de propagation.

Lorsque nous avons présenté la terminaison en section II.2.4, nous avons été volontairement imprécis en situant le point de reprise "au niveau de l'association du traitant". Or l'exception marque la non réalisation du traitement normal attendu de la méthode signalante. Le traitement alternatif fourni doit donc remplacer l'invocation signalante, quel que soit l'élément de programme auquel le traitant est associé. Cela est valable même si l'invocation signalante fait partie d'une expression. Il faut dans ce cas prévoir des mécanismes permettant au traitant de fournir une valeur de remplacement à celle normalement attendue de l'invocation signalante. En conséquence, dans l'exemple ci-dessous, après que l'invocation $o_1.M_1$ a signalé une exception et qu'un traitement alternatif a été exécuté, l'exécution reprend à l'invocation $o_2.M_2$. L'absence des traitants dans l'exemple est volontaire, leur déclaration ne doit pas influer sur la sémantique de la politique de terminaison.

```
...
o1.M1;
o2.M2;
...
```

```
...
o2.M2(o1.M1);
...
```

Le grand avantage de ce mécanisme est qu'il dissocie la syntaxe de l'association du traitant et la sémantique du traitement qu'il propose. On peut donc choisir l'emplacement où le traitant est déclaré, cela n'en modifie pas l'effet. Cela permet de répondre à deux objectifs signalés en section II.1.2, qui sont d'une part la séparation syntaxique du code exceptionnel et du code de l'algorithme principal, et d'autre part la factorisation maximale du code de traitement.

La politique de réessai doit de la même manière s'appliquer à l'invocation signalante. Un mot clef permet de réexécuter l'invocation de manière générique, sans avoir à la réécrire. Cela augmente d'autant les possibilités de factorisation d'un tel traitant.

La propagation doit satisfaire aux mêmes exigences que le signalement. En conséquence l'exception ne peut pas être réellement propagée telle quelle, puisqu'elle perd toute signification au niveau supérieur ; la propagation est en fait un resignalement. D'autre part l'exception propagée (resignalée) doit être déclarée dans la signature de la méthode. La propagation d'une exception, signalement d'une exception dans un traitant, est donc équivalente au signalement d'une exception dans la méthode associée.

résumé

La syntaxe d'association des traitants ne doit pas influer sur leur sémantique, ce qui autorise la séparation syntaxique du code de l'algorithme principal et de celui de traitement des situations exceptionnelles. Les politiques de terminaison et de réessai sont définies par rapport à l'appel de méthode signalant.

III.3 Traitants généraux

Un objectif d'un SGE est de permettre la déclaration de traitants généraux, permettant de factoriser des traitants, ou de définir des traitements par défaut. Un traitant est normalement associé à un élément de programme, qui en précise la portée. On ne peut bien sûr pas l'associer à un processus, à cause de la portée de l'exception réduite au strict appelant. Par contre on peut exploiter la notion de classe.

III.3.1 Associés à la classe de l'objet appelant

On peut déjà parler d'un traitant associé à la méthode appelante. Ce n'est que la généralisation de l'association à un bloc d'instructions ; le traitant protège toutes les invocations contenues dans la mise en œuvre de la méthode. De la même manière on peut associer un traitant à la classe appelante. Il protège alors toutes les invocations contenues dans la mise en œuvre des méthodes de la classe. Naturellement, plus le traitant est général, moins il a accès à un environnement précis. Un traitant de méthode perd l'accès aux variables des blocs internes. Un traitant de classe perd l'accès aux variables de travail de la méthode.

Cela paraît simple mais cache déjà un certain nombre de choses. La classe est une notion importante du langage à laquelle se rattache l'héritage ; il faut donc préciser les liens entre traitants de classe et héritage. Nous travaillerons à partir de l'exemple d'une classe C_0 qui définit une méthode M_0, et d'une classe C_1 qui hérite M_0 de C_0, et qui définit une nouvelle méthode M_1. Chaque classe possède un traitant de classe.

```
class C0 is
    method M0;
        o2: ref C2;
    begin
        o2.M2;
    end M0;
except
    erreur: <traitant0>;
end.
```

```
class C1 subclass of C0 is
    method M1;
        o3: ref C3;
    begin
        o3.M3;
    end;
except
    erreur: <traitant1>;
end.
```

La première question qui se pose est de savoir si le traitant de classe est hérité dans les sous-classes. Supposons que C_1 n'ait pas de traitant de classe, et que l'on invoque M_0 sur un objet de classe C_1. Il est alors normal d'exécuter *<traitant 0>* si $o_2.M_2$ signale l'exception *erreur*. Par contre, on supposant toujours que C_1 n'a pas de traitant de classe mais en invoquant cette fois M_1 sur l'objet de classe C_1, il est moins évident d'exécuter *<traitant 0>* si $o_3.M_3$ signale *erreur*. Cela dépend en fait du degré de dépendance qui existe entre super-classe et sous-classe. Si le langage impose à la sous-classe de ne manipuler les structures qu'elle hérite de sa super-classe que par son interface, alors le traitant ne peux pas être hérité. En effet ce traitant fait partie de la mise en œuvre de la classe, pas de son interface. Par contre si la sous-classe entretient des rapports privilégiés avec sa super-classe, alors l'héritage automatique du traitant peut-être un outil très utile pour offrir un traitement par défaut.

Reprenons les invocations précédentes sur l'objet de classe C_1, mais en prenant cette fois en compte l'existence du traitant de classe de C_1. Lorsqu'on exécute M_1, <*traitant* $_1$> est bien sûr prioritaire en cas de signalement de *erreur*, mais <*traitant* $_0$> peut être exécuté dans certaines circonstances, si bien sûr les relations entre sous-classe et super-classe sont particulières. Par contre le cas de l'exécution de M_0 est moins évident ; il pose la question de la liaison dynamique du traitant de classe. Si on cherche le traitant de classe à partir de la classe effective de l'objet, alors <*traitant* $_1$> est prioritaire sur <*traitant* $_0$>. Réciproquement si on cherche le traitant de classe à partir de la classe de définition de la méthode appelée, alors seul <*traitant* $_0$> peut être appelé. La réponse, encore une fois, dépend des relations qui existent entre une classe et sa super-classe. Si elles entretiennent des relations privilégiées alors la liaison dynamique du traitant de classe est possible. Dans ce cas le programmeur de la sous-classe doit vérifier que le traitant qu'il fournit remplace bien les fonctions offertes par le traitant qu'il cache. Si par contre la sous-classe ne voit pas le code des traitants de sa super-classe, alors la seconde solution doit être choisie.

résumé

Un traitant associé à une classe protège les instructions de toutes les méthodes définies dans la classe. En outre s'il existe des relations privilégiées entre une classe et sa super-classe, alors un traitant de classe protège également les instructions des méthodes définies dans les super-classes et les sous-classes de la classe.

III.3.2 Traitants système

D'autres traitants par défaut sont fournis par le système. Ils sont toujours valides. Ces traitants s'insèrent naturellement dans un modèle où les traitants de classe sont hérités. Si le langage possède une classe *Top*, racine de l'arbre d'héritage de toutes les classes du système, alors les traitants système sont assimilés à des traitants de cette classe. La règle de précédence s'en déduit naturellement : les traitants systèmes sont moins prioritaires que les traitants de classe.

Un traitant système peut servir à libérer le programmeur de l'obligation de fournir un traitant pour chaque exception potentiellement signalable. Il est appelé à chaque fois qu'une exception est signalée sans traitant spécifique fourni par le programmeur. Le traitement proposé doit garantir le principe énoncé en section II.1.2, qui veut qu'une exception, une fois détectée, ne puisse pas être oubliée. La solution la plus simple et qui doit être exécutée en dernier recours est la propagation d'une exception particulière. Un tel traitant pourrait s'écrire :

```
class Top is
    ...
    except
        all: raise UNCAUGHT_EXCEPTION;
end.
```

Une possibilité plus riche est d'appeler le metteur au point, ou un outil similaire qui engage un dialogue avec l'utilisateur pour lui permettre de fournir un traitant adéquat.

résumé

Le système fournit au moins un traitant par défaut, appelé en dernier recours, qui propage une exception particulière. Cela libère le programmeur de l'obligation de fournir un traitant pour chaque exception potentiellement signalable.

III.3.3 Propositions de traitement

Dans le cadre d'un dialogue que nous venons d'esquisser, le traitant système peut présenter à l'utilisateur un choix de traitants, ou plutôt de traitements proposés par les différents acteurs du signalement. Ces acteurs sont l'appelant, le signalant et l'exception elle-même.

Voyons sur un exemple la différence entre traitant et proposition de traitement. La méthode *envoi* d'*émetteur* cherche à envoyer un message. En cas d'échec le programmeur d'*émetteur* propose deux solutions : réessayer après un moment d'attente ou diffuser le message. L'appelé est donc capable de proposer des traitements à l'exception qu'il signale, mais il ne peut pas prendre lui-même la décision de les exécuter. C'est la première différence entre un traitant et une proposition de traitement : la proposition est ignorée lors du signalement. Seul un traitant explicite comme <*traitant*> dans l'exemple qui suit, ou un traitant système peut être exécuté, et c'est le traitant qui, après avoir demandé à l'utilisateur de choisir, exécute la proposition de traitement *répète* .

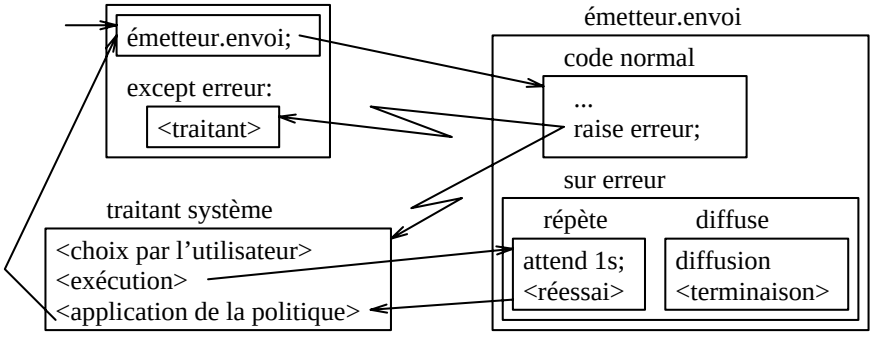

Fig. 3.4 : propositions de traitement

D'autre part la proposition de traitement ne peut pas appliquer elle-même une politique particulière. Cela se voit bien pour la propagation : l'appelé ne connaît pas les exceptions que l'appelant a le droit de propager. La proposition de traitement peut seulement suggérer une politique, que le traitant applique s'il le peut. De la même manière une proposition de traitement attachée à l'exception (dans le cas où l'exception est une classe) ne peut pas connaître le type de l'éventuelle valeur de remplacement attendue par l'appelant. Enfin pour ce qui est de la politique de réessai, la proposition de traitement ne peut pas réexécuter elle-même l'appel signalant en définissant de nouveaux traitants prioritaires lors de cette exécution (cf section V.4.1) ; seul le traitant peut le faire.

Nous venons de montrer que traitants et propositions de traitement sont deux concepts distincts. La possibilité de définir les seconds dans un langage impose l'introduction de nouveaux mots-clefs et la description des différences existant entre les politiques d'un traitant et les demandes d'exécution de politique d'une proposition de traitement. Nous pensons que cette complication n'est justifiable que dans un environnement interactif où l'utilisateur est censé être capable de prendre une décision.

Une variante de ce mécanisme consiste à demander à chacun des acteurs du signalement une seule proposition de traitement par défaut, qui sera appelée automatiquement par le traitant système en l'absence de traitants explicites. Mais les problèmes cités plus haut demeurent, aussi pensons nous que l'avantage gagné ne justifie pas le coût d'introduction de ce nouveau concept.

résumé

Le concept de proposition de traitement permet d'enrichir un environnemnet de traitements par défaut, mais il complique le mécanisme de gestion des exceptions. Nous pensons qu'il n'est justifiable que dans un environnement où un opérateur averti est capable de prendre des décisions lorsque des exceptions sont signalées.

III.4 Quid de la reprise

Le but de cette section est de discuter de la validité de la politique de reprise dans l'environnement objet. Nous regarderons les trois cas d'utilisation cités en section II.3, à savoir la reprise après le traitement d'une exception vraie, l'extension du "domaine d'appel normal" d'une procédure, l'appel explicite à l'appelant.

III.4.1 Traitement d'une exception vraie

L'exception vraie est la rencontre par l'appelé en cours d'exécution d'une circontance qui l'empêche de satisfaire la partie normale de ses postconditions. Lorsque l'appelant exécute le traitant dans son contexte puis demande la reprise au point de signalement, il suppose d'une part que les postconditions normales de l'appelé peuvent de nouveau être satisfaites, et d'autre part que l'exécution du traitant n'influe pas sur la validité des instructions déjà effectuées par l'appelé. Pour ce faire il est obligé de connaître la mise en œuvre de l'appelé. Cela viole le principe d'encapsulation des langages à objets.

Ce comportement peut toutefois être accepté dans une circonstance particulière. En cours de mise au point, le programmeur a le droit de faire beaucoup de choses qui lui sont habituellement interdites. On peut donc lui permettre de choisir son point de reprise, qui d'ailleurs ne sera pas forcément le point de signalement mais plus généralement un point compris entre le point de signalement et le point d'association. C'est lui qui est responsable de la validité de la politique qu'il choisit. Mais comme on vient de le dire les possibilités offertes par un outil de mise au point dépassent ce qui est ordinairement utilisable en cours de programmation. Cette utilisation de la reprise n'est donc pas suffisante pour offrir la politique de reprise en standard.

De manière similaire on pourrait accepter la reprise lorsque l'application s'exécute dans un environnement interactif. Là encore l'utilisateur serait responsable de vérifier la validité du

traitement choisi. Mais c'est plus dangereux puisque l'utilisateur en question est justement plus utilisateur que programmeur, et n'est donc pas qualifié pour cette vérification.

En fait cette politique correspond à un mécanisme intelligent de réessai, qui pourrait reprendre la partie de la pile entre point de signalement et point de traitement qu'il sait valide. On rejoint ainsi la sémantique du réessai, qui pose elle moins de problèmes, et on justifie même la possibilité de reprise en un point intermédiaire, chose impossible dans la politique de reprise habituelle.

III.4.2 Extension du "domaine d'appel normal"

La différence avec le cas précédent est que le traitant ne cherche pas à corriger les causes de l'exception pour atteindre les postconditions normales de l'appelé, mais il change ces postconditions. Il s'agit donc d'une extension de l'appelé, comme il est montré dans l'exemple de la section II.3.2.

Cette politique suppose au même titre que la précédente que le traitant connaît la mise en œuvre de l'appelé. Il y a donc violation du principe d'encapsulation. D'autre part l'extension des fonctions d'une classe dans un langage à objets passe normalement par la définition d'une sous-classe. Savoir si la sous-classe peut réutiliser la mise en œuvre de sa super-classe en fournissant un traitant de reprise dépend maintenant des relations privilégiées qui peuvent exister entre les deux classes. Mais de toutes façons cette utilisation est marginale, car elle suppose de nombreuses conditions rappelées en section II.3.2, et ne peut donc pas justifier la fourniture en standard de la politique de reprise.

III.4.3 Appel explicite à l'appelant

La différence avec le cas précédent est que cette fois l'appelé attend une réponse de l'appelant lorsqu'il signale l'exception. Nous voulons montrer que la sémantique de ce signalement est celle d'une invocation de méthode, méthode éventuellement passée en paramètre, plutôt que celle d'un signalement avec exécution d'un traitant de reprise.

Une différence entre signalement et appel de méthode a trait à la gestion des exceptions. Si un traitant signale une exception, celle-ci ne peut être traitée que par l'appelant de l'appelant. Dans le cas d'un appel de méthode, par contre, l'exception signalée est traitée par l'appelé (du premier appel). Cette différence a déjà été décrite dans la figure Fig. 2.3.

Or dans le cas qui nous intéresse, l'appelé (méthode *Normale* de la figure Fig. 2.3) est parfaitement capable et peut avoir envie de traiter une éventuelle exception signalée par le "traitant de reprise". Il ne s'agit donc pas d'un vrai signalement, mais d'un appel de méthode. Toutefois cet appel peut être plus ou moins complexe et demander des fonctions particulières au langage, comme nous allons maintenant le voir.

La principale caractéristique de l'exécution de cet appel est qu'elle utilise l'environnement de l'appelant. Pour que l'appelé soit capable d'effectuer l'appel, il faut donc qu'il connaisse cet environnement. Pour cela on peut dégager deux niveaux de complexité. Si l'environnement en question se restreint aux variables d'état de l'objet, alors il suffit que l'appelant passe son identification en paramètre à l'appelé. Ce dernier est alors capable d'appeler une méthode particulière sur son appelant. Une variante de cette technique est la possibilité pour l'appelant de passer en paramètre et son identification et la méthode à

appeler. Si par contre l'environnement nécessaire inclut des variables de travail de la méthode appelante, alors le langage doit fournir un tout autre mécanisme : les continuations. Une continuation est une notion complexe mais très puissante, qui recouvre à la fois du code et l'environnemnet dans lequel il doit s'exécuter. Quand l'appelant définit la continuation qu'il passera en paramètre à l'appelé, il y adjoint son environnement.

En résumé la fonction recherchée n'est pas celle d'un traitant de reprise. Il s'agit en fait pour les cas simples de l'appel d'une méthode sur l'objet appelant passé en paramètre, et pour les cas complexes de l'activation d'une continuation passée en paramètre. L'offre de la politique de reprise par un SGE ne peut pas se justifier comme substitut à un mécanisme de continuations absent.

III.4.4 Résumé

Nous pensons donc que la politique de reprise ne doit pas être proposée dans un environnement à objets, bien que cette déclaration doive être tempérée. En effet, le principe d'encapsulation nie toute imbrication intime de la mise en œuvre de deux types, or il existe des cas où ce n'est pas vrai. Quelles seraient les raisons des *friend*s de C++ ? Quelles sont les relations entre une classe et sa super-classe ? Comment la récursivité se place-t-elle dans ce schéma ? Nous ne répondons pas ici à ces questions, qui demandent à raffiner la définition de l'encapsulation dans le formalisme objet. Nous avons strictement réagi par rapport au formalisme existant, préférant être trop stricts que pas assez.

III.5 Besoin de cohérence

Lorsque nous avons présenté les politiques de terminaison, soit traitement alternatif puis terminaison, soit correction puis réessai, nous avons signalé la difficulté de contrôler l'effet de la part de l'appelé exécutée avant le signalement. Nous regardons maintenant comment on peut traiter ce problème.

III.5.1 Cohérence de l'objet

La sémantique idéale de ces politiques demande que tout ce qui a été incomplètement fait soit effacé, autrement dit que l'appel de méthode soit une opération atomique. Mais cela mène à définir un mécanisme lourd, c.f. le "backward recovery" de [15]. D'un autre côté, vouloir spécifier précisément les postconditions d'une méthode lors du signalement de l'exception peut se révéler difficile, puisque fortement dépendant de la mise en œuvre. Nous choisissons donc une solution intermédiaire, supposant l'existence d'états cohérents de l'objet.

La cohérence de l'objet est d'autre part nécessaire pour toute utilisation future de l'objet, qu'il soit partagé ou persistant.

La cohérence de l'objet peut être définie à partir d'assertions portant sur les données de l'objet. Ces assertions sont les invariants de l'objet ; ils sont explicitement déclarés dans le langage Eiffel. Ils doivent être toujours vérifiés, excepté en des moments bien précis contrôlés par l'objet. Dans Eiffel ils sont contrôlés en début et en fin de méthode. Nous pensons qu'ils doivent également être vérifiés en cas de sortie d'une méthode par signalement d'une

exception, garantissant ainsi des préconditions minimales à l'exécution des traitants. Or la nature d'un SGE rend difficile la satisfaction de cette contrainte, comme nous allons le voir.

III.5.2 Risques induits par les exceptions

Le premier effet des exceptions est de multiplier les points de sortie de la méthode. Sans exceptions, la seule sortie possible se fait par l'instruction **return**, et il est aisé de se restreindre à un point de sortie unique par méthode ; le code de restauration peut alors être placé à ce niveau. Avec les exceptions, il existe une deuxième instruction de sortie qui est le **raise**, signalement d'une exception, et il faut en outre gérer les points de sortie définis dans les traitants. Or le traitement principal qu'offre un mécanisme de traitement d'exceptions est la propagation de l'exception. Les points de sortie par signalement se trouvent alors répartis dans de nombreux traitants, et il n'est donc pas possible de réduire le nombre de points de sortie à deux, un normal et un exceptionnel.

De plus la présence de traitants généraux complique le travail. Comme nous l'avons vu en section III.3, de tels traitants peuvent être définis sur les super-classes de l'objet, voire même sur la super-classe commune *Top* qui peut invoquer des propositions de traitement diversement définies. Le point de sigalement se trouve alors fort éloigné de la classe effective de l'objet signalant, et il devient difficile d'en vérifier ou restaurer la cohérence.

Vouloir mélanger les deux notions de traitement de l'exception et de maintien de la cohérence de l'objet mène aux deux problèmes suivants. D'une part le code de restauration de l'objet est dupliqué aux multiples points de sortie, en particulier aux points de propagation de différents traitants. D'autre part les traitants généraux ne peuvent pratiquement plus être réutilisés, en particulier le traitant d'une classe dans une sous-classe, car la modification des invariants impose la redéfinition de la partie restauration du traitant. Un mécanisme spécifique, distinct du mécanisme d'exceptions, est donc nécessaire pour gérer la cohérence des objets.

III.5.3 Finalisation

La finalisation est un outil qui garantit l'exécution d'un morceau de code en fin de méthode, ou plus généralement en fin de bloc d'instructions, quoi qu'il ait pu se passer durant l'exécution de ladite méthode (respectivement dudit bloc d'instructions). Cet outil vise à aider le programmeur à maîtriser la multiplication des points de sortie due aux exceptions. Mais nous pensons que dans un langage à objets il faut aller plus loin, et traiter le problème spécifique du maintien de la cohérence.

III.5.4 Résumé

En cas de signalement d'une exception l'objet risque d'être dans un état incohérent. Plutôt que d'offrir un mécanisme lourd d'atomicité, nous pensons suffisant d'exécuter du code de restauration fourni par le programmeur. La multiplication des points de sortie d'une méthode due aux exceptions d'une part, le besoin d'indépendance du code de restauration par rapport aux traitants d'exception d'autre part, imposent la définition d'un mécanisme spécifique de

restauration. Un simple outil de finalisation n'est pas suffisant car il ne cherche pas à résoudre le problème du maintien de la cohérence des objets.

III.6 Les exceptions comme objets typés

Nous allons maintenant étudier l'intérêt d'une représentation des exceptions par des objets. Pour ce faire on peut soit créer une classe *Exception* dont toutes les occurrences d'exception seront des instances, soit créer une classe par genre d'exception dont chaque occurrence sera une instance. Nous n'envisageons que la seconde solution, car la première n'offre pas de fonctions réellement différentes de la représentation par des chaînes de caractères.

III.6.1 Mécanisme de base

Les exceptions signalables par une méthode sont en fait des noms de classes d'exception. Le programmeur signale une exception en donnant le nom de la classe associée. Le système crée alors une instance de cette classe. Le traitant est ensuite recherché et exécuté, en ayant la possibilité de manipuler une référence sur l'objet exception créé.

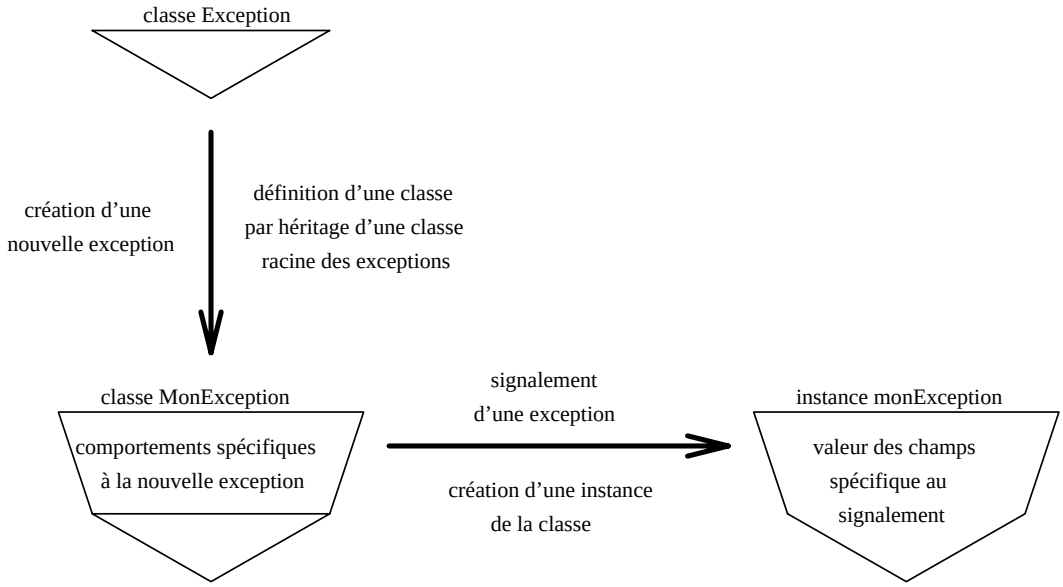

Fig. 3.5 : classe et instance d'exception

L'intérêt de cette représentation réside dans la possibilité de personnaliser l'état des classes et instances d'exception, et d'associer aux genres d'exceptions des comportements spécifiques. Par contre elle présente déjà un premier inconvénient, en universalisant le nom d'une classe d'exception. Le lien entre une exception et le type qui la signale disparaît, et n'importe quel autre type peut reprendre et signaler l'exception.

III.6.2 Champs de l'objet exception

L'état d'un objet exception peut (doit ?) servir à représenter le contexte du signalement. Ce contexte peut n'être que la référence de l'objet signalant, acompagnée du nom de la méthode exécutée. Il peut aussi contenir un cliché de l'état de l'objet au moment du signalement, qui sera ainsi accessible au traitant. Ces informations sont fournies par le système, qui les met à jour dans l'objet exception créé lors du signalement. Elles peuvent être utiles en donnant accès de manière générique à l'entité appelée, permettant éventuellement la définition de traitants généraux comme par exemple l'impression d'un message d'erreur décrivant l'objet et la méthode signalants.

Le programmeur peut également ajouter des champs à cet état. Il doit donc les définir dans la classe de l'exception, et il les initialise dans l'instance par des valeurs qu'il donne lors du signalement. Comme ces variables seront accessibles au traitant, ce sont bien des paramètres du signalement, dont nous avons parlé en section II.2.1. Pour contrôler la validité de ces paramètres, le signalement devient une méthode de l'exception, dont la signature est définie dans le type de l'exception (voir en section suivante).

Si la mise en œuvre des exceptions suit ce modèle, c'est-à-dire qu'un objet est effectivement créé et initialisé à chaque exception signalée, alors on risque d'alourdir le mécanisme. On peut en effet supposer que la manipulation de l'objet sera plus coûteuse que la manipulation directe d'informations contextuelles. Réciproquement si la mise en œuvre ne suit pas le modèle, alors on peut mettre en doute la validité de ce dernier.

Cette lourdeur risque aussi de transparaître au niveau du langage. En effet déclarer une nouvelle exception demande de définir une classe. Cela peut toutefois être fait automatiquement par le système (compilateur) dans le cas d'une exception simple qui ne définit rien de nouveau.

III.6.3 Méthodes de la classe exception

III.6.3.1 Signalement

La première méthode d'une classe d'exception a trait au signalement. Le signalement par un mot-clef devient un appel de méthode "classique", ou du moins apparaît comme tel syntaxiquement. Notons en premier lieu que cet appel de méthode est réalisé sur l'objet classe d'exception, puisque, à l'instar de la méthode *New*, elle en crée une instance. Il devient donc beaucoup moins classique si le langage ne représente pas les classes par des objets.

L'intérêt de faire du signalement une invocation de méthode est bien sûr de pouvoir le personnaliser suivant l'exception. Pour cela plusieurs possibilités s'offrent. La première est la définition de paramètres de signalement. Comme pour toute méthode, le programmeur définit la signature de la méthode *Raise* de signalement lorsqu'il déclare le type de l'exception.

Mais on se rappelle que la méthode *Raise* est invoquée sur un objet classe, donc qu'elle est définie sur une méta-classe. Cela mène donc au schéma décrit dans Fig. 3.6. La méta-classe déclare une méthode *Raise* avec les paramètres voulus. Cette méthode crée une instance d'exception et invoque sur cette instance la méthode *Raise* définie sur la classe et qui accepte les mêmes paramètres. Cette mécanique est classique pour la création d'objet, mais elle reste compliquée.

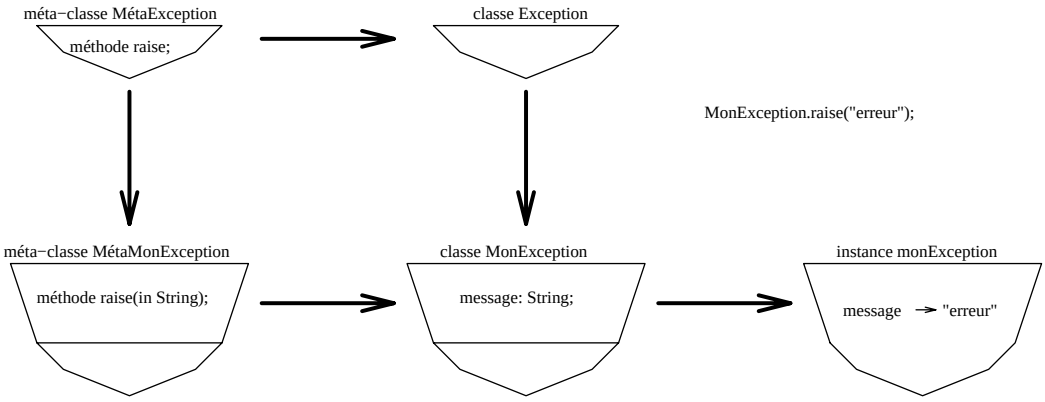

Fig. 3.6 : classe et méta-classe d'exception

Allons plus loin. Puisque le signalement est devenu une invocation classique de méthode, on peut certainement créer plusieurs méthodes de signalement sur une classe d'exception, acceptant par exemple des paramètres différents. On peut aussi donner à ces méthodes un autre nom que *Raise*. Cela veut dire que le compilateur ne peut plus reconnaître une méthode de signalement ; c'est normal puisque c'est une méthode classique ! Et pourtant une méthode de signalement doit suivre un comportement particulier ; elle doit rechercher puis invoquer un traitant. La solution consiste à définir ce comportement dans la méthode *Raise* d'une classe système *Exception*, à déclarer toute nouvelle exception comme sous-classe de *Exception*, et d'appeler cette méthode *Raise* à la fin de toute méthode de signalement.

Poussons encore un peu plus loin la standardisation du signalement d'exception. Comme nous l'avons vu, le signalement inclut la recherche du traitant. Ce nouvel aspect peut également être paramétré, par exemple pour résoudre le problème du choix entre un traitant et un autre moins précis mais plus interne, problème signalé en section II.2.2. Comme ce genre de mécanisme est difficilement programmable, les différentes politiques de recherche du traitant peuvent être offertes par différentes méthodes de signalement définies sur la classe système *Exception*.

La principale critique qui peut-être faite de la banalisation du signalement est justement le manque de caractérisation externe d'un mécanisme qui reste et restera particulier. On donne la syntaxe d'un appel de méthode à une instruction qui ne retourne pas (politique de terminaison). On met en oeuvre des mécanismes lourds (manipulation de méta-classes). Pire que tout, la méthode de signalement doit invoquer une méthode de signalement définie sur une classe système pour être valide. Si le programmeur l'oublie, alors le premier principe d'un SGE n'est plus respecté, i.e. une exception détectée est oubliée (cf section II.1.2).

III.6.3.2 Traitement

Les différentes politiques de traitement peuvent également emprunter la syntaxe d'un appel de méthode sur l'objet exception. On peut donc trouver des méthodes *Resume* pour la

reprise, *Exit* pour la terminaison, ou *Retry* pour le réessai, définies sur les classes d'exception.

Un intérêt de cette technique est la caractérisation possible des traitements autorisés pour une classe d'exception. Il suffit en effet de définir une classe avec la seule méthode *Exit* pour interdire la reprise après le signalement d'une telle exception. Réciproquement la définition de la seule méthode *Resume* impose l'application de la politique de reprise.

Toutefois on peut faire à propos de ces méthodes la même remarque que pour la méthode de signalement, à savoir qu'elles n'ont pas la sémantique d'une méthode normale. Il est en effet inutile de faire suivre un appel à ces méthodes par une instruction ; elle ne serait jamais exécutée.

D'autre part la représentation des politiques de terminaison et de reprise sous forme de méthodes tend à cacher des dépendances qui deviennent alors difficiles à comprendre. En particulier le travail du compilateur d'un langage fortement typé devient difficile. Tout d'abord comment expliquer que le paramètre d'entrée de la méthode *Resume* doive être du même type que le paramètre de sortie de la méthode *Raise* ? Nous le montrons dans la figure suivante : le paramètre *val* passé à la méthode *Resume* est bien celui qui est affecté à *ret*, donc celui attendu en retour de *Raise*.

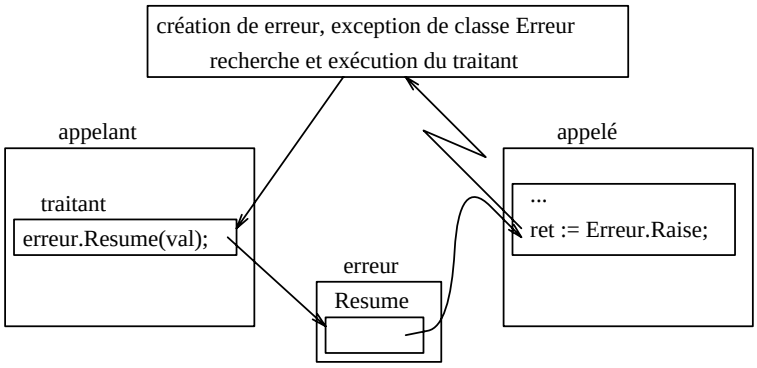

Fig. 3.7 : la reprise est une méthode

On doit même aller plus loin et imposer que toutes les méthodes de signalement aient le même type de paramètre de retour, type égal à celui du paramètre d'entrée de toutes les éventuelles méthodes de reprise. En effet en présence d'un appel à une primitive de reprise, le compilateur doit se contenter de vérifier que l'appel est bien conforme à la déclaration d'une des primitives. Il serait incapable de détecter l'appel à une primitive qui ne correspondrait pas au signalement sans traiter de manière particulière ces méthodes.

On peut trouver un problème analogue avec la méthode *Exit*. La politique de terminaison peut en effet demander un paramètre pour remplacer celui attendu de la méthode signalante. Or le type de ce paramètre doit être conforme au type du paramètre attendu. Il paraît difficile pour le compilateur de vérifier cette contrainte lors de la déclaration du type de l'exception, à moins que la dépendance entre l'exception et la méthode qui la signale ne soit explicitée.

III.6.3.3 Proposition de traitement

On peut enfin utiliser des méthodes d'une classe d'exception pour proposer des traitements à l'exception signalée. Ces propositions peuvent être invoquées par le programmeur utilisateur de la classe signalante, par l'utilisateur dans le cadre d'un outil interactif, ou par le système qui peut utiliser une méthode de nom prédéfini comme traitant par défaut (c.f. section III.3.3).

III.6.4 Hiérarchie d'exceptions

L'intérêt majeur de la représentation des exceptions par des classes réside dans les possibilités offertes par l'héritage.

Tout d'abord la sous-classe hérite des méthodes de signalement de sa super-classe. Cela permet non seulement de récupérer une politique de recherche, comme il est dit à la section précédente, mais surtout d'isoler ces mécanismes spécifiques dans la classe système *Exception*, évitant ainsi leur trop large diffusion.

La sous-classe hérite également des propositions de traitement définies sur la super-classe.

Enfin, et surtout, la sous-classe peut être considérée comme une spécialisation de sa super-classe. C'est particulièrement utile dans l'exemple suivant, où l'on veut raffiner dans le sous-type l'exception *err* signalable par la méthode du super-type.

```
type A is
    method M;
        signals err;
end A.

type err
subtype of Exception is
end err.
```

```
type B subtype of A is
    method M;
        signals e1, e2;
end B.

type e1 subtype of err is
end e1.
type e2 subtype of err is
end e2.
```

Un appelant qui invoque la méthode *M* sur un objet à travers une variable de type *A* peut alors traiter l'exception *e1* éventuellement signalée par l'objet, dont le type effectif peut être *B*, cette exception étant une spécialisation de l'exception *err* connue de l'appelant. Cet propriété est longuement discutée en section VI.1, et est reliée aux règles de conformité développées en section V.8. Cela permet une plus grande factorisation du traitant à l'association. Il faut toutefois remarquer que l'on peut obtenir cette propriété intéressante de hiérarchie d'exceptions sans pour autant imposer aux exceptions la nature d'objets typés.

III.6.5 Résumé

La représentation complète des exceptions sous forme d'objets typés présente un certain nombre d'avantages. La structure hiérarchique des exceptions qui en découle permet la spécialisation dans un sous-type d'une exception déclarée dans un type. Les champs de l'objet exception sont un moyen propre de passer des paramètres lors du signalement. Les méthodes définies sur la classe d'exception permettent de contrôler les traitements qui y sont applicables.

Toutefois cette représentation pose de nombreux problèmes. La manipulation des exceptions est alourdie. Au niveau programmatif elle impose la manipulation de classes, voire de méta-classes. Au niveau exécutif on peut penser que la manipulation d'un objet est plus coûteuse que celle d'un paramètre de retour supplémentaire. D'autre part la banalisation des mécanismes de signalement et de traitement amène la création de méthodes dont la sémantique est très étrange, cache les dépendances qui existent entre exception signalée et méthode signalante, et rend difficile la vérification par le compilateur des contraintes qui y sont liées.

Nous pensons que l'importance des remarques qui s'opposent à la représentation complète des exceptions sous forme d'objets typés montre que cette représentation n'est pas adaptée. Il faut si possible rechercher une représentation qui accepte une structure hiérarchique, et éventuellement permet de passer des paramètres de signalement. Il ne faut pas chercher à traduire les mécanismes d'un SGE sous forme de méthodes.

III.7 Conclusions

Nous rappelons ici les principaux points discutés dans le chapitre.

Le mécanisme doit être bâti autour de l'appel de méthode. Une exception signalable par une méthode doit être déclarée dans sa signature. Le traitement ne doit concerner que l'appel de méthode signalant ; le point de reprise de la politique de terminaison est indépendant du point d'association du traitant.

On utilise l'association de traitants aux classes et l'héritage pour définir des traitants généraux.

Offrir la politique de reprise ne convient pas au principe strictement appliqué d'encapsulation des objets. Pour l'accepter il faudrait pouvoir définir des relations privilégiées entre classes.

Il est important de fournir un mécanisme spécifique, distinct du mécanisme de gestion des exceptions, qui aide le programmeur à garantir la cohérence de ses objets.

Il ne faut pas représenter les mécanismes de gestion des exceptions sous forme de méthodes d'une classe d'exception. Cela banalise des mécanismes spécifiques et pose de nombreux problèmes.

Chapitre IV

Propositions existantes

Nous avons dans le chapitre précédent tiré un certain nombre de conclusions quant à un mécanisme de gestion d'exceptions dans un langage à objets. Nous n'en avons abordé que les grandes lignes. Nous allons maintenant étudier les propositions d'une large palette de langages existants, en les confrontant à ces conclusions. Nous avons découpé cet état de l'art en trois principales parties. Tout d'abord nous regardons le modèle choisi par beaucoup de langages, le modèle avec terminaison. Ce modèle dérive des mécanismes des langages modulaires, dont nous parlons en premier lieu. Ensuite nous étudions les propositions pour le langage Lisp et ses dérivés, en développant particulièrement le langage Lore développé au LITP. Nous avons ensuite regroupé dans une même section quatre propositions hors classement proposant des solutions originales. La première n'est pas liée à un langage, bien qu'étant décrite par rapport à la sémantique d'Algol 68. Les trois suivantes sont celles de Smalltalk-80, de C++ et d'Eiffel.

IV.1 Modèles avec terminaison

IV.1.1 Langages modulaires

Les langages modulaires qui ont intégré un mécanisme de gestion des exceptions ont tous plus ou moins suivi le même modèle. Mais l'outil proposé est essentiellement l'extension des primitives de déroutement classiques, et n'intègre pas les contraintes d'un langage modulaire comme le principe d'encapsulation dont nous avons largement parlé au chapitre précédent. Nous envisageons ici les propositions de Ada, CLU, et Modula-2+, qui ont été reprises par nombre de langages à objets.

IV.1.1.1 Mécanisme de base

Voyons tout d'abord le signalement. Les remarques faites en section III.2.1 sont ici encore applicables. Un module est l'unité d'encapsulation ; son utilisation est conditionnée par l'interface qu'il exporte. Il est donc normal de déclarer les exceptions signalables par les opérations du module dans cette interface, et de garantir (à la compilation) que seules ces exceptions sont effectivement signalées. Or seul CLU le fait complètement. L'exception dans CLU est une simple chaîne de caractères, éventuellement accompagnée d'un ensemble de paramètres. Cette chaîne doit se retrouver dans la signature de l'opération signalante, au signalement, et dans le bloc de traitement de l'opération appelante. Le langage fournit aussi un traitant de propagation par défaut. Pour assurer la validité de la règle précédente dans ce cas, le système ne propage pas l'exception telle quelle, mais resignale l'exception prédéfinie *failure* .

Ada et Modula-2+ ont laissé à l'exception le sens global au flot d'exécution d'une instruction *setjump/longjump* . A l'inverse de CLU, où la déclaration de l'exception dans le

signalant est automatiquement récupérée pour tout appelant, la déclaration doit ici être globale aux deux modules signalant et traitant, donc en fait globale au programme entier. En fait l'exception est déclarée dans un module, par exemple le signalant, et importée dans le module traitant. Mais ce n'est pas pour autant que l'exception est locale au module où elle est déclarée. Il est en effet possible à un module de signaler une exception déclarée dans un autre.

Le sens global de l'exception dans Ada et Modula-2+ est également visible à l'exécution, car le traitant de l'exception est recherché dans la liste des appelants, successivement du plus proche jusqu'au premier appelant de l'activité, comme il est montré dans la figure Fig. 2.1. Cela revient en fait à un traitant par défaut qui propage l'exception inchangée, sans se préoccuper si elle est déclarée dans l'interface de l'opération. En fait dans Modula-2+ il est possible de déclarer dans l'interface d'une procédure, grâce au mot clef **raises**, la liste des exceptions qu'elle peut signaler. Cette vérification est faite à l'exécution. Le problème est que, par défaut, la procédure peut signaler n'importe quelle exception. La politique d'utilisation de cette propriété est d'imposer la déclaration d'une clause **raises** pour toute procédure utilisée depuis un module externe, mais de laisser libres les procédures utilisées en interne par le module. Notons également, mais nous y reviendrons dans le paragraphe suivant, que l'exception peut être traitée par l'opération même qui l'a signalée. Ces remarques ne sont pas valables pour CLU, où le traitant est recherché dans le strict appelant, conformément aux conclusions de la section III.2.2.

IV.1.1.2 Sémantique de l'association

Le deuxième point, caractéristique cette fois de nos trois langages de référence, est lié à la sémantique de l'association du traitant. Comme il est dit dans [46], la déclaration du traitant ne doit pas être collée à l'appel d'opération signalante. Mais les auteurs ne vont pas jusqu'au bout de ce principe, et donnent un sens au point d'association du traitant. Nous sommes allés plus loin en section III.2.3, en disant que le point de déclaration du traitant ne doit pas influer sur son effet. Les trois langages étudiés associent grossièrement un traitant à un bloc d'instructions. Si le traitant choisit la politique de terminaison, alors l'exécution reprend à l'instruction suivant immédiatement le bloc d'instructions, et non pas seulement l'instruction signalante. Cela, lié à la possibilité de traiter une exception dans l'opération où elle est signalée, permet de sortir élégamment de la boucle suivante, quel que soit le nombre de boucles imbriquées dans lesquelles on peut se trouver. On prend la syntaxe d'Ada.

```
begin
    while true loop
    begin
        ...
        while ... loop
        begin
            ...
            raise sortie;
            ...
            raise erreur;
    end; end;
exception
when sortie =>
    -- sortie normale
when erreur =>
    -- sortie anormale, propagation de l'exception
    raise;
end;
```

Mais il faut noter que cette technique revient à associer un déroutement local à la politique de terminaison du traitant suivant son point d'association. La technique prônée en III.2.3 impose la définition de la sémantique d'une instruction de déroutement local dans un traitant, liant éventuellement l'effet d'un **break** au point d'association du traitant mais uniquement dans ce cas, et permettant dans tous les autres cas la déclaration du traitant indépendamment du traitement qu'il propose. Le coût dans l'exemple précédent serait de rajouter un **break** à la fin du traitant de l'exception *sortie* ; le gain est la séparation syntaxique du code de tous les traitants qui ne proposent pas de déroutement local.

IV.1.1.3 Paramètres du signalement

Le problème des paramètres de signalement est diversement traité. Dans Ada il est impossible d'en fournir ; dans Modula-2+ on autorise un paramètre, déclaré comme variable de la procédure ; dans CLU un nombre quelconque de paramètres est permis, et ils sont déclarés comme paramètres formels du traitant à l'association. La première solution n'est pas forcément la moins bonne (c'est celle que nous avons retenue pour Guide) ; elle évite en tout cas les problèmes que nous allons préciser. Le paramètre d'une exception de Modula-2+ n'apparaît pas dans la signature de la procédure. C'est contraire au principe énoncé en section III.2.1. Cela provient d'un certain amalgamme entre deux exceptions qui portent le même nom mais ont des paramètres différents. Que ce soit en Modula-2+ ou en CLU, il est possible de traiter une exception avec paramètres par un traitant sans paramètre. Le fait-on pour des procédures ? D'autre part, la non limitation du nombre de paramètres de signalement en CLU doit conduire, d'après l'auteur, à la fourniture par le signalant d'un maximum d'informations sur l'exception. Or, comme nous le disons en section III.2.1, il faut veiller à ne passer que ce qui a un sens au niveau de la spécification de l'opération.

IV.1.1.4 Politiques de traitement

La seule politique, autre que la propagation, offerte par les trois langages est la terminaison. L'intérêt, déclaré au moins par les auteurs de CLU, est la simplicité de la sémantique du mécanisme résultant. Cet abandon de la politique de reprise est bien sûr conforme à la discussion de la section III.4.

IV.1.2 Trellis-Owl

Trellis est le type même des langages à objets qui ont intégré un mécanisme de gestion d'exceptions sur la base des mécanismes des langages modulaires. Il se contente de la politique de terminaison et reprend la technique de traitement liée au bloc d'association du traitant. La caractéristique de ce langage est qu'il respecte le principe de déclaration dans l'interface des exceptions signalées dans la méthode, et que la recherche du traitant se limite au strict appelant de la méthode signalante. Ces deux principes sont ceux respectés par CLU et adoptés dans les sections III.2.1 et III.2.2, et ils sont repris par les différents langages abordés dans la suite de cette section IV.1, mis à part Modula-3. Ils sont assurés dans Trellis par le traitant suivant, ajouté par défaut à la fin de chaque opération, qui propage l'exception système *Failure* lorsqu'aucun autre traitement n'est proposé (on reprend l'expression de ce traitant directement depuis le manuel du langage, donc avec la syntaxe de Trellis).

```
except
    on Failure do resignal;
    otherwise do signal Failure
        with ("Unhandled exception: " & exception_name);
```

Pour ce qui est des paramètres de signalement, le langage en autorise un seul. Son type apparaît dans la signature de la méthode. Si l'exception est déclarée avec un paramètre, celui-ci doit être fourni lors du signalement. Le traitant le récupère en déclarant un paramètre formel à l'association. Par contre l'exception peut être traitée par un traitant qui n'a pas spécifié de paramètre.

IV.1.3 Modula-3

Modula-3 reprend le mécanisme proposé pour Modula-2+ ; il en reprend aussi les défauts déjà cités. Mais il en intègre également l'outil de finalisation que nous allons maintenant décrire, et reprend le module *Thread* qui introduit le parallélisme, et dont nous allons voir les liens avec les exceptions.

IV.1.3.1 Finalisation

Cet outil, introduit par les mots clefs **try** et **finally**, assure que le bloc marqué par **finally** sera exécuté quoi qu'il se passe durant l'exécution du bloc marqué par **try**. L'exemple suivant en montre une utilisation possible, transposée dans l'environnement C/Unix. On ouvre un fichier temporaire avant d'exécuter l'algorithme principal et on est sûr de détruire le fichier quoi qu'il advienne, y compris lors de la phase de mise au point de *FaitLeTravail* .

```
try
    fichierTemp = open("/tmp/##", O_RDWR | O_CREAT, 0777);
    FaitLeTravail();
finally
    unlink(fichierTemp);
end;
```

Comme nous l'avons dit en section III.5.3, cet outil n'est pas réellement un outil de restauration. Il permet de libérer des ressources en fin de méthode, mais la restauration vise à garantir que l'objet est dans un état cohérent. Lorsqu'une exception survient, cet objectif de restauration peut conduire à défaire ce qui a été fait jusqu'alors, ou même à prendre des décisions plus draconiennes comme dans l'exemple de la section V.7.2. Ce type d'action ne doit pas être réalisé en cas de sortie normale, car l'objet est supposé être dans un état cohérent. C'est la différence qui existe entre un mécanisme de finalisation comme celui de Modula-3, et un mécanisme de restauration dont nous avons exprimé le besoin en section III.5.

IV.1.3.2 Parallélisme

Le parallélisme apparaît dans Modula-3 par l'intermédiaire du module *Thread*, qui permet de manipuler des activités. La création d'une activité est asynchrone, par la primitive *Fork*, mais deux activités peuvent se resynchroniser grâce à la primitive *Join*. D'autre part, les différentes activités coopèrent par l'intermédiaire de verrous d'exclusion mutuelle, manipulables par *Acquire* et *Release*, et de variables condition au sens des moniteurs de Hoare, manipulables par *Wait*, *Signal* et *Broadcast*.

Comme l'activité fille est lancée en asynchrone, une exception signalée durant son exécution et non rattrapée en interne n'est pas propagée à l'activité mère. En effet, le signalement d'une exception est un évènement synchrone. Si on veut la propager à l'activité mère, ce n'est plus une exception mais une interruption. Le seul point d'action possible est donc au niveau du *Join*.

Les concepteurs de Modula-3 ont choisi d'associer à l'activité une variable d'état supplémentaire manipulable par deux procédures d'écriture et de lecture, *Alert* et *TestAlert*. Cet état est rendu accessible à l'activité mère par l'intermédiaire d'une procédure de rendez-vous spéciale, *AlertJoin*, qui signale l'exception *Alerted* si ou lorsque l'activité fille est ainsi marquée. La terminaison "incorrecte" de l'activité fille est donc réalisée explicitement.

Par contre, rien n'est prévu dans le manuel du langage pour traiter la terminaison d'une activité par une exception non traitée. En fait les auteurs font semblant de se garantir contre ce cas en imposant à la procédure initiale de l'activité de ne pas signaler d'exception. La signature de cette procédure inclue une clause **raises** vide. Or cette vérification est faite à l'exécution, et rien n'est dit sur ce qui se passe lorsqu'une telle situation survient.

D'autre part les auteurs ont voulu fournir un outil minimal d'interruption basé sur le mécanisme précédent. La primitive *Alert* accepte une activité en paramètre, permettant à une activité de modifier l'état d'une autre. Cette possibilité marche de concert avec la procédure *AlertWait*, qui est un *Wait* modifié pour signaler l'exception *Alerted* si ou lorsque l'activité qui s'exécute est marquée.

IV.1.4 Argus

Le langage Argus est construit sur la base de CLU. Il reprend donc les grandes lignes du mécanisme de traitement d'exceptions de ce dernier, en prenant en compte les nouvelles notions d'*action*, de parallélisme, et surtout de *guardian*.

Argus reprend donc un modèle avec terminaison. Cela se voit tout particulièrement sur l'impact du signalement sur une *action*. L'*action* est une entité nouvelle d'Argus, qui traduit la notion d'atomicité. Lorsqu'une exception est signalée dans une *action* mais n'y est pas traitée, le programmeur doit choisir entre le commit ou l'abort de l'*action*. Il ne peut pas choisir la solution intermédiaire qu'offrirait un mécanisme de reprise. On peut d'ailleurs se rendre compte grâce à ce point de la sémantique floue de la reprise, puisque la nouvelle notion d'*action* aurait bien du mal à être intégrée de manière orthogonale à un mécanisme qui offrirait cette politique.

Les auteurs ont également repris tel quel le mécanisme d'association du traitant de CLU, qui ajoute au traitement de terminaison un déroutement local lié au point d'association. Ils notent simplement que cela peut mener à la duplication du code de traitement, en particulier dans le cas de l'instruction de parallélisme. Le fait que le traitant soit déclaré à l'extérieur du bloc parallèle ou à l'intérieur de chacune des branches a un impact important lors de l'exécution du traitant pour le compte d'une des branches.

La règle de propagation automatique de l'exception système *failure* a également été modifiée. Dans le cas où l'exception *failure* n'est pas traitée, le *guardian* dans lequel l'exécution a lieu est détruit (*crash*). De manière générale, l'arrêt de l'exécution d'un point d'entrée du *guardian* par une exception amène la destruction du *guardian*. Ce traitant par défaut garantit de la même manière que celui de CLU qu'une exception, une fois détectée, ne peut pas être involontairement oubliée. Nous verrons plus loin que la destruction du *guardian* n'est pas définitive, mais qu'elle joue un rôle important dans le mécanisme de restauration offert par Argus.

L'instruction de parallélisme offerte par Argus est le **coenter**. Il est synchrone, c'est-à-dire que l'exécution de la mère ne reprend que lorsque les filles ont terminé. La terminaison normale de ce bloc parallèle résulte de la terminaison normale de chacune des branches. Mais elle peut aussi résulter de la terminaison d'une branche particulière qui le désire. Par contre elle ne peut pas être exprimée en fonction d'une combinaison de la terminaison de certaines branches. La terminaison en exception d'une branche amène automatiquement la terminaison du bloc entier, donc de toutes les autres branches, avant de permettre à la mère de traiter l'exception. On peut donc exprimer la condition : "dès qu'une branche termine correctement je veux terminer le bloc", mais on ne peut pas dire : "tant qu'une branche peut encore terminer correctement je veux laisser le bloc continuer de s'exécuter". En fait pour réaliser cela on est obligé de faire comme si les branches qui terminent incorrectement sont en fait correctes vis-à-vis du bloc, ce qui impose un test à l'issue du bloc pour savoir si effectivement une des branches a terminé correctement. La partie exceptionnelle de l'algorithme n'est pas différenciée.

Enfin Argus propose un mécanisme de restauration bien particulier, entièrement dépendant de la structuration de son modèle autour du *guardian*. Ce dernier est la seule entité permanente. Il possède un état et un certain nombre de points d'entrée qui permettent sa manipulation. Il est mis en œuvre par un serveur qui veille pour répondre aux demandes d'exécution de ces points d'entrée. Lorsque le *guardian* est détruit, le système le relance en réinitialisant ses variables d'état en fonction des dernières actions validées, puis en exécutant un code de restauration fourni par le programmeur du *guardian*. Argus reconnaît donc le besoin d'un mécanisme de restauration spécifique, distinct du mécanisme

de gestion d'exceptions, que nous avons demandé en section III.5. Mais les concepteurs du langage ont choisi de ne pas fournir ce mécanisme au niveau de l'appel de méthode, mais uniquement au niveau du *guardian*, entité de grain bien supérieur.

IV.1.5 Nil

Les exceptions dans Nil sont très proches du modèle de Trellis-Owl. La sémantique de l'association est inchangée, le signalement de l'exception est contrôlé. La règle de propagation du traitant par défaut est toutefois légèrement modifiée. D'autre part, les auteurs ont accordé une importance toute particulière à la destruction d'une activité, en créant une exception aux vertus spécifiques.

Le traitant par défaut de Trellis propage en dernier recours l'exception système *Failure*. Dans Nil, on regarde tout d'abord si l'exception n'est pas déclarée dans la méthode appelante. Si c'est le cas, alors elle est propagée telle quelle. Sinon, l'exception système *ERROR* est signalée. Le traitant décrit en section IV.1.2 est donc toujours valide, et garantit la fonction première d'un SGE, mais il est caché par (il est moins prioritaire que) des traitants sur le modèle suivant, un par exception déclarée (on prend la syntaxe de Nil).

```
on (<exceptionId>) return exception (<exceptionId>);
```

On note au passage que dans Nil on ne dispose pas de paramètre supplémentaire au signalement. On doit se contenter des paramètres de sortie normaux de la procédure, définis dans le message de retour, qui est renvoyé en même temps qu'une éventuelle exception est signalée. L'exception éventuelle en Nil qualifie l'instruction de retour.

```
return <message> [exception (<exceptionId>)];
```

Venons en maintenant à l'exception *CANCEL*. Dans certains cas, liés au parallélisme, le système veut stopper l'exécution d'une activité. Il le fait d'une manière douce, en forçant le signalement de l'exception *CANCEL* dans l'activité à stopper. L'originalité de Nil est que le système garantit que cette exception sera signalée dans un temps fini, même en cas de deadlock, et que l'exécution qui s'ensuit est contrôlée pour éviter les boucles infinies et autres cas de blocage. Pour ce faire, les points suivants sont respectés :

- la seule politique autorisée par le système dans un traitant de l'exception *CANCEL* est la propagation de l'exception *CANCEL*,
- les opérations exécutées après le signalement de *CANCEL* et non habituellement limitées en temps (boucles, blocages de synchronisation) sont interrompues au bout d'un temps fini,
- les appels inter-programmes sont interdits,
- les opérations de restauration à l'état non initialisé de l'objet exécutées par le système sont assurées de terminer.

Ces contraintes permettent d'effectuer à chaque niveau d'imbrication les opérations de restauration nécessaires aussi bien du point de vue du système que de celui du programmeur.

IV.2 Langages dérivés de Lisp

La particularité de Lisp est la chute de la barrière qui existe dans les langages procéduraux entre données et code. Le code est aisément manipulable, transformable, exécutable dans un environnement dynamique ; il peut être créé dynamiquement et passé en paramètre de fonction. C'est peut-être pour cette raison que les SGE dans cet environnement proposent la politique de reprise (voir section III.4).

Lisp possède un nombre important de dialectes, et certains se sont orientés vers le formalisme objet. Nous en étudierons deux, Common Lisp, qui possède une extension orientée-objet CLOS, et Lore, développé aux Laboratoires de Marcoussis.

IV.2.1 Common Lisp

Common Lisp n'est pas à proprement parler un langage à objets ; il en existe une extension, CLOS, qui est, elle, orientée objet. Common Lisp propose un mécanisme d'exceptions en cours de normalisation. C'est celui-ci que nous allons étudier, sachant qu'il existe déjà certaines notions d'un langage à objets dans Common Lisp, comme les types (qui définissent un format minimal de données et qui donnent naissance aux classes de CLOS), le sous-typage, l'instanciation d'un type. D'autre part, comme des données peuvent être évaluées, cela peut être utilisé pour définir du code dépendant des données ce qui est une première approche des méthodes d'un langage à objets. Savoir comment ce mécanisme d'exceptions sera intégré à CLOS est une autre question.

IV.2.1.1 Exceptions typées

La première caractéristique du mécanisme est que l'exception est une structure typée, et les types d'exception sont hiérarchisés. Il existe un type racine, *condition*, qui possède un certain nombre de champs comme *parent-type*, qui permet de générer la hiérarchie, *report-function*, qui permet de définir une fonction d'impression dépendant de la structure de l'exception, ou *make-function*, qui permet de définir un constructeur spécifique. Comme Common Lisp n'est pas réellement un langage orienté-objets, il ne bénéficie pas des avantages liés à l'héritage des méthodes et décrits en section III.6.4. Par contre la possibilité qu'il offre de spécialiser un type d'exception permet la factorisation à l'association décrite à la fin de cette même section.

Le signalement est également un peu différent de ce que nous avons développé en sections III.6.1 et III.6.3. Nous avons dit en effet que le signalement est une opération sur la classe, qui en génère une instance et recherche le traitant à exécuter. La fonction **error** de signalement de Common Lisp accepte aussi bien une "classe" qu'une "instance" d'exception. Cela autorise un traitant à propager l'exception qu'il traite, mais cela permet aussi, même si l'utilité n'en est pas très claire, de signaler une exception créée précédemment et conservée dans une variable. Dans le premier cas, il est clair que le principe d'encapsulation des langages objets est violé, puisque le traitant (niveau 1) rend visible à l'appelant (niveau 0) l'exception qui provient d'une opération qu'il appelle (niveau 2).

L'exception signalée est passée en paramètre au traitant, qui la déclare formellement comme paramètre de la fonction anonyme qu'il est. L'utilisation des champs de l'exception permet donc le passage de paramètres quelconques au signalement, comme il est dit en section III.6.2.

Enfin le langage propose différents types de signalement. Une première différence apparaît dans le cas où l'exception signalée n'est pas traitée. Si elle est signalée par **error** ou **cerror**, alors on entre dans une boucle interactive de rattrapage de l'exception (voir section IV.2.1.4). Si elle est signalée par **signal** ou **warn**, alors on applique la politique de reprise, c'est-à-dire que l'exécution reprend juste après le point de signalement. L'instruction de signalement se comporte alors comme un appel de procédure vide. On se rapproche là des exceptions de contrôle de la section II.3.4, qui sont peut-être agréables mais certainement pas nécessaires. D'autre part, **warn** et **cerror** définissent des points de reprise particuliers auxquels le traitant peut se dérouter (voir section IV.2.1.3).

IV.2.1.2 Association et traitement

Le mécanisme d'association présente l'avantage de ne pas relier la syntaxe d'association à la sémantique du traitant, avantage que nous avons appuyé en section III.2.3. Mais les moyens employés pour ce faire sont différents. On se rappelle que ce lien peut apparaître lors de l'application de la politique de réessai, ou de celle de terminaison sans déroutement local explicite. Dans ce dernier cas, après l'exécution du traitant, l'exécution reprend après l'entité à laquelle le traitant est syntaxiquement associé. La solution que nous proposons est de reprendre l'exécution juste après l'invocation signalante. Celle proposée dans Common Lisp est d'imposer en fin du traitant l'exécution d'un déroutement, local ou non, comme **return**, **go**, ou **throw**. Cette dernière solution garantit bien évidemment l'indépendance entre syntaxe et sémantique de l'association, mais au prix d'une perte importante de précision dans la définition des traitants. En effet, pour atteindre le même objectif que le mécanisme que nous proposons, il faudrait établir des étiquettes à chaque point de reprise éventuel, pour pouvoir s'y dérouter en fin de traitant, sans compter que la sémantique d'un tel déroutement, syntaxiquement placé en dehors du bloc de définition de l'étiquette, peut être difficile à établir. Quant à la politique de réessai, elle est tout simplement refusée. En fait nous verrons en section IV.2.1.3 que les mécanismes de reprise proposés ne respectent pas le principe d'indépendance entre syntaxe d'association et sémantique du traitement, et qu'ils se ramènent dans certains cas à la politique de terminaison des langages modulaires.

Le mécanisme de recherche conserve la propriété de prendre en compte tous les traitants comme dans Ada (voir section IV.1.1.1), sans limitation du niveau d'imbrication des appels. Le mécanisme va même plus loin en autorisant un traitant à propager l'exception qu'il traite après qu'il a exécuté une partie de traitement. Cette propagation de la même exception (voir section précédente IV.2.1.1) est d'ailleurs le traitement par défaut appliqué lorsque le traitant ne se termine pas par une instruction de déroutement.

Notons en passant, avant d'aborder le point important de la politique de reprise, qu'il est possible en Common Lisp d'associer un traitant à l'exception *NO-ERROR*, traitant qui est invoqué si le bloc associé termine sans signalement d'exception. L'utilité en paraît douteuse, d'autant plus que cela rend en quelque sorte exceptionnelle la terminaison normale du bloc !

Nous allons maintenant étudier le mécanisme de reprise proposé par Common Lisp.

IV.2.1.3 Reprise

On se rappelle que la politique de reprise "classique" amène à reprendre l'exécution après le point de signalement. Comme nous l'avons dit en section III.4.1, cette politique relève

plus d'un comportement de mise au point, et est dans ce cas plutôt limitée puisque le point de reprise devrait se situer généralement entre le point de signalement et le point de traitement. Les auteurs de Common Lisp ont poussé cette remarque assez loin, et proposent un mécanisme qui autorise la définition de points de reprise variés, en faisant la distinction entre ceux utilisables par programmation, et ceux restreints à l'outil interactif de mise au point.

L'astuce consiste à invoquer dans le traitant une fonction qui est définie dans le contexte du signalement. On exploite complètement la possibilité offerte par Lisp de modifier dynamiquement le code d'une fonction ou, ce qui revient au même, d'exécuter du code contenu dans une variable. Le mécanisme utilisé est décrit dans l'exemple suivant. *Entité-traitante* protège l'exécution de *entité-de-reprise* par un traitant associé à l'exception *exception*, grâce au mot-clef **handler-case**. *Entité-de-reprise* définit ensuite grâce au mot-clef **restart-case** un certain nombre de points de reprise, dont en particulier celui nommé *point-de-reprise* , et qui sont valides durant l'exécution de *entité-signalante* . Lorsque *entité-signalante* signale l'exception *exception*, le traitant défini par *entité-traitante* a accès à l'ensemble des points de reprise dynamiquement définis au moment du signalement, et il peut invoquer *point-de-reprise* .

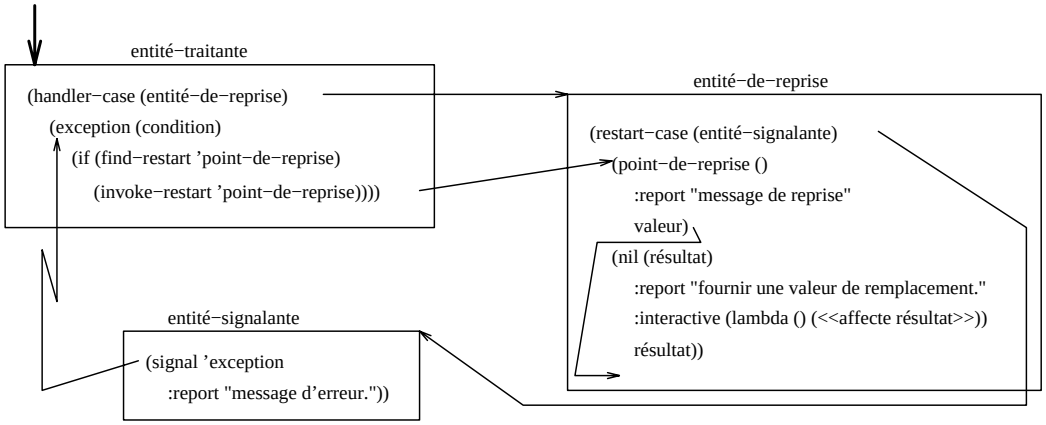

Fig. 4.1 : reprise en Common Lisp

Le grand intérêt de cette technique est que les points de reprise sont explicitement fournis par le programmeur, et qu'il peut ainsi les contrôler. Par contre, plusieurs points sont discutables. Le point de reprise n'est pas circonscrit au signalant. Il n'est même pas limité à l'intervalle entre point de signalement et point de traitement, au cas ou on accepterait de traiter l'exception par un appelant indirect. L'instruction **invoke-restart** devient alors une primitive de déroutement supplémentaire, difficilement contrôlable. D'autre part le point de reprise est l'instruction suivant le bloc **restart-case**, bloc qui définit par ailleurs la portée du point de reprise. La sémantique du traitement est donc de nouveau liée à la syntaxe de déclaration du bloc de reprise, ce qui est gênant en particulier pour fournir la reprise classique juste après le point de signalement. Pour réaliser cela on est obligé d'invoquer le signalement depuis un bloc de reprise, ce qui est réalisé pour les deux primitives **cerror** et **warn** qui définissent respectivement les reprises *continue* et *muffle-warning*.

Fig. 4.2 : reprises prédéfinies

IV.2.1.4 Interactivité

Le deuxième aspect de la reprise en Common Lisp est la possibilité de définir des points de reprise accessibles uniquement via un outil interactif de mise au point, celui-là même appelé par défaut lors d'un signalement par **error** non rattrapé. Ces points sont simplement non nommés (introduits par **nil**), comme celui qui suit *point-de-reprise* dans la figure Fig. 4.1. Lorsque cet outil est invoqué, un message décrivant l'exception et tiré du champ *report* de l'exception est imprimé, puis la liste des points de reprise possibles est affichée, chaque point étant numéroté et décrit par le champ *report* donné lors de sa déclaration. Ainsi l'exécution du code de la figure Fig. 4.1 pourrait résulter en la proposition suivante :

```
Signal : message d'erreur.
pour continuer, entrer :continue suivi d'un numéro de choix:
    1: fournir une valeur de remplacement.
    2: message de reprise.
    3: revenir au Toplevel Lisp.
debug>
```

IV.2.2 Lore

Le langage Lore est développé aux Laboratoires de Marcoussis. C'est un langage à objets issu de Lisp. Il propose la notion de classe, génératrice d'instances, et qui définit une structure de données avec des méthodes associées, mais conserve d'autres notions de Lisp comme les ensembles, les variables globales et les fonctions anonymes. Une différence marquante entre ce langage et ceux plus classiques abordés en section IV.1 est que tout est objet, y compris les classes, les méthodes, les relations. Ainsi on peut invoquer *has method* sur un objet classe pour lui ajouter une nouvelle méthode, on peut modifier la méthode *herit* définie sur les propriétés pour redéfinir les règles d'héritage, on peut invoquer *of?* sur une relation pour obtenir l'ensemble des objets qui la vérifient. Ces caractéristiques ont permis la définition d'un mécanisme de gestion des exceptions très intéressant, qui utilise complètement le modèle d'objets [26]. Lore a été mis en œuvre une première fois sur Common Lisp, puis sur Smalltalk.

IV.2.2.1 Modèle général

Comme Common Lisp, Lore définit une structure hiérarchique de classes d'exception. Il existe une classe racine *Exceptional-event* , qui factorise la déclaration des champs *xobj*, *xprop*, *xargs*, et *protocol-for-resumption* . Les champs *xobj* et *xprop* décrivent

l'environnement d'exécution au moment du signalement, i.e. objet et méthode actifs. Le champ *xargs* permet le passage de paramètres quelconques.

Créer un nouveau type d'exception implique créer une classe, c'est-à-dire instancier la méta-classe *Exception-class*. Cette méta-classe définit la méthode **signal**, qui peut alors être invoquée sur un objet classe d'exception pour signaler une exception de cette classe. La méthode **signal** est à rapprocher de la méthode **new** définie sur la méta-classe *Class*, super-classe de *Exception-class*. Ce schéma respecte la présentation de la section III.6.3.

Par contre, peu a été fait pour le respect du principe d'encapsulation de l'appel de méthode, principe que nous avons admis en section III.2. En effet, si une méthode a la possibilité de déclarer, grâce aux mots clefs **signals** et **propagates**, les exceptions qu'elle est susceptible de signaler, cela n'a qu'une valeur informative et n'a aucune conséquence sur le signalement d'une exception non déclarée. D'autre part la recherche du traitant n'est pas restreinte au strict appelant, mais s'étend comme dans Common Lisp depuis le point de signalement jusqu'au premier "toplevel" où un traitant par défaut est défini. La possibilité de propager depuis un traitant l'exception même en cours de traitement est également conservée. La seule structuration notable par rapport à Common Lisp est le rattachement de la reprise au point de signalement, qui redonne à ce traitement la sémantique d'une réponse au signalement et non pas celle d'une instruction de déroutement supplémentaire.

Les traitants peuvent user de tous les traitements classiques, depuis la propagation jusqu'à la reprise, en passant par la terminaison et le réessai. Ils peuvent être associés à une expression, grâce au mot clef **protect**, ou à une classe, grâce au mot clef **set-protect**. Malheureusement la syntaxe de l'association influe sur la sémantique du traitement, dans les cas de la terminaison et du réessai, et nous en avons déjà vu les conséquences sur la possible définition de traitants généraux.

Les deux types d'association ont deux sémantiques différentes. Un traitant associé à une expression protège classiquement toutes les opérations invoquées pendant l'exécution de l'expression. Un traitant associé à une classe protège toutes les invocations de méthode sur un objet de cette classe. Il faut bien voir que le deuxième cas n'est pas la factorisation du premier pour toutes les méthodes de la classe, comme nous le proposons en section III.3.1. Dans le second cas (Lore) l'entité traitante est l'appelant de l'objet de la classe considérée, dans le premier c'est l'objet lui-même. On peut voir la différence dans l'exemple suivant, où on transpose la situation à un langage acceptant l'instruction **return** (politique de terminaison avec déroutement) dans un traitant. Dans un cas le **return** amène la reprise de l'exécution à l'instruction *<suite>*, alors que dans l'autre il provoque l'abandon de l'appel *appel2*.

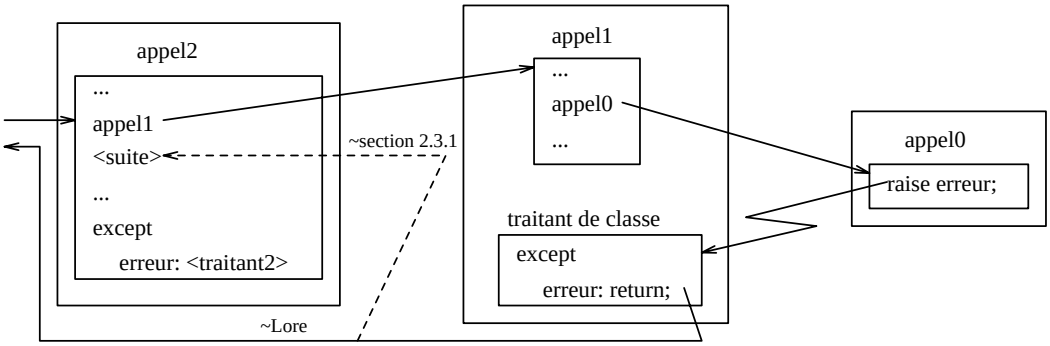

Fig. 4.3 : traitant de classe

D'autre part vu depuis l'appelant, ici *appel2* , ce traitant est prioritaire par rapport à tout traitant *<traitant2>* qu'il peut définir. Si la recherche de traitant était limitée au strict appelant, un tel traitant serait lié statiquement au point de signalement.

Tous les traitements sont des méthodes définies sur des classes d'exception. Ainsi la terminaison, méthode **exit**, est proposée par la classe *Fatal-event* , et la reprise, méthode **resume**, par la classe *Proceedable-event* . De ces deux classes le système génère par héritage simple les classe *Error* et *Warning*, et par héritage multiple la classe *Exception*. Ainsi on ne peut pas reprendre après le signalement d'une erreur, comme on ne peut pas terminer lors de la détection d'un avertissement. La méthode **retry**, considérée comme une variation de la terminaison, est également définie sur la classe *Fatal-event* . Enfin la propagation inchangée d'une exception est disponible via la méthode **instance-signal** définie sur la classe *Exceptional-event* .

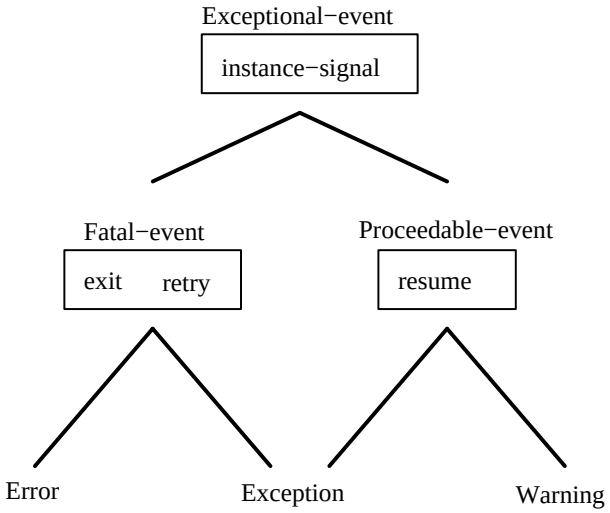

Fig. 4.4 : hiérarchie d'exceptions

Le signalant participe en quelque sorte au traitement, en indiquant ceux qui sont applicables de par le choix de l'exception qu'il signale. Cette participation est encore plus importante

dans le cas de la reprise, comme nous le verrons dans la section suivante. Toutes ces méthodes de traitement sont invoquées par le traitant sur l'exception grâce à la variable prédéfinie **xself**, qui référence dans le traitant l'exception en cours de traitement.

IV.2.2.2 Reprise

La reprise est provoquée par l'invocation de **xself**.**resume** suivie de la valeur qui remplace l'expression de signalement. Mais cette politique peut également être proposée par une autre méthode de l'exception, qui se termine en invoquant **resume** sur elle-même. La particularité de la proposition de Lore est que le signalant peut, grâce au champ *protocol-for-resumption*, n'autoriser l'invocation par le traitant que d'une partie des méthodes qui proposent la reprise. Il y a en quelque sorte restriction dynamique de l'interface de l'exception. Mais par ce moyen les différents types de reprise sont banalisés du point de vue du signalant. Comme celui-ci doit certainement traiter différemment les cas, il faut mettre en place un protocole supplémentaire entre signalant et traitant. Ainsi le premier champ du paramètre retourné par le traitant peut servir d'identificateur de la politique de reprise employée, comme le montre l'exemple ci-dessous tiré de [26]. Lorsque la méthode *selector* n'est pas trouvée sur l'objet *receiver*, alors l'exception *Unknown-selector* est signalée. Ce signalement peut retourner en cas d'application par le traitant d'une des trois méthodes de reprise *new-selector*, *new-receiver*, ou **resume**, définies sur cette classe d'exception. Le premier champ du paramètre de retour est ensuite testé, et la continuation appropriée est choisie.

```
(defun ask (receiver selector args)
    ; recherche de la méthode selector
    (let ((property (search-property selector [receiver type])))
    (if property
        ; elle est trouvée; on l'invoque
        (apply-property property receiver args)
        ; elle n'est pas trouvée; on signale Unknown-selector
        (let ((signal-res (Unknown-selector signal
                ; en fixant les différents champs de l'exception
                wmr receiver
                wms selector
                wma args
                protocol-for-resumption
                    '(new-selector new-receiver resume))))
        ; puis on teste le retour en cas de reprise
        (if (atom signal-res)
            signal-res
            ; en particulier le premier champ du code retour
            ; qui détermine la politique choisie, soit
            (case (car signal-res)
                ; new-selector
                (new-selector (ask receiver
                        (cadr signal-res) args))
                ; new-receiver
                (new-receiver (ask (cadr signal-res)
                        selector args))
                ; resume
                (t signal-res)))))))
```

IV.2.2.3 Traitants généraux

Lorsqu'aucun traitant dynamique ne le cache, le traitant associé au "toplevel" est exécuté, qui invoque un traitant général. Trois types de traitants généraux peuvent être définis, que nous allons décrire par ordre de priorité. Le premier est la méthode de nom prédéfini *handles-default* définie sur la classe de l'objet responsable du signalement. Cette méthode a également accès à la variable **xself** qui référence l'exception courante, ce que nous trouvons incohérent car elle peut être a priori appelée en dehors d'un traitant. En effet cette méthode n'est pas un traitant ordinaire puisqu'elle n'est pas introduite par un des deux mots clefs **protect** et **set-protect**. Il s'agit plutôt d'une proposition de traitement, comme nous l'avons dit en section III.3.3. Toujours est-il que grâce à cette variable la méthode est capable d'invoquer directement les différentes politiques de traitement proposées sur l'exception. Il faut donc être prudent quant à son invocation en dehors d'un traitant.

Cette méthode *handles-default* peut être héritée d'une super-classe, et en dernier recours de la classe racine, où elle invoque le second type de traitant par défaut. Il s'agit cette fois de la méthode *handles-default* définie sur la classe de l'exception signalée. Celle-ci peut appliquer toute politique de traitement en usant de la variable prédéfinie **oself** référençant l'objet courant.

De la même manière que précédemment cette méthode peut être héritée d'une super-classe, et en dernier recours de la classe *Exceptional-event* , où elle invoque un outil interactif de l'inspiration de celui proposé pour Common Lisp (voir section IV.2.1.4).

IV.2.2.4 Finalisation

De la même manière qu'en Modula-3 (voir section IV.1.3.1) seul un outil de finalisation est offert dans Lore. La restauration en tant que telle n'est pas traitée, même si bien sûr les deux fonctions sont assez proches. Un bloc **when-exit** peut être associé à toute expression, et est exécuté quand on quitte la portée de cette expression. Cet outil a la particularité d'être dans un environnement où les deux politiques de terminaison et de reprise sont autorisées. Cela veut dire en particulier que si une exception signalée dans la portée de l'expression associée au bloc **when-exit** est traitée par un traitant extérieur à cette portée, alors le bloc ne doit être exécuté que si la politique de terminaison est choisie. Rien ne doit être fait si l'exécution reprend au point de signalement.

IV.3 Propositions originales

Nous allons maintenant présenter quatre propositions qui n'ont pas trouvé leur place dans les deux grands groupes précédents.

IV.3.1 Yemini/Berry

Le mécanisme qu'ils proposent dans [78] et [79] a été mis en œuvre sur Algol 68 mais ne fait véritablement partie d'aucun langage. Le principe directeur de leur conception est le respect de l'encapsulation d'une entité de base ; les conséquences qui en découlent sont similaires aux conclusions de la section III.2. Seules les exceptions préalablement déclarées par une entité peuvent être signalées ; l'entité responsable du traitement est celle strictement appelante ; la syntaxe d'association du traitant n'influe pas sur la sémantique du

traitement ; on peut fournir une valeur de remplacement dans le cas où l'entité signalante doit rendre une valeur. La différence est que, contrairement à un modèle objet, l'entité n'est pas l'appel de méthode mais le bloc d'instructions.

D'autre part les auteurs ont complètement formalisé le passage de paramètres de signalement et de reprise, puisque cette dernière politique est offerte. Cela amène à considérer une exception non plus comme une simple chaîne de caractères, mais comme une opération avec des paramètres d'entrée et de sortie. La déclaration de son interface est faite au début de la portée de l'entité signalante, et est dupliquée avec la définition de paramètres formels lors de la définition du traitant. Les auteurs ont également voulu prendre en compte dans cette interface le type de la valeur de remplacement dans le cas de la politique de terminaison sur une expression. Cela paraît curieux car ce type est connu de l'appelant, c'est celui de l'expression signalante. Toujours est-il que cela permet l'écriture du programme suivant, tiré de [78], qui donne la chaîne de caractères correspondant à un tableau de codes numériques.

```
proc convert=(ref []int code) string
    signals(exc(int)(char, string) badcode)
begin
    string s := "";
    for i from lwb code to upb code do
        int code_i = code[i];
        s := s +
            if code_i <= char_hi and code_i >= char_lo then
                << caractère correspondant à code_i >>
            else
                badcode(i)
                << signalement de l'exception >>
            fi
    od;
    s
end
```

La procédure *convert* déclare une exception *badcode*, qui accepte en paramètre de signalement un entier, l'index du mauvais code dans le tableau, et qui déclare la reprise possible pourvu que le traitant fournisse une valeur de type **char** utilisable en remplacement de la valeur attendue par l'algorithme. Dans ce cas le signalement prend la forme d'un appel procédural classique, comme la syntaxe le montre. Enfin le dernier type déclaré dans la clause **signals** est **string**, c'est le type de la valeur que doit fournir le traitant s'il veut remplacer la valeur attendue du signalant, c'est-à-dire la valeur de retour de *convert*. Le traitant demande la reprise simplement en fournissant une valeur du type attendu, ou demande la terminaison en fournissant la valeur de remplacement suivie de **replace**.

```
on badcode=(int i)(char, string):
    "?"
no
on badcode=(int i)(char, string):
    "" replace
no
```

IV.3.2 Smalltalk/80

Le langage offre à partir de la version 2.5 un mécanisme de gestion des exceptions très sophistiqué. Il se rapproche de celui de Lore, en ce sens que les exceptions sont structurées en hiérarchie d'objets typés. Il existe une méta-classe *Signal*, qui est l'équivalent de *Exception-class* de Lore, et qui définit entre autres une méthode **raise** de signalement. Il existe également une classe racine des exceptions, *genericSignal*, qui a pour équivalent *Exceptional-event* en Lore. Le signalement correspond à l'envoi d'un message à une classe d'exception, qui en crée une instance dans laquelle le contexte de signalement est stocké, puis qui recherche et exécute le traitant approprié.

La recherche du traitant utilise la pile entière en partant normalement de l'appelant du signalant. D'ailleurs les exceptions signalables par une méthode ne sont pas déclarées, et pour cause au vu du programme de la figure Fig. 4.5. D'autre part le problème de l'universalisation des exceptions noté en section III.6.1 trouve ici un début de réponse intéressant. En effet pratiquement aucune des classes d'exception système n'a de nom, mais ces classes sont référençables par le biais d'un message envoyé à une autre classe. Ainsi l'exception générale qui est la super-classe conseillée des exceptions utilisateur est accessible via le message *errorSignal* de la classe *Object* ; en hérite la classe des exceptions arithmétiques accessible par le même message sur *ArithmeticValue* , et qui est à son tour spécialisée en une classe référencée par le message *divisionByZeroSignal* sur *ArithmeticValue* . Cela semble rattacher un type d'exception au type d'objet qui la signale, mais il n'en est rien, l'exception peut être signalée par n'importe qui. Un détail est lié à ce fait, c'est la spécialisation possible d'une exception en une sous-classe "rattachée" à une classe sans relation particulière avec la classe de rattachement de la première exception. Ainsi *errorSignal* de *Objet* est la spécialisation de *genericSignal* de *Signal*.

Toutes les politiques de traitement déjà décrites sont autorisées, et même d'autres comme nous allons le voir. Elles sont proposées comme des méthodes de l'exception. La terminaison est amenée par l'invocation de **return** sur l'exception, l'option réessai par **restart**, la reprise par **proceed**. Pour ce qui est de la terminaison et du réessai, la syntaxe d'association du traitant influe sur la sémantique du traitement ; c'est le bloc auquel le traitant est associé qui est concerné. La propagation peut se faire classiquement par resignalement, mais un traitant peut également forcer la poursuite de la recherche de traitant pour la même exception, en invoquant **reject**, qui est à rapprocher de **instance-signal** de Lore. La nouveauté réside dans la possibilité qu'a le traitant de demander l'exécution d'un bloc de code après l'application de la politique choisie. En effet dans Smalltalk il est possible d'encapsuler du code dans un objet de classe *Block*, que l'on peut passer en paramètre, et dont on peut ensuite demander l'évaluation. Cette possibilité est utilisée par la méthode **proceedDoing**, qui force l'évaluation par le signalant et dans le contexte de signalement, de code fourni par le traitant. Le schéma d'exécution suivant est donc possible, où le traitant de l'exception signalée par *signalant* demande le resignalement par ce dernier d'une autre exception *exception*, qui est cette fois traitée par *<<traitant>>* .

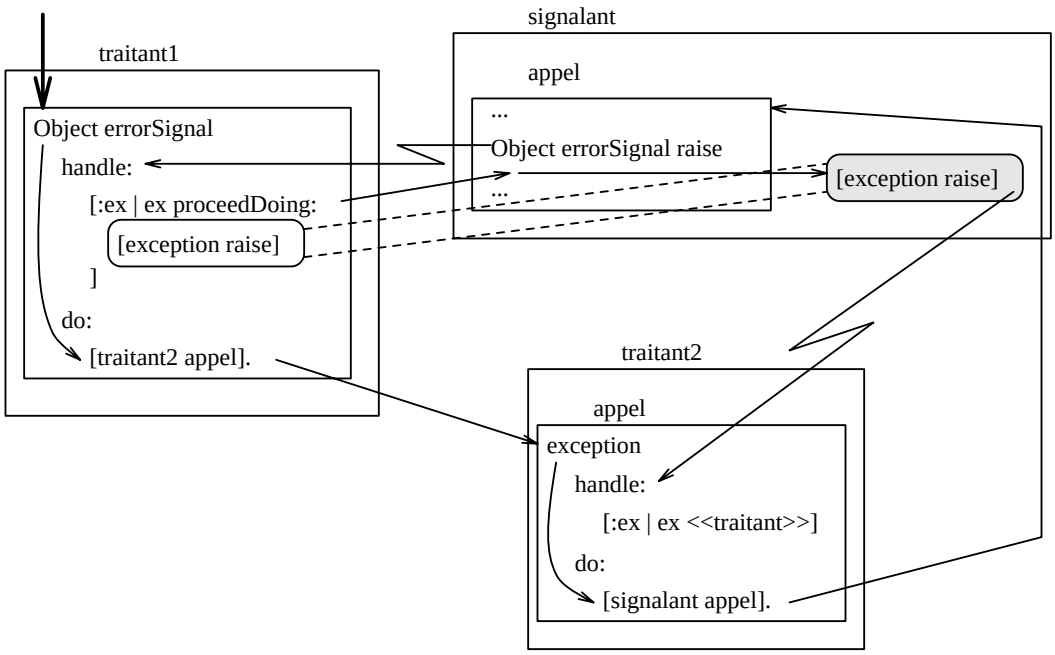

Fig. 4.5 : traitement additionnel après la reprise

Une classe d'exception peut interdire l'application de la politique de reprise en traitement de toutes ses instances. Smalltalk offre ainsi l'équivalent des classes *Error* et *Exception* de Lore. De plus, le signalant peut préciser au traitant s'il est prêt ou non à accepter la politique de reprise, en utilisant au signalement une des deux primitives **raiseRequest** ou **raise**. C'est le pendant d'un des objectifs du champ *protocol-for-resumption* de Lore.

Smalltalk offre un mécanisme de finalisation, qui associe un bloc de finalisation à un bloc corps grâce au mot clef **valueNowOrOnUnwindDo**. Ce bloc est exécuté après le corps quelle que soit l'issue, exceptionnelle ou non, de ce dernier. Ce mécanisme présente une variante curieuse, qui utilise le mot clef **valueOnUnwindDo**, et qui n'exécute le bloc de finalisation que lorsqu'une exception est signalée, ou que l'on sort du bloc par l'équivalent Smalltalk de **return**. Rappelons que nous pensons que le bloc de restauration ne devrait être exécuté qu'en cas de sortie exceptionnelle, et que **return** amène une sortie normale de la méthode.

Enfin notons l'utilisation d'une exception, *terminateSignal* sur *Process*, utilisée par le système pour stopper en douceur une activité. Celle-ci a ainsi le temps d'exécuter d'éventuels blocs de finalisation avant de sortir.

IV.3.3 C++

L'exception en C++ est un objet typé, mais bien différent de ce qui est décrit en section III.6, ou de ce qui est offert dans Lore. En effet, tout les mécanismes de sigalement ou de traitement ne sont pas des méthodes de l'objet exception. En fait une exception dans C++ est essentiellement un ensemble de champs. Cela se voit particulièrement dans le fait qu'il

n'existe pas de classe racine des exceptions. N'importe quelle classe peut être utilisée comme une exception, comme c'est le cas de la chaîne de caractères.

La recherche du traitant n'est pas limitée au strict appelant, et se satisfait d'un traitant qui déclare en unique paramètre une super-classe de l'exception signalée, ou un type dans lequel l'exception peut être convertie de manière standard. C'est le cas dans l'exemple suivant, où l'exception *tableau*, de type tableau de *T*, est acceptée par le traitant qui prend en paramètre *exc*, un pointeur vers un *T*.

```
try {
    signalant();
}
catch (T *exc) {
    <<traitant>>
}
```

```
void signalant()
{
    T tableau[max];

    throw tableau;
}
```

De manière similaire à Modula-3, C++ n'oblige pas ses programmeurs à déclarer les exceptions qu'une fonction peut signaler. Ceux-ci ont la possibilité de le faire grâce à une clause **throw** décrite avant le corps de la fonction. Quand cette clause existe, la fonction ne peut pas signaler une exception qui n'y apparaît pas. Si le cas survient, le système invoque la fonction *unexpected()* (voir plus loin). Quand la clause **throw** n'existe pas, la fonction peut signaler n'importe quelle exception. Le peu de cas fait de cette clause est apparent dans sa non prise en compte dans la définition du type de la fonction. Cela veut dire en particulier qu'une fonction virtuelle qui déclare une clause **throw** vide peut être surchargée par une fonction sans clause **throw**, ce qui est en complète contradiction avec la section III.2.1.

Les traitements proposés se résument à la terminaison, dont la sémantique est liée à la syntaxe d'association du traitant, et à la propagation par resignalement. Le signalement sans paramètre permet de resignaler l'exception en cours de traitement. Le langage permet d'ailleurs curieusement l'emploi de cette instruction en dehors d'un traitant, depuis une fonction appelable depuis un traitant.

La fonction *unexpected()* invoquée lors du signalement d'une exception interdite est modifiable par le programmeur grâce à la primitive *set_unexpected()* . Elle invoque par défaut la fonction *terminate()* , elle-même modifiable par le programmeur, et qui invoque par défaut *abort()*. Une politique par défaut est donc programmable, mais cela est lié au fait que la propagation des exceptions est peu contrôlée. On peut le voir dans l'exemple suivant, où on simule la politique par défaut de Trellis qui est la propagation de l'exception prédéfinie *Failure*.

```
void Trellis_default_handling() { throw Failure; }
set_unexpected(&Trellis_default_handling);
```

Enfin une ébauche de mécanisme de finalisation est fournie par l'intermédiaire des destructeurs de chacun des objets qui sortent de leur portée du fait du signalement. Ces destructeurs sont exécutés avant l'exécution du traitant, le cas du paramètre de signalement étant traité de manière particulière.

IV.3.4 Eiffel

Le langage a accepté comme principe directeur la programmation par contrat. Cela veut dire qu'une méthode garantit qu'elle fournit un service caractérisé par des postconditions, pourvu qu'elle soit invoquée avec certaines préconditions vérifiées. L'appelant est responsable de la vérification des préconditions. Ce principe n'est pas nouveau, il est directement issu du formalisme décrit en section II.1.1. L'originalité dans Eiffel est que préconditions et postconditions sont effectivement décrites dans un langage d'assertions, et qu'elles peuvent être vérifiées à l'exécution. Comme ce mécanisme coûte cher, il est activé suivant une option de compilation. Lorsqu'une telle assertion n'est pas vérifiée, une exception est signalée.

Le mécanisme de gestion des exceptions décrit dans [53] est très sobre. Signalement et traitement sont bien rattachés à l'appel de méthode, mais d'une manière très originale. En effet, le signalement d'une exception par un appelé conduit à l'abandon de l'algorithme de la méthode appelante qui est considéré avoir failli. Le seul traitement autorisé est le réessai de cette méthode appelante entière, ou bien sûr la propagation de l'exception. L'exemple suivant illustre le mécanisme. Le signalement de l'exception par *signalant* provoque l'abandon de l'algorithme de l'appelant et le déroutement dans la clause **rescue** de celui-ci. L'instruction **retry** de cette clause permet de reprendre l'exécution au début de la méthode appelante. Si l'instruction **retry** n'est pas présente à la fin de cette clause, l'exception est automatiquement propagée.

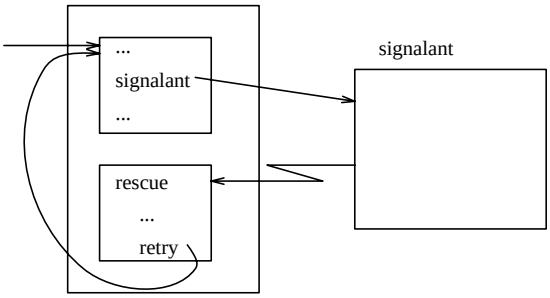

Fig. 4.6 : la clause **rescue** *d'Eiffel*

Ce mécanisme garantit bien sûr que le rattrapage d'une exception respecte l'algorithme principal, mais il limite très fortement les possibilités de traitement par l'appelant. En effet, la politique de traitement alternatif est refusée. Il est impossible proposer une séquence de code pour remplacer l'appel signalant ; la seule solution consiste à demander un **retry**, en ayant prévu dans le corps de la méthode un deuxième algorithme, et en usant de certaines propriétés spéciales d'initialisation. Ainsi la variable locale *essai* est-elle initialisée à 0 à l'entrée de la méthode, comme toute variable entière, mais n'est pas réinitialisée lors des éventuels passages suivants demandés par **retry**. Son incrémentation dans la clause **rescue** permet de contrôler le nombre de tentatives de réessai demandées.

```
méthode is
    require
        ...
    local
        essai: INTEGER
    do
        if essai = 0 then
            miseEnOeuvre1
        elsif essai = 1 then
            miseEnOeuvre2
        end
    ensure
        ...
    rescue
        essai := essai + 1;
        if essai < 2 then
            ...
            retry
        end
    end
```

L'inconvénient évident de ce mécanisme est que l'exception signalée par l'appelé amène automatiquement une sorte d'exception dans l'appelant, sans interprétation possible. Par contre il résoud complètement le problème de la restauration de la section III.5, ce pour quoi il a en fait été construit.

Le manuel de la version 2.2 [54] explicite un certain nombre de points, tout en conservant les principes généraux énoncés ci-dessus. On apprend en particulier que l'exception est une simple chaîne de caractères, accessible dans la clause **rescue**, ce qui permet un traitement différencié suivant l'exception. Le manuel précise d'autre part que l'utilisateur peut signaler ses propres exceptions, autres que des violations d'assertions, par l'instruction **raise**.

Chapitre V

Proposition pour Guide

Nous allons maintenant regarder comment les conclusions de la section III.7 ont été mises en pratique dans Guide. Nous commençons par une présentation du projet, contexte de travail, points principaux du langage. Puis nous passons en revue les différents aspects du mécanisme proposé pour Guide, en détaillant certains points particuliers comme les exceptions dans un traitant, l'association, la restauration, ou liés au langage comme le parallélisme, l'héritage ou les exceptions système.

V.1 Système et language Guide

V.1.1 Contexte du projet

Le langage Guide est développé au laboratoire Bull-IMAG à Grenoble, dans le cadre du projet Guide. Ce projet, lancé en 1986, avait pour but l'étude et la réalisation d'un prototype de système réparti qui servirait de base à l'écriture d'applications réparties. La première phase s'est terminée en 1990 et a vu la réalisation d'une maquette sur Unix. Dans le même temps un langage a été développé pour rendre compte plus parfaitement des nouvelles fonctions offertes au programmeur. La seconde phase du projet doit utiliser le micro-noyau Mach comme support. Le projet Guide participe au projet ESPRIT COMANDOS.

V.1.2 Caractéristiques du langage

Système et langage Guide sont construits sur une base objet. Le langage reprend les principes reconnus structurants des langages à objets, en choisissant comme Emerald de séparer explicitement mises en œuvre et interfaces. Un type Guide décrit l'ensemble des opérations possibles sur les objets de ce type. Il peut être mis en œuvre par une ou plusieurs classes, qui décrivent un format de données et donnent le code des opérations du type.

Les objets sont manipulés dans le langage par des références vers un type donné. Le type de l'objet (le type mis en œuvre par la classe de l'objet) n'est pas obligatoirement celui de la référence qui le manipule, mais il doit y être conforme. La relation de conformité syntaxique est essentiellement vérifiée statiquement à la compilation, pour tout ce qui est affectation et passage de paramètre.

Types et classes acceptent une relation d'héritage simple. Sous-typage et héritage respectent la règle de conformité.

Les objets sont tous persistants. Il est à la charge d'un ramasse-miettes de libérer la place occupée par ceux qui ne sont plus accessibles.

Enfin le langage offre une instruction de parallélisme permettant la création de plusieurs flots d'exécution parallèles, et qui se termine suivant une condition portant sur la terminaison des différentes branches.

Nous allons dans la suite décrire comment nous avons intégré un mécanisme d'exceptions à ce langage, en décrivant à chaque fois la syntaxe utilisée. Pour cela nous utiliserons les conventions suivantes, reprises du manuel du langage [57] :

- terminaux en gras, non terminaux entre <>
- (x) = un x, [x] = au plus un x, {x}* = au moins un x, {x} = zéro x ou plus
- | marque l'alternative

D'autre part nous utiliserons différentes parties d'un même exemple pour illustrer chaque aspect de notre proposition. Les types (interfaces) utilisés sont donnés ci-dessous. Ils sont introduits par le mot clef **type**. Ils décrivent un *Editeur* simple, qui parcourt un document *Page* par *Page*. Il cache les *Page*s qu'il a déjà lues dans des *PageTampon*s jusqu'à *Validation* . Il demande les *PageTampon*s dont il a besoin à un *Gestionnaire* de *PageTampon*s, et les lui rend en fin d'utilisation. La méthode *Nettoyage* libère les *PageTampon*s qu'il a gardées propres. Deux autres méthodes permettent de visualiser la *PageTampon* courante et d'en créer une nouvelle. Une *Page* est lue caractère par caractère, et une *PageTampon* est une *Page* avec un indicateur de "saleté". Un *Gestionnaire* fournit et récupère des objets avec les méthodes *Demande* et *Libère*. Le mot clef **class** introduit une mise en œuvre. Nous ne donnons ici que les variables d'état de la classe *Editeur*, les méthodes étant décrites au fur et à mesure de la description du mécanisme.

```
type Editeur is
    method Validation;
    method VoirPage;
    method CréerPage;
    method Nettoyage;
end Editeur.
```

```
type Page is
    method LitCaract: Char;
        signals erreur,
            fin_de_ligne,
            fin_de_page;
end Page.
```

```
type Document is
    pages: ref List
        of ref Page;
end Document.
```

```
type PageTampon
subtype of Page is
    sale: Boolean;
end PageTampon.
```

```
class Editeur implements Editeur is
    // variables d'état
    pagesTampon: ref List of ref PageTampon;
    gestionnairePages: ref Gestionnaire of ref PageTampon;
    pageCourante: ref PageTampon;
end Editeur.
```

V.2 Signalement d'exception

V.2.1 Déclaration

<liste signaux> = **signals** <ident exception> { **,** <ident exception> } **;**

<signature méthode> = **method** <ident méthode> [**(**<liste param>**)**]
[**:** <type retour>] **;** [<liste signaux>]

Comme précisé en section III.2.1, la méthode déclare dans sa signature les exceptions qu'elle peut signaler. Cette déclaration est faite dans un type Guide. Les exceptions en Guide sont de simples chaînes de caractères. Dans l'exemple, le type *Page* déclare que sa méthode *LitCaract* peut signaler les trois exceptions *fin_de_ligne* , *fin_de_page* , et *erreur*.

V.2.2 Signalement

<instruction raise> = **raise** <ident exception> **;**

Cette instruction permet de signaler l'exception <*ident exception*>. Le compilateur vérifie que cette exception est bien déclarée dans l'interface de la méthode qui la signale. Dans le cas contraire il signale une erreur.

Conformément à ce que nous avons dit en section III.2.2, le traitant de cette exception est recherché dans le strict appelant de la méthode signalante. Du point de vue du signalant cette instruction est donc semblable à un **return**.

L'instruction de signalement peut également se trouver dans le code d'un traitant. S'il s'agit d'un traitant associé à un bloc d'instructions ou à une méthode entière, le compilateur vérifie que cette méthode ou que celle qui comprend le bloc d'instructions a bien déclaré l'exception dans son interface. S'il s'agit d'un traitant de classe (voir section V.3.1), alors la vérification est plus complexe. En effet, un tel traitant peut être invoqué durant l'exécution d'une quelconque des méthodes de la classe où il est défini. Il faudrait donc que chacune de ces méthodes déclare l'exception. Or on peut être un peu plus fin en exploitant le masque du traitant (voir section V.3.2) qui précise la portée de l'exception. Le compilateur est alors capable, en analysant le code de chaque méthode, de déterminer si le traitant est susceptible d'être invoqué durant l'exécution de cette méthode. Si ce n'est pas le cas, alors la méthode n'est pas obligée de déclarer l'exception.

Le compilateur pourrait également vérifier la seconde étape du signalement, à savoir le rattrapage de l'exception. En effet, lorsqu'il analyse la méthode appelante il dispose de la signature des méthodes appelées, donc des exceptions signalables. Il pourrait alors vérifier que pour chaque exception signalable il existe un traitant prêt à la rattraper. Or imposer au programmeur de fournir un tel traitant peut se révéler très contraignant, et nous avons vu en section III.3.2 qu'un traitant système répond parfaitement au problème. Le compilateur ne doit donc pas signaler d'erreur lorsqu'il ne trouve pas de traitant explicite à une exception donnée.

V.3 Association de traitant

V.3.1 Déclaration

<bloc traitement> = **except** { <liste masques> **:** <traitant> }*

<bloc instructions> = **begin** <corps> [<bloc traitement>] **end;**

<instruction repeat> = **repeat** <corps> [<bloc traitement>] **until** <...> **;**

<méthode> = **method** <...> **begin** <corps> [<bloc traitement>] **end;**

<classe> = **class** <...> **begin** <corps> [<bloc traitement>] **end.**

Un bloc de traitement d'exceptions est introduit par le mot clef **except**, et se termine en général au terminateur de l'entité associée. Grâce à l'instruction bloc d'instructions, nous permettons une association de traitant à un ensemble d'instructions quelconques. Pour des raisons de souplesse, il est également possible d'associer un bloc de traitement à toute autre instruction qui possède une structure de bloc, en le plaçant juste avant le terminateur. En particulier, les instructions incluant une expression de test comme l'instruction **repeat** sont entièrement protégées par les traitants du bloc, expression de test comprise.

Mais le mode d'association privilégié de Guide reste l'association au corps de la méthode tout entier. On suit pour cela une syntaxe similaire ; le bloc est placé juste avant le terminateur. La sémantique de cette association reste celle de l'association à un bloc d'instructions ; toutes les instructions du corps de la méthode sont protégées par les traitants.

L'association à une classe suit le même schéma syntaxique, et conserve également la même sémantique (voir section III.3.1). Il s'agit d'une factorisation de l'association à toutes les méthodes de la classe. La seule différence d'avec l'association à une méthode provient des relations avec l'héritage. Les traitants déclarés dans ce bloc protègent les instructions de toutes les méthodes définies dans cette classe, dans toute sous-classe, et dans les super-classes (voir section V.8.2).

V.3.2 Portée

<masque exception> = (<ident exception> | **all**)
[**from** ([<ident méthode> **.**] <ident type> | **system** | **all**)]
<liste masques> = <masque exception> { **,** <masque exception> }

Pour aider le programmeur à préciser la portée des traitants qu'il déclare au niveau de la méthode, ce qui est le mode de déclaration que nous préconisons, le langage permet d'associer à chaque traitant un ou plusieurs masques d'exceptions. Un masque comprend bien sûr le nom de l'exception visée, qui peut être factorisé par **all**, mais surtout il peut préciser un nom de type et un nom d'opération. Ces deux derniers champs permettent de ne traiter que les exceptions signalées par l'exécution de l'opération donnée sur un objet référencé par une référence d'un type conforme à celui donné. Ainsi les deux traitants associés à la méthode *VoirPage* ne protègent que l'appel de méthode *pageCourante.LitCaract* puisque le masque associé impose le type *Page*. L'invocation d'impression *output.WriteChar* n'est pas concernée. Notons au passage que les deux traitants protègent bien l'invocation de lecture, puisque le type de *pageCourante* est *PageTampon*, qui n'est pas *Page*, mais qui lui est conforme.

```
method VoirPage;    // de la classe Editeur
begin
    while TRUE do
        output.WriteChar(pageCourante.LitCaract);
    end;
except
    fin_de_ligne from Page: replace '\n';
        // caractère imprimable pour un retour à la ligne
```

```
        fin_de_page from Page: return;
            // sort normalement de la méthode VoirPage
    end VoirPage;
```

Si le nom de méthode est absent du masque, comme c'est le cas dans l'exemple, alors toutes les méthodes du type sont concernées. Le mot clef **system** employé au lieu d'un nom de type permet d'adresser les exceptions système (voir section V.5).

Comme nous l'avons précisé en section III.3.1, un traitant de méthode est invoqué prioritairement par rapport à un traitant de classe. Plus généralement, un traitant syntaxiquement plus proche de l'instruction signalante qu'un autre traitant, est prioritaire par rapport à ce dernier. Ce choix est conservé même si le second traitant est associé à un masque plus précis, par exemple en donnant un nom de type alors que le premier ne le fait pas.

Enfin la portée d'un traitant inclut les traitants invoqués pendant l'exécution des instructions associées au bloc de traitement où il est défini (voir section V.6.1).

V.4 Traitements possibles

V.4.1 Cas standard

Nous avons discuté de la politique de reprise en section III.4 et nous l'avons refusée pour Guide. Par contre sont offertes les politiques de propagation et toutes les variations de la terminaison, à savoir remplacement, réessai, déroutement local. Ces traitements particuliers sont offerts en supplément des autres structures du langage, qui restent utilisables. Les traitants ont accès à l'environnement défini au point où ils sont déclarés, donc en particulier aux variables de travail de la méthode pour ceux déclarés dans une méthode.

La propagation, par resignalement, est discutée plus avant en section V.6.1. Rappelons ici toutefois que la propagation de l'exception système *UNCAUGHT_EXCEPTION* est le traitement par défaut fournit par le compilateur. Les vertus de ce traitant système sont rappelées en section III.3.2.

La terminaison, comme nous l'avons dit en section III.2.3, doit s'appliquer à la seule invocation de méthode signalante, et ne doit pas dépendre du point de déclaration du traitant. C'est ce que nous avons voulu faire pour Guide. Aussi, après que l'invocation *pageCourante.LitCaract* dans la méthode *VoirPage* a signalé l'exception *fin_de_ligne* et que le traitant correspondant a été exécuté, alors l'exécution reprend à l'invocation *output.WriteChar* de la même instruction. Ce traitant aurait pu être déclaré dans le bloc de traitement associé à la classe, le résultat aurait été le même. La terminaison est la politique appliquée par défaut, et n'est pas invoquée explicitement.

Comme dans le cas de l'exemple la méthode *LitCaract* doit rendre un caractère, le traitant ne peut pas terminer brutalement. Il doit fournir un caractère de remplacement utilisable par la méthode *WriteChar*. Le mot clef **replace** lui permet de le faire. Le compilateur a la charge de vérifier que la valeur fournie est conforme au type de retour normal de la méthode signalante. Ce travail de vérification peut se révéler important puisque par le jeu de masques généraux le traitant peut protéger un grand nombre d'appels différents.

Notons au passage que dans le cas d'une terminaison simple sans remplacement, le compilateur doit vérifier que les possibles invocations signalantes ne fournissent pas de valeur de retour.

La politique de réessai est offerte de manière générique par le mot clef **retry**. Cela provoque une nouvelle exécution de l'invocation signalante. Cette réexécution devrait être faite exactement dans le même contexte que la première, mais cela amènerait un risque important de bouclage pernicieux entre invocation signalante et traitant de réessai. Nous avons pris deux mesures pour contrecarrer cet effet. Tout d'abord nous imposons au traitant qui demande le réessai de rester non activable durant la nouvelle exécution. D'autre part nous autorisons le traitant à définir de nouveaux traitants qui incluent l'instruction **retry** dans leur portée, et qui protègent alors la réexécution de l'appelé en étant prioritaires par rapport aux traitants précédemment définis. A titre d'exemple de la politique de réessai, le traitant suivant essaie de continuer malgré le retour en exception d'une demande de *PageTampon* au *Gestionnaire*.

```
class Editeur implements Editeur is
    ...
    method  Nettoyage;
        page: ref PageTampon;
    begin
        page := pageTampons.First;
        while page do
            if page.sale then
                gestionnairePages.Libère(page);
                page := pageTampons.Delete;
            else
                page := pageTampons.Next;
        end; end;
    end;
except
    plus_d_objet from Gestionnaire.Demande: begin
        self.Nettoyage;
        retry;
        end;
end Editeur.
```

Il faut noter que le traitement **retry** n'est applicable qu'aux exceptions utilisateur. Il est interdit en traitement d'exceptions système, pour des raisons de difficulté de mise en œuvre.

Enfin le **return**, seule primitive Guide de déroutement local, est aussi admise à l'intérieur d'un traitant. Sa sémantique est celle d'un **return** normal, appliquée après la terminaison du traitant. Il est équivalent à un **return** écrit à la place de l'invocation signalante. Le compilateur doit alors garantir que la valeur éventuellement retournée est conforme au type du paramètre formel de retour de la méthode en cours d'exécution.

V.4.2 Traitant de classe

Les traitants de classe (définis dans un bloc de traitement associé à la classe) ont un statut un peu à part. En effet ils sont susceptibles d'être invoqués durant l'exécution d'un nombre important de méthodes de la classe, aussi certaines vérifications devront être faites en tenant compte de chacune de ces méthodes. D'autre part ils n'ont bien évidemment accès qu'à l'état de l'objet.

Nous avons déjà vu en section V.2.1 qu'une exception signalée depuis un traitant de classe doit être vérifiée entrer dans la déclaration de toute méthode où le traitant peut être appelé. L'exemple suivant montre une situation tordue que le compilateur doit reconnaître incorrecte. En effet, l'appel *self.m* de la méthode *p* peut amener le signalement de l'exception *e*, qui a son tour provoque l'exécution du traitant de classe de *c* hérité dans *s*, donc le signalement de l'exception *x*. La méthode *p* doit donc déclarer *x* dans sa signature comme une exception signalable.

```
type t is
    method m;
        signals e;
end t.
class c implements t is
    method m;
        signals e;
    begin
        raise e;
    end m;

    except
        e: raise x;
end c.
```

```
type s subtype of t is
    method p;
end s.

class s subclass of c
implements s is
    // hérite m de c

    method p;
    begin
        self.m;
    end p;
end s.
```

Le compilateur doit donc analyser le code des méthodes de la classe, et des méthodes héritées des super-classes. Mais il doit répéter ce travail pour toute sous-classe, puisque le traitant de classe est hérité (voir section V.8.2).

Le même type de travail devrait être réalisé pour chacune des politiques de terminaison simple, avec **replace**, ou avec **return**. Seul le **retry** ne demande aucun contrôle particulier. Or nous pensons que **return**, politique de déroutement local, devrait être rattaché à sa méthode, et non pas se retrouver factorisé dans un traitant de classe. C'est pourquoi nous (dans le compilateur) interdisons l'emploi de **return** dans un traitant de classe. D'autre part le **replace** dépend de manière très étroite de la méthode signalante. Il a pour but en effet de donner une valeur de remplacement à sa valeur normale de retour. Nous imposons donc à un traitant qui utilise cette politique et qui est déclaré dans le bloc de traitement de la classe de s'associer à un masque précis qui donne nom de type et nom de méthode décrivant les invocations qu'il protège. Ainsi le compilateur n'a plus besoin de parcourir le code pour réaliser ses vérifications ; il peut se contenter d'analyser le masque. Nous pensons que cette contrainte est réellement minime.

V.5 Exceptions système et hardware

V.5.1 Liste

Comme nous l'avons vu en section II.3.1, une source importante d'exceptions est la défaillance d'un composant interne utilisé. Un composant bien particulier et fortement utilisé est le système sous-jacent. Il faut donc pouvoir rendre compte des exceptions système. Celles du système Guide se résolvent en trois catégories.

On trouve tout d'abord celles qui ont un sens clair pour le programmeur, et qui sont généralement liées aux informations que celui-ci donne au système.

CALL_ON_NILREF

Cette exception sanctionne une invocation sur la référence **nil**. En effet **nil** est conforme à tout type construit Guide. Le compilateur, par le biais des vérifications de conformité, ne peut donc pas empêcher l'invocation d'une méthode sur cette référence.

CONFORMANCE_ERROR

Le compilateur vérifie la majeure partie des règles de conformité statiquement. Mais il existe des cas où une vérification dynamique est nécessaire. La vérification est demandée par le mot clef **assertype** dans le programme Guide ; il en existe actuellement deux utilisations principales. Premier point, le programmeur peut avoir besoin de reprendre plus de sémantique sur un objet qu'il manipule à travers une référence peu précise. L'outil principal pour ce faire est **typecase**, mais **assertype** peut être utilisé dans ce but, comme on le voit dans l'exemple suivant où on veut appliquer la méthode *proc* définie sur le sous-type, sur un objet du sous-type mais accédé par une référence vers le super-type.

```
superType: <référence vers le super-type>;
sousType: <référence vers le sous-type>;

// essai direct, interdit par les règles de conformité
superType.proc;

// utilisation de assertype
sousType := assertype(superType);
sousType.proc;

// utilisation de typecase
typecase superType of
<sous-type>:
    superType.proc;    // maintenant autorisé
    end;
others:
end;
```

La deuxième utilisation est liée à la récupération d'un objet depuis le serveur de noms. Il s'agit d'un cas similaire, puisque le type de retour d'un appel au serveur de noms est *Top*, le type le plus général possible ; mais l'utilisation est différente puisque le programmeur s'attend à recevoir un objet d'un type connu.

```
objet: <référence vers le type effectif de l'objet>;
objet := assertype(catal.Search(<nom de l'objet>));
```

Après l'exécution de cette instruction, le programmeur manipule l'objet à travers la référence *objet*. Lorsque le type effectif de l'objet n'est pas conforme au type de la référence, alors l'exception *CONFORMANCE_ERROR* est signalée.

JOIN_FAILED

L'exception est signalée lorsque la condition de terminaison du bloc parallèle ne peut plus être réalisée. Ce mécanisme est décrit en détail en section V.9.

RECOVERY_FAILED

Cette exception marque l'exécution incorrecte d'une clause de restauration. La restauration est développée en section V.7.

UNCAUGHT_EXCEPTION

Nous avons déjà parlé de cette exception (voir sections III.3.2 et V.4.1). Elle est signalée par le système dès lors qu'une exception n'est pas autrement traitée par l'appelant.

QUIT

Cette exception traduit la demande de terminaison d'une activité. Cela peut être le résultat d'une demande directe, par la méthode *Kill* de la classe *Activity* ou par la destruction du domaine englobant, mais aussi de la terminaison d'un bloc parallèle (voir section V.9). Un tel évènement, typiquement asynchrone, est resynchronisé en un signalement de l'exception système (voir section A.5.2).

Le deuxième groupe d'exceptions traduit un dysfonctionnement plus ou moins grave du système.

CREATION_FAILED

Cette exception marque l'occurrence d'une erreur pendant la création d'un objet, primitive *guideCreate* du système.

LINK_FAILED

Celle-ci a trait au chargement d'un objet, objet étant pris au sens large puisque le code d'une classe est un objet pour le système.

UNKNOWN_METHOD

Cette exception est signalée par le système lorsqu'il ne trouve pas la méthode demandée dans la classe de l'objet. Nous le considérons ici comme une erreur puisque l'utilisation exclusive du compilateur Guide interdit ce genre de situation.

EXTENSION_FAILED

Celle-ci détecte l'extension manquée d'un domaine.

REMOTE_CALL_FAILED

Celle-là rend compte d'une erreur durant l'exécution d'un appel à distance. Il s'agit bien d'une erreur et non d'une exception langage signalée par la méthode appelée exécutée à distance. Dans ce dernier cas l'exception est normalement ramenée à l'appelant qui peut en réponse exécuter un traitant.

FORK_FAILED

Cette exception marque une erreur durant la création d'une nouvelle activité. Elle peut être signalée aussi bien lors d'un **co_begin** que lors d'une création de domaine.

SEGMENTATION_FAULT

Cette exception indique un écrasement incorrect d'une zone mémoire. Cette exception n'est signalée que lorsque le système s'exécute dans un mode particulier, appelé vérification des bornes. Dans ce mode, le système vérifie que les bornes de la zone mémoire où est chargé l'objet en cours d'exécution ne sont pas écrasées, avant et après l'exécution de la méthode. Cela ne permet bien sûr pas de détecter toutes les erreurs de débordement, mais c'est tout de même fort utile.

EXCEPTION_COLLISION

Celle-ci permet au système de rendre compte du signalement "simultané" de plusieurs exceptions. Elle est utile en particulier lorsque une erreur système intervient dans le *guideCall* , après l'appel effectif au code de la méthode, alors qu'une exception utilisateur a été signalée (voir section A.4.3).

TRACE_DEMO_ERROR

Celle-ci a trait au fonctionnement en mode démonstration. Elle indique une erreur dans la communication avec l'objet *GuideEvents* de l'Observer [63].

SYSTEM_ERROR

Enfin cette dernière exception sert de fourre-tout pour les cas non répertoriés. Elle permet également de rapporter les exceptions du noyau sur lequel le système est construit. De telles exceptions n'ont pas à priori à apparaître plus explicitement, si le langage n'a pas la visibilité du noyau.

Le dernier type d'exception système regroupe celles introduites par le noyau sous forme d'exception hardware. Le modèle d'une telle exception est la division par zéro, détectée par le processeur. Ces exceptions n'ont pas été traitées dans la maquette actuelle, mais certaines idées sont développées en section VI.3.

V.5.2 Visibilité

Les exceptions système ont souvent été présentées comme des exceptions prédéfinies sur chaque classe. Cela présente l'inconvénient de ne pas faire traiter l'exception par l'entité responsable, ou au choix de ne pas présenter un modèle homogène. En effet, si l'exception (fictive) *DIVISION_PAR_0* signalée lors de l'exécution de *1/0* est assimilée à une exception de *appelé* , alors l'entité responsable du traitement est le strict appelant *appelant* , alors que ce devrait être *appelé* .

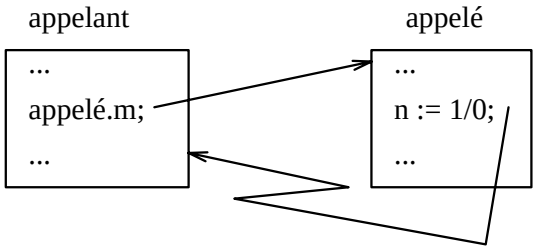

Fig. 5.1 : division par 0, le mauvais choix

En fait l'erreur vient d'une mauvaise représentation des interactions avec le système. Nous pensons qu'il faut considérer celui-ci comme une entité appelée par chaque méthode et à chaque instruction. L'instruction *1/0* devient ainsi l'invocation de la division du système, qui devient du coup l'entité signalante alors que *appelé* devient responsable du traitement.

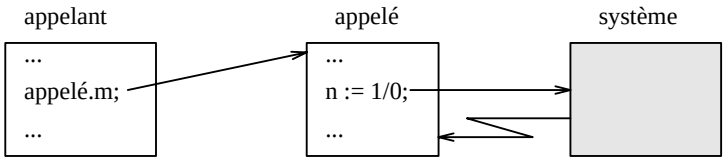

Fig. 5.2 : division par 0, le bon choix

Toutes ces exceptions sont rattrapables par un masque de traitant qui donne le mot clef **system** comme nom de type. L'ensemble des exceptions système peut être traité grâce au masque **all from system**. Le masque **all** comprend toutes les exceptions, système ou utilisateur. Il va de soi que ces deux derniers masques doivent être utilisés avec une extrême prudence, vu qu'ils peuvent servir à cacher des dysfonctionnements du système.

V.5.3 Traitements

Comme nous l'avons dit en section V.4.1, le traitement de **retry** n'est pas autorisé sur une exception système, pour une raison de difficulté de mise en œuvre. La propagation est toujours possible. Le **return** est également acceptable. La terminaison simple est tolérée par endroits au lieu d'un **replace**, encore une fois pour des raisons de mise en œuvre.

V.6 Exceptions dans un traitant

V.6.1 Cas général

Lorsqu'une exception est signalée par l'invocation d'une méthode dans un traitant, que doit-on en faire ? En particulier, les traitants définis au niveau de la méthode ou de la classe associée au traitant sont-ils valides ?

Disons pour commencer que le traitant peut définir dans son corps de nouveaux traitants qui protègent tout ou partie de son code. Quant aux autres traitants de la méthode, la réponse habituelle est oui, et la solution couramment adoptée est qu'un traitant est protégé par les traitants qui l'englobent syntaxiquement. Ainsi le traitant de l'exception *erreur2*, associé à la méthode *Init* et dont l'invocation va générer le signalement de *erreur3*, est protégé par le traitant de *erreur3* associé à la classe *Objet*. La procédure *Opération* sert de générateur d'exceptions dans cet exemple.

```
class Objet is
    method Init;
    begin
        Opération(1);
    except
        erreur2:  Opération(3);
    end Init;

    procedure  Opération(in n: Integer);
        signals erreur1, erreur2, erreur3;
    begin
        case n of
            1: raise  erreur1;
            2: raise  erreur2;
            3: raise  erreur3;
        end;
    end  Opération;

    except
        erreur1:  Opération(2);
        erreur3:   output.WriteString("ok\n");
end  Objet.
```

Or, pour différentes raisons, nous avons voulu fournir des traitants à portée dynamique. Cela veut dire que le traitant est armé (i.e. il devient activable si une exception correspondant à son masque est signalée) au début de l'exécution de l'entité à laquelle il est associé, et désarmé à la fin de cette exécution. La conséquence est que la trace de l'exécution de *Init* sur un objet de classe *Objet* est :

ok

En effet, lors de l'exécution de *Opération(1)* le traitant associé à *erreur2* dans *Init* est armé. Les deux autres traitants associés à la classe le sont d'ailleurs également. Lorsque la méthode signale l'exception *erreur1*, alors le traitant associé à la classe est invoqué, exécute *Opération(2)* qui signale à son tour *erreur2*. Le premier traitant est alors exécuté et invoque *Opération(3)*. La fin est évidente. Le point à noter est que pour cette exécution particulière, le traitant associé à *erreur2* dans la méthode *Init* a protégé le traitant associé à *erreur1* dans la classe. Voyons maintenant les justifications de ce choix.

Historiquement la portée du traitant est dynamique (voir section II.2.1). Elle le reste d'ailleurs tant qu'il s'agit d'exceptions signalées pendant l'exécution du corps normal d'une méthode. Cela vient du fait que lorsqu'on associe un traitant à une méthode, on veut signifier que si tel cas survient durant l'exécution de la méthode, alors tel traitement doit être appliqué. Le traitant est associé à la méthode non pas syntaxiquement, mais par une notion temporelle de durée d'exécution. L'association de ce traitant aux traitants syntaxiquement inclus dans la déclaration de la méthode n'a dès lors plus de sens. Nous pensons donc que seuls deux choix sont cohérents, soit fournir des traitants à portée complètement dynamique, soit particulariser le code de traitement d'exception et interdire à un traitant de protéger un autre traitant de même niveau (voir section VI.4.2). En fait il faut tempérer cette déclaration, car s'il est vrai qu'elle tient debout tant qu'il s'agit d'exceptions vraies, elle est plus discutable pour ce qui est des exceptions programmées.

V.6.2 Traitements

L'instruction du traitant est considérée remplacer dans la méthode l'instruction signalante. La sémantique du traitement d'une exception signalée par une instruction de traitant s'en déduit simplement ; elle est résumée dans le schéma suivant.

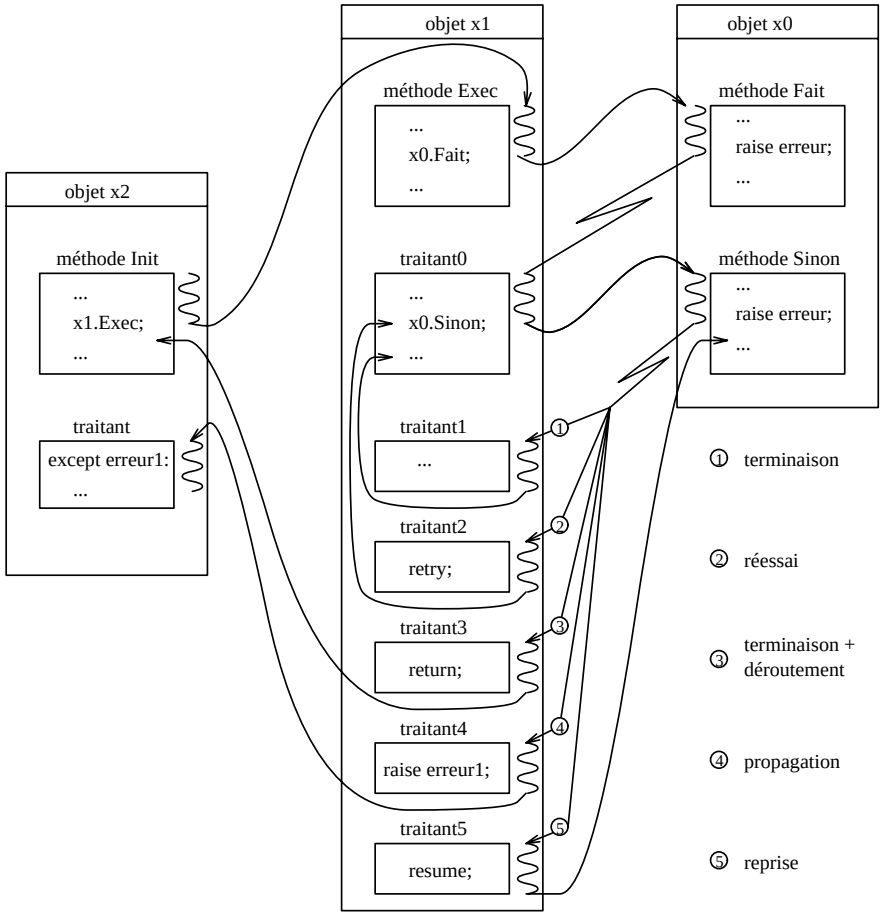

Fig. 5.3 : traitements d'une exception signalée depuis un traitant

Il faut simplement noter l'effet du **return**, qui parait délicat à manipuler car l'effet est le même que le traitant soit appelé directement ou en réponse à une exception signalée dans un traitant (voir section VI.4.2).

V.7 Restauration

Nous avons conclu en section III.5 à la nécessité d'un mécanisme de restauration qui puisse garantir la cohérence des objets en cas de signalement d'exception. Les sections qui suivent décrivent une première approche du problème, que nous avons mise en œuvre dans la maquette actuelle, mais qui est rediscutée en section VI.2.

V.7.1 Déclaration et association

<bloc restauration> = **restore** <déclaration variables locales>
begin <corps> **end;**
<méthode> = **method** <...> **begin** <corps>
[<bloc traitement>] [<bloc restauration>] **end;**
<classe> = **class** <...> **begin** <corps>
[<bloc traitement>] [<bloc restauration>] **end.**

Le programmeur peut associer un bloc de restauration à une méthode ou à une classe. La déclaration se fait simplement grâce au mot clef **restore**, qui introduit une définition semblable à celle d'une procédure. Elle se place avant le terminateur de la méthode ou de la classe.

Le bloc de restauration d'une classe est automatiquement hérité dans une sous-classe, si celle-ci n'en définit pas. Dans le cas contraire, le bloc de la sous-classe remplace celui de sa super-classe.

V.7.2 Exécution

Le code de restauration est systématiquement exécuté lorsqu'une méthode sort en signalant une exception. Le système commence par le bloc de restauration de la méthode, s'il existe, puis exécute celui de la classe. Le bloc de méthode concerné est bien sûr celui défini en même temps que la méthode exécutée. Par contre le bloc de classe est celui associé à la classe effective de l'objet, et non à celle où la méthode exécutée est définie.

Pourquoi exécuter les deux blocs ? Nous avions le choix entre cette solution ou considérer le bloc de méthode comme une surcharge du bloc de classe. Pourquoi lier dynamiquement le bloc de classe ? Nous aurions pu choisir d'exécuter le bloc de la classe de définition de la méthode exécutée. Il n'y a pas véritablement de raisons à ces choix, qui sont d'ailleurs fortement remis en cause (voir section VI.2.1). Néanmoins l'idée directrice a été de permettre à l'hypothétique vérificateur formel de prouver la vérification des invariants de classe en cas d'exception uniquement à partir du bloc de restauration de la classe effective de l'objet. D'autre part le choix de lier dynamiquement le code de restauration de classe est en conformité avec le choix fait pour les traitants de classe, comme pour d'autres structures du langage.

L'exemple suivant montre une utilisation possible de la restauration, qui permet à notre *Editeur* de sauver ses *Pages* sales quand il sort en exception.

```
class Editeur implements Editeur is
    ...
restore
    page: ref PageTampon;
    documentDelta: ref Document;
begin
    page := pageTampons.First;
    while page # nil do
        if page.sale then
            documentDelta.pages.Append(page);
        else
            gestionnairePages.Libère(page);
        end;
        page := pageTampons.Delete;
    end;
    if documentDelta.pages.nbItem > 0 then
        catal.Insert(<nom>, documentDelta);
        output.WriteString("sortie anormale\n" +
            "modifications sauvées dans " + <nom>);
    end;
end;
end Editeur.
```

V.7.3 Exceptions dans la restauration

La manipulation des exceptions reste possible dans le code de restauration. Celui-ci peut définir de nouveau traitants, mais il est aussi protégé par les traitants de classe, et pour le bloc de restauration de la méthode par les traitants de la méthode. Tous les mécanismes habituels s'appliquent, excepté pour la propagation ou le signalement direct d'une exception. En effet le bloc de restauration est censé remettre l'objet dans un état cohérent. L'occurrence d'une exception non traitée pendant l'exécution du code de restauration veut dire que cet objectif n'est pas atteint, et il faut donc empêcher tout traitement de l'appelant qui le supposerait. Nous avons choisi pour cela de signaler une exception système particulière, *RECOVERY_FAILED*.

Pour résumer, après que l'appelé signale ou propage une exception il exécute son code de restauration. Si celui-ci termine correctement l'exception est transmise à l'appelant. Dans le cas contraire l'appelant voit le signalement de l'exception système *RECOVERY_FAILED*.

V.8 Conformité et héritage

Nous allons détailler dans cette section les nombreuses implications de la conformité et de l'héritage dans les mécanismes d'exception et de restauration. Certains choix ont été dictés par ceux déjà faits pour d'autres structures du langage, comme les conditions d'activation.

V.8.1 Déclaration

La déclaration des exceptions signalables par une méthode doit suivre les règles de conformité. Avant de décrire ces règles, un exemple simple permet d'en comprendre les fondements.

```
type LigneTampon
subtype of PageTampon is
    method LitCaract: Char;
        signals erreur,
            fin_de_ligne;
end LigneTampon.
// ce type est correct
```

```
type PageTamponSpéciale
subtype of PageTampon is
    method LitCaract: Char;
        signals erreur,
            fin_de_ligne,
            fin_de_page,
            non_imprimable;
end PageTamponSpéciale.
// celui-ci ne l'est pas
```

Dans la méthode *VoirPage*, l'appel *pageCourante.LitCaract* ne peut pas signaler d'autre exception que *fin_de_ligne* , *fin_de_page* , ou *erreur* qui sont déclarées dans et héritées du type *Page*. Comme *pageCourante* peut désigner un objet de tout type conforme à *PageTampon*, cela veut dire que la méthode *LitCaract* d'un tel type ne peut signaler qu'un sous-ensemble de ces trois exceptions. C'est pourquoi le type *PageTamponSpéciale* n'est pas conforme à *PageTampon*, et sa définition comme sous-type de *PageTampon* est incorrecte.

La règle de conformité pour les exceptions s'énonce donc comme suit. Une méthode A est conforme à une méthode B si et seulement si chaque exception déclarée par A est aussi déclarée par B.

V.8.2 Traitants

Le bloc de traitement associé à une méthode est lié au code de cette méthode. Si celle-ci est surchargée dans une sous-classe, alors le bloc est abandonné, même si la surcharge ne définit pas un nouveau bloc de traitement. La politique adoptée est différente que celle des conditions d'activation, mais les deux ne sont pas réellement comparables. La clause d'activation attachée à une méthode contrôle les interactions de la méthode avec les autres méthodes de la classe. Elle a donc un sens global qu'un traitant de méthode ne possède pas du tout.

Par contre la règle d'héritage s'applique pour le bloc de traitement associé à la super-classe. Le cas le plus simple est celui d'une sous-classe qui ne définit pas de nouveau bloc de traitement ; le bloc de la super-classe est alors hérité intégralement. Si elle en définit un, il faut décider si le nouveau bloc écrase l'ancien, ou s'il y a héritage partiel comme ce qui est fait pour les conditions d'activation. La règle de priorité suivant la proximité permet d'établir la règle d'héritage suivante. Le bloc de la super-classe est intégralement repris, mais ses traitants sont moins prioritaires que ceux de la sous-classe. Il y a bien héritage partiel des traitants pour le masque desquels aucun nouveau traitant n'est défini dans la sous-classe, les autres traitants étant masqués par ceux nouvellement définis. En fait, par un effet de bord du désarmement d'un traitant en cours d'exécution, ces traitants peuvent se trouver invoqués.

Une question plus difficile a trait à l'invocation potentielle d'un traitant défini dans la sous-classe, lors de l'exécution d'une méthode définie dans la super-classe. On fait la même distinction entre traitants de classe et de méthode pour la surcharge que pour l'héritage. On ne peut donc pas surcharger dans la sous-classe le bloc de traitement associé à une méthode sans surcharger également la méthode. Par contre, les traitants de classe protègent

l'exécution de toutes les méthodes de la classe, y compris de celles héritées de super-classes. Cette dernière politique respecte le choix général du langage, qui présuppose de la part de la sous-classe une connaissance approfondie de sa super-classe, et qui se traduit en particulier par la possible surcharge des conditions d'activation d'une méthode sans surcharge de la méthode elle-même. Ce résultat est conforme aux conclusions de la section III.3.1.

V.8.3 Restauration

De manière similaire à ce qui est fait pour le bloc de traitement, le bloc de restauration associé à une méthode n'est pas hérité dans une sous-classe qui surcharge la méthode, même si celle-ci ne définit pas de nouveau bloc de restauration. Cela se comprend en considérant que la part de restauration associée à la méthode dépend strictement de son code, et que la redéfinition de ce dernier impose la redéfinition du premier. Il faut relier ceci au mécanisme offert par Guide pour récupérer dans la surcharge de la méthode le code de sa définition initiale dans la super-classe : celui-ci prend la forme d'un appel procédural sur le mot clef **super**, qui fixe statiquement la classe de définition de la méthode appelée. Dans ce cas l'exécution de la méthode version super-classe est effectivement protégée par le code de restauration associé à la méthode dans la super-classe.

Le bloc de restauration de la classe est lui hérité par défaut dans les sous-classes qui n'en redéfinissent pas un en propre. Par contre il n'existe pas l'équivalent de l'héritage partiel des conditions d'activation ou de l'héritage avec moindre priorité du bloc de traitement. En effet le code de restauration est considéré comme un tout. On peut toutefois s'interroger sur la concaténation possible dans une classe des blocs de restauration associés à la classe et à ses super-classes. Sur ce point nous restons cohérents avec les choix adoptés pour les autres structures du langage, et ne prenons en compte que le bloc de restauration défini le plus proche.

La même question difficile que pour le bloc de traitement porte sur la possible exécution du code de restauration redéfini dans une sous-classe, lors de la sortie en exception d'une méthode définie sur une super-classe. Comme nous l'avons dit en section V.7.2, nous avons répondu oui à cette question de la liaison dynamique du code de restauration de classe. La raison est la volonté d'autonomie d'un objet, au niveau de sa classe effective, dans la détermination des conditions qui maintiennent sa cohérence en cas d'exception. Ce choix est par ailleurs cohérent avec celui fait pour l'héritage du code de traitement d'exceptions de la classe.

V.9 Exceptions et parallélisme

Nous regardons dans ce chapitre comment les exceptions s'intègrent aux mécanismes de gestion du parallélisme dans Guide.

V.9.1 Sémantique

Une première primitive que nous ne décrivons pas ici permet de créer un domaine, entité d'exécution asynchrone. Comme il n'existe aucun moyen de resynchronisation du domaine avec son créateur, l'éventuelle terminaison en exception de l'activité principale du domaine

reste ignorée. Nous nous concentrerons donc sur la deuxième primitive Guide, introduite par le mot-clef **co_begin**, et qui offre la création de plusieurs activités parallèles resynchronisées en fin d'exécution.

Le code de chaque branche parallèle se réduit dans l'état actuel du langage à un appel de méthode. Or un mécanisme que nous ne voulons pas développer ici permet en fait de donner une suite d'instructions, ce qui nous offre l'équivalent de l'instruction "idéale" décrite ci-dessous, et que nous allons commenter.

<instruction co_begin idéale> ≅ **co_begin**
{ <identificateur> **: begin** <corps> [<bloc traitement>] **end;** }*
[<bloc traitement>] **co_end** [<condition de terminaison>] **;**

Chaque nouvelle activité se voit attribuer un identificateur qui peut être utilisé dans la condition de terminaison. Le but de cette condition est d'exprimer le cas de terminaison correcte du bloc parallèle en fonction de la terminaison correcte de chacune des branches. Chaque identificateur de branche est une variable booléenne, vraie si la branche a terminé correctement, fausse si elle n'a pas encore terminé ou si elle a terminé en exception. La condition de terminaison est une combinaison en **and** et **or** de ces variables. Elle permet d'affiner la condition offerte par défaut qui impose la terminaison correcte de chaque branche pour accepter une terminaison correcte de l'ensemble. On peut ainsi exprimer une condition sur deux branches *a* et *b*, statuant la terminaison correcte du bloc parallèle d'après la terminaison correcte d'une des deux branches (la première qui termine).

```
co_begin
    a:  miseEnOeuvre1(paramètres);
    b:  miseEnOeuvre2(paramètres);
co_end a or b;
```

A partir d'une telle condition, le système peut non seulement détecter le moment où elle est satisfaite mais aussi celui où elle ne peut plus l'être. Ainsi, en supposant que dans l'exemple la branche *a* est la première à terminer mais signale une exception, comme la branche *b* peut toujours terminer correctement et vérifier la condition de terminaison du bloc l'exécution continue. Par contre, si *b* termine également en signalant une exception, alors la condition ne peut plus être vérifiée. La sémantique du **co_begin** s'énonce donc comme suit : lorsque la terminaison correcte d'une branche amène la terminaison correcte du bloc, alors les branches restantes sont arrêtées (voir section V.9.4) et l'instruction **co_begin** termine normalement ; lorsque la terminaison en exception d'une branche amène à l'impossibilité d'une vérification future de la condition de terminaison, alors les branches restantes sont arrêtées et l'instruction **co_begin** termine anormalement, en signalant l'exception système *JOIN_FAILED*.

V.9.2 Traitants

On note dans le mécanisme de signalement de l'exception *JOIN_FAILED* que l'on a perdu de l'information, puisqu'on ne connaît plus après ni la branche "coupable" du ratage, ni l'exception signalée dans cette branche.

Il faut nuancer cette déclaration. Dans l'exemple précédent c'est la terminaison anormale des deux branches qui amène le signalement de *JOIN_FAILED*. Il devient dès lors difficile de rapporter cette double exception par le mécanisme standard monovalué. Par ailleurs l'objet appelant peut adapter son comportement suivant l'exception signalée et suivant la branche qui la signale, en déclarant des traitants spécifiques que nous allons maintenant décrire.

Chaque branche peut déclarer un bloc de traitement pour protéger les instructions qui composent son corps. Ces traitants sont invoqués classiquement dans l'environnement de l'objet appelant, mais logiquement par l'activité fille qui a signalé l'exception ; ils permettent donc au bloc d'interpréter localement le signalement d'une exception par une de ses branches. Les autres traitants qui sont armés lors de l'exécution du **co_begin** sont également susceptibles d'être appelés pour traiter l'exception d'une branche. Les traitants de classe et de méthode en particulier sont concernés. Cela offre un second niveau de traitement, cette fois indépendant de la branche signalante. Enfin si l'exception n'est pas rattrapée, la branche termine anormalement et le bloc interprète cette terminaison à travers sa condition de terminaison.

V.9.3 Traitements

Deux traitements ne prennent pas leur sens habituel. Il s'agit de **return** et **raise** qui, s'ils étaient traités comme à l'habitude, amèneraient la terminaison de toutes les branches du bloc parallèle puis la sortie de la méthode elle-même, soit normalement, soit en exception. Nous avons choisi d'imposer un traitement moins sévère en confinant l'exception au bloc parallèle. Dans ces cas l'activité fille considérée est terminée anormalement, mais cela peut être rattrapé par la clause de terminaison du bloc parallèle.

V.9.4 Terminaison des activités parallèles

Que ce soit en cas de terminaison normale ou anormale du bloc parallèle, les activités restantes sont arrêtées. Une terminaison brutale de l'activité aurait l'inconvénient de laisser nombre d'objets dans un état incohérent. Le système décide donc d'arrêter les activités plus doucement, en leur "signalant" une exception *QUIT*.

Le mécanisme parait étrange puisque l'action est asynchrone, de l'activité mère sur l'activité fille, alors que le signalement d'exception est synchrone. En fait cela se fait très simplement en resynchronisant ce signalement lors de l'appel système suivant. Comme tout est supposé être un appel système, cela peut se faire théoriquement n'importe quand. Les problèmes de mise en œuvre amènent à restreindre ces possibilités, et l'exception *QUIT* est actuellement resynchronisée sur l'appel d'une primitive système Guide (voir section A.5.2). Il va de soi que cela implique un risque de non traitement à bref délai de cette exception, en cas de bouclage important sans appel au système.

Le grand intérêt de ce choix est d'autoriser l'exécution du code de restauration de toutes les méthodes en cours d'exécution et donc interrompues, et cela permet du même coup au système de remettre certaines choses en place très naturellement, comme les compteurs de synchronisation.

Pour l'instant l'activité mère attend que toutes ses filles aient terminé, après les avoir averties, pour continuer. Mais nous pourrions envisager de reprendre l'exécution après la détection de la terminaison, en laissant les activités filles se terminer seules.

V.10 Conclusions et résultats

Le langage Guide est un langage orienté-objets fortement typé. Les exceptions y ont une nature simple de chaînes de caractères, ce qui fait perdre les avantages liés à une structure hiérarchique, mais cela garde aux mécanismes de création et de signalement d'exception leur simplicité. Cela permet d'autre part de conserver l'attachement d'une exception à la méthode qui la signale. Les exceptions signalées par une méthode doivent être déclarées dans la signature de la méthode. Cela permet la mise en œuvre de tests à la compilation, qui débouchent sur des avertissements ou sur la détection d'erreurs.

Les traitants peuvent être associés à des blocs d'instructions, mais le mode d'association que nous préconisons est l'association à la méthode ou à la classe. Pour permettre de préciser sa portée, le traitant est attaché à un masque qui précise les appels de méthode qu'il protège. De plus quel que soit le mode de déclaration du traitant, la sémantique du traitement qu'il propose reste inchangée. Cette technique d'association permet de dissocier le code de l'algorithme principal et le code de traitement des situations exceptionnelles. Ce dernier se trouve alors regroupé en fin de méthode ou en fin de classe, améliorant ainsi la lisibilité de l'algorithme principal. Cela favorise également la factorisation des traitants.

La reprise n'est pas offerte. Seuls la terminaison, avec fourniture d'une valeur de remplacement, le réessai et la propagation sont possibles. Le traitement par défaut fourni par le système est la propagation de l'exception *UNCAUGHT_EXCEPTION*. Ce traitement par défaut garantit qu'une exception, une fois détectée, n'est pas involontairement oubliée. Cette remarque est particulièrement importante appliquée aux exceptions signalées par le système, car elle augmente la sécurité des programmes qui ne vérifient pas que les appels système qu'ils effectuent se déroulent correctement.

L'utilisation de **return** est interdite dans un traitant de classe. L'utilisation de **replace** dans un tel traitant est contrôlée par le masque associé au traitant.

Un certain nombre d'exceptions système permettent de rendre compte de cas particuliers non détectables à la compilation, ou de gérer la propagation d'une exception non traitée. D'autres traduisent un dysfonctionnement du système et ont en principe peu de signification pour le programmeur. Toutes ces exceptions sont signalées à l'utilisateur comme provenant d'une opération faite sur le type **system**, autorisant ainsi leur traitement. Le grand intérêt de ces exceptions système, avant même la possibilité offerte de les traiter, est d'armer un ensemble de détecteurs qui augmentent les chances de remarquer les erreurs à l'exécution. Combinées avec le traitant de propagation par défaut, elles réduisent le danger de diffusion de ces erreurs. D'autre part le fait d'homogénéiser la gestion des exceptions système et celle des exceptions utilisateur permet de reprendre pour ces exceptions système les possibilités de restauration offertes dans les autres cas. Ainsi même dans le cas de problèmes importants qui nécessitent l'arrêt d'une tâche, chaque objet impliqué a la possibilité de se remettre dans un état cohérent.

La portée des traitants est dynamique, correspondant à un armement et un désarmement respectivement au début et à la fin de l'exécution de l'entité associée. Seul le traitant en cours d'exécution est désarmé, les autres continuent de protéger le traitant exécuté.

Il est possible d'associer du code de restauration à une méthode et à une classe. En cas de sortie d'une méthode en exception ce code est exécuté. En cas de sortie normale il est ignoré. Si l'exécution du code de restauration provoque le signalement d'une exception non traitée, alors l'exception système *RECOVERY_FAILED* est signalée. Ce mécanisme garantit que les différentes politiques de traitement ne sont exécutées que si l'objet signalant est dans un état cohérent, mais l'expérience montre qu'il présente certaines limitations que nous essayons de résoudre en section VI.2.

La règle de conformité pour les exceptions impose que seules les exceptions déclarées dans la signature d'une méthode peuvent être déclarées dans la signature de la surcharge de la méthode. Cette règle est rigide mais nécessaire si on veut garantir statiquement la validité des manipulations d'objets au travers des variables typées.

Un traitant de classe protège les instructions des méthodes définies dans la classe, dans ses super-classes et dans ses sous-classes. Le code de restauration de classe est également recherché dynamiquement. Ces règles de comportement des mécanismes de gestion des exceptions et de restauration vis-à-vis de l'héritage suivent la politique générale du langage Guide. Elles permettent un partage de code plus important, autorisant en particulier la définition de traitants généraux associés aux classes.

Le programmeur peut interpréter la terminaison correcte d'un bloc parallèle en fonction de la terminaison correcte des différentes branches. A ce niveau les éventuelles exceptions signalées par les branches sont perdues. Par contre le programmeur est capable de fournir des traitants spécifiques à chaque branche, ce qui l'autorise à interpréter localement l'exception. Il est d'autre part averti quand le bloc parallèle ne peut plus se terminer correctement, suite à la fin anormale d'une des branches. Les branches en cours d'exécution lors de la sortie du bloc parallèle sont arrêtées "doucement" grâce au signalement de l'exception système *QUIT*. Nous retirons deux résultats importants de cette nouvelle sémantique du bloc parallèle. D'une part nous pouvons reconnaître le cas où la terminaison incorrecte d'une des branches interdit la terminaison correcte future du bloc parallèle. D'autre part l'arrêt en douceur des branches restant en activité lors de la terminaison du bloc facilite grandement la remise dans un état cohérent des objets manipulés par ces branches, ainsi que des structures du système concernées.

Chapitre VI

Développements futurs

Les choix de base de notre mécanisme ne sont pas remis en cause. Sa construction sur la base de l'appel de méthode, la sémantique de l'association, le besoin d'un mécanisme de restauration, sont autant de points positifs que nous voulons garder. Par contre certains choix secondaires demandent à être revus. Le but de ce chapitre est de développer et d'argumenter les modifications que nous voudrions intégrer à la deuxième conception du mécanisme pour Guide.

VI.1 Hiérarchie d'exceptions

Le premier point porte sur la nature des exceptions. Le choix actuel est une chaîne de caractères. Nous avons discuté en section III.6 des exceptions comme objets typés, mais nous ne pensons toujours pas que cette solution compliquée soit adéquate pour Guide. Nous voulons par contre reprendre la hiérarchie sur les exceptions que cette solution implique (voir section III.6.4).

VI.1.1 Limitations des simples noms

Nous avons vu en section V.8.1 que la structure des exceptions impose une stricte règle de conformité. Or la définition du type *PageTamponSpéciale* est typique des besoins du programmeur en matière d'exceptions dans un sous-type. Le sous-type précise le super-type, éventuellement le complète par de nouvelles méthodes, il aimerait donc pouvoir disposer de nouvelles exceptions.

La seconde restriction due à la structure actuelle des exceptions est l'impossibilité de signaler une exception dans un traitant de classe. Or c'est encore une fois un besoin important du programmeur, qui aimerait pouvoir regrouper dans la classe les traitants de propagation. Ce besoin est d'autant plus impérieux que notre mécanisme impose la propagation explicite d'une exception sous peine de propager l'exception système anonyme *UNCAUGHT_EXCEPTION*.

C'est parce que ce besoin est réel que nous autorisons dans le modèle actuel le signalement d'une exception dans un traitant de classe, bien que cela amène à des vérifications relativement complexes (voir section V.4.2). C'est aussi parce que ces vérifications sont trop complexes à réaliser que nous n'offrons pas cette fonction dans la réalisation actuelle (voir section A.2.1.1).

Nous n'avons pas expliqué en quoi des exceptions non structurées empêchent le signalement depuis un traitant de classe, ou plutôt nous n'en avons abordé qu'un aspect. L'autre possibilité est de pouvoir déclarer des exceptions signalables au niveau du type. Cela

paraît naturel, car si un traitant de classe est la factorisation d'un traitant pour les méthodes de la classe, une exception de type doit être la factorisation d'une exception pour les méthodes du type. Et c'est là que les règles de conformité entrent en jeu. Si une exception est signalable par un type, elle doit l'être par son super-type, donc par le super-type commun *Top*. Il est alors impossible de déclarer de nouvelles exceptions de type, autres que celles éventuellement déclarées sur *Top*.

VI.1.2 Structure hiérarchique

Nous pensons que la solution est une structure hiérarchique d'exceptions, qui autorise la spécialisation d'une exception en plusieurs autres. On peut pour cela user de la syntaxe suivante dans la partie déclarative.

<liste signaux> = **signals** <exception> { **,** <exception> } **;**

<exception> = <ident exception> [**isa** <ident exception>]

La déclaration peut être faite au niveau de la signature de méthode, ou factorisée au niveau du type. Par le mot clef **isa**, l'exception donne le nom de l'exception qu'elle spécialise.

L'exception reste néanmoins associée à son type de déclaration (voir section III.6.5). En conséquence, l'identificateur donné après le mot clef **isa** doit être celui d'une exception déclarée dans le super-type. Une exception déclarée dans une méthode du sous-type peut spécialiser une exception du super-type, ou de la même méthode du super-type en cas de surcharge. Une exception du sous-type peut spécialiser une exception du super-type. La déclaration du type *PageTamponSpéciale* devient possible.

```
type PageTamponSpéciale subtype of PageTampon is
    method LitCaract: Char;
        signals erreur, fin_de_ligne, fin_de_page,
            non_imprimable isa erreur;
end PageTamponSpéciale.
```

Pour ce qui est des exceptions de type, il suffit d'initialiser le processus en déclarant une exception générale au niveau de *Top*, comme par exemple *error*.

Les implications sur l'association de traitant sont en partie celles attendues. Dans l'exemple ci-dessous, les deux références *pt* et *pts* référencent le même objet, qui est d'un type conforme à *PageTamponSpéciale* . On suppose que l'invocation de la méthode *LitCaract* sur cet objet signale l'exception *non_imprimable*. Le traitant *<traitant1>* , associé à l'exception *erreur* signalée par le type *PageTampon*, est capable de traiter l'exception, qu'elle soit signalée par l'invocation sur *pt* ou *pts*.

```
    c: Char;
    pt: ref PageTampon;
    pts: ref PageTamponSpéciale;
begin
    pt := pts;
    c := pt.LitCaract;
    c := pts.LitCaract;
except
    erreur from PageTampon: <traitant1>;
    non_imprimable from PageTamponSpéciale: <traitant2>;
end;
```

Le problème se complique quand on réfléchit sur la portée du second traitant, et pour deux raisons. Première question, *<traitant2>* protège-t-il l'invocation sur *pt* ? Nous pensons que non, car formellement une invocation sur *pt* ne peut pas signaler cette exception, et ne peut voir dans le signalement effectif que l'exception *erreur*. Autoriser l'association reviendrait en quelque sorte à lui donner la fonction d'un **assertype**, car un fois dans le traitant le programmeur a d'une certaine manière gagné de l'information sur *pt* ; il peut considérer cette variable comme étant référence de type *PageTamponSpéciale* . Mais cette décision est vraiment subtile, et est sujette à controverse.

Ce qui est sûr, c'est que *<traitant2>* , comme *<traitant1>* , protège l'exécution de l'invocation sur *pts*. Mais lequel des deux est prioritaire sur l'autre ? L'un est plus précis quant à l'exception signalée, mais l'autre est plus proche syntaxiquement, et c'est ce dernier critère qui décide de la priorité entre deux traitants qui diffèrent de par leur type associé (voir section V.3.2). Nous penchons donc pour le second choix, pour une raison d'homogénéité. Mais il est vrai que nous ne donnons pas de raison très forte au choix de la section V.3.2, aussi celui-ci est-il également discutable. Le seul impératif est d'avoir un choix homogène, c'est-à-dire donner plus d'importance à la syntaxe d'association ou à la précision du masque. Il faut noter tout de même que notre choix est "définitif", alors que le choix opposé demande la définition supplémentaire d'une règle de priorité entre les différents champs du masque.

VI.2 Restauration

Le mécanisme offert actuellement est relativement primitif ; il assure l'exécution du code de restauration en cas de sortie en exception. D'autre part il traite de manière particulière la terminaison en exception de ce code. Nous allons d'abord regarder les limitations de ce mécanisme, avant de proposer quelques directions de travail.

Les descriptions qui suivent peuvent sembler très précises, mais il ne s'agit que de propositions. Elles peuvent être par moments incohérentes et doivent être développées, et confrontées aux autres structures du langage.

VI.2.1 Limitations actuelles

La contrainte qui apparaît en premier lieu est l'exécution systématique du code de restauration en cas de signalement. Comme il a été dit en section III.5.1, le but de ce code est d'assurer la validité des invariants (virtuels) de classe, et présuppose un minimum sur l'état de l'objet. Or, surtout en cas de traitement explicite d'une exception utilisateur, le programmeur est souvent au fait de l'état de l'objet, et est capable de le restaurer simplement, par exemple en défaisant ce qu'il a commencé. Dans ce cas il n'a aucune envie de payer le coût d'un traitement général.

Le deuxième point a trait à l'exception système *RECOVERY_FAILED*. Le but de cette exception est d'avertir l'appelant de l'objet que ce dernier n'a pas pu se remettre dans un état cohérent. Or ce mécanisme est incomplet, pour deux raisons.

Il n'avertit que l'appelant de l'objet, alors que les mécanismes de Guide autorisent son partage soit par accès concurrent, soit par accès différé. L'objet doit donc être capable d'avertir tous ces appelants concurrents ou futurs de l'incohérence de son état.

D'autre part, aucun mécanisme de synchronisation n'est offert pour contrôler l'exécution du code de restauration. Or il est raisonnable de penser que l'exécution concurrente d'autres méthodes a un impact important sur la restauration de l'état de l'objet.

VI.2.2 Restauration de méthode orientée finalisation

Le premier problème, écriture de code de restauration qui suppose connu l'état de l'objet, est à rapprocher des objectifs d'un outil de finalisation. Nous proposons donc de rechercher un mécanisme intermédiaire, qui tient de la finalisation tant que le code s'exécute correctement, mais qui prend le sens de restauration manquée en cas d'exécution incorrecte.

Ce code est déclaré dans un bloc associé à la méthode. Cela se justifie par sa nature, car il dépend du code exécuté jusqu'au point de sortie. Par contre il est inutile de fournir un mécanisme de déclaration plus précis, au niveau de l'instruction, car, de par la nature du mécanisme d'exceptions, une majorité de points de sortie sera trouvée dans les traitants de méthode ou de classe.

Pour permettre tout de même la différenciation suivant le point de sortie, le code est dépendant de la nature de la sortie. Nous proposons une syntaxe semblable à celle d'un bloc de traitement, avec des blocs de code associés à des noms d'exceptions. On peut également utiliser le mot clef *NO_EXCEPTION* pour déclarer du code exécutable en cas de sortie normale.

Le code ainsi défini est exécuté en cas de sortie de la méthode, normalement ou en exception, depuis le corps de la méthode ou depuis un traitant de bloc, de méthode ou de classe.

Pour autoriser la factorisation de code entre les différents cas de sortie, les différents blocs déclarés dans le bloc de restauration de méthode ne sont pas exclusifs. Tous les blocs associés à un masque correspondant au type de sortie sont exécutés. La factorisation par nom d'exception hiérarchiquement englobant est également valide.

L'aspect restauration de ce bloc de méthode apparaît dans les points suivants. La terminaison en exception du code de restauration de méthode met l'objet dans un état incohérent. Cela est valide y compris lors de l'exécution du code de restauration associé à *NO_EXCEPTION*, provoquée par une sortie dite normale. Si une exception est signalée et qu'il n'existe pas de bloc associé dans le code de restauration de méthode, alors l'objet est dans un état incohérent. L'absence volontaire de code de restauration pour l'exception est explicitement déclarée par un bloc vide.

VI.2.3 Etat incohérent de l'objet

Nous avons vu dans quelles circonstances l'état de l'objet est déclaré incohérent. A partir de ce moment des mécanismes coûteux peuvent être activés sans problème pour le programmeur, qui a pu le contrôler par l'intermédiaire de la restauration de méthode.

On exécute alors le bloc de restauration de classe. Si celui-ci s'exécute correctement, alors l'appelant est averti de la restauration et de ses causes, par exemple par l'intermédiaire des exceptions *UNCAUGHT_EXCEPTION* et *FINALIZATION_FAILED*. Dans le cas contraire, l'objet retourne à l'appelant l'exception système *BAD_OBJECT_STATUS*.

Les méthodes en cours d'exécution lors de la détection de l'état incohérent de l'objet sont suspendues le temps de l'exécution du code de restauration. Si cette exécution termine correctement, alors les différents appelants en sont avertis par l'intermédiaire de l'exception système *OBJECT_RESTORED*. Dans le cas contraire, l'exception *BAD_OBJECT_STATUS* est signalée.

La demande d'exécution d'une autre méthode, postérieure à la détection de l'état incohérent de l'objet, est bloquée pendant l'exécution de la restauration. Dans le cas d'une restauration correcte, l'exécution reprend normalement. Dans le cas contraire, l'exception *BAD_OBJECT_STATUS* est signalée.

Ces principes simples répondent aux remarques de la section VI.2.1. Ils demandent bien sûr à être développés, en relation avec les autres structures du langage. Un autre axe de développement est de faire participer à la restauration les différentes activités qui s'exécutent concurrement sur l'objet, à la différence du schéma maître/esclaves présenté ici.

VI.3 Exceptions hardware

Le problème des exceptions hardware ne se situe pas vraiment au niveau du modèle. Nous pensons en effet que notre choix de les faire apparaître comme des exceptions système est correct (voir section V.5.1). Le problème est dans la mise en œuvre.

VI.3.1 Problème de mise en œuvre sur Unix

Une exception hardware, par exemple provoquée par une division par 0, est traduite en Unix par un signal. Le programmeur peut avoir prévu un traitement particulier en réponse à ce signal, sous la forme d'une procédure que le système invoque automatiquement. En conséquence, les seules politiques de traitement réalisables directement sont la terminaison, avec un **return** C, ou la destruction de l'activité, avec un **exit** C. La propagation ou la terminaison avec déroutement, troisième flèche du schéma, ne sont pas réalisables ainsi.

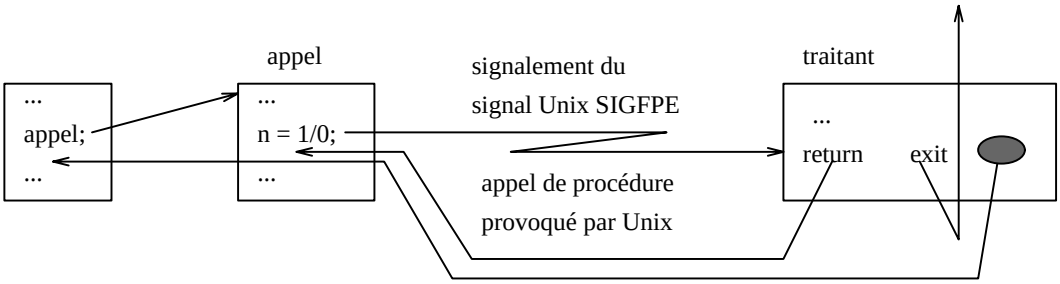

Fig. 6.1 : signaux Unix

Pour réaliser ces politiques il faut utiliser les primitives de déroutement de C, *setjump* et *longjump*. La première permet de capturer un environnement, un état de pile, et le second permet d'y retourner. Il faut en prévention appeler *setjump* à chaque début de méthode,

pour pouvoir le cas échéant invoquer *longjump* depuis le traitant du signal Unix. Mais ce choix est particulièrement coûteux, et surtout le prix est à payer tout le temps, même en l'absence de signalement.

VI.3.2 Mise en œuvre sur micro-noyau

La seconde version de Guide sera mise en œuvre sur micro-noyau, Mach ou Chorus [57]. Ces noyaux offrent un mécanisme de base pour le traitement des exceptions, mécanisme qui permet une mise en œuvre plus facile du traitement des exceptions hardware. Nous décrivons ci-dessous le mécanisme offert par Mach.

L'occurrence d'une exception hardware est transformée par le noyau en un message envoyé au port exception du thread qui l'a provoquée. Ce thread est alors suspendu pendant qu'un autre thread traite le message. Le thread de traitement peut manipuler le thread signalant, et en particulier lui faire reprendre son exécution, après avoir éventuellement modifié ses registres. C'est cette propriété que nous utilisons dans le schéma qui suit, pour dérouter le thread signalant depuis le thread de traitement.

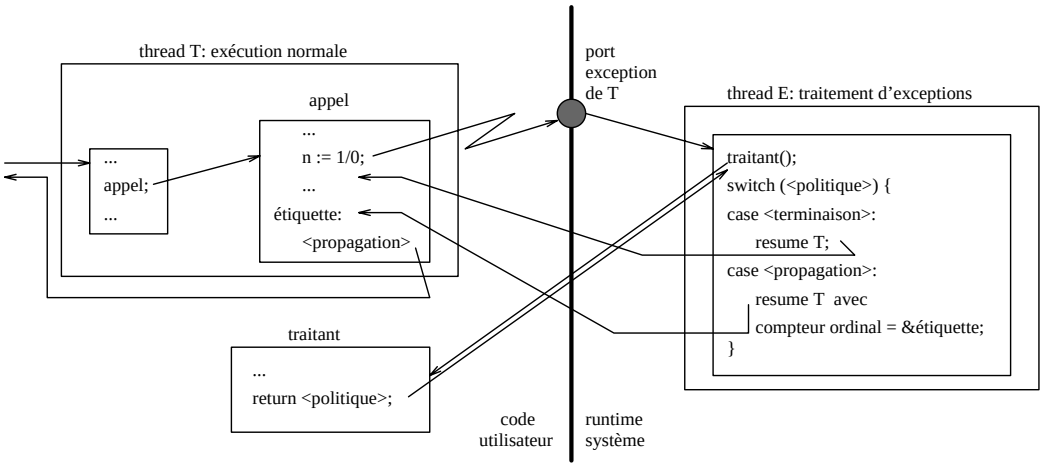

Fig. 6.2 : exceptions sur Mach

Il s'agit là d'un schéma de principe encore incomplet. Il faut en particulier savoir passer au thread de traitement l'adresse du traitant qu'il doit exécuter, ou plutôt faire exécuter, déterminer par qui le traitant doit être exécuté (thread *T* ?), et savoir rechercher dans le thread de traitement la politique donnée en retour. On peut aussi essayer d'optimiser le schéma pour qu'il n'y ait qu'un seul aller-retour entre les threads.

VI.4 Autres points

Nous présentons pour terminer quelques idées qui sont dans un état encore embryonnaire.

VI.4.1 Type et classe Core

Nous n'avons pas abordé le compte rendu par le système à l'utilisateur de la terminaison en exception d'une commande. Le point d'entrée de l'utilisateur au système Guide est un interpréteur d'appels de méthode, *Shell*. Lorsque la méthode appelée signale ou propage une exception l'utilisateur en est averti mais n'obtient aucune information sur ses causes, en particulier il ne connait pas la première exception signalée dont la propagation termine la méthode initiale. C'est une conséquence naturelle du principe d'encapsulation que nous avons choisi d'appliquer, mais la dernière exception, celle visible par l'utilisateur, risque d'être souvent *UNCAUGHT_EXCEPTION*. L'utilisateur doit être capable, dans un objectif de recherche d'erreur, d'obtenir plus de renseignements sur les causes de l'exception.

Pour cela nous pensons utile de créer un type *Status*, qui conserve les points importants de l'exécution d'une activité. Chaque objet représentant une activité possède dans son état une référence vers un objet *Status*. En cas d'exécution normale, cette référence vaut **nil**. C'est également le cas lors du signalement d'une exception qui est rattrapée par l'appelant, pour ne pas surcharger inutilement le mécanisme d'exceptions. Ainsi, lorsque *obj0.Op0* signale *erreur*, rien n'est fait car *erreur* pourrait être rattrapée par *obj1*. Par contre, si l'appelant propage explicitement ou implicitement l'exception, alors un objet de type *Status* est créé, et est affecté à la référence dans l'état de l'activité. Cet objet contient essentiellement référence de l'objet courant, méthode en cours d'exécution, ainsi que le chaînage vers l'objet *Status* qui décrit l'appel signalant.

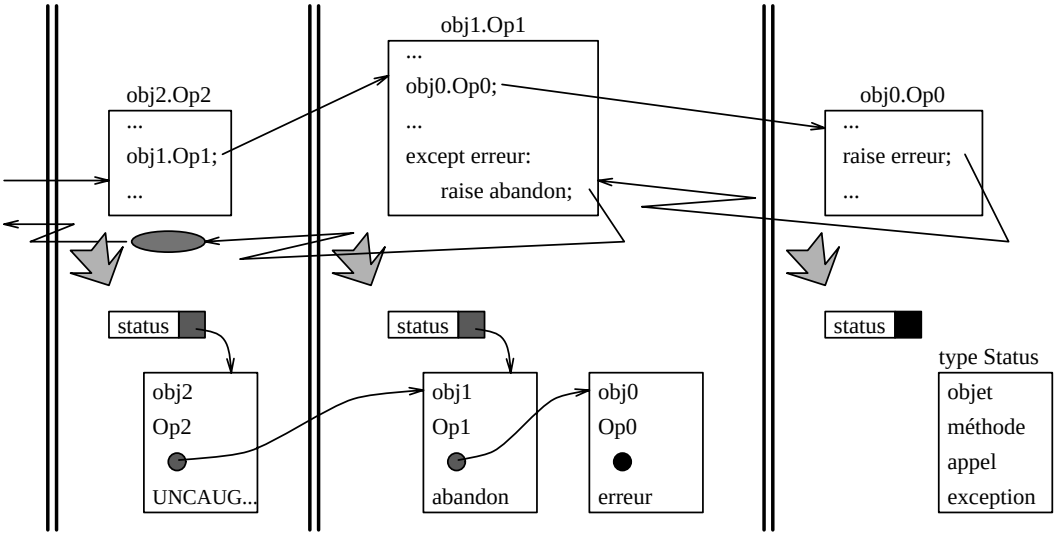

Fig. 6.3 : objet Status d'une activité

Lorsque *obj1* propage l'exception *abandon*, il y a création d'un objet *Status* pour *obj1* mais aussi pour *obj0*. La création de celui pour *obj0* est retardée pour ne pas pénaliser l'utilisation du mécanisme pour les exceptions dites programmées.

VI.4.2 Exceptions dans un traitant

Nous développons ici l'idée donnée en section V.6.1 au sujet de la protection du code des traitants. Le principe est cette fois de dire qu'il existe différents niveaux de code exceptionnel, et que le code d'un niveau ne peut protéger que le code du niveau inférieur.

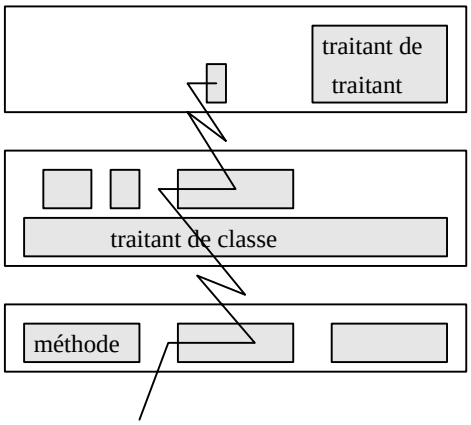

Fig. 6.4 : niveaux de traitants

Les traitants de premier niveau sont ceux associés à une structure du code principal, classe, méthode, instruction. Les traitants de second niveau sont ceux associés à, ou dans le code de, un traitant de premier niveau.

Cette technique a plusieurs avantages du point de vue du programmeur. Le principal est qu'elle élimine le cas pathologique de protection croisée donné en exemple en section V.6.1. D'autre part elle supprime le risque de bouclage lié en particulier à la politique de réessai, puisque un traitant ne peut se protéger lui-même. Enfin elle supprime le problème de la sémantique bizarre d'un **return** dans un traitant de traitant V.6.2. Il reste toutefois à déterminer si elle est préférable pour le programmeur Guide.

VI.4.3 Classes et applications

Nous voulons terminer en rappelant que le mécanisme que nous proposons repose sur la stricte application du principe d'encapsulation des objets. Or l'expérience montre que l'on a régulièrement envie d'écrire une application par l'intermédiaire de classes coopérantes. L'exemple typique est l'utilisation des **friend**s de C++. Ce mot clef permet à une classe d'accéder directement aux champs des objets d'une autre classe, sans se préoccuper de l'interface de cette autre classe. Dès lors la question se pose d'offrir à ces classes coopérantes des possibilités plus vastes quant au mécanisme d'exceptions. Une exception pourrait alors être traitée par un appelant plus éloigné que le strict appelant. Il faudrait également revoir notre position vis à vis de la politique de reprise.

Mais cette notion de classe coopérante est encore informelle, et il existe beaucoup de détracteurs à l'usage des **friend**s de C++. Cet aspect des langages à objets est loin d'être élucidé.

Chapitre VII

Conclusion

Ce travail nous a amené à rappeler les objectifs d'un SGE, et à proposer une classification des exceptions. Cela nous a permis de discuter des contraintes qu'un environnement à objets apporte sur les différents mécanismes de signalement et de traitement, et en particulier sur la politique de reprise. Nous avons également écrit les avantages de l'héritage dans la déclaration de traitants généraux. Nous avons ensuite défini un SGE pour le langage Guide en répondant aux points spécifiques suivants : traitement des exceptions système, définition d'un mécanisme de restauration et traitement de l'instruction de parallélisme.

VII.1 Exceptions et SGE

Nous détaillons en cinq points dans le chapitre II les objectifs qu'un SGE doit remplir et que nous rappelons :

- Il doit permettre de séparer le code de l'algorithme principal et le code de traitement des exceptions. Cela augmente la lisibilité de l'algorithme principal, et correspond au schéma de pensée du programmeur.
- Il doit favoriser la factorisation du code de traitement. Cela facilite la définition de traitants généraux et diminue la charge du programmeur.
- Il ne doit rien coûter tant qu'aucune exception n'est signalée. Cela favorise l'exécution de l'algorithme principal supposée plus fréquente. Le coût du traitement d'une exception peut être plus élevé.
- Il doit garantir qu'une exception ne peut être involontairement oubliée une fois détectée. C'est le point le plus important. Cela supprime une source importante d'erreurs difficiles à retrouver.
- Il doit particulariser le signalement vis-à-vis des autres structures de contrôle. C'est un peu la conséquence du point précédent. Cela permet de distinguer l'exception du cas normal.

Ces objectifs sont atteints par la majeure partie des SGE existants, et ils sont respectés aussi bien par notre proposition générale du chapitre III que par notre proposition pour Guide.

Nous proposons dans le même chapitre II une classification des exceptions en trois catégories :

- Le signalement d'une exception vraie ne doit en principe pas survenir. Appelant et appelé sont indépendants, et l'appelant espère que l'appelé ne signalera pas d'exception.
- Le signalement d'une exception programmée est au contraire prévu et forcé. L'appelant provoque le signalement de l'exception par l'appelé ; cela fait partie de

son algorithme. A la limite appelant et appelé coopèrent pour utiliser les fonctions du SGE.

- Le signalement d'une exception peut enfin marquer la détection d'une erreur. L'erreur n'est pas détectable par elle-même, mais elle peut provoquer l'exécution d'un appel en dehors de ses préconditions. Le SGE conduit le programmeur à signaler une exception en présence d'un tel appel.

A chacune de ces catégories correspond des types de traitements. Cette classification des exceptions et les traitements associés nous ont servi dans l'analyse des contraintes qu'un environnement à objets impose sur la conception d'un SGE.

VII.2 Conclusions sur l'environnement à objets

L'opération élémentaire d'un langage orienté-objets est l'appel de méthode. C'est autour de cette opération élémentaire qu'un SGE pour un tel langage doit être construit. Or la nature de l'appel de méthode, qui inclut la notion de polymorphisme, amène une contrainte forte d'encapsulation du code qui dirige la définition du SGE. Cela débouche sur les caractéristiques suivantes des fonctions de base du SGE :

- Le signalement d'une exception marque la non réalisation du traitement normal attendu de la méthode. Il a pour la méthode appelée la sémantique d'un retour. Dans un langage fortement typé les exceptions signalées par une méthode doivent apparaître dans la signature de cette méthode, et doivent être prises en compte dans la règle de conformité.
- Un traitant doit être sémantiquement associé à un appel de méthode. Les politiques de terminaison, de remplacement, de réessai sont alors définies par rapport à l'appel de méthode signalant.
- Un traitant peut être syntaxiquement associé à différentes entités du langage, en particulier une méthode ou une classe, mais il s'agit d'une factorisation de l'association à tous les appels de méthode "inclus" dans cette entité. Cette indépendance entre la syntaxe d'association d'un traitant et la sémantique du traitement qu'il propose permet d'une part la déclaration du traitant en dehors du code de l'algorithme principal, et favorise d'autre part la factorisation des traitants.

La définition de traitants généraux bénéficie de la fonction nouvelle d'héritage. Cela permet de disposer dans un langage à objets d'un environnement riche de traitants. On peut en distinguer trois types :

- Les traitants associés à une classe ont une portée très importante, si toutefois les relations entre une classe et une sous-classe sont privilégiées dans le langage. Ils protègent les instructions des méthodes de la classe ; ils sont hérités et protègent donc les instructions des méthodes des sous-classes ; ils sont recherchés dynamiquement et protègent donc les instructions des méthodes des super-classes.
- Les traitants système sont constamment armés. Ils sont conceptuellement associés à une classe racine du langage, s'il en existe une. Il doit exister un traitant système particulier, exécuté quand il n'existe aucun autre traitant pour une exception donnée, et qui propage une exception particulière. Ce traitant est très important ; il

garantit qu'une exception ne peut être involontairement oubliée une fois détectée, tout en supprimant l'obligation faite au programmeur de fournir un traitant pour chaque exception potentiellement signalée.

- Les propositions de traitement ne sont pas de vrais traitants, mais elles peuvent enrichir l'environnement des traitements applicables à une exception. Toutefois ce nouveau concept complique le SGE, et ne se justifie que dans un environnement interactif où ces différentes propositions sont soumises à un utilisateur capable de choisir.

La politique de reprise est très difficile à justifier. L'analyse porte sur les différents types d'emploi décrits dans le chapitre II :

- Dans le cadre du traitement d'une exception vraie, la politique de reprise entre en contradiction avec le principe d'encapsulation du code. La fonction recherchée est celle d'un mécanisme intelligent de réessai. On peut toutefois accepter la reprise dans un environnement de mise au point, mais dans ce cas seulement.
- Dans le cadre de l'extension du "domaine d'appel normal" d'une méthode, la reprise ne peut être acceptée que dans un cas d'héritage, si les relations entre une classe et sa super-classe sont privilégiées. Mais cette utilisation est marginale, car elle suppose de nombreuses conditions pour être applicable.
- Dans le cadre d'un appel explicite à l'appelant, la reprise est un palliatif au concept de continuation. Cette utilisation détournée ne peut pas justifier la fourniture de la reprise par le SGE.

L'utilisation du formalisme objet pour représenter les exceptions et les mécanismes qui s'y rattachent présente quelques avantages :

- La structure hiérarchique des exceptions qui en découle permet la spécialisation dans un sous-type d'une exception déclarée dans un type.
- L'utilisation des champs de l'objet exception permet le passage de paramètres lors du signalement.

Mais les inconvénients sont trop importants et emportent la décision. Ils sont principalement dûs aux dangers liés à la banalisation des exceptions :

- La manipulation des exceptions est alourdie, nécessitant l'emploi de classes, voire de méta-classes. Cela doit amener un surcoût à l'exécution.
- Les méthodes de signalement et de traitement ont une sémantique très particulière puisqu'elles peuvent ne pas retourner.
- On perd la dépendance qui existe entre une exception et la méthode qui la signale. Cela rend difficile l'explication et la vérification par le compilateur des contraintes qui s'y rattachent.

La définition d'un SGE dans un langage à objets complique le maintien de la cohérence des objets. Cela impose la définition d'un mécanisme de restauration associé qui présente les caractéristiques suivantes :

- Assurer l'atomicité des appels de méthode remplit l'objectif de cohérence, mais c'est lourd. Nous pensons que la possibilité offerte au programmeur d'écrire du code de restauration est suffisante.

- Le mécanisme doit être distinct du SGE. Cela permet d'une part d'éviter la duplication du code de restauration aux multiples points de sortie impliqués par le SGE, et d'autre part de conserver la validité dans une classe des traitants généraux définis dans ses super-classes.

VII.3 Conclusions sur Guide

Le SGE que nous définissons pour Guide reprend les conclusions précédentes. Nous y apportons les précisions suivantes :

- La déclaration des exceptions dans la signature des méthodes autorise le compilateur à effectuer un certain nombre de tests, qui débouchent soit sur des message d'erreur, soit sur des avertissements qui peuvent être le fait d'erreurs.
- La règle de conformité est complétée par une contrainte qui restreint aux exceptions déclarées dans la signature d'une méthode celles déclarables dans la signature d'une méthode conforme.
- Un masque d'exceptions associé à un traitant permet de préciser la portée de celui-ci. C'est un complément appréciable à la déclaration de traitants généraux au niveau de la classe.
- Les politiques de remplacement et de réessai sont offertes de manière générique grâce aux mots clefs **replace** et **retry**.

Nous répondons ensuite à certains problèmes spécifiques au langage Guide, en commençant par donner une réponse au problème des exceptions système.

- Les exceptions système sont intégrées au SGE pour apparaître de la même manière que les exceptions utilisateur. Les appels système sont considérés comme des appels de méthode sur un objet fictif de type **system**. Les exceptions signalées par ce type sont prédéfinies, et elles peuvent être traitées.
- Il en découle un gain important du point de vue de la cohérence des objets, car chaque objet concerné par la tâche dans laquelle l'exception est signalée a la possibilité de se remettre dans un état cohérent grâce à l'exécution d'un code de restauration.

Nous présentons ensuite notre proposition pour un mécanisme de restauration. Elle présente les caractéristiques suivantes :

- Du code de restauration peut être défini au niveau de chaque méthode et chaque classe. Ce code est exécuté quand une exception est signalée. Il n'est pas exécuté en cas de sortie normale, car dans ce cas l'objet est supposé être dans un état cohérent.
- Si une exception est signalée et non traitée durant l'exécution du code de restauration, alors l'exception système *RECOVERY_FAILED* est signalée. Cela garantit que les différentes politiques de traitement ne sont exécutées que si l'objet signalant est dans un état cohérent.

Enfin les exceptions sont intégrées à l'instruction de parallélisme de Guide. Nous apportons les fonctions supplémentaires suivantes :

- Il est possible de détecter une situation où la terminaison anormale d'une des branches du bloc parallèle interdit la terminaison normale future du bloc.
- Les branches restant en activité lorsque le bloc se termine sont arrêtées en douceur grâce à l'exception système *QUIT*. Chaque objet en cours d'exécution dans ces branches peut alors exécuter du code de restauration.

VII.4 Perspectives

Cette conception a donné lieu à une réalisation qui valide la majeure partie des choix faits. Nous voyons toutefois trois améliorations importantes qui pourraient être apportées au mécanisme.

Nous avons noté les avantages d'une structure hiérarchique d'exceptions, mais nous ne l'avons pas mise en œuvre. Nous proposons une solution qui reprend cette structure sans pour autant représenter les exceptions comme des objets typés.

Le mécanisme de restauration proposé se révèle par certains côtés contraignant. Nous proposons de le séparer en deux niveaux. Un premier niveau se comporte comme un mécanisme de finalisation, mais en cas d'erreur active le second niveau qui permet de mettre en œuvre des mesures plus fortes. Nous prenons d'autre part en compte le problème de la concurrence d'accès à un objet dans un état incohérent.

Les exceptions hardware ne sont pas traitées dans la réalisation actuelle. Nous proposons une mise en œuvre dans la maquette future sur le micro-noyau Mach.

Annexe A

Mise en œuvre

Nous avons pu mettre en œuvre le mécanisme décrit dans le chapitre V sur la maquette réalisée pendant la première phase du projet Guide. Cette maquette a été construite au dessus du système Unix, et est décrite dans [30]. Un certain nombre de points de la mise en œuvre diffèrent du modèle du chapitre V, mais l'essentiel a été réalisé, et est fourni dans la livraison actuelle du système et du langage.

A.1 Schéma général de la maquette

Nous ne décrivons ici que les aspects de la maquette Guide qui ont influé sur la mise en œuvre du mécanisme de gestion des exceptions. Cette présentation risque toutefois d'être insuffisante pour qui ne connaît pas le système, mais nous n'avons pas voulu trop nous étendre sur ce qui est déjà décrit dans d'autres documents.

A.1.1 Architecture

La distribution est cachée par le système. L'activité, concept distribué du langage, est mise en œuvre par des processus communicants, représentants de l'activité sur chaque site. Ces représentants sont créés à la demande grâce au démon *guided*, processus qui préexiste sur chaque site.

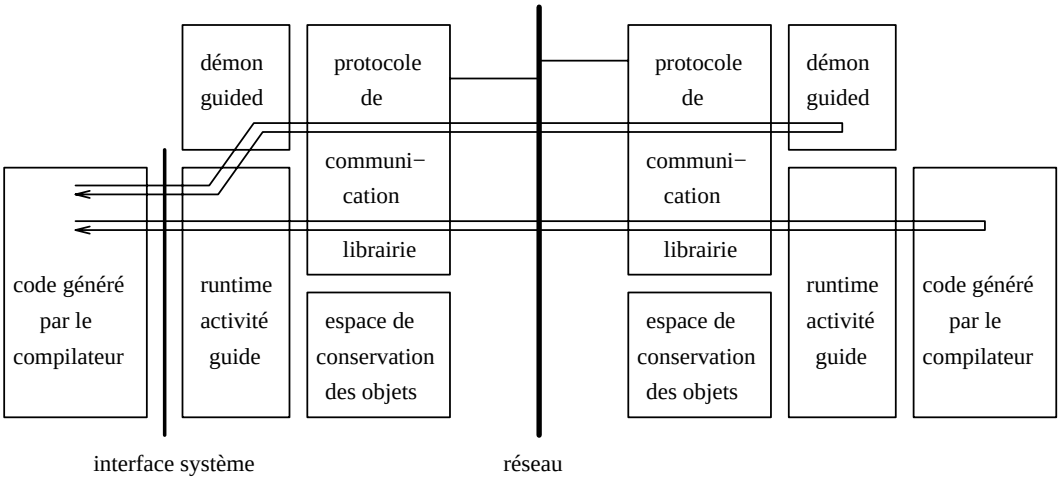

Fig. 1.1 : schéma d'exécution

Le schéma de compilation se décompose lui en cinq phases. La première analyse le source Guide et génère un arbre syntaxique. La seconde réalise des vérifications sémantiques, et complète l'arbre. La troisième génère du code C. La quatrième compile le code C et résoud les éventuelles références externes restantes pour générer un code objet intermédiaire. Ce code est transformé en un objet classe au format Guide durant la cinquième phase, et est inséré dans l'espace de conservation des objets.

Dans la suite nous allons préciser le format d'un objet classe Guide, ainsi que le schéma de compilation et d'exécution d'un appel de méthode. Ce sont en effet sur ces points que nous avons dû greffer la mise en œuvre de la gestion des exceptions.

A.1.2 Relations entre compilateur et système

Le format d'un objet classe Guide a pour origine les contraintes du chargement dynamique. Les classes Guide sont en effet chargées à la demande par le système. Pour laisser à ce dernier la liberté de l'adresse de chargement, le code généré est relogeable. Les méthodes définies dans la classe sont compilées en procédures, dont l'adresse par rapport au début de la classe est conservée dans l'entête de celle-ci, et plus précisément dans la Table des Définitions Externes (ou TDE).

Le format de l'entête d'une classe est une structure connue du système. Cela lui permet d'invoquer des fonctions spécifiques à la classe, et en particulier la fonction de recherche de l'index d'une méthode à partir de son nom. Chaque entrée de la TDE décrit ensuite les données propres à une méthode. On y trouve en particulier l'adresse du code de la méthode, mais aussi celle d'une éventuelle fonction de synchronisation associée. Cette dernière est également invoquée directement depuis le système.

Le compilateur peut également définir des procédures internes à la classe, qui seront directement manipulées par le code généré. C'est la solution adoptée pour la compilation des procédures Guide. Les éventuels pointeurs vers ces procédures sont automatiquement mis à jour lors du chargement de la classe. Par automatiquement nous voulons dire qu'il n'y a rien de particulier à faire lors de l'ajout d'une procédure interne dans la classe, contrairement à l'ajout d'une procédure appelable depuis le système qui impose la changement de la structure de l'entête.

La connaissance inverse du système par le code généré par le compilateur est condensée dans l'adresse de la *primTab*, table des primitives du système. Dans cette table se retrouvent adresses de fonctions mais aussi de variables globales. La table est dépendante de l'activité.

Les fonctions sont les primitives du système Guide, comme *guideCall*, mais peuvent également servir d'intermédiaires pour fournir au code généré les fonctions du runtime Unix, comme *strcpy*. Les variables permettent de partager un environnement, comme *myDomain* qui référence le domaine courant.

A.1.3 L'invocation de méthode

Elle est réalisée par la primitive système *guideCall*. Cette primitive accepte en paramètre un bloc d'appel de type *T_callBlock*. Le système n'utilise qu'une partie des informations du bloc d'appel, qui sont la référence de l'objet visé, le nom codé de la méthode

invoquée, et une structure de taille variable précisant nombre, nature, type et valeur des paramètres d'appel. Il charge alors l'objet, sa classe, éventuellement la classe de définition de la méthode en cas d'héritage, et complète le bloc d'appel avec les adresses de liaison. Si besoin il met également à jour le champ *primTabStart* avec l'adresse de la *primTab*. Il invoque ensuite le code de la méthode en lui fournissant le bloc d'appel complété en paramètre.

On se retrouve ensuite dans le code généré par le compilateur. La méthode exploite alors les informations contenues dans le bloc d'appel pour mettre à jour certaines variables locales. Elle récupère en particulier l'adresse de la *primTab* qui lui permet d'accéder au système.

La sortie du code de méthode est contrôlée par le compilateur. Celui-ci génère une étiquette de sortie et transforme toute demande de retour en un déroutement à ce point. Cela lui permet d'être sûr de l'exécution de certain code.

Dans le cas d'une exécution à distance, les champs utilisés par *guideCall* sont transférés au représentant distant, qui complète lui-même la structure. Seuls les paramètres **in** et **inout** de l'appel sont transférés. Réciproquement, en fin d'exécution, le bloc d'appel local est mis à jour avec les paramètres **out** et **inout** retournés par le représentant distant. Certaines informations supplémentaires sont ajoutées au message par le protocole de communication. Cela permet en particulier le maintien de la cohérence de la variable d'activité *output*, qui référence le canal de sortie standard, entre les différents représentants de l'activité.

A.2 Vérifications syntaxiques et sémantiques

La première phase de compilation réalise l'analyse syntaxique et lexicale du programme Guide. Elle est réalisée à l'aide des outils Unix classiques Lex et Yacc. La deuxième phase effectue ensuite toute une série de vérifications quant à sa sémantique. Le compilateur est décrit plus en détail dans la thèse de Hiep Nguyen Van <Bibliographie>.

Les modifications pour le traitement des exceptions faites dans cette partie du compilateur portent essentiellement sur les vérifications sémantiques. La partie syntaxique se résume à l'ajout de quelques règles dans la grammaire du langage.

A.2.1 Erreurs

Le compilateur peut détecter deux types d'erreurs. Le plus simple, à première vue, est la violation de la règle de déclaration de l'exception avant son signalement. Le second est la violation des règles de conformité lors d'un **return** ou un **replace** dans un traitant.

A.2.1.1 Exception non déclarée

Après la première phase, le compilateur dispose d'un arbre syntaxique qui structure les informations du programme. Une méthode est un nœud qui possède en autres champs un pointeur vers une table de symboles. L'analyseur syntaxique y insère les exceptions déclarées dans la signature de la méthode, entre autres variables d'état ou de travail. Lorsque le compilateur parcourt le sous-arbre du corps de la méthode et qu'il rencontre une instruction **raise**, il vérifie que l'exception signalée est bien dans la table des symboles. Dans le cas contraire il signale l'erreur.

La table des symboles est en fait une liste chaînée de tables de symboles, qui se termine sur une table de symboles globale. Dans cette table globale le compilateur insère les exceptions système, signalables à tout moment. Les vérifications liées aux exceptions système s'insèrent donc naturellement dans l'algorithme.

Cette technique est correcte tant que l'exception est signalée depuis le corps de la méthode, ou depuis un traitant d'instruction ou de méthode. Lorsque l'instruction de signalement est dans un traitant de classe, cela ne convient plus. Nous avons suggéré en section V.2.2 une solution pour traiter ce cas. Il s'agit d'utiliser le masque auquel le traitant est associé pour déterminer les instructions susceptibles de signaler une exception acceptée par ce masque, et donc susceptibles de provoquer l'exécution du traitant. Les méthodes où ces instructions apparaissent doivent déclarer l'exception dans leur signature.

Or cette politique a deux inconvénients majeurs. Pour commencer le code à parcourir peut être très important. Il couvre bien sûr toutes les méthodes de la classe, mais aussi toutes celles de toutes ses super-classes. Mieux, il couvre également toutes les méthodes de toutes les futures sous-classes. Ce dernier point montre que l'analyse doit se faire en (au moins) deux temps, parcours des méthodes de la classe et des super-classes en liaison avec les traitants de classe définis dans la dernière classe, puis parcours des traitants des super-classes en liaison avec les méthodes de la dernière classe.

Le problème se complique encore lorsqu'il faut prendre en compte le fait que ce traitant de classe, qui signale une exception, peut être invoqué à cause de l'exécution d'un autre traitant de classe. Il faut alors itérer, ou tout au moins compliquer l'algorithme. Nous n'avons pas réalisé cette vérification, et le compilateur interdit actuellement le signalement d'une exception depuis un traitant de classe. Nous pensons que cette fonction doit se résoudre autrement que par un algorithme de vérification compliqué, et nous donnons en section VI.1 quelques idées sur la manière de procéder.

A.2.1.2 Paramètre non conforme

Lorsque le traitant exécute un **return**, le paramètre qu'il fournit (éventuellement son absence) doit être conforme au type de retour déclaré dans la signature de la méthode associée. De la même manière lorsque le traitant exécute un **replace**, le paramètre qu'il fournit doit être conforme au type de retour déclaré dans la signature de la méthode signalante. Enfin lorsque le traitant choisit la politique de terminaison simple (par défaut), le compilateur doit vérifier que la méthode signalante n'a pas déclaré de paramètre de retour (voir section V.4.1).

Pour ce qui est du **return**, le code de vérification a pu être repris du **return** classique depuis un corps de méthode. Un travail équivalent est réalisé pour les vérifications d'un traitant de terminaison, avec ou sans **replace**.

A.2.2 Avertissements

Une part importante de vérifications ne peut déboucher sur des erreurs, mais permettent la détection de situations suffisamment étranges pour mériter un avertissement du compilateur. Ces vérifications concernent la mise en correspondance des exceptions signalables, de par leur présence dans l'interface des opérations appelées, avec les traitants potentiels, repérés par leur masque associé.

Le principal problème de ces vérifications est la détermination de la portée des traitants, sous la forme d'un ensemble d'opérations invoquées. L'algorithme que nous décrivons maintenant est mis en œuvre dans la phase 3 du compilateur, car il sert également à la génération de code C. Comme il est relativement coûteux, nous n'avons pas voulu le dupliquer en phase 2 qui est la phase normale d'analyse sémantique. Nous utilisons deux variables liées, *_excUsedTypes* et *_excUsedMeths*, sous forme de pile avec un index *_excNumUsedTypes* de haut de pile, pour conserver les opérations concernées par le bloc de traitement courant. Le parcours de l'arbre commence par la classe, et nos trois variables sont mises à jour pour conserver les appels contenus dans le code des traitants de classe. Les méthodes sont ensuite analysées une à une, en remettant à chaque fois les trois variables dans cet état. Le code d'une méthode est une instruction de bloc, et à chaque instruction peut être associée un bloc de traitement dans l'arbre syntaxique. Le traitement des traitants de classe est ainsi homogène avec le traitement des traitants d'instruction.

Une instruction est traitée de la manière suivante. Les opérations invoquées dans les traitants associés sont empilées dans les variables *_excUsedTypes* et *_excUsedMeths*, qui reçoivent ensuite les opérations invoquées dans le corps de l'instruction. Ces dernières incluent les traitants associés aux instructions internes. A ce moment, nos deux variables contiennent un sur-ensemble des opérations potentiellement exécutées pendant que le traitant associé à l'instruction est armé. On prend en effet en compte toutes les opérations internes, ainsi que les opérations de tous les traitants englobants qui, on l'a vu en section V.6.1, sont dynamiquement protégés par le traitant local. Il s'agit d'un sur-ensemble car on ne tient pas compte du masque associé aux traitants englobants, masque qui réduit en quelque sorte la portée du traitant associé. La situation est illustrée par l'exemple ci-dessous, où le traitant *<traitant>* sera supposé protéger l'appel *r1.Op* de la boucle et l'appel *r2.Po* du traitant de méthode, alors que ce dernier ne risque pas d'être appelé puisque personne ne peut signaler l'exception *bourde.*

```
method Op;
    r1: ref type1;
    r2: ref type2;
begin
    while true do
        r1.Op;
    except
        erreur: <traitant>;
    end;
except
    bourde: r2.Po;
end Op;
```

```
type type1 is
    method Op;
        signals erreur;
end type1.
type type2 is
    method Po;
        signals erreur;
end type1.
```

A partir des deux variables on effectue deux tests sur les traitants associés à ce niveau. On détecte tout d'abord le traitant qui ne peut pas être appelé, simplement parce que les exceptions définies par le masque qui lui est associé ne sont signalées par aucune opération de la liste. Ce cas peut survenir à la suite d'une erreur de frappe dans le nom de l'exception donné dans le masque. Le compilateur signale alors un avertissement.

Le cas inverse est celui d'un traitant qui protège plusieurs types d'invocations. Cela est bien sûr parfaitement autorisé, mais comme notre mécanisme favorise la factorisation des

traitants au niveau de la méthode, il est possible que des factorisations involontaires soient introduites. Cela est probable en particulier si le masque se réduit à un nom d'exception. Dans ce dernier cas, le compilateur signale un avertissement, incitant le programmeur à compléter le masque du traitant, éventuellement avec le mot clef **all** si la factorisation est volontaire.

Le compilateur analyse ensuite récursivement les instructions internes à l'instruction qu'il vient de vérifier, en remettant l'index *_excNumUsedTypes* à la valeur qu'il avait juste après l'empilement des opérations des traitants de l'instruction.

A.3 Architecture du mécanisme d'exceptions

Nous allons maintenant nous intéresser à la phase 3, c'est-à-dire à la génération de code C.

A.3.1 Variable exceptionId

A.3.1.1 Nature

Tant que l'on reste au niveau des exceptions utilisateur, l'exception peut être considérée comme un paramètre de retour supplémentaire géré par le système. Elle pourrait donc très bien à première vue être gérée entièrement par le compilateur. Mais il est intéressant de donner au système la visibilité de cette variable, car cela permet d'optimiser sa gestion et de faciliter la mise en œuvre d'autres mécanises comme les exceptions système ou la liaison dynamique de la restauration.

Nous avons donc ajouté une variable *exceptionId* dans l'état de l'activité. Cette variable est gérée par le système, qui la partage entre les différentes invocations réalisées par l'activité tant qu'elle est représentée par le même processus Unix, et qui l'ajoute au message de réponse d'un appel à distance quand l'activité s'est étendue. Mais elle est aussi accessible depuis du code de méthode généré par le compilateur. Elle est déclarée comme variable globale dans la mise en œuvre du représentant sur un site d'une activité. Le code généré y accède via la table des primitives *primTab* .

Cette variable contient un identificateur d'exception et un code. L'identificateur est la chaîne de caractères qui identifie l'exception dans le source Guide. Le code prend la valeur *EXC_NO_EXCEPTION*, *EXC_SYSTEM_EXCEPTION*, ou *EXC_USER_EXCEPTION*. Il détermine la nature de l'exception signalée.

A.3.1.2 Signalement

Le signalement d'une exception utilisateur se réalise simplement par l'affectation de *exceptionId* suivie d'un déroutement au point de sortie de la méthode. Il s'ensuit un certain nombre d'actions, que nous détaillerons plus loin, puis un retour du *GuideCall* effectué au niveau appelant. La variable est alors testée et l'exception détectée.

Les informations passées dans *exceptionId* sont l'identificateur de l'exception signalée, et le code *EXC_USER_EXCEPTION*.

Le signalement dans un traitant se compile également par l'affectation de *exceptionId* , mais qui est suivie par une sortie du traitant avec demande de propagation (voir section A.3.2.1).

A.3.2 Traitant

A.3.2.1 Nature

Un traitant est compilé sous forme de fonction C. Il est chargé et relogé en même temps que sa classe de définition. Comme une procédure il accepte en paramètre d'entrée un *callBlock*, qui lui apporte l'environnement nécessaire à son exécution. Il retourne la politique de traitement choisie.

Les deux retours actuellement possibles sont *EXC_CONTINUE*, pour la politique de terminaison, ou *EXC_RETURN* pour la propagation. Le **retry** et le **replace** sont réalisés par le traitant lui-même, tandis que le **return** est une propagation sans exception (voir section A.3.3.2).

Les traitants sont compilés avant les corps des méthodes, ce qui permet d'une part de les générer dans le même fichier (les fonctions imbriquées n'existent pas en C), et d'autre part de leur donner un nom qui est utilisé dans le corps de la méthode (voir section A.3.2.2). Pour la même raison les traitants internes à un autre sont compilés avant ce dernier.

A.3.2.2 Association

L'ensemble des traitants potentiellement activables à un instant donné de l'exécution est regroupé dans la variable *_excHandlersTab* gérée sous forme de pile. L'index *_excHandlersTop* en repère le sommet. Ces deux variables sont locales à chaque fonction C compilation d'une méthode ou procédure. Elles sont générées par le compilateur.

Les entrées de cette pile ont le format C suivant :

```
typedef struct _excHandler {
    T_byte *excId;      /* nom de l'exception          */
    T_uShort modId;     /* identificateur du type      */
    T_uShort mask;      /* champ de bits               */
        /* codage de la méthode suivant ses paramètres */
    T_uLong minOpCode[NAME_SIZE_IN_LONG];   /* sans     */
    T_uLong maxOpCode[NAME_SIZE_IN_LONG];   /* maximum  */
    T_long (*code)();   /* code du traitant            */
} T_excHandler;
```

Les champs *excId*, *modId*, *minOpCode* et *maxOpCode* encodent le masque du traitant, la fonction elle-même est pointée par *code*, tandis que *mask* offre un espace de marquage. Tous ces champs sauf *mask* sont fixes et calculés à la compilation. Les valeurs **all** pour l'exception ou le nom de type sont codés par 0 dans *excId* ou *modId*, tandis que **all** pour le nom de méthode est codé dans *mask*. Un bit de mask sert également à armer ou désarmer un traitant (voir section A.3.3.1).

Chaque déclaration d'un bloc de traitement est compilée par l'empilement des traitants dans *_excHandlersTab* et la mise à jour de *_excHandlersTop* . La sortie de portée du bloc demande simplement la restauration de *_excHandlersTop* à sa valeur précédente. Pour ce dernier point le compilateur ne sauvegarde pas l'ancienne valeur mais la retrouve en décrémentant l'index du nombre connu de traitants insérés au début.

Le compilateur peut déclarer *_excHandlersTab* comme un tableau de taille fixe car il peut calculer une borne supérieure au nombre de traitants activables à un instant quelconque durant l'exécution de la méthode.

A.3.2.3 Traitants généraux

Les traitants les plus généraux sont les traitants système. Le compilateur n'en définit qu'un, qui propage l'exception système *UNCAUGHT_EXCEPTION* et qui traite toutes les exceptions. Il est placé au fond de la pile *_excHandlersTab* . Au dessus sont placés les traitants de classe, en commençant par les moins prioritaires hérités des super-classes.

Les traitants des super-classes sont recompilés dans la sous-classe à cause du codage du nom de type qui apparaît dans le masque, et de l'acceptation de types conformes à ce dernier (voir section A.3.2.5). Les traitants des sous-classes sont par contre valides dans la super-classe car le codage des types qui sont utilisés dans cette dernière peut être repris.

Cette initialisation est factorisée dans une méthode supplémentaire de la classe, *_excSetClassControlBlock* , ajoutée par le compilateur. Cette méthode est exécutée au début de chaque autre méthode ou procédure de la classe. Elle rend le nombre de traitants empilés, ce qui permet la mise à jour de *_excHandlersTop* . L'intérêt de faire une méthode de cette fonction d'initialisation est de pouvoir récupérer le mécanisme de liaison dynamique. Ainsi les traitants de la classe sont repris pour toute méthode héritée d'une super-classe. Par contre la technique n'est acceptable que dans un objectif de prototypage du mécanisme d'exceptions, puisqu'elle introduit un surcoût d'un appel de méthode pour chaque appel de méthode.

A.3.2.4 Algorithme de recherche

De par sa politique de gestion, la pile *_excHandlersTab* est naturellement ordonnée par ordre de priorité décroissante du sommet de pile vers le fond. L'algorithme de recherche consiste donc à parcourir cette pile depuis son sommet à la recherche d'un traitant armé avec un masque adéquat.

On recherche d'abord un masque sans type ou l'égalité entre type du masque et type de la référence signalante, puis un masque sans exception ou l'égalité entre les noms d'exceptions. Enfin si le bit prévu de *mask* le demande, on teste la validité du nom de la méthode signalante.

A.3.2.5 Difficultés

Le premier point concerne la mise en œuvre des traitants par des fonctions C. Cela rend actuellement impossible l'accès aux variables de travail de la méthode depuis le traitant.

Un autre aspect du modèle a beaucoup compliqué la mise en œuvre. Il s'agit de la prise en compte par un masque des exceptions signalées par un type conforme au type qu'il donne. Pour éviter un test de conformité dynamique, à supposer qu'il soit possible ce qui n'est pas le cas actuellement, nous générons une entrée dans la pile par type conforme effectivement utilisé dans le code protégé. C'est une des raisons qui imposent la recompilation dans la sous-classe des traitants hérités de la super-classe.

On peut ensuite regretter que l'exception ne soit pas codée. La comparaison de chaînes de caractères est en effet coûteuse. Mais ce point est également lié au phénomène de conformité cité au-dessus. Il faut dire tout d'abord que la duplication d'un traitant suivant les types conformes à celui donné dans le masque n'est pas réalisée quand le type est **all**. En effet dans ce cas un seul traitant est empilé, avec un code de type particulier. En

conséquence le codage d'une exception donnée doit être le même pour tous les types, passés et à venir, ce qui est fort difficile à garantir.

Reconnaître le type de la référence invoquée pose également problème, car toute information de type disparaît normalement à l'exécution. Nous avons donc rajouté cette information en la codant à partir de l'ensemble des types utilisés par la classe compilée et ses super-classes. Le code du type est l'index du type dans cet ensemble. Il est pour l'instant conservé dans la variable *_excModuleId*, locale à une méthode, qui est mise à jour avant chaque appel de méthode.

Enfin le test suivant le nom de méthode utilise les deux champs *minOpCode* et *maxOpCode*. Cela vient du fait que le nom de la méthode que l'on connaît est codé, dans le bloc d'appel, et qu'il possède un suffixe dépendant du nombre de paramètres de la méthode par rapport au nom donné dans le source Guide. Plutôt que rechercher ce nom depuis le code et faire la comparaison entre noms, nous avons préféré coder le nom donné dans le masque pour le comparer au code du bloc d'appel. Comme deux méthodes peuvent porter le même nom dans le source Guide si elles ont un nombre de paramètres différents, nous ne faisons pas un test d'égalité mais une comparaison par rapport à deux bornes.

A.3.3 Détection et traitement

A.3.3.1 Mécanisme

La détection d'un signalement d'exception se résume donc à un test bien placé de la variable globale *exceptionId*. Le compilateur doit en générer un après chaque *guideCall*, pour détecter les exceptions utilisateur.

Lorsque l'exception est détectée, le compilateur génère un appel à la fonction *_excHandleException* définie dans chaque classe. Cette fonction factorise la recherche du traitant à partir des différentes variables dont nous avons parlé jusqu'à présent, le désarme, l'exécute, le réarme, puis renvoie le code correspondant au traitement choisi.

La politique de traitement ne peut être exécutée par *_excHandleException*. En effet, la politique de propagation, code de retour *EXC_RETURN*, est facilement mise en œuvre par un déroutement à l'étiquette de sortie qui existe dans chaque méthode. Faire ce travail directement depuis *_excHandleException* serait beaucoup plus difficile.

Le test de *exceptionId* n'est pas non plus inclu dans *_excHandleException*, mais cette fois pour une raison de performance. L'exception est supposée rarement signalée ; on évite ainsi un appel de procédure inutile.

A.3.3.2 Traitements

On en vient maintenant à la mise en œuvre des différents traitements spécifiques aux exceptions.

La terminaison sans remplacement est automatique. Il s'agit d'une simple continuation après le traitement de l'exception. Quant au déroutement par **return**, cela dépend si on se trouve dans le corps de la méthode ou déjà dans un traitant. Dans le second cas, il faut propager la demande en retournant le même code *EXC_RETURN*. Dans le premier cas, on force le déroutement à l'étiquette de sortie de la méthode. Chaque méthode définit en effet une étiquette, point de passage obligé pour toute sortie de la méthode.

La propagation est mise en œuvre comme un signalement simple. Nous avons déjà vu en section A.3.1.2 qu'il s'agit d'un déroutement à l'étiquette de sortie de la méthode après une affectation de *exceptionId*. Quand le signalement a lieu depuis un traitant, *exceptionId* est affectée puis on demande la politique *EXC_RETURN* qui fait ce qui est requis.

La terminaison avec remplacement par **replace** se fait naturellement car on dispose dans le traitant d'un chemin d'accès générique à la valeur de retour de la méthode signalante. On obtient cette adresse à partir du bloc d'appel, qui est conservé inchangé pendant l'exécution du traitant.

Le réessai est également mis en œuvre à partir de ce bloc d'appel conservé en l'état pendant l'exécution du traitant. Il suffit de réexécuter un appel de méthode (*guideCall*) avec ce bloc en paramètre.

A.4 Exceptions système

A.4.1 Mécanisme

Le système utilise la même variable pour signaler une exception. Il doit bien sûr gérer la propagation de cette exception jusqu'au code utilisateur. Le compilateur doit alors générer un test, ainsi qu'une séquence de traitement pour le cas où le test est positif, de la variable après chaque appel système.

Dans le cas particulier du *guideCall*, une exception système ou une exception utilisateur peuvent être signalées. Dans le cas d'une exception utilisateur, le type de la référence signalante est donné par la variable *_excModuleId* (voir section A.3.2.5). Pour différencier le cas de l'exception système, on utilise le code qui est dans *exceptionId* (voir section A.3.1.1) et qui prend la valeur *EXC_SYSTEM_EXCEPTION*. Lorsque l'exception est détectée, la fonction *_excHandleException* (voir section A.3.3.1) en teste le code, et remet à jour la valeur de *_excModuleId* dans le cas d'une exception système pour permettre un traitement homogène dans la recherche du traitant.

A.4.2 Lien avec le compilateur

L'ensemble des exceptions signalables par le système est déclaré dans un module partagé entre système et compilateur. Cela permet à ce dernier de les prendre en compte dans les vérifications qu'il réalise en phase 2 (voir section A.2). Pour cela, il suppose qu'une exception système est signalable à tout moment.

A.4.3 Cohérence

Puisqu'il y a partage de la variable *exceptionId* entre les deux entités système et code utilisateur, il faut s'assurer de la cohérence de ses accès.

On peut tout d'abord noter que les accès à *exceptionId* ne sont pas concurrents. Le code utilisateur contrôle les appels au système et réciproquement. Le test de la variable juste après chaque appel système garantit qu'une exception utilisateur ne peut pas écraser une exception système. Réciproquement le seul appel du système à du code utilisateur

susceptible de signaler une exception est l'invocation du code effectif de la méthode dans le *guideCall*. Avant que l'exception ne soit remontée à l'utilisateur, le système exécute un certain nombre d'actions. Si dans cet intervalle une autre exception est signalée, il faut choisir laquelle est transmise à l'utilisateur, ou en signaler une troisième. Nous avons choisi la dernière solution, et nous signalons dans ce cas l'exception système *EXCEPTION_COLLISION*.

A.5 Interruptions

Une interruption est un évènement asynchrone. Elle peut survenir à tout moment durant la suite d'invocations de méthodes exécutées par l'activité. Elle recouvre en fait deux concepts, qui sont traduits en Unix par les signaux.

L'interruption est un moyen de communication entre activités. La primitive Unix *kill* permet ainsi à un processus de générer un signal dans un autre processus. C'est également l'outil utilisé par le système pour rendre compte à l'utilisateur de l'occurrence d'une exception hardware, comme le signal *SIGSEGV*d'Unix.

A.5.1 Exceptions hardware

Nous avons dit en section V.5.1 qu'une exception hardware doit être transformée en une exception système. Nous nous heurtons alors à un important problème de mise en œuvre.

Le premier problème concerne la détection de l'exception par le mécanisme de traitement. Si on adopte une mise en œuvre qui utilise la variable *exceptionId*, la détection se fait par un test. Or il est impensable d'effectuer un tel test après chaque instruction. Il faut alors envisager d'attendre la détection en un point déjà prévu, comme après un appel de méthode ou après un appel système, ou en de nouveaux points judicieusement choisis, comme lors du test d'une instruction de boucle. Mais cela soulève alors un certain nombre de questions.

Est-on sûr que l'exception sera un jour détectée, et dans quelle limite de temps ? On ne peut bien sûr rien garantir quant à une date limite de détection. Par contre on pourrait, à un certain prix, garantir la détection en un temps fini, par exemple en insérant les tests déjà cités à l'intérieur des boucles qui ne font pas d'appels système. Il restera néanmoins toujours un intervalle de temps peut-être important durant lequel l'exception sera signalée mais non détectée.

En conséquence il y a risque important de collision avec une autre exception. On peut éventuellement reprendre dans ce cas l'exception système EXCEPTION_COLLISION (voir section A.4.3).

Plus grave, l'exception hardware peut rendre absurde la continuation de l'exécution, comme c'est le cas lors de la manipulation d'une adresse invalide. Dans ce cas on risque de générer une suite d'exceptions système ou hardware, et même de créer ou propager des dommages sur les données manipulées.

Cette solution est donc peu attrayante, et il nous semble qu'il faut s'orienter vers une autre solution, avec traitement immédiat de l'exception hardware dans le respect du schéma d'exécution d'une exception système (voir section VI.3).

A.5.2 Communications asynchrones

Quant à l'utilisation des interruptions comme moyen de communication entre activités, elle est plutôt réduite en Guide. En effet, le modèle n'offre comme moyen de communication que l'utilisation d'objets partagés. La seule action directe autorisée sur une activité par une autre est la terminaison, par l'appel explicite ou implicite de la méthode *Kill* sur l'activité ou le domaine englobant. Pour ce cas particulier nous avons adopté la méthode décrite dans la section précédente, car les inconvénients qu'elle implique pour les exceptions hardware ne sont peut-être plus aussi cruciaux ici.

Il existe toutefois une différence, puisqu'il faut retrouver l'extension active de l'activité pour signaler l'exception dans cette extension. Une fois qu'elle est trouvée, le système envoie le signal Unix *SIGQUIT* au processus qui la représente. Le traitant de ce signal peut ensuite affecter la variable *exceptionId* du processus.

La demande de terminaison est donc transformée en exception système *QUIT*. La terminaison de l'activité repasse ainsi naturellement par le mécanisme standard de signalement, et en particulier par le traitement de propagation par défaut qui assure un dépilement en douceur de tous les appels de méthode en cours. Le code de restauration est donc naturellement invoqué (voir section V.9.4).

Pour ce qui est d'un temps maximum de détection, rien ne peut être garanti. Si une telle fonction est nécessaire, il faut assortir le signalement de *QUIT* de l'armement d'une alarme, et mettre en œuvre des techniques plus violentes pour terminer l'activité si l'alarme arrive à terme. C'est ce qui doit être fait actuellement dans Guide, avec comme technique plus violente l'arrêt du processus Unix qui représente l'extension active de l'activité.

Par contre, le problème de la cohérence de l'exécution entre le point de signalement et le point de détection n'existe plus.

Les problèmes viennent plutôt cette fois de la concurrence d'accès à *exceptionId*. Comme elle est manipulée depuis un traitant de signal Unix, la concurrence est réelle. Cette fois l'exception *QUIT* peut écraser, ou être écrasée par une exception utilisateur, ou une exception système. Soit l'exception *QUIT* est détectée et tout va bien, soit on détecte l'exception originale et l'exécution continue de manière cohérente, soit on détecte une exception invalide et l'exécution se poursuivra par l'exécution d'un traitant général. Dans tous les cas on ne risque pas de faire des actions incohérentes. On risque simplement de faire sonner l'alarme, et de perdre le bénéfice d'un arrêt en douceur, mais n'oublions pas que c'est dans un cas supposé rare de collision.

A.6 Restauration

A.6.1 Nature et appel

Aux deux types de code de restauration correspondent deux mises en œuvre différentes.

Un bloc de restauration de méthode est compilé comme partie intégrante de la fonction C qui représente la méthode. Ce code est placé après l'étiquette de sortie de la méthode, et son exécution est protégée par un test de *exceptionId* pour ne pas l'exécuter en cas de sortie

normale. De cette manière, il a accès aux variables de travail de la méthode, ce qui est demandé dans le modèle.

Le bloc de restauration de la classe est par contre compilé comme la clause d'activation d'une méthode, voire comme une méthode à part entière. Il s'agit donc d'une fonction qui apparaît dans la TDE de la classe. Elle n'a accès qu'à l'état de l'objet, ce qui est conforme au modèle. Elle est invoquée directement par le système, après l'appel effectif de la méthode, et si celle-ci sort en signalant une exception.

Pour permettre la compilation normale des instructions du bloc de restauration de la classe, la fonction qui le représente reprend la séquence d'initialisation d'une méthode. Elle met ainsi à jour des variables prédéfinies comme **self**, et reprend l'adresse de la *primTab*.

A.6.2 Exceptions

Certains détails doivent être mis en place pour que la compilation des exceptions dans le code de restauration reste valide.

Pour ce qui est du code de restauration de la méthode, il y a peu de choses à faire. On se trouve en effet à l'intérieur de la méthode, donc l'environnement des traitants, i.e. les variables *_excHandlersTab* et *_excHandlersTop*, est valide. Il suffit de ne pas dépiler les traitants de méthode avant l'exécution de la restauration.

Il faut ensuite contrôler la validité de la compilation des politiques de traitement. La continuation ne pose pas de problème, mais la propagation d'une exception si. Cette politique est normalement compilée par un déroutement au point de sortie de la méthode. Plutôt que d'en faire un cas particulier, nous avons préféré ajouter un contrôle au point de sortie qui vérifie si on vient normalement de la méthode ou du code de restauration. Le test se fait sur la variable booléenne *_excInMethodRestorationBlock*, mise à vrai quand on passe dans le code de restauration. C'est à ce point que l'on propage l'exception *RECOVERY_FAILED* en cas de signalement pendant la restauration.

Pour ce qui est du code de restauration de la classe, tout est à faire. On reprend donc les déclarations de variables de la méthode, ainsi que l'appel à la méthode *_excSetClassControlBlock* qui met en place les traitants de classe. Ensuite tout va bien, et le système n'a plus qu'à tester la sortie de l'appel de la fonction de restauration, et à propager *RECOVERY_FAILED* en cas d'exception.

A.7 Join

A.7.1 Traitants

Dans la mise en œuvre actuelle, la création d'une activité prend en paramètre un bloc d'appel, qui recouvre un objet, une méthode, et des paramètres. La suite d'instructions du **co_begin** idéal décrit dans la section V.9.1 est en fait transformée en une méthode cachée de la classe exécutant le **co_begin**. Il est alors facile d'ajouter dans cette méthode les traitants spécifiques à la branche, mais aussi de reprendre les traitants de la méthode mère.

Le seul travail restant consiste à remonter à la méthode mère les exceptions éventuellement signalées par les activités filles. Cela se fait naturellement car les communications entre activité mère et activité fille au sujet de l'invocation initiale et du retour des paramètres sont gérées de la même manière que lors d'une invocation à distance. Le problème du passage en retour de *exceptionId* ayant déjà été réglé pour le second cas, les mécanismes sont en place pour le premier.

A.7.2 Terminaison

L'activité mère est avertie quand une de ses filles termine. Elle récupère alors le code d'exception de celle-ci, mais n'en conserve que l'information de signalement ou non d'une exception. Le type de l'exception signalée n'est pas exploité (voir section V.9.1). L'information recueillie est conservée dans deux tableaux, qui tiennent à jour respectivement pour *finishedAct* les activités ayant correctement terminé, et pour *noCrashedAct* celles ayant terminé en signalant une exception. Les tableaux possèdent chacun une entrée pour chaque activité fille créée.

A partir de la condition de terminaison, le compilateur peut générer une fonction booléenne qui, acceptant *finishedAct* en paramètre, indique si la condition est remplie ou non. La première fonction de la condition de terminaison est donc réalisée, on peut déterminer le moment où la terminaison correcte d'une branche amène la terminaison correcte du bloc. Il reste à réaliser la contraposée, c'est-à-dire détecter le moment où la terminaison incorrecte d'une branche interdit la terminaison correcte du bloc dans le futur.

Pour cela on utilise le second tableau, *noCrashedAct*. L'information y est conservée d'une manière inverse à celle utilisée dans *finishedAct* . Dans celui-ci, une branche est marquée correctement terminée par la mise à vrai du booléen qui la représente. Dans celui-là, une branche est marquée terminée en exception par la mise à faux du booléen qui la représente. La réutilisation de la fonction de terminaison avec cette fois *noCrashedAct* en paramètre indique si le bloc peut encore terminer correctement ou non. La démonstration informelle de cette propriété est la suivante. Du fait de l'utilisation exclusive des opérateurs **et** et **ou** sans l'opérateur **not**, si la fonction de terminaison donne le résultat **vrai** sur un tableau de booléens, alors la changement dans ce tableau d'un **faux** en **vrai** ne change par le résultat. La contraposée de cette remarque, avec le fait que *noCrashedAct* représente l'ensemble des branches ayant terminé ou pouvant encore terminer correctement, donne le résultat.

Une fois la terminaison détectée, il ne reste plus qu'à terminer les branches qui s'exécutent encore, ce qui est fait par le moyen décrit en section A.5.2.

A.8 Evaluation des mécanismes

Nous allons maintenant évaluer notre mécanisme à travers trois filtres différents. Nous comparons le modèle Guide avant et après l'introduction du mécanisme. Nous évaluons les fonctions gagnées, le coût supplémentaire imposé à un programme qui n'utilise pas explicitement les exceptions, et enfin le coût de traitement des différents aspects du

mécanisme. Les points moins satisfaisants sont largement développés dans le dernier chapitre VI.

A.8.1 Fonctions

Les gains portent essentiellement sur le point principal marqué dans la section II.1.2, c'est-à-dire la séparation syntaxique et sémantique du code exceptionnel et du code de l'algorithme principal. La meilleure preuve en est que les programmes de démonstration écrits avant l'avènement du mécanisme de gestion d'exceptions tombent tous dans le piège du non traitement d'une condition d'exception détectée. Cela se voit particulièrement dans la manipulation des objets de bibliothèque, comme le catalogue, quand le retour d'une demande d'insertion n'est pas contrôlé. Beaucoup d'objets de la bibliothèque ne savent d'ailleurs pas vraiment comment rendre une condition d'exception. Le mécanisme fourni avec propagation automatique de l'exception garantit que l'exception, une fois détectée, ne peut pas être involontairement oubliée.

Le deuxième point fort est la présentation d'une exception système à travers le mécanisme standard. La solution précédemment retenue était l'arrêt pur et simple du (des) processus représentant l'activité. Maintenant l'arrêt se fait en douceur, ce qui permet au "compilateur" ainsi qu'au système d'exécuter du code de restauration.

A.8.2 Surcoût constant hors exception

Nous regardons ici ce que coûte le mécanisme lorsqu'on ne s'en sert pas explicitement dans le programme (pas d'instruction **raise**, **except**, ou **restore**). Le surcoût existe et provient essentiellement de la génération, après chaque appel système, du code de test d'exception et d'application de la politique de traitement décrit en section A.3.3.1.

Cela amène donc un surcoût en temps de compilation, d'autant plus qu'il y a, à chaque invocation de méthode, calcul du code du type de la référence manipulée.

Il y a bien évidemment une augmentation associée de la taille de l'objet classe généré, avec les coûts supplémentaires que cela implique.

Cela amène enfin un surcoût à l'exécution, d'un test de la variable *exceptionId* par appel système. Ce dernier coût, plus que les autres, doit être modulé par la remarque suivante. En l'absence d'un mécanisme d'exceptions, le programmeur doit ajouter manuellement un test après chaque appel système pour en vérifier le retour correct. Dans ces conditions, le test de *exceptionId* ne doit pas être considéré comme un surcoût.

Il reste à considérer deux mécanismes. Lors de chaque appel distant, le système transmet *exceptionId* en paramètre de retour supplémentaire. Mais pour la même raison que précédemment, on peut difficilement appeler ceci un surcoût. Enfin, chaque méthode commence par mettre en place le traitant système. Il s'agit là d'un surcoût réel.

A.8.3 Coût du traitement

On peut maintenant analyser le coût du traitement des différentes primitives du système d'exceptions, en regardant comme précédemment les différents aspects du temps passé à la compilation, de la taille de la classe générée, et du temps d'exécution.

A.8.3.1 Temps à la compilation

Le surcoût induit à la compilation est dû pour une part à l'ensemble des vérifications sémantiques décrites en section A.2. Il est important car il suppose la relecture de la partie de l'arbre syntaxique protégée par chaque bloc de traitement. Le bloc associé à la méthode impose la relecture du corps de la méthode. Le bloc associé à la classe amène lui une lecture supplémentaire du code des toutes les méthodes de la classe et de ses superclasses.

Le traitement de l'instruction de signalement est beaucoup plus raisonnable. Les vérifications associées ne demandent qu'un parcours dans la table des symboles de la méthode. De même, le traitement d'un bloc de restauration est comparable, en coût, à celui d'un bloc de code normal.

A.8.3.2 Taille de l'objet classe

Le code de la classe inclut le code des traitants qui y sont définis. C'est normal. Par contre il contient également les traitants définis dans les super-classes. Ces derniers sont donc dupliqués, ce qui est moins bien, mais les raisons en sont données en section A.3.2.3.

A.8.3.3 Temps à l'exécution

Le surcoût à l'exécution dû à l'utilisation des exceptions dans le programme mais en l'absence de signalement est concentré dans l'association des traitants. Chaque association génère la mise à jour d'une entrée de la table *_excHandlersTab* et de son index *_excHandlersTop* . Cette opération est relativement coûteuse, à cause du format de l'entrée. Elle pourrait être réduite en conservant la partie statique d'une entrée dans la classe, et en ne mettant à jour, lors de l'association, qu'un index et la partie variable de l'entrée.

Lors du signalement, la recherche du traitant demande un temps appréciable. Il faut en effet parcourir la pile *_excHandlersTab* à la recherche d'un traitant au masque adéquat. Or l'opération de vérification du masque est coûteuse, et cela est dû en partie au non codage de l'exception. Il faut aussi prendre en compte le test sur la méthode signalante, coûteux lorsqu'on le demande, mais qui demande au moins un test dans le cas contraire. Enfin il faut noter que notre choix de mise en œuvre pour traiter la conformité sur le type signalant (voir section A.3.2.5) a tendance à faire grossir la taille de la pile, et donc augmenter son temps de parcours.

Enfin, nous avons déjà parlé du coût prohibitif de la restauration de classe. Payer un appel de méthode n'est pas vraiment raisonnable, même si c'est uniquement dans les cas exceptionnels. Il faut noter que la restauration dans le cas normal induit tout de même deux tests supplémentaires, un dans la méthode pour contrôler l'exécution du code de restauration de la méthode, l'autre dans le système pour contrôler celle du code de classe.

Bibliographie

[1] E. Allman et D. Been, ''An Exception Handler for C'', *Proceedings USENIX '85*, pp. 25-45, 1985.

[2] J. Alves Marques, R. Balter, V. Cahill, P. Guedes, N. Harris, C. Horn, S. Krakowiak, A. Kramer, J. Slattery et G. Vandôme, ''Implementing the Comandos architecture'', *Proc. 5th Annual ESPRIT Conference*, pp. 1140-1157, Bruxelles. North-Holland, november 1988.

[3] T. Andrews et C. Harris, ''Combining Language and Database Advances in an Object-Oriented Development Environment'', *OOPSLA '87 Proceedings*, special issue of SIGPLAN Noticespp. 430-440, October 1987.

[4] *ANSA Reference Manual*, Architecture Projects Management Limited, 24 Hills Road, Cambridge CB2 1JP, United Kingdom, March 1989.

[5] A. Arnold, ''Systèmes de transitions finis et sémantique des processus communicants'', *Technique et Science Informatiques*, pp. 193-216, Mars 1990.

[6] R. Balter, J. Bernadat, D. Decouchant, A. Duda, A. Freyssinet, S. Krakowiak, M. Meysembourg, P. Le Dot, H. Nguyen Van, E. Paire, M. Riveill, C. Roisin, X. Rousset de Pina, R. Scioville et G. Vandôme, ''Architecture and Implementation of Guide, an Object-Oriented Distributed System'', *Computing Systems*, 4(1), pp. 31-67, 1991.

[7] J.P. Banâtre, B. Gamatie et F. Ployette, ''Traitement d'Exceptions et de Fautes Résiduelles dans les Langages de Programmation'', *RAIRO Informatique/Computer Science*, 15(1), pp. 3-38, 1981.

[8] G. Bernot, M. Bidoit et C. Choppy, ''Abstract Data Types with Exception Handling: an Initial Approach Based on a Distinction between Exceptions and Errors'', *Theoretical Computer Science*, 46(1), pp. 13-47, 1986.

[9] M. Bidoit, B. Biebow, M.C. Gaudel, C. Gresse et G.D. Guiho, ''Exception Handling: Formal Specification and Systematic Program Construction'', *IEEE Transactions on Software Engineering*, SE-11(3), pp. 242-251, March 1985.

[10] A. Black, N. Hutchinson, E. Jul et H. Levy, ''Object Structure in the Emerald System'', *ACM SIGPLAN Notices, Proc. OOPSLA*, 21(11), pp. 78-86, november 1986.

[11] D.L. Black, D.B. Golub, K. Hauth, A. Tevanian et R. Sanzi, *The Mach Exception Handling Facility*, (CMU-CS-88-129, Carnegie Mellon University, Computer Science Department, Pittsburgh, PA 15213, April 1988.

[12] A. Borgida, ''Exceptions in Object-Oriented Languages'', *ACM Sigplan Notices*, 21(10), pp. 107-119, October 1986.

[13] H. Bromley et R. Lamson, *Lisp lore: a guide to programming the Lisp machine*, second edition Boston, MA, 1987.

[14] W.F. Burger, N. Halim, J.A. Pershing, R. Strom et S. Yemini, *Draft NIL Reference Manual*, (42993), IBM, IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598, December 1982.

[15] R.H. Campbell et B. Randell, ''Error Recovery in Asynchronous Systems'', *IEEE Transactions on Software Engineering*, SE-12(8), pp. 811-826, August 1986.

[16] R. Campbell, V. Russo et G. Johnston, ''The Design of a Multiprocessor Operating System'', *Proceedings Usenix C++ '87*, pp. 109-125, 1987.

[17] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow et G. Nelson, *Modula-3 Report (revised)*, October 1989.

[18] *CHORUS Kernel V3.1 Specification and Interface*, Chorus Systèmes, April 1989.

[19] P. Coupey, *Etude d'un réseau sémantique avec gestion des exceptions : interprétation logique et implantation informatique*, Thèse, Université Paris 13, 1989.

[20] F. Cristian, *Le Traitement des Exceptions dans les Langages Modulaires*, Thèse de Doctorat, Université Scientifique et Médicale de Grenoble, 1979.

[21] F. Cristian, ''Exception Handling and Software Fault Tolerance'', *IEEE Transactions on Computers*, C-31(6), pp. 531-540, June 1982.

[22] F. Cristian, *Dependability of Resilient Computers (chapter 4, pp. 68-97)*, PSP Professional Books, Oxford England, 1989.

[23] D. Decouchant, A. Duda, A. Freyssinet, E. Paire, M. Riveill, X. Rousset de Pina et G. Vendôme, ''Guide: an implementation of the Commandos object-oriented architecture on Unix'', *Proc. EUUG Automn Conference*, Lisbon, october 1988.

[24] D. Decouchant, P. Le Dot, M. Riveill, C. Roisin et X. Rousset de Pina, ''A synchronization mechanism for an object-oriented distributed system'', *Proc. ICDCS-11*, May 1991.

[25] C. Dony, ''An Object-Oriented Exception Handling System for an Object-Oriented Language'', *Lecture Notes in Computer Science*, 322pp. 146-161, August 1988.

[26] C. Dony, *Langages à Objets et Génie Logiciel, Application à la Gestion des Exceptions et à l'Environnement de Mise Au Point*, Thèse de Doctorat, Université Paris 6, Mars 1989.

[27] C. Dony, ''Exception Handling and Object-Oriented Programming: towards a synthesis'', *ECOOP/OOPSLA '90 Proc., Sigplan Notices*, 25(10), pp. 322-330, October 1990.

[28] M.A. Ellis et B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, 1990.

[29] C. Esculier, *Introduction à la Tolérance Sémantique*, doctorat, Joseph Fourier - Grenoble 1, Juillet 1989.

[30] A. Freyssinet, *Architecture et Réalisation d'un Système Réparti à Objets*, Thèse de Doctorat, Université Joseph Fourier Grenoble 1, Juillet 1991.

[31] Goodenough, ''Exception Handling: Issues and a Proposed Notation'', *Communications of the ACM*, 18(12), pp. 683-696, December 1975.

[32] C.A.R. Hoare, ''An Axiomatic Basis for Computer Programming'', *Communications of the ACM*, 12(10), pp. 576-583, October 1969.

[33] J. Hogg et S. Weiser, ''OTM: Applying Objects to Tasks'', *ACM special issue of Sigplan Notices*, 22(12), pp. 388-393, December 1987.

[34] C.J. Horn et S. Krakowiak, ''An object-oriented architecture for distributed office systems'', *Proc. 4th Annual ESPRIT Conference*, Brussels, september 1987.

[35] J.J. Horning, H.C. Lauer, P.M. Melliar-Smith et B. Randell, ''A Program Structure for Error Detection and Recovery'', *Lecture Notes in Computer Science*, 16pp. 171-187, April 1974.

[36] J.D. Ichbiah, J.C. Heliard, O. Roubine, J.G.P. Roubine, B. Krieg-Brueckner et B.A. Wichmann, ''Preliminary Ada Reference Manual'', *ACM Sigplan Notices*, 14(6), June 1979.

[37] V. Issarny, *Le traitement d'exceptions Aspects théoriques et pratiques*, (Rapport de recherche INRIA 1118), IRISA-INRIA, Campus de Beaulieu, 35042 Rennes Cedex, Novembre 1989.

[38] V. Issarny, ''Exception Handling in Communicating Sequential Processes'', *Proc. joint conference on Vector and Parallel Processing*, september 1990.

[39] A. Koenig, Bjarne Stroustrup, ''Exception Handling for C++ (revised)'', *Proc. USENIX C++ Conference*, pp. 149-176, 1990.

[40] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill et C. Roisin, ''Design and implementation of an object-oriented, strongly typed language for distributed applications'', *Journal of Object-Oriented Programming*, 3(3), pp. 11-22, September-October 1990.

[41] S. Lacourte, ''Exceptions in Guide, an object-oriented language for distributed applications'', *Proc. ECOOP '91, Lecture Notes in Computer Science (512)*, édité par Springer-Verlag pp. 268-287, Geneva, July 1991.

[42] L. Lamport, ''The 'Hoare Logic' of Concurrent Programs'', *Acta Informatica*, 14(1), pp. 21-37, 1980.

[43] O. Lehrmann Madsen, ''Block Structure and Object Oriented Languages'', *ACM Sigplan Notices*, 21(10), pp. 133-142, October 1986.

[44] Y. Lesperance, B.M. Kramer et P.F. Schneider, *Exceptional Condition Handling in PSN*, (CSRG-112), University of Toronto, Computer Systems Research Group, Toronto, Ontario, Canada, M5S 1A1, April 1980.

[45] J. Lindskov Knudsen, ''Better Exception-Handling in Block-Structured Systems'', *IEEE Software*, pp. 40-49, May 1987.

[46] B. Liskov et A. Snyder, ''Exception Handling in CLU'', *IEEE Transactions on Software Engineering*, SE-5(6), pp. 546-558, November 1979.

[47] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J.C. Schaffert, R. Scheifler et A. Snyder, *Lecture Notes in Computer Science*, 114 CLU Reference Manual Springer-Verlag Berlin Heidelberg New York, 1981.

[48] B. Liskov, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler et W. Weihl, *Preliminary Argus Reference Manual*, October 1983.

[49] M.D. MacLaren, ''Exception Handling in PL/1'', *ACM Sigplan Notices*, 12(3), pp. 101-104, March 1977.

[50] G. Masini, A. Napoli, D. Colnet, D. Leonard et K. Tombre, *Les langages à objets*, InterEditions, 1989.

[51] B. Meek, ''Failure Is Not Just One Value'', *SIGPLAN Notices*, 25(8), pp. 80-83, August 1990.

[52] P.M. Melliar-Smith et B. Randell, ''Software Reliability: the Role of Programmed Exception Handling'', *ACM Sigplan Notices*, 12(3), pp. 95-100, March 1977.

[53] B. Meyer, *Object-Oriented Software Construction*, Series in Computer Science Prentice Hall International, 1988.

[54] B. Meyer, *Eiffel, the Language V 2.2*, Août 1989.

[55] M.L. Meysembourg-Männlein, *Modèle et Langage à Objets pour la Programmation d'Applications Réparties*, Thèse de doctorat, Institut National Polytechnique de Grenoble, Juillet 1989.

[56] *MIT Scheme Reference Manual*, Massachusetts Institute of Technology, 1991.

[57] H. Nguyen Van, M. Riveill et C. Roisin, *Manuel du langage Guide*, Bull-IMAG, Z.I. de Mayencin, 2 av Vignate, 38610 Gières, Décembre 1990.

[58] H. Nguyen Van, *Compilation et Environnement d'Exécution d'un Langage à base d'Objets*, Thèse de Doctorat, Institut National Polytechnique de Grenoble, Février 1991.

[59] *Objectworks Smalltalk-80 V2.5, Advanced User's Guide*, Parc Place Systems, 1550 Plymouth Street, Mountain View, California 94043, 1989.

[60] G. Padiou, ''Nested Coroutines for Exception Handling in Modula-2'', *Structured Programming*, 11(2), pp. 79-83, 1990.

[61] K.M. Pitman, ''Error / Condition handling. Contribution to WG16. Revision 18'', *Propositions pour ISO-LISP. AFNOR, ISO/IEC JTC1/SC 22/WG 16N15*, Avril 1988.

[62] E.S. Roberts, *Implementing Exceptions in C*, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, March 1989.

[63] C. Roisin et M. Santana, *The observer : a tool for observing distributed applications* , (Rapport Technique 8-91), Bull-IMAG, 2, rue de Vignate, ZI de Mayencin, 38610 Gières - France, Janvier 1991.

[64] P. Rovner, ''Extending Modula-2 to Build Large, Integrated Systems'', *IEEE Software*, 3(6), pp. 46-57, November 1986.

[65] V. Russo, G. Johnston et R. Campbell, ''Process Management and Exception Handling in Multiprocessor Operating Systems Using Object-Oriented Design Techniques'', *OOPSLA '88 Proceedings, special issue of Sigplan Notices*, 23(11), pp. 248-258, November 1988.

[66] C. Schaffert, T. Cooper et C. Wilpolt, *Trellis Object-Based Environment, Language Reference Manual*, (DEC-TR-372), Digital Equipment Corporation, Eastern Research Lab, Hudson, Masachusetts, November 1985.

[67] R. Seliger, ''Extending C++ to Support Remote Procedure Call, Concurrency, Exception Handling, and Garbage Collection'', *Proc. Usenix C++ '90*, pp. 241-264, 1990.

[68] A.V. Shah, J.E. Rumbaugh, J.H. Hamel et R.A. Borsari, ''DSM: An Object-Relationship Modeling Language'', *OOPSLA '89 Proceedings*, special issue of SIGPLAN Noticespp. 191-202, October 1989.

[69] C. Stirling, ''A Generalization of Owicki-Gries's Hoare Logic for a Concurrent While Language'', *Theoretical Computer Science*, 58pp. 347-359, 1988.

[70] R.E. Strom et S. Yemini, ''Typestate: A Programming Language Concept for Enhancing Software Reliability'', *IEEE Transactions on Software Engineering*, 12(1), pp. 157-171, january 1986.

[71] A. Szalas et D. Szczepanska, ''Exception Handling in Parallel Computations'', *ACM Sigplan Notices*, 20(10), pp. 95-104, October 1985.

[72] M.D. Tiemann, ''An Exception Handling Implementation for C++'', *Proc. USENIX C++ Conference*, pp. 215-232, 1990.

[73] I. Van Horebeek, J. Lewi, E. Bevers, L. Duponcheel et W. Van Puymbroeck, ''An Exception Handling Method for Constructive Algebraic Specifications'', *Software practice and experience*, 18(5), pp. 443-458, May 1988.

[74] F. Veillon et J.M. Cagnat, *Cours de Programmation en Langage PL/1 (1)*, Collection U Armand Colin, 1971.

[75] H.K.T. Wong, *Design and Verification of Interactive Information Systems Using TAXIS*, (CSRG-129), University of Totonto, April1981.

[76] L. Wong et B.C. Ooi, ''Treating Failure As Value'', *SIGPLAN Notices*, 25(1), pp. 29-32, January 1990.

[77] L. Wong et B.C. Ooi, ''Treating Failure As State'', *SIGPLAN Notices*, 25(8), pp. 24-26, August 1990.

[78] S. Yemini et D.M. Berry, ''A Modular Verifiable Exception-Handling Mechanism'', *ACM Transactions on Programming Languages and Systems*, 7(2), pp. 214-243, April 1985.

[79] S. Yemini et D.M. Berry, ''An Axiomatic Treatment of Exception Handling in an Expression-Oriented Language'', *ACM Transactions on Programming Languages and Systems*, 9(3), pp. 390-407, July 1987.

[80] A. Yonezawa, J.P. Briot et E. Shibayama, ''Object-Oriented Concurrent Programming in ABCL/1'', *Proceedings OOPSLA '86 special issue of SIGPLAN Notices*, 21(11), pp. 258-268, November 1986.

Chapitre I

lntroduction

Chapitre II

Exceptions : définitions et objectifs

Chapitre III

Les exceptions dans le modèle objet

Chapitre IV

Propositions existantes

Chapitre V

Proposition pour Guide

Chapitre VI

Développements futurs

Chapitre VII

Conclusion

Annexe A

Mise en œuvre

Exceptions dans les langages à objets

résumé

Dans les langages de programmation une exception apparaît comme conséquence des limites qu'une mise en œuvre introduit par rapport à un modèle idéal. Les structures de contrôle traditionnelles ne conviennent pas pour traiter ces cas limites, et sont secondées dans certains langages par un système spécifique de gestion des exceptions qui sépare le le traitement des exceptions de l'algorithme principal.

Nous analysons dans le contexte plus précis des langages à objets la forme que doit prendre un tel système et les contraintes qu'il doit satisfaire. Nous regardons en particulier les implications de la modularité, de l'héritage et de la conformité. Nous proposons ensuite un système de gestion des exceptions pour le langage Guide, langage à objets conçu pour la construction d'applications réparties. Nous proposons des solutions aux problèmes de la cohérence des objets et de la concurrence. Ce travail a donné lieu à une mise en œuvre sur le système Guide.

abstract

Exceptions in programming languages result from the limitations introduced by a realisation compared to the model it implements. Usual control structures are not adequate to handle these special cases and specific exception handling mechanisms have been designed which separate the handling of exceptions from the main algorithm.

We study the characteristics such a mechanism must satisfy in object-oriented languages. Relations with modularity, inheritence and conformity are developped. We then describe the mechanism we have built in Guide, an object-oriented language for the programming of distributed applications. Object consistency and concurrency issues are addressed. This design has been implemented on top of the Guide distributed system.

mots clefs

exception, langage à objets, cohérence, concurrence, déroutement, modularité