HAL
archives-ouvertes.fr

Aide à la mise au point des applications parallèles et réparties à base d'objets persistants

Hervé Jamrozik

▶ To cite this version:

Hervé Jamrozik. Aide à la mise au point des applications parallèles et réparties à base d'objets persistants. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I, 1993. Français. tel-00005129

HAL Id: tel-00005129
https://tel.archives-ouvertes.fr/tel-00005129

Submitted on 26 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THESE

présentée par

Hervé Jamrozik

pour obtenir le titre de

Docteur de l'Université
Joseph Fourier – Grenoble1

(arrêté ministériel du 5 juillet 1984 et du 30 mars 1992)

Spécialité : INFORMATIQUE

Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants

Thèse soutenue devant la commission d'examen le :

25 mai 1993

Jacques Mossiere	Président
Guy Bernard	Rapporteur
Claude Bétourné	Rapporteur
Sacha Krakowiak	Directeur de thèse
Roland Balter	Examinateur
Miguel Santana	Examinateur

Thèse préparée au sein du Laboratoire Unité Mixte Bull−IMAG

Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants

Résumé

L'objectif de ce travail est d'offrir une aide à la mise au point des applications parallèles et réparties, à base d'objets persistants, permettant une mise au point cyclique et offrant une observation de l'exécution d'un haut niveau d'abstraction.

Le non−déterminisme et la sensibilité à toute perturbation de ce type d'exécution rendent très difficile la correction des erreurs liées aux conditions d'exécution. Les limitations de l'analyse statique des programmes et des approches dynamiques fondées sur une exécution courante nous conduisent à préconiser la mise en œuvre de méthodes basées sur la reproduction d'une exécution qui apportent une solution au non−déterminisme en fixant une exécution. La mise au point s'effectue alors dans un contexte particulier où le comportement de l'exécution à corriger est déjà connu et peut être observé à l'aide de vues de l'exécution adaptées aux particularités de l'environnement d'exécution.

Nous définissons, dans le contexte des systèmes à objets, un système de mise au point basé sur la reproduction (dirigée par le contrôle) d'une exécution, permettant une mise au point cyclique et une observation de l'exécution au niveau des objets. Nous spécifions le service de réexécution, le service d'observation, et proposons une architecture modulaire pour l'assemblage des composants logiciels réalisant ces services.

Nous présentons ensuite l'application concrète des propositions précédentes au système Guide. Nous avons réalisé un noyau de réexécution, structuré en objets Guide, qui se charge de manière automatique de l'enregistrement et de la reproduction d'une exécution Guide.

Mots−clés

Mise au point, système à objets, concurrence, exécution déterministe, observation, trace, reproduction d'une exécution

Assistance on Debugging Parallel and Distributed Applications based on Persistent Objects

Abstract

Our goal is to provide a debugging assistance for parallel and distributed applications based on persistent objects which allows cyclical debugging and offers a high level of abstraction for the observation of an execution.

The nondeterminism and the probe effect of this kind of execution make bugs caused by race conditions difficult to correct. The limitations of both static analysis and dynamic approaches based on a current execution justify the choice of a debugging methodology based on replay, which solves the problem of nondeterminism by recording an execution. In this case the behavior of an incorrect execution is well known and can be visualised in specific views according to the particularities of the execution environment.

In the context of object−oriented systems, we define a debugging record/replay (control driven) system based on the objects which allows cyclical debugging and object execution observation. The reexecution and the observation services are designed and integrated in a modular architecture.

These concepts have been applied to the Guide system. A reexecuting kernel has been implemented using Guide objects, which automatically records and replays an execution.

Key words

Debugging, object−oriented system, concurrency, deterministic execution, observation, trace, execution replay.

Je tiens à remercier

Jacques Mossière, Professeur à l'Ecole Nationale Supérieure d'Informatique et de Mathématiques Appliquées de Grenoble, de me faire l'honneur de présider le jury de cette thèse.

Guy Bernard, Professeur à l'Institut National des Télécommunications, et Claude Bétourné, Professeur à l'Université Paul Sabatier de Toulouse, de l'attention qu'ils ont accordé à mon travail et de leur évaluation.

Sacha Krakowiak, Professeur à l'Université Joseph Fourier de Grenoble, de m'avoir accueilli au sein de l'équipe qu'il dirige, et de m'avoir donné la possibilité de mener à bien cette thèse.

Roland Balter, Docteur ès Sciences et Directeur de l'Unité Mixte Bull−IMAG de son constant soutien et de ses encouragements.

Miguel Santana, Docteur de l'Université Joseph Fourier de Grenoble et Ingénieur Bull de m'avoir encadré tout au long de ce travail. Je le remercie tout particulièrement pour ses encouragements, ses conseils et son amitié. Je lui dois la réussite de cette thèse.

Je tiens aussi à remercier toutes les personnes du laboratoire Bull−IMAG qui, par leurs conseils, leurs encouragements ou leur aide, ont contribué à l'aboutissement de ce travail. Je remercie tout particulièrement Fabienne et Annie, mes charmantes collègues de bureau, pour m'avoir supporté au quotidien, André, Dominique et Jean−Yves pour leur participation à la réalisation de Thésée, Miguel, Serge et Xavier pour la correction de la première version de cette thèse, Jacques et Michel pour leur lecture finale.

Je tiens aussi à remercier

Adrianaalainalexandraalfredoanaandrzejandréandréaanneanniearnaudbabethbichabéatr-icebernardbozbrigittebroutchbrunocatherinecécilecharlottechristianchristineclaudiaclaco-rinnecrevettedamousedanieldenisdidierdominiqueelianeelizabethevelineextasefabiennefab-ricefelipeflorencefrançoisfrançoisefrédéricgédéongérardhankhervéirèneisabellejacquesje-anlouisjeanyvesjoëljoëllejuankarinekatchalaurencelaurentleonlepatronlesuisselouisalucm-alcolmmarcmarcelmariemarienoëllemichelmiguelmoimêmemauricionabilnathalienickyolivi-erpascalpascalepaulpépéphilippepierrepierreyvesraymonderémirémyriquetritonrobertrolan-dsachasamersboubserchiosergeslimstéphanesylviethierryvalérievincentvirginiaxavier.

Chapitre I

Introduction

Depuis la nuit des temps, l'homme cherche à améliorer sa condition. Le développeur d'applications informatiques n'échappe pas à cette aspiration et est demandeur, depuis les débuts de l'informatique, d'une aide qui guide son travail et simplifie sa tâche. La motivation des environnements de développement est de répondre à ces besoins, par le biais d'outils qui assistent le développeur durant chacune des phases du développement d'une application.

Cette demande d'aide se fait d'autant plus pressante que les difficultés rencontrées dans le développement des programmes augmentent. C'est le cas actuellement, et le besoin de nouveaux outils offrant une assistance plus grande est largement reconnu. En particulier, les outils doivent prendre en considération les différents facteurs d'évolution des applications : croissance de leur taille, utilisation de nouveaux services d'affichage (interfaces graphiques), exécution parallèle et répartie, etc.

La **mise au point** est l'une des tâches intervenant dans le développement d'une application. L'objectif de cette phase est de localiser et corriger les erreurs contenues dans l'application développée. De l'avis de tous ceux qui ont une expérience de la programmation, cette tâche est l'une des plus délicates du développement des applications. C'est en effet une tâche pour laquelle le programmeur ne dispose pas d'une aide suffisante, ce qui lui demande un savoir−faire important pour arriver à ses fins.

L'aide la plus complète que l'on puisse offrir au programmeur consisterait en une mise au point automatique, qui trouverait la cause d'un comportement erroné sans qu'il ait à intervenir. Une telle mise au point n'est pas actuellement envisageable, au regard des difficultés rencontrées par les approches telles que l'analyse statique et la mise au point semi−automatique, limitées à la détection de certains cas d'erreurs. A l'opposé, la mise au point dynamique est fondée sur l'exploitation des informations liées à une exécution. Elle a pour objectif de donner la plus grande liberté au programmeur pour contrôler et observer l'exécution de ses programmes.

La méthode cyclique de mise au point que permet la mise au point dynamique facilite la découverte d'une erreur d'un programme en réduisant progressivement l'espace de recherche de cette erreur. Cette recherche s'effectue alors en exécutant plusieurs fois de suite le programme, et en améliorant la finesse des informations observées. L'aide apportée par la méthode cyclique de mise au point dans le cas de programmes séquentiels offre une assistance très appréciée, bien qu'elle ne soit pas entièrement satisfaisante. Cette méthode ne peut malheureusement pas être appliquée avec autant de succès au cas des applications parallèles et réparties. Le non−déterminisme de ce type d'exécution et sa sensibilité à toute

perturbation se traduisent par un comportement variable de l'exécution, ce qui rend très difficile la correction des erreurs sensibles aux conditions d'exécution.

Les vues de l'exécution qui sont offertes au développeur jouent un rôle primordial dans la mise au point cyclique : les informations montrées, ainsi que la forme de leurs présentations (textuelle, graphique, etc), aident le programmeur à mieux comprendre le comportement de son application. L'impact de l'observation est encore plus important dans le cas d'applications parallèles et réparties, où les notions manipulées sont plus nombreuses et plus complexes. L'observation doit alors permettre de suivre le comportement d'une exécution selon plusieurs critères, montrant l'évolution des processus, leurs communications, etc. Elle doit cependant rester assez souple pour ne pas noyer le programmeur sous une trop grande quantité d'informations et pour lui permettre d'aller à l'essentiel. Il doit donc disposer d'une totale liberté d'action en ce qui concerne le contrôle de la visualisation de l'exécution.

Cette thèse s'intéresse à ces deux aspects de la mise au point dans le cadre du projet Guide. Ce projet a pour objectif de développer un système réparti pour le support d'applications coopératives. Les applications visées sont parallèles, réparties et utilisent des objets persistants. Notre objectif est de permettre une **mise au point cyclique des programmes** Guide. Nous utilisons la méthode de reproduction d'une exécution pour apporter une solution au non−déterminisme de ces exécutions et reproduire fidèlement une exécution. La reproduction d'une exécution s'effectue en deux phases ("record/replay") : la première phase conserve le contexte initial et l'historique d'une exécution dite initiale, qui sont ensuite utilisés lors de la deuxième phase pour diriger les réexécutions successives et reproduire le même comportement. Les informations conservées dans l'historique d'une exécution rendent possible une **observation de l'exécution à un haut niveau d'abstraction** qui tire parti au maximum des objets pour fournir à l'utilisateur des vues très synthétiques de l'exécution, et lui permettre d'appréhender facilement le comportement de son programme.

Présentation du plan de la thèse :

État de l'art de la mise au point

Nous présentons un état de l'art des principales approches suivies par les travaux sur la mise au point des programmes.

Nous présentons tout d'abord les difficultés émanant des aspects parallèles et répartis d'une application. Nous étudions ensuite les différentes approches qui ont été suivies pour faciliter la mise au point, en fonction de leur adéquation à résoudre les problèmes posés par le parallélisme et la répartition. Nous présentons plus en détail celle basée sur la reproduction d'une exécution, car dans l'état actuel, c'est celle qui correspond le mieux à nos objectifs.

Définition d'un système de mise au point évolué

Nous présentons la traduction de nos objectifs en un système de mise au point évolué pour un environnement parallèle et réparti à base d'objets persistants.

Nous justifions tout d'abord notre choix d'un système de reproduction de l'exécution offrant une observation de l'exécution basée sur les objets. Nous présentons ensuite les principales propriétés nécessaires à la reproduction d'une exécution et à son observation. Nous présentons enfin l'organisation d'un système de mise au point complet, intégrant ces deux services.

Un metteur au point pour Guide

Nous reprenons nos propositions pour les appliquer de façon concrète au système Guide.

Après une présentation du contexte de réalisation et une étude des caractéristiques d'une exécution Guide, nous analysons les problèmes posés par la reproduction d'une exécution en Guide ainsi que par son observation. Nous présentons notamment les choix de conception que nous avons fait en ce qui concerne les services de bases nécessaires à notre système de mise au point : l'enregistrement de l'historique, la conservation du contexte initial, l'isolation de l'exécution et la reproduction de l'exécution.

Thésée : mise en œuvre du noyau de réexécution

Nous présentons une mise en œuvre du composant central de notre outil de mise au point, le noyau de réexécution, dans le cadre du système Guide.

Après une présentation générale de Thésée, en termes d'objets Guide, de services offerts et de clients potentiels, nous présentons les problèmes rencontrés et les choix de réalisation du noyau de réexécution.

Évaluation

Nous évaluons l'approche que nous avons suivi ainsi que la réalisation qui a été faite de Thésée. Nous mentionnons quelques perspectives en présentant les utilisations possibles de Thésée ainsi que des évolutions souhaitables.

Chapitre II

État de l'art de la mise au point

La mise au point est une étape importante dans le cycle de développement d'un logiciel, en raison de son rôle de correction des erreurs d'exécution, mais aussi en termes de temps consacré et d'efforts consentis par le programmeur. Nous montrons dans ce chapitre que la mise au point de programmes est une tâche difficile, dont la complexité est en rapport avec celle de l'exécution rencontrée. Nous présentons ensuite les différentes approches qui ont été suivies pour faciliter la mise au point de programmes. Nous nous intéressons plus particulièrement à celle basée sur la reproduction d'une exécution, car elle nous semble être la plus intéressante pour mettre au point les applications parallèles et réparties.

II.1 Mise au point de programmes

II.1.1 Tâche ingrate

Il est généralement admis que la mise au point est une tâche difficile qui requiert une grande activité mentale s'effectuant le plus souvent en situation de (dé) pression. Les causes de cet état de fait sont multiples, les principales étant :

- L'idée que le programmeur se fait du fonctionnement de son programme n'est pas forcément conforme à la réalité.
- Les outils de mise au point disponibles ne sont, la plupart du temps, pas adaptés aux besoins de l'utilisateur.
- Le savant mélange entre les deux modes de pensées analytique et intuitif que nécessite cette activité n'est pas facile à obtenir.
- L'intensité de l'effort produit ne se traduit pas toujours par des résultats immédiats, ce qui peut provoquer un certain découragement.
- La découverte d'une erreur d'exécution, a fortiori de plusieurs, revient à admettre que le programme qui vient d'être créé n'est pas parfait, ce qui sur le plan psychologique peut être perturbant.

Les erreurs (ou bogues) rencontrées dans les programmes sont intuitivement classées en deux catégories : les faciles et les difficiles. Les erreurs faciles représentent 80% des cas et sont localisées et corrigées en quelques heures. Alors qu'il faut compter en jours pour corriger les 20% restantes. Globalement, le programmeur passe donc plus de temps à corriger les bogues difficiles que les faciles.

La mise au point est donc une tâche difficile qui prend du temps et demande beaucoup d'efforts au programmeur. Une aide à la mise au point efficace se doit donc de faciliter la localisation des erreurs d'exécution en prenant en compte ces deux facteurs d'irritabilité du programmeur (une localisation rapide, demandant peu d'efforts).

II.1.2 Difficultés liées au type de l'exécution

Suivant la complexité de l'exécution rencontrée (séquentielle, parallèle, répartie), la mise au point d'un programme se fera avec plus ou moins de facilité. En effet, lors de la mise au point d'une exécution parallèle ou répartie, le programmeur se trouve confronté à des problèmes supplémentaires, par rapport à une exécution séquentielle, qui rendent encore plus difficile son travail. Nous considérons le cas où l'outil de mise au point n'introduit pas de comportement erroné, qu'il est lui−même au point.

II.1.2.1 Exécution séquentielle

Une exécution séquentielle est composée d'un seul processus, qui exécute en séquence une suite ordonnée d'instructions du programme, appelée flot d'exécution.

Du point de vue de la mise au point, cette exécution est caractérisée par les propriétés suivantes :

- Un état global du programme que l'on peut observer à tout moment.

  La progression de l'exécution peut être stoppée à tout instant, obtenant un état cohérent qu'il est possible d'observer.

- Un contrôle de l'exécution qui n'influe pas sur le déroulement du programme.

  L'arrêt de l'exécution ne modifie pas le comportement de la suite de l'exécution. Nous considérons le cas où les réponses des services externes utilisés par le programme (allocation de ressources, requêtes à une base de données, etc) sont stables.

- Une reproductibilité de la même exécution.

  Les instruction exécutées et leur ordre d'exécution dépendent des données initiales et sont immuables dans le cas où le programme ne comporte pas d'instruction non−déterministe (fonction génératrice de nombres aléatoires, fonction date, fonction heure, etc).

II.1.2.2 Exécution parallèle

Une exécution parallèle est constituée de plusieurs processus asynchrones qui communiquent et se synchronisent, et qui peuvent partager des ressources. Aucune supposition ne peut être faite sur la vitesse relative des différents processus. Le parallélisme est réel lorsque chaque processus s'exécute sur un processeur, il est simulé dans le cas contraire.

Les difficultés rencontrées dans la mise au point de programmes parallèles sont liées principalement à la concurrence de l'exécution des processus :

- Le comportement d'un tel programme n'est pas toujours reproductible : son exécution est souvent non−déterministe.

Le comportement de deux exécutions peut être différent pour des données initiales identiques. Il peut en effet dépendre des conditions d'exécution (ordonnancement, charge des médiums de communication, etc), qui peuvent modifier l'ordre d'exécution des opérations de communication et de synchronisation. Prenons l'exemple de deux processus concurrents dont l'un modifie la valeur d'une variable et l'autre la lit : la valeur lue par ce dernier sera différente suivant l'ordre d'exécution de ces deux processus. Ce résultat différent peut influer sur la suite de l'exécution.

Ce phénomène de variation du comportement de l'exécution rend plus difficile la correction des erreurs. La répétabilité du comportement erroné n'est plus assurée : il va se reproduire de façon aléatoire, avec une probabilité plus ou moins grande, fonction des conditions d'exécution particulières. Ces erreurs seront d'autant plus difficiles à localiser que ces conditions d'exécution seront singulières.

- Toute intervention lors de l'exécution, visant à l'observer ou à la contrôler, provoque une perturbation supplémentaire qui amplifie le phénomène de variation du comportement de l'exécution.

  On est confronté au principe d'incertitude d'Heisenberg selon lequel : "Le déroulement d'un phénomène physique qui est observé est affecté par cette observation". Ce phénomène appliqué à la mise au point de programme est connu sous le nom de "probe effect" (effet de sonde ou perturbation de l'exécution) [27].

  Toute intrusion du metteur au point peut donc modifier le cours de l'exécution, au point que le comportement erroné attendu ne se reproduise plus. La mise au point de programmes séquentiels n'est pas sensible à ce phénomène car l'ordre d'exécution des instructions est identique d'une exécution à l'autre et les changements d'état global du programme sont observables après chaque instruction.

Les causes de non−déterminisme des exécutions parallèles peuvent être de différentes natures :

- Les interactions du programme avec son environnement extérieur. Ces événements correspondent essentiellement aux entrées effectuées pendant l'exécution.
- Les opérations non−déterministes du programme, qui correspondent soit à des fonctions dont l'exécution peut rendre des valeurs différentes à chaque fois (génération de nombres aléatoires, date, heure, etc), soit à des constructions du langage dont l'exécution peut provoquer un contrôle de l'exécution différent (commandes gardées dans CSP, Ada, Estelle, etc).
- Les communications internes au programme qui sont sensibles à la concurrence d'exécution des processus. La nature de ces événements dépend des caractéristiques du système concerné : dans un système à mémoire partagée, ils correspondent à l'accès aux variables partagées, tandis que dans un système à mémoire répartie, ils correspondent aux envois et aux réceptions de messages.

Dans la suite de ce chapitre, nous considérons uniquement, lorsqu'aucune précision n'est donnée, le cas de programmes n'utilisant pas d'opérations non−déterministes. Remarquons que celles−ci peuvent être considérées comme des entrées en ce qui concerne les fonctions

non−déterministes, et comme des opérations internes au programme en ce qui concerne les constructions non−déterministes du langage.

II.1.2.3 Exécution répartie

Une exécution répartie fait intervenir plusieurs sites reliés entre eux par un système de communication. Elle est donc constituée de plusieurs processus (généralement asynchrones) qui s'exécutent sur des sites différents et qui communiquent et se synchronisent via le système de communication.

Les difficultés rencontrées dans la mise au point de programmes répartis sont liées essentiellement à l'utilisation de référentiels de temps différents sur les sites visités :

- L'observation d'un instantané de l'état global de l'exécution, comme nous l'avons défini précédemment, n'est plus possible.

  L'état global d'une exécution répartie se compose, à un moment donné, de l'état de l'exécution sur chaque site visité et de l'état du système de communication. En premier lieu, on ne sait pas définir l'état d'un système de communication à un instant précis. De plus, les sites utilisant des référentiels de temps différents, on est incapable d'arrêter l'exécution globale à un instant précis. Même avec une horloge globale exacte, il est impossible d'obtenir des informations globales concernant un ensemble de processus à un instant précis : les délais de communication, les vitesses relatives et les états différents sur les multiples sites vont provoquer l'exécution des opérations de collecte des informations globales à différents moments.

  On peut seulement avoir un état global approximatif, réalisé dans un certain laps de temps.
- Le contrôle de l'exécution globale ne peut plus être immédiat. Il peut induire de graves perturbations de l'exécution, en raison des délais de communication.

II.1.3 Influence du type de programmation

Lors de la phase de mise au point, le programmeur a envie de travailler avec les mêmes abstractions que celles utilisées lors de la phase de programmation. Il veut notamment manipuler les mêmes entités et retrouver la structure de son programme. Pour faciliter cette tâche, le metteur au point doit donc s'efforcer de faire ressortir cette structure, surtout lorsque celle−ci a disparu au cours des transformations du programme source par le compilateur et l'éditeur de liens.

Les entités que l'utilisateur a envie de manipuler diffèrent suivant le type de programmation utilisé. Dans la programmation classique (structurée et modulaire), le programmeur s'attend à manipuler des modules, des procédures, des instructions, des variables globales et locales, etc. Dans la programmation logique, ce sont des prédicats et des faits. Dans la programmation fonctionnelle, il s'attend à manipuler essentiellement des fonctions, etc.

La programmation à base d'objets apporte une grande modularité des programmes et utilise un niveau d'abstraction (l'objet), que le metteur au point se doit de montrer. Le programme est décomposé en classes qui définissent des points d'entrés et l'état des objets utilisés. Le principe d'encapsulation assure que les données d'un objet sont accessibles

uniquement par l'intermédiaire des points d'entrées définis pour l'objet. Le metteur au point, lorsqu'il montre les variables utilisées, doit aussi permettre de connaître les objets qui les contiennent. De même, un point d'entrée est défini dans une classe et s'applique sur un objet particulier.

II.2 Différentes approches

Nous présentons dans cette section les principales approches qui ont été explorées dans le domaine de la mise au point des programmes. Nous distinguons deux familles d'approches, suivant qu'elles utilisent une exécution du programme ou une analyse statique du flot des données comme support direct de l'activité de mise au point.

Parmi les approches basées sur une exécution du programme, nous distinguons des autres approches celles qui apportent une solution au problème du non−déterminisme, car elles facilitent la tâche de mise au point des programmes parallèles et répartis. Elles garantissent un comportement identique de l'exécution pendant toute la phase de mise au point en permettant de travailler sur une exécution fixée du programme, alors que les autres sont basées sur l'exécution courante du programme dont le comportement peut varier.

Nous donnons une description des différentes approches en présentant leurs points forts et leurs faiblesses. Nous tenons à signaler l'existence des travaux de synthèse de McDowell [59], Leu [50], Cheung [15], Moukeli [63] et Valot [80], qui constituent de très bonnes lectures.

II.2.1 Analyse statique des programmes

Les approches regroupées sous l'appellation d'analyse statique sont principalement fondées sur une analyse du flot des données des programmes. Elles permettent de détecter certaines erreurs dans un programme en se servant uniquement du résultat de l'analyse qui constitue une certaine abstraction du comportement du programme. Une description complète des techniques d'analyse statique peut être trouvée dans McDowell [59]

L'analyse statique est à distinguer des preuves formelles de programmes. En effet, ces dernières ont pour but de vérifier la conformité des programmes vis à vis d'une spécification de leur comportement, alors que le but de l'analyse statique est de détecter certaines catégories d'anomalies dans les programmes.

L'analyse statique de programmes séquentiels permet de détecter des cas d'erreurs connus qui sont souvent difficiles à localiser à l'exécution. Parmi ces problèmes, on peut citer ceux liés à l'utilisation de variables non initialisées, à l'utilisation de boucles, aux appels de fonctions, aux débordements de structures, etc.

L'analyse statique de programmes parallèles permet de détecter principalement deux classes d'erreurs :

- Les erreurs de synchronisation qui correspondent aux situations d'interblocage (blocage indéfini de plusieurs processus) et aux situations de famine (processus en attente infinie d'une ressource).
- Les erreurs d'accès concurrents à une même donnée.

L'analyse statique se base généralement sur la modélisation du comportement du programme sous la forme de graphes (graphes d'états, graphes d'événements ou réseaux de Pétri) qui sont construits à la compilation du code source. On obtient ainsi des graphes de contrôle pour chaque sous−programme et un graphe des appels pour le programme complet. Chaque chemin du graphe correspond à un schéma d'exécution possible du programme, c'est−à−dire à une trace des états possibles.

Les anomalies du programme sont détectées à partir de ces graphes. Les interblocages et les attentes infinies correspondent à des nœuds qui ont des tâches actives et qui n'ont pas de successeurs. Si pour chaque nœud de l'arbre, on associe aux processus les variables partagées qu'ils utilisent, la présence dans un même nœud de deux processus actifs qui possèdent une variable en commun indique une concurrence d'accès potentielle.

Le principal problème auquel est confrontée l'analyse statique de programme est l'explosion combinatoire du nombre des états des graphes engendrés. Appelbe [6] propose une solution qui consiste à regrouper certains nœuds en un état virtuel. Young [82] propose de réduire la taille de ces graphes par une exécution symbolique qui consiste à détecter les branches non utilisées du graphe du flot de contrôle en assignant des valeurs symboliques aux variables d'entrée et en les propageant.

L'analyse statique seule n'est pas suffisante pour détecter certains cas d'erreurs, repérés comme potentiels : elle doit être combinée avec une exécution du programme. C'est le cas notamment de certaines erreurs liées au partage des variables qui ne sont détectables qu'à l'exécution, comme par exemple les accès à un même élément d'un tableau ou d'une structure (accès indirects).

En conclusion, l'analyse statique n'est pas actuellement une approche entièrement satisfaisante pour la mise au point de programmes parallèles : elle ne permet de détecter que certains cas d'erreurs, et ce n'est que combinée avec d'autres outils que cette approche est intéressante.

II.2.2 Mise au point dynamique basée sur l'exécution courante

Les approches permettant une mise au point dynamique sont basées sur l'exploitation des informations liées à une exécution des programmes. Elles aident le programmeur à localiser et à corriger les erreurs qui se manifestent à l'exécution en lui permettant d'obtenir des renseignements sur l'exécution : observer son état, suivre son déroulement, etc.

Le programmeur peut obtenir ces informations soit en désignant directement les adresses mémoires utilisées par l'exécution, soit de manière plus évoluée en se servant des symboles utilisés dans son programme source. L'utilisateur désigne alors les différentes entités de l'exécution du programme de manière symbolique : la liaison symbole/adresse mémoire est réalisée par l'intermédiaire d'informations produites à la compilation (tables de symboles).

Les approches basées sur l'exécution courante du programme renseignent le programmeur à partir de l'exécution en cours. Les exécutions successives vont montrer un comportement identique dans le cas de programmes séquentiels (dont le comportement est stable pour des données en entrée identiques), mais peuvent montrer des comportements différents dans le cas des programmes parallèles et répartis.

Ces approches peuvent s'intéresser à des moments différents de l'exécution : l'observation de l'état du programme après une terminaison anormale, l'observation du comportement du programme à un moment donné de son exécution, et l'observation interactive de l'exécution.

II.2.2.1 Analyse de l'image mémoire d'un processus

Cette approche permet d'observer l'image mémoire d'un processus lorsque celui−ci s'est terminé de manière anormale (en raison, par exemple, d'un accès mémoire interdit ou de l'exécution d'une instruction illégale). Cette image ("dump") est créée par le système d'exploitation au moment de la terminaison anormale.

Cette approche possède deux inconvénients majeurs :

- Seules les erreurs graves, qui ont pour conséquence une terminaison anormale du programme, peuvent être corrigées. Ceci n'est plus vrai dans le cas particulier où le programmeur peut provoquer cette terminaison anormale de manière contrôlée.
- Rechercher une erreur à partir de l'état de l'exécution à un moment donné n'est pas facile. Les informations recueillies peuvent ne pas être pertinentes, si le moment de terminaison ne coïncide pas exactement avec celui de l'erreur (celui−ci a pu être modifié). De plus, l'interprétation de cet état et la mise en évidence de la cause de l'erreur n'est pas une tâche aisée.

II.2.2.2 Trace de l'exécution du programme

Cette approche permet de suivre le déroulement d'une exécution en surveillant une trace d'événements représentant son comportement. La localisation des erreurs s'effectue en comparant le comportement observé avec le comportement attendu du programme.

La trace de l'exécution peut être réalisée de manière simple, en ajoutant aux endroits adéquats dans le programme le code nécessaire à l'impression d'informations pertinentes (informations sur le flot d'exécution et modifications des objets). Ce service de trace peut aussi être offert par le système d'exploitation ou l'environnement de programmation (cf II.3.1.1). La trace peut être alors activée ou désactivée selon le désir de l'utilisateur.

La trace de l'exécution peut être constituée de deux types d'événements : les événements prédéfinis, dont la nature dépend des caractéristiques du système et du niveau d'abstraction choisi (accès mémoire, envois et réceptions de messages, accès aux objets, appels de procédures, etc), et ceux définis et placés dans le programme par l'utilisateur.

L'observation de l'exécution est réalisée au travers d'un affichage textuel des événements, ou de façon plus élaborée par animation d'une représentation graphique de cette trace (cf II.2.3.1).

Cette approche est intéressante pour la mise au point de programmes séquentiels. Elle peut être utilisée avec succès pour corriger les erreurs des programmes parallèles et répartis

qui ne sont pas liées au non−déterminisme de leur exécution. Pour faciliter la correction des autres cas d'erreur, il faut pouvoir observer de manière répétée une même exécution parallèle et répartie, et donc enregistrer la trace de cette exécution, qui sert alors de support aux prochaines observations. Cette approche est présentée en section II.2.3.1.

II.2.2.3 Mise au point interactive

La mise au point interactive permet de contrôler le déroulement d'une exécution (arrêt/reprise de l'exécution, exécution pas à pas) et d'observer le comportement de l'exécution du programme de façon précise (consultation et modification de son état lors d'un arrêt de l'exécution).

La mise au point interactive de programmes séquentiels ([1], [77]) se base sur les deux concepts suivants [80] :

- Le point d'arrêt ("breakpoint"), qui permet d'arrêter l'exécution à un endroit précis du programme.
- Le pas à pas, qui correspond à un mode d'exécution dans lequel le metteur au point retrouve le contrôle après l'exécution d'un pas (instruction source ou machine, appel de fonction, etc).

Elle offre généralement les services offerts par les approches précédentes, à savoir la possibilité d'observer une image mémoire d'un processus et la possibilité de tracer une exécution.

La méthode cyclique de mise au point (exécutions successives, observation et pose de points d'arrêt cernant à chaque fois l'erreur d'un peu plus près) que permet cette approche a remporté un franc succès pour la correction d'exécutions séquentielles, dû en grande partie à son efficacité (localisation rapide des erreurs, demandant peu d'efforts). Cette méthode ne peut malheureusement pas s'appliquer avec autant de succès lorsque des processus peuvent s'exécuter en parallèle ; dans ce cas, certaines erreurs ne peuvent pas être corrigées :

- Les exécutions successives du programme peuvent avoir des comportements différents (cf II.1.2).
- Les points d'arrêt ne permettent plus d'obtenir un instantané cohérent de l'état global du système (cf II.1.2).
- Le rythme d'exécution n'étant pas a priori identique pour chaque processus, la notion de pas au niveau de l'application devient imprécise. Si un processus séquentiel se déroule instruction par instruction, une application parallèle exécute plusieurs instructions au même pas (chaque processus fait un pas, l'application en fait donc plusieurs).

Différents travaux de recherche ont essayé d'apporter des réponses à ces problèmes, leur but étant d'étendre la méthode cyclique à la mise au point des programmes parallèles et répartis. Deux axes de recherches se distinguent : ceux visant à étendre le contrôle de l'exécution aux flots d'exécution parallèles, et ceux visant à étendre la notion de point d'arrêt aux exécutions parallèles et réparties. Nous présentons ci−après ces deux approches.

II.2.2.4 Mise au point traditionnelle appliquée au parallélisme

La mise au point traditionnelle appliquée au parallélisme [59] consiste à associer un metteur au point séquentiel à chaque processus parallèle. Les principales différences entre les diverses réalisations qui ont vu le jour résident dans la façon d'effectuer le contrôle de ces metteurs au point indépendants et dans la façon dont sont présentées leurs informations.

La mise au point de programmes parallèles nécessite, en particulier, de pouvoir :

1. Contrôler un processus particulier.
2. Contrôler un ensemble de processus.
3. Différencier les informations en provenance des différents processus.

Le premier et le dernier points peuvent être réalisés facilement en utilisant un metteur au point séquentiel par processus, et en utilisant un environnement multi−fenêtres tels que Suntools [64] ou X−Window [72]. Le deuxième point est alors satisfait en répétant la même commande dans les fenêtres des différents metteurs au point.

Cette solution n'est cependant qu'un palliatif au manque de metteur au point parallèle, car elle introduit une perturbation de l'exécution très importante. Il n'existe pas en effet de mécanisme de coordination entre les différentes activations du metteur au point séquentiel pour effectuer le contrôle global de l'exécution. Des commandes telles que l'arrêt et la reprise de l'exécution vont donc provoquer une perturbation de l'exécution en rapport avec le laps de temps entre l'envoi de la première et de la dernière commande aux différentes activations du metteur au point séquentiel.

Pour réduire la perturbation de l'exécution, le metteur au point pdbx [22] permet de commander les metteurs au point séquentiels à partir d'une seule fenêtre. Les commandes s'appliquent alors au processus courant ou à celui spécifié dans la commande, ou à tous les processus susceptibles d'être intéressés par cette commande. Par exemple, la commande *Continuer* s'applique à tous les processus suspendus. Griffin [30] généralise ce mécanisme en permettant de contrôler des ensembles de processus modifiables dynamiquement. Le programmeur peut ainsi appliquer des commandes alternativement à des ensembles de processus. Les commandes sont reçues par un ensemble de processus dans un délai minimum lié au temps de communication (broadcast d'une commande).

Cette approche apporte une solution simple, mais partielle, à l'extension de la méthode cyclique de mise au point aux programmes parallèles. Elle permet de contrôler, dans une certaine mesure, le cours d'une exécution et d'examiner son état, mais introduit une perturbation de l'exécution importante. Elle ne résout pas non plus les problèmes de la non reproductibilité du comportement de l'exécution, de l'arrêt de l'exécution dans un état cohérent.

II.2.2.5 Extension de la notion de point d'arrêt aux exécutions parallèles et réparties

Nous présentons ci−après les approches qui visent à étendre la notion de point d'arrêt aux exécutions parallèles et réparties. L'arrêt de l'exécution permet au programmeur d'examiner et de modifier l'état du programme (la valeur des données utilisées, l'état des processus), et

de contrôler la suite de l'exécution (exécution pas à pas, ajout de points d'arrêt, reprise de l'exécution, lancement d'une nouvelle exécution).

On peut distinguer différents éléments auxquels peuvent être associés des points d'arrêt :

- A un point précis du programme.
- A l'occurrence d'un événement, qui peut être simple ou composé, ou associés à un état particulier du programme défini par une condition [35].

La politique d'arrêt de l'exécution peut aussi être multiple [59] :

- Arrêter tous les processus participant à l'exécution du programme.
- Arrêter un seul processus ou un groupe de processus (les autres continuant leur exécution).

L'arrêt de tous les processus peut être difficile à atteindre dans un intervalle de temps réduit, et l'arrêt d'un seul processus ou d'un groupe de processus peut provoquer de graves perturbations pour les systèmes avec des contraintes de temps.

La réalisation du mécanisme de point d'arrêt est différente suivant les caractéristiques du système d'exploitation sous−jacent, et en particulier, de l'organisation de la mémoire [50] : partagée ou répartie.

Dans les architectures à mémoire partagée, les processus coopèrent essentiellement au travers de variables partagées. La définition des points d'arrêt peut inclure des conditions faisant intervenir ces variables (par exemple : 18 <= nana <= 45 et lolo > 90). La détection de tels points d'arrêt peut s'effectuer de manière simple en modifiant le code source pour ajouter, après chaque modification d'une des variables intervenant dans la condition, le test de la condition (Lewis [53]). Une façon plus évoluée de détecter ces points d'arrêt est d'utiliser des programmes parasites (Aral [8]), liés dynamiquement au programme cible, qui s'exécutent sur leurs propres processeurs et observent l'exécution du programme cible pour le suspendre lorsqu'une condition est vérifiée.

L'évaluation de ces conditions n'est pas atomique, ce qui pose le problème de la validité de l'état observé pour un point d'arrêt faisant intervenir plusieurs variables. La condition globale peut être validée alors qu'une partie de celle−ci n'est plus vérifiée.

Dans les architectures à mémoire répartie, les processus coopèrent principalement en échangeant des messages. La définition des points d'arrêt peut alors faire intervenir des événements d'émission et/ou de réception de messages s'effectuant sur des sites différents (par exemple : P1 sur S1 reçoit m1 avant que P2 sur S2 envoie m2). L'absence de temps physique global à tous les sites ne permet pas de détecter ces types de points d'arrêt. Une première façon de résoudre ce problème est d'essayer de simuler un temps physique global : certains algorithmes permettent d'obtenir une bonne approximation [40]. L'autre approche consiste à utiliser un temps logique comme support pour la détection des points d'arrêts globaux. Lamport [46] a introduit la notion d'horloge logique et la relation "<" ("happened−before") qui permettent d'obtenir un ordre total des événements de l'exécution et donc de comparer l'ordre d'occurrence des événements entre eux. Cette approche ne permet cependant pas d'exprimer la concurrence entre les événements. Fidge [25] et

Mattern [58] résolvent ce problème en définissant des horloges vectorielles qui permettent d'obtenir un ordre partiel entre les événements.

Différents travaux de recherche ont eu pour but de permettre l'utilisation des points d'arrêts en utilisant la notion de temps logique :

- Cooper [18] s'est intéressé au maintien de la cohérence du temps lors de l'utilisation de points d'arrêts dans les systèmes répartis avec des contraintes de temps. Lors de l'occurrence d'un point d'arrêt sur un site, les horloges logiques de tous les sites participant à l'exécution sont arrêtées. L'exécution des processus est ainsi stoppée, de façon immédiate pour le site initiateur, le plus tôt possible pour les autres (dû au délai de communication). L'évaluation des contraintes de temps (qui s'effectue avec le temps logique) est donc suspendue pendant l'arrêt de l'exécution, provoquant ainsi le minimum de perturbation.
- Miller et Choi [61] ont travaillé à la définition des points d'arrêt répartis en utilisant un référentiel de temps global. Ils ont proposé un algorithme de détection de l'occurrence de points d'arrêt répartis et un algorithme d'arrêt cohérent du programme pour un système avec communication par messages.

  Un point d'arrêt a été défini en termes de prédicats de différents types : prédicat simple, basé sur l'état d'un seul processus ; disjonction et conjonction de prédicats simples ; et prédicat lié qui spécifie une séquence de prédicats qui peut être ordonnée par la relation "happened−before" de Lamport.

  L'algorithme de détection d'un point d'arrêt défini par un prédicat lié nécessite que le processus réalisant la mise au point communique avec tous les processus de l'exécution. Pour détecter ce point d'arrêt, un message spécifique transportant le prédicat lié (exemple de prédicat lié : Pred1−>Pred2−>Pred3) est envoyé à tous les processus impliqués dans l'évaluation du premier sous prédicat (Pred1). Lorsque la condition associée à un processus est vérifiée, ce dernier envoie le reste du prédicat à tous les autres. Ce procédé est répété tant que le dernier prédicat n'est pas vérifié. A ce moment, le dernier processus sait qu'il faut lancer l'algorithme d'arrêt.

  L'algorithme d'arrêt cohérent proposé est dérivé de l'algorithme de Chandy et Lamport [14], qui permet de déterminer un état global cohérent dans un système réparti. L'algorithme garantit que tous les processus s'arrêtent au même temps logique, bien que les instants physiques soient différents. Un processus s'arrête soit de sa propre initiative (si il détecte l'occurrence d'un point d'arrêt), soit s'il reçoit un message d'arrêt.

  Ce type d'algorithme n'est pas efficace lorsque les processus ne sont pas fortement connectés. Dans ce cas, un processus peut être arrêté longtemps après que la décision d'arrêt ait été prise. Ce délai correspond au temps mis par le message d'arrêt pour parvenir jusqu'au processus le moins accessible. De plus, cet algorithme demande une communication qui garantisse que la réception des messages s'effectue dans le même ordre que l'émission.

L'extension de la notion de point d'arrêt aux exécutions parallèles et réparties permet d'obtenir un état cohérent lors de l'arrêt d'une exécution parallèle et répartie. Cependant, la

perturbation de l'exécution introduite par cette approche est importante. De plus, le problème de la non répétabilité de l'exécution n'est pas abordé et reste le principal inconvénient.

II.2.2.6 Autres évolutions de la mise au point interactive

En parallèle aux travaux visant à étendre la méthode cyclique de mise au point aux programmes parallèles et répartis, différentes améliorations ont été apportées à la mise au point interactive de programmes séquentiels. Nous nous limitons à les décrire sommairement ci−après, car ces travaux constituent des améliorations ponctuelles et ne concernent pas directement notre propos.

Mise au point de code optimisé

Certains travaux se sont attachés à permettre la mise au point symbolique de programmes séquentiels dont le code a été optimisé à la compilation. L'objectif est de ne plus être obligé de revenir à la version non optimisée pour rechercher la cause d'un comportement erroné. La mise au point directe du programme optimisé est réalisée par ajout d'informations à la compilation (Brooks [13]) ou par dé−optimisation dynamique (Hölzle [33]).

Filtre contextuel sur les instructions du source

Certains travaux se sont employés à simplifier l'observation du comportement du programme en fonction d'une variable particulière, à un emplacement particulier du programme ("program slicing"). Une analyse du flot des données permet de déterminer les instructions dont l'exécution peut avoir une conséquence directe ou indirecte sur la valeur de la variable à cet endroit là du programme (Agrawal [3]). Cette analyse peut être faite à partir du source ("static program slicing") ou à partir d'une exécution du programme ("dynamic program slicing").

Exécution en arrière

Certains travaux se sont intéressés à permettre le retour en arrière dans l'exécution de programmes séquentiels. L'objectif est de ne plus avoir à réexécuter entièrement un programme pour arriver à l'endroit désiré. Agrawal [2] conserve pour cela l'ordre d'exécution des instructions et pour chaque instruction les valeurs initiales des variables qu'elle modifie. Le retour arrière est effectué en remontant l'enchaînement des instructions et en réinstallant les valeurs des variables conservées.

II.2.3 Mise au point dynamique basée sur une exécution fixée

Les approches présentées dans cette section apportent une réponse au problème du non−déterminisme de l'exécution des programmes parallèles et répartis, en garantissant le même comportement de l'exécution pendant toute la phase de mise au point. Le programmeur travaille donc à la correction d'une exécution fixée, qui ne varie pas.

Nous pouvons distinguer essentiellement deux catégories de travaux : l'observation et l'analyse a posteriori d'une trace de l'exécution (appelée historique) et la reproduction d'une exécution, permettant une mise au point interactive.

II.2.3.1 Observation et analyse de l'historique d'une exécution

Cette approche permet d'observer et d'analyser une exécution particulière dont la trace d'exécution (historique) a été au préalable enregistrée. L'historique de l'exécution est ordonné pour connaître la précédence des événements entre eux.

Pour que le programmeur puisse détecter des erreurs à partir de cet historique, il faut d'une part que celui−ci contienne des informations relatives à cette erreur, d'où une tendance pour l'historique à être volumineux, et d'autre part qu'il dispose d'une aide de qualité pour analyser et observer cet historique. En effet, plus l'historique conservé est fin, plus son observation peut nous donner des renseignements sur la cause probable du comportement erroné, facilitant ainsi sa correction. Par contre, la conservation d'une grande quantité d'informations a comme inconvénient majeur de provoquer une perturbation de l'exécution importante.

Différents travaux se sont employés à diminuer le volume des informations conservées dans l'historique, pour diminuer la perturbation de l'exécution :

- Garcia−Molina [28] propose de ne tracer que les informations importantes traduisant l'activité d'un programme réparti, sous la forme d'une trace ordonnée d'événements de haut niveau. Ces événements correspondent à des actions macroscopiques faisant intervenir le système, comme par exemple la création et la destruction d'un processus, l'envoi et la réception de messages, l'allocation et la libération d'une ressource, etc. Par contre, ils ne concernent pas les actions microscopiques ou internes au programme, telles que le changement de valeur d'une variable.
- Miller et Choi [62] [17] ont introduit la technique de la trace incrémentale ("incremental tracing"). Cette technique a été tout d'abord développée pour des programmes séquentiels, puis étendue aux programmes parallèles communiquant par mémoire partagée. Elle génère dans un premier temps une trace grossière de l'exécution, qui peut être ensuite affinée lors d'une session interactive, avec l'aide d'informations produites à la compilation. L'idée de cette approche est de reporter le coût de la création de la trace de l'exécution sur les phases non critiques de compilation et de mise au point.

  Une analyse sémantique du programme permet de découper celui−ci en blocs, selon des critères de taille et de temps d'exécution probable. Au début et à la fin de chaque bloc, le compilateur insère du code pour produire à l'exécution un prologue ("prelog") et un épilogue ("postlog") contenant respectivement, les valeurs des variables susceptibles d'être lues dans le bloc, et les modifications effectuées dans le bloc. Le compilateur associe aussi une portion de code à chaque bloc, permettant d'émuler son comportement et d'obtenir, à partir de son prologue, une trace fine de son exécution.

Pour utiliser cette technique avec des programmes parallèles, il faut en plus connaître la valeur des variables partagées au moment de leur utilisation dans le bloc. Cette valeur peut en effet évoluer entre le moment de l'entrée dans le bloc et le moment de son utilisation. Cette information est conservée au moment de l'accès à la variable partagée dans la trace de l'exécution.

L'aide qui est ensuite fournie au programmeur pour observer et analyser un historique est de différentes natures. Elle consiste au minimum en un service de visualisation et d'analyse de l'historique, et de manière plus évoluée en un service d'observation du déroulement de l'exécution.

- Visualisation et analyse de l'historique.

  Ce service peut être offert de manière simple par édition de l'historique. Pour faciliter la manipulation des historiques volumineux, l'édition peut offrir des fonctions de filtrage des événements, permettant de cacher les événements jugés de moindre importance (filtrage par type d'événement et par processus) et, d'ordonnancement (temporel ou causal) des événements.

  Certains travaux facilitent l'analyse de l'historique, en le conservant dans une base de données. Snodgrass [75] et Garcia−Molina [28] conservent l'historique dans une base de données relationnelle répartie. Le programmeur utilise un langage relationnel pour créer de nouvelles relations et consulter l'historique. LeDoux et Parker [48] conservent l'historique totalement ordonné d'une exécution sous forme de prédicats Prolog, qui définissent les relations temporelles entre les événements (avant, pendant et après). L'utilisateur peut définir de nouveaux prédicats et connaître facilement, par exemple, l'ordre des accès à une variable.

  Certains travaux permettent d'analyser le comportement du programme à partir de graphes qui représentent le flot des données du programme. Choi et Miller [17] se basent sur de tels graphes pour permettre de suivre aussi bien en avant, qu'en arrière ("flowback analysis"), l'utilisation qui est faite des données. Le programmeur dispose pour cela d'un graphe des dépendances dynamique de l'exécution, généré à partir d'un graphe des dépendances statiques (construit par analyse sémantique du programme) et d'informations collectées à l'exécution.

- Observation du déroulement de l'exécution.

  Ce service est offert par l'intermédiaire de représentations graphiques qui sont alimentées par les événements de l'historique. On distingue deux types de représentations graphiques complémentaires [59] : les diagrammes de temps, qui montrent l'activité des processus en fonction du temps et guident ainsi le programmeur vers les moments de l'exécution qui l'intéressent, et l'observation de l'exécution, qui montre avec plus de détails l'état de l'exécution à un moment donné. Les diagrammes de temps sont des représentations à deux dimensions de l'évolution de l'exécution dont les axes sont le temps et les processus. Un point du diagramme donne certaines informations sur l'état d'un processus particulier à un temps donné (Griffin [30], Harter [32]). De façon complémentaire, l'observation de

l'exécution se concentre sur un instantané de l'exécution à un moment donné, et peut donc montrer plus d'informations. Celle−ci peut prendre différentes formes, adaptées aux particularités du système, et peut être animée pour montrer l'évolution de l'état de l'exécution (Hough [34], Socha [76], Jamrozik [38]).

Signalons l'approche intéressante de Eichholz [23] qui permet de suivre le déroulement d'une exécution parallèle sur le texte source du programme. Une fenêtre est associée à chaque processus et met en évidence l'instruction qui correspond à l'événement considéré. Ceci nécessite de conserver pour chaque événement une référence vers les instructions du source concernées.

La visualisation et l'analyse de l'historique d'une exécution est intéressante pour tracer l'exécution de petits programmes : elle permet de saisir le fonctionnement du programme, de détecter les erreurs de synchronisation et parfois les erreurs de conception. De plus, on peut obtenir à partir de l'historique des informations quantitatives (nombre de communications, temps d'activité des processus, etc) et qualitatives (communication effectuée, procédures exécutées, etc) sur l'exécution considérée. Par contre, l'historique généré lors de l'exécution de gros programmes devient rapidement encombrant et difficile à utiliser.

Cette approche permet de s'affranchir du problème du non−déterminisme des exécutions parallèles et réparties en permettant d'observer plusieurs fois de suite le comportement d'une même exécution. On peut arrêter l'observation dans un état cohérent en se servant des informations conservées dans l'historique (ordre des événements). La sensibilité de cette approche au phénomène de perturbation de l'exécution dépend de la méthode employée pour enregistrer l'historique (cf II.3.1.1).

II.2.3.2 Reproduction d'une exécution

La reproduction d'une exécution permet d'obtenir un comportement identique de l'exécution mise au point lors d'exécutions successives d'un programme. Les réexécutions du programme vont suivre, pour des données initiales identiques, le même cheminement de l'exécution et produire les mêmes résultats que l'exécution initiale : chaque processus participant à l'exécution va donc utiliser les mêmes données, exécuter la même séquence d'instructions et produire les mêmes résultats que l'exécution initiale.

La reproduction d'une exécution s'effectue généralement en deux phases distinctes : une première phase d'enregistrement de l'historique d'une certaine exécution dite initiale, suivie par une phase de reproduction de cette exécution. Pendant cette seconde phase les informations conservées dans l'historique sont utilisées pour contrôler les réexécutions et obtenir un comportement identique. La mise au point du programme est effectuée lors de cette deuxième phase en utilisant la méthode cyclique de mise au point.

La reproduction d'une exécution parallèle et répartie permet d'obtenir des exécutions équivalentes à celle enregistrée : les exécutions elles−mêmes ne sont pas identiques mais équivalentes, car elles reproduisent le même comportement que celui de l'exécution initiale. Les réexécutions ne peuvent pas être identiques en raison des conditions d'exécution des programmes parallèles et répartis (ordonnancement des processus, délais des communications entre sites, etc) particulières à chaque exécution. Ces conditions d'exécution, liées au

taux de charge des différentes sites, au taux d'encombrement du réseau, etc, ne sont pas reproductibles.

Les différents travaux permettant la reproduction d'une exécution se différencient par les informations conservées (le support utilisé pour enregistrer l'historique ainsi que les informations conservées, l'utilisation de photographies de l'état cohérent de l'exécution), et le type de reproduction effectuée (dirigée par les données, dirigée par le contrôle). Ces travaux sont présentés plus en détails en section II.3.

L'avantage principal de cette approche est qu'elle permet une véritable mise au point cyclique des programmes, puisqu'elle apporte une solution au problème du non−déterminisme des programmes parallèles et répartis en reproduisant des exécutions équivalentes à une exécution initiale. Par rapport à l'observation et l'analyse de l'historique, elle permet au programmeur d'obtenir plus d'informations sur l'exécution puisqu'il dispose d'une véritable exécution qu'il peut examiner dans le détail : il n'est donc plus limité aux informations contenues dans l'historique. Comme précédemment, l'historique est une source très intéressante d'informations quantitatives et qualitatives sur l'exécution.

Cette approche a pour inconvénient principal de nécessiter la conservation d'une quantité importante d'informations (l'historique), qui peut entraîner une perturbation de l'exécution. Elle nécessite par ailleurs la mise en place d'un mécanisme complexe de contrôle de l'exécution.

II.2.4 Autre approche

Parmi les autres approches intéressantes, nous tenons à signaler celles permettant une mise au point automatisée des programmes. Nous présentons ci−après la mise au point algorithmique de programmes séquentiels qui fait intervenir à la fois une analyse et une exécution du programme et qui permet une localisation semi−automatique des erreurs.

Mise au point algorithmique

La mise au point algorithmique, introduite par Shapiro [74] pour la programmation logique, est une méthode semi−automatique de détection des erreurs qui utilise des décisions du programmeur comme oracle. Cette méthode est basée sur une technique standard de trace et permet de détecter des procédures incorrectes. Elle génère une première hypothèse sur le résultat de l'appel à une procédure et demande au programmeur de la vérifier ou de la réfuter. De nouvelles hypothèses sont générées en accord avec le résultat des vérifications précédentes, pour trouver finalement la procédure incorrecte.

Le travail de Shahmehri [73] étend la mise au point algorithmique aux langages impératifs. Cette approche comporte deux phases principales :

- La première phase transforme le programme source en une représentation fonctionnelle interne, qui sera utilisée pour effectuer la mise au point algorithmique. Cette transformation est guidée par une analyse du flot de données et produit un programme intermédiaire qui ne contient plus d'effets de bord ni de boucles. Ce programme est ensuite utilisé pour construire l'arbre des appels d'une exécution particulière.

- La deuxième phase correspond à la mise au point algorithmique elle−même. Pendant cette phase, le metteur au point questionne le programmeur sur le comportement normal du programme. Les questions sont construites à partir de l'arbre des appels généré lors de la première phase. Les réponses à ces questions sont conservées dans une base de connaissances sous forme d'oracle et sont utilisées pour diriger la recherche de l'erreur et déterminer les prochaines questions.

L'utilisateur commence la mise au point de son programme en signalant un appel de procédure qu'il juge incorrecte : il précise alors les paramètres d'entrée de l'appel et son résultat. Commence alors une série de questions (oui/non) qui vont diriger la recherche pour trouver la procédure responsable de ce comportement erroné.

On peut remarquer que la précision dans la recherche de l'erreur est liée à la granularité des procédures. Plus le programme source est divisé en constructions primitives, plus la localisation des erreurs est précise.

II.2.5 Conclusion

La mise au point de programmes séquentiels à fait l'objet de nombreux travaux, qui proposent des approches complémentaires (analyse statique, mise au point symbolique, etc). Ces différentes approches offrent une aide de qualité au programmeur et facilitent dans une large mesure la correction des programmes.

Nous avons vu que la mise au point de programmes parallèles et répartis pose de nouveaux problèmes (non−déterminisme de l'exécution, perturbation de l'exécution par le metteur au point, pas d'état global cohérent, difficulté du contrôle de l'exécution, etc), qui rendent inefficaces les méthodes séquentielles et nécessitent des aménagements aux solutions existantes.

La reproduction d'une exécution est à notre avis la solution la plus intéressante pour la mise au point des programmes parallèles et répartis. D'une part, elle apporte une réponse aux problèmes posés par le parallélisme et la répartition de l'exécution. D'autre part, elle rend possible l'utilisation de la méthode cyclique de mise au point qui permet une correction efficace des programme.

II.3 Reproduction d'une exécution parallèle et répartie

La reproduction d'une exécution facilite la correction des erreurs d'un programme en permettant d'obtenir un comportement identique de l'exécution mise au point lors des exécutions successives du programme et d'appliquer donc la méthode cyclique de mise au point.

Nous avons choisi de présenter les différents travaux de ce domaine, en fonction de critères orthogonaux qui correspondent aux choix de conception du mécanisme de reproduction : informations conservées et support d'enregistrement utilisé, type de reproduction effectuée.

II.3.1 Informations utilisées pour reproduire une exécution

Les informations conservées pour reproduire une exécution sont essentiellement de deux natures : elles comprennent au minimum un historique des événements de l'exécution, qui est dans certains cas complété de photographies périodiques de l'état cohérent de l'exécution.

II.3.1.1 Historique des événements

L'historique d'une exécution conserve une trace des événements qui caractérisent une exécution. Ces événements qui sont utilisés pour reproduire une exécution correspondent :

- Aux entrées provenant de l'environnement extérieur au programme.
- Aux opérations significatives de communication et de synchronisation effectuées par le programme.

La récolte des événements d'une exécution peut être effectuée en utilisant différents supports [56] :

1. Instrumentation du code source

   La trace d'une exécution peut être obtenue en ajoutant du code aux endroits appropriés dans le source du programme.

   L'ajout de code peut être effectué de manière manuelle ou automatique. L'ajout manuel (Marzullo [57]) a pour inconvénient de demander beaucoup de temps et d'effort au programmeur et peut être la source d'erreurs supplémentaires. L'ajout automatique de code (Lumpp [52]) s'effectue par l'intermédiaire d'un langage de spécification d'événements qui est ensuite compilé. La compilation de ces spécifications ajoute automatiquement le code nécessaire pour tracer les événements ainsi définis dans le code source du programme.

   Cette approche donne une entière liberté au programmeur pour adapter la trace à ses besoins. Par contre, toute modification demande une recompilation du programme.

2. Instrumentation du code objet

   La trace d'une exécution peut être obtenue en instrumentant le code objet du programme. Le compilateur détermine, par une analyse du programme, les endroits où il doit ajouter le code nécessaire à la création de la trace (Malony [55], Pan [67]).

3. Instrumentation des primitives d'une librairie

   L'exécution du programme peut être tracée en utilisant des primitives d'une librairie particulière, qui ont été modifiées pour signaler des événements en plus de leur fonctionnement normal. Pour tracer l'exécution, il suffit de lier le programme avec cette librairie (Joyce [42], Pan [67]).

   Cette approche permet d'ajouter et d'enlever facilement la trace à un programme, en liant celui−ci avec la version correspondante de la librairie, sans nécessiter de recompilation. Par contre, les événements observables sont figés et correspondent à ceux associés aux primitives de la librairie.

4. Instrumentation du système

   L'exécution du programme peut être tracée par le système lui−même, lorsque les fonctions du système ont été modifiées en conséquence. La trace de l'exécution est alors obtenue dans un mode d'exécution particulier.

   Différentes réalisations ont été effectuées, dont la plus simple consiste à faire conserver les événements par les processus de l'application eux−mêmes (Socha [76]). Une réalisation plus intéressante consiste à dissocier la détection des événements de leur sauvegarde, qui est alors réalisée par un processus différent de ceux mis en œuvre par l'exécution (Tokuda [79]).

   Cette approche a comme avantage de ne nécessiter aucune modification du programme. Par contre, seuls les événements relatifs aux appels système sont observables.

5. Utilisation de matériel spécialisé

   La trace de l'exécution peut être obtenue en utilisant un matériel spécialisé qui se charge de détecter les événements.

   Différentes solutions ont été développées, dont la plus intéressante correspond à l'utilisation d'une machine espion. Celle−ci récupère les informations soit par observation du bus de la machine cible (Haban [31]), soit par une communication étroite avec celle−ci (Rubin [70]).

   Les solutions matérielles ont comme inconvénient d'être mises en œuvre sur des machines particulières, dédiées exclusivement à la trace. Leur coût est en effet trop important pour les généraliser à l'ensemble des machines. De plus, les informations qu'elles permettent d'obtenir sont souvent de trop bas niveau par rapport à celles qui intéressent l'utilisateur.

Dans le cas des applications parallèles et réparties, la création de la trace peut introduire une perturbation de l'exécution (cf II.1.2.2) dont l'importance dépend du support utilisé. Seul le support matériel ne provoque pas (ou très peu) de perturbation de l'exécution. Les ressources utilisées pour générer la trace sont en effet propres à l'activité de trace et sont indépendantes de celles utilisées par l'exécution.

II.3.1.2 Photographies de l'état

Une photographie de l'exécution ("snapshot") conserve un état cohérent de tout ou partie de l'exécution qui peut être utilisé comme point de reprise lors de la reproduction. Ces photographies sont prises de manière périodique lors d'une première exécution, et définissent une succession d'états cohérents. Ces points de reprise permettent de ne reproduire qu'une portion de l'exécution. La reproduction s'effectue alors en réinstallant l'état intermédiaire précédant cette portion et en contrôlant la réexécution grâce aux événements correspondants conservés dans l'historique.

Pan [67] effectue, à intervalles de temps réguliers, des photographies de l'état d'un ensemble de processus d'une exécution parallèle. La photographie d'un processus est réalisée en créant un processus image (un processus fils représentant une copie exacte de l'état du processus père) dont l'exécution est suspendue jusqu'au moment de la réexécution.

Jones [41] [19] effectue régulièrement une photographie de l'ensemble d'une exécution parallèle. Il utilise pour cela des processus TRACE, un par processus utilisateur, qui interrompent régulièrement leur exécution. La photographie d'un processus est constituée de l'ensemble des pages mémoires qu'il utilise.

L'utilisation de ces photographies permet de réduire le temps pris par la réexécution du programme pour se retrouver à un endroit précis de l'exécution. Ceci est d'autant plus intéressant, que la situation qui intéresse l'utilisateur se manifeste longtemps après le début de l'exécution.

II.3.2 Types de reproduction

La reproduction d'une exécution peut être effectuée principalement selon deux méthodes [51] : dirigée par les données ou dirigée par le contrôle. Ces deux approches ont un même objectif : fournir à chaque processus les mêmes données que lors de l'exécution initiale. La première approche retrouve ces données en les conservant de manière explicite, alors que la deuxième fait en sorte de les reproduire par l'exécution elle−même.

II.3.2.1 Reproduction dirigée par les données

Dans cette approche, la phase d'enregistrement conserve, pour chaque processus de l'exécution, la valeur des données qu'il utilise. La reproduction de l'exécution est effectuée en réexécutant le programme et en utilisant les données conservées pour nourrir les processus (Curtis et Wittie [19], Jones [41], Pan [67]).

L'historique de l'exécution peut être créé sous la forme d'un historique global ou, de manière plus simple, par processus. Il conserve pour chaque processus les données pouvant être la cause d'un comportement non−déterministe (cf II.1.2.2) :

- Les valeurs des entrées effectuées par le processus (lecture dans un fichier, etc).
- Les valeurs des communications inter−processus : le contenu des variables, dans le cas d'une communication par variables partagées, ou le contenu des messages, dans le cas d'une communication par messages.

Ces données sont utilisées lors de la reproduction à la place de celles normalement rendues par les instructions correspondantes.

Pan [67] permet de reproduire l'exécution d'un groupe de processus parallèles dans le cas d'une communication par mémoire partagée. Les interactions entre processus s'effectuent soit par l'intermédiaire d'appels au système, soit par accès à la mémoire partagée. Le résultat des appels systèmes est sauvegardé (et récupéré) dans l'historique de chaque processus par le biais d'une librairie particulière. Les accès aux variables partagées sont détectées à la compilation, par analyse du graphe des dépendances des données, et le résultat des lectures est conservé (et récupéré) par ajout d'instructions dans le code source du programme. La reproduction d'un processus s'effectue à partir d'un point de reprise (cf II.3.1.2) en réveillant le processus image associé à ce point de reprise et en dirigeant son exécution avec les informations de l'historique.

Jones [41] permet de reproduire l'exécution de programmes parallèles dans le cas d'une communication par messages. Les interactions entre processus s'effectuent

uniquement par des messages, qui sont interceptés et conservés lors de la phase d'enregistrement. Lors de la reproduction, l'utilisateur peut choisir les processus qu'il désire réexécuter, et parmi eux préciser ceux qui vont utiliser les informations conservées et ceux qui vont s'exécuter normalement. La reproduction s'effectue à partir d'un point de reprise (cf II.3.1.2), en restaurant l'état des processus concernés et en dirigeant leurs exécutions avec les informations de l'historique.

L'inconvénient de ce type de réexécution est de provoquer une perturbation de l'exécution importante, due à la conservation d'un volume important d'information.

Ce mode de reproduction possède cependant l'avantage de permettre une réexécution partielle de l'exécution : le programmeur peut reproduire une partie des processus qui constituent le programme. L'exécution de chaque processus peut être reproduite de façon indépendante du reste de l'exécution.

II.3.2.2 Reproduction dirigée par le contrôle

L'objectif de la reproduction dirigée par le contrôle (Leblanc [49], Leu [51]) est de limiter la perturbation de l'exécution engendrée par l'enregistrement de l'historique, en diminuant le volume des informations conservées. Les données liées à la communication inter−processus ne sont plus conservées dans l'historique, mais sont régénérées par l'exécution elle−même.

L'historique de l'exécution conserve, comme précédemment, les valeurs des entrées effectuées par chaque processus. Par contre, il ne conserve plus la valeur des communications inter−processus, mais l'ordre des opérations liées à ces communications : l'ordre des accès aux variables pour une communication par variables partagées , et l'ordre des envois et des réceptions de messages pour une communication par messages.

La reproduction de l'exécution est réalisée en reproduisant les mêmes événements inter−processus dans le même ordre et en utilisant les valeurs conservées pour les entrées.

Leblanc et Mellor−Crummey [49] ont introduit la technique de reproduction nommée "Instant Replay" qui n'est orientée vers aucun mécanisme de communication inter−processus, mais qui est plus particulièrement adaptée au cas des architectures à mémoire partagée. L'idée de base est de modéliser toutes les interactions entre processus comme des opérations sur des objets partagés (tout envoi de message à un processus est alors considéré comme un accès à la boite aux lettres de ce processus). A chaque objet est associé un compteur de versions, incrémenté à chaque opération sur cet objet. Une série de modifications sur un objet partagé correspond donc à une séquence totalement ordonnée de versions de cet objet. La phase d'enregistrement conserve pour chaque processus le numéro de version de chaque objet accédé. Lors de la reproduction, chaque processus attend avant d'accéder à un objet que son numéro de version corresponde à celui conservé dans son historique. La technique de l'Instant Replay nécessite la reproduction de l'exécution entière, et ne permet donc pas la reproduction isolée d'un processus. Elle ne nécessite pas par contre l'utilisation d'un référentiel de temps global, puisque la notion de temps est remplacée par celle de séquence d'événements.

Leu et Schiper [51] proposent un ordonnancement des événements qui soit plus adapté au cas des architectures à mémoire répartie avec communication par messages.

L'adaptation de la proposition précédente à ce type d'architectures est pénalisée en termes de performances, parce qu'elle demande d'une part la gestion du partage d'une variable (compteur de version) entre processus situés sur des sites différents, et parce qu'elle nécessite d'autre part l'emploi de messages supplémentaires pour récupérer le numéro de version, dans le cas ou le processus destinataire est sur un autre site. Leu et Schiper proposent d'associer les numéros de versions aux événements d'un processus. Ils font la distinction entre les événements explicites, correspondant aux instructions exécutées par le processus, et les événements implicites, qui correspondent à l'émission et à la réception effective d'un message. L'historique d'un processus conserve uniquement les événements implicites, sous la forme d'un numéro de séquence et du type de l'événement, car ceux−ci définissent effectivement la communication inter−processus.

La reproduction dirigée par le contrôle nécessite la réexécution complète de l'exécution (la reproduction d'une partie de l'exécution n'est plus possible). Elle est aussi plus complexe à mettre en œuvre, car elle nécessite d'une part un ordonnancement des événements de l'exécution et d'autre part demande un contrôle de la réexécution complète du programme.

Le principal avantage de la reproduction dirigée par le contrôle est de diminuer la perturbation de l'exécution introduite par la conservation de l'historique.

II.3.2.3 Reproduction spéculative

La reproduction spéculative (Stone [78]) permet de reproduire le comportement de l'exécution initiale par approximations successives. Elle ne garantit pas un comportement identique lors de la reproduction, mais permet à l'utilisateur de modifier le cours des réexécutions pour obtenir finalement le même comportement. Les modifications apportées par l'utilisateur sont censées lui permettre de se faire une idée précise sur la cause du comportement erroné.

L'historique conserve les événements correspondant aux dépendances identifiées par une analyse préalable du programme, qui partitionne le programme en blocs sérialisables et en blocs concurrents et détecte les dépendances inter−processus probables. Les dépendances de l'exécution initiale sont présentées au programmeur sous la forme d'un tableau de concurrence ("concurrency map") et sont utilisées pour contrôler le bon déroulement des réexécutions. Un point d'arrêt est posé au début de chaque bloc intervenant dans une dépendance inter−processus, pour lui permettre d'attendre la fin de son bloc prédécesseur.

Si une réexécution diverge (elle s'écarte de l'exécution enregistrée), elle est arrêtée. L'utilisateur doit alors intervenir pour rajouter de nouvelles dépendances inter−processus afin d'empêcher cette divergence. L'analyse statique du programme ne permet pas en effet de déterminer toutes les dépendances inter−processus existantes, notamment celles non explicites qui peuvent être la cause de comportements erronés. L'utilisateur rajoute de nouvelles dépendances de temps à la table de concurrence pour forcer la reproduction de l'exécution initiale.

II.4 Conclusion

Les caractéristiques des programmes parallèles et répartis rendent difficile leur mise au point dynamique : leur exécution est non−déterministe ; toute intrusion dans l'exécution provoque une perturbation de l'exécution ; et les opérations de contrôle global de l'exécution, telles que l'arrêt de l'exécution, ne sont plus immédiates.

Les différentes approches suivies pour la mise au point des programmes séquentiels s'adaptent avec plus ou moins de bonheur au cas des programmes parallèles et répartis : l'analyse statique des programmes parallèles est limitée dans le spectre des erreurs qu'elle permet de corriger ; la mise au point dynamique ne souffre pas de ce handicap, mais est confrontée aux problèmes liés à l'exécution de ces programmes.

Les approches dynamiques basées sur une exécution courante du programme ne résolvent pas le non−déterminisme de ces exécutions, dont le comportement peut varier d'une exécution à l'autre, et provoquent une perturbation importante de l'exécution par leurs actions intrusives liées à l'observation de l'exécution. Signalons quand même l'approche qui vise à étendre la notion de point d'arrêt de l'exécution et qui permet d'observer des états cohérents d'une exécution parallèle et répartie.

Les approches dynamiques basées sur une exécution fixée garantissent le même comportement de l'exécution pendant toute la phase de mise au point, et apportent ainsi une solution au non−déterminisme de l'exécution des programmes parallèles et répartis. L'observation et l'analyse de l'historique d'une exécution permettent de corriger dans des conditions confortables uniquement des programmes dont la taille reste modeste ; de plus, les renseignements que l'on peut obtenir sur l'exécution sont limités. La reproduction d'une exécution est à notre avis l'approche la plus intéressante car elle met en œuvre une véritable exécution, que le programmeur peut examiner à loisir, et permet une mise au point cyclique des programmes.

La reproduction d'une exécution peut s'effectuer principalement selon deux méthodes : une reproduction dirigée par les données ou une reproduction dirigée par le contrôle. La deuxième méthode est plus complexe à mettre en œuvre que la première, mais provoque une perturbation de l'exécution moins importante, du fait qu'elle conserve moins d'informations pour diriger la reproduction.

Nous pensons que la reproduction de l'exécution dirigée par le contrôle est l'approche la plus intéressante pour la mise au point des programmes parallèles et répartis, pour toutes les raisons présentées ci−dessus. Nous ne voulons pas utiliser de photographies de l'exécution pendant la phase d'enregistrement, car le coût de ce mécanisme est important et anéantirait tous les efforts effectués jusqu'à présent pour diminuer la perturbation de l'exécution. Le support idéal pour enregistrer l'historique de l'exécution est bien entendu un support matériel, car c'est celui qui introduit la plus faible perturbation de l'exécution. L'inconvénient de ce type de support est d'une part le bas niveau des informations obtenues (adresses mémoires) et d'autre part son caractère exceptionnel, dû à son coût de réalisation qui fait qu'en pratique, il n'est pas disponible sur l'ensemble des machines.

Chapitre III

Définition d'un système de mise au point évolué

Nous avons présenté au chapitre précédent l'état actuel des principales propositions dans le domaine de la mise au point. Nous présentons dans ce chapitre notre conception d'un système de mise au point d'applications parallèles, réparties et à base d'objets persistants.

Nous détaillons tout d'abord les objectifs de notre travail pour analyser ensuite les deux principaux services que doit offrir un système de mise au point parallèle et répartie : la reproduction d'une exécution et une observation simplifiée de l'exécution. Nous présentons ensuite l'organisation de notre outil de mise au point et terminons par une évaluation de notre approche.

III.1 Objectifs

L'objectif de notre travail est de permettre une mise au point plus simple et plus confortable des programmes dans un environnement parallèle, réparti et à base d'objets persistants. Notre intérêt s'est porté sur une mise au point interactive basée sur une observation de l'exécution à un haut niveau d'abstraction, offrant à l'utilisateur des vues très synthétiques du déroulement de cette exécution.

Nous montrons dans la suite comment nous avons procédé pour atteindre notre objectif en tenant compte des particularités de notre environnement.

III.1.1 Observation de l'exécution basée sur les objets

III.1.1.1 Importance de l'observation

Toute décision se base sur un ensemble d'informations. La pertinence d'une décision est relative à la quantité et à la qualité des informations disponibles au moment de la prise de cette décision. Notre propos n'est pas de disserter sur le minimum d'informations nécessaires pour prendre une décision appropriée, mais d'insister sur le rôle central de l'information dans ce processus qui est l'aboutissement de l'étape d'analyse et de compréhension d'un problème.

L'observation en général est un moyen pour obtenir cette information et certainement le meilleur pour appréhender un phénomène dynamique tel que l'exécution d'une application. L'observation d'une exécution consiste essentiellement en la visualisation de ses composants, processus et données, de leur évolution et de leurs interactions. Une bonne

compréhension du comportement de l'exécution ne peut que faciliter la découverte d'une erreur et ainsi faciliter sa correction. Encore faut−il que les informations montrées soient pertinentes !

La démonstration d'applications et la mise au point sont deux domaines pour lesquels la qualité de l'observation est primordiale. Dans les deux cas, les utilisateurs potentiels s'intéressent au fonctionnement interne de l'application et à ses détails : choix de base, structures de données centrales, flots d'exécution, etc. Ces exigences correspondent à un niveau élaboré de l'observation dont la difficulté dépend du type de programmation utilisé :

- Si l'application a été développée dans un langage séquentiel traditionnel, la description du programme peut s'avérer dans certains cas suffisante pour sa compréhension. Cependant, l'observation de son exécution peut accélérer cette compréhension et même être d'un grand secours dès que le programme dépasse une certaine taille.
- L'observation d'une exécution parallèle est indispensable pour saisir la finesse de son comportement. Le parallélisme est un aspect critique de l'exécution qu'il est très intéressant d'illustrer avec un outil adapté, pour montrer, par exemple, l'évolution des branches parallèles, leurs interactions et leur synchronisation (mécanisme de rendez−vous, communication synchrone ou asynchrone, etc).
- Si l'exécution est répartie sur différentes sites, son observation peut être plus complexe mais néanmoins très riche en renseignements. Par exemple, celle−ci peut montrer la localisation des objets et leur migration en cours d'exécution, même si le système est construit pour justement cacher ces détails à l'utilisateur.
- Si l'application a été développée dans un langage orienté objet, il est sans aucun doute intéressant de montrer les objets utilisés par son exécution et d'observer tout particulièrement leurs interactions.

En conclusion plus l'application est complexe, plus il est essentiel de fournir une observation qui facilite la compréhension de l'exécution de l'application.

III.1.1.2 Différents niveaux d'abstraction

Le manque d'information comme le surplus d'informations sont préjudiciables à une bonne compréhension de l'exécution observée. Dans un cas, on ne comprend pas ce qui se passe. Dans l'autre, on est submergé par la quantité de renseignements montrés, rendant difficile voire impossible l'extraction des informations importantes.

Pour ne pas subir ces inconvénients, l'observation de l'exécution doit s'effectuer avec un niveau d'abstraction adapté au but recherché. Trois niveaux d'abstraction peuvent être choisis :

1. **Processus/Flot de contrôle**

   Une observation au niveau des processus offre une vue très générale du comportement d'une application, qui permet de comprendre les relations macroscopiques d'un programme avec son entourage (environnement, système, etc) et les relations entre les processus qui composent le programme (création, terminaison, etc). Elle met en évidence la structuration de l'exécution en termes de flots de contrôle.

Cette observation ne donne cependant pas la possibilité de suivre avec précision le déroulement de l'exécution, car elle ne montre pas les opérations exécutées.

2. **Procédure/Opération**

   Une observation au niveau des opérations dans un langage orienté−objet, ou des procédures dans un langage procédural, offre une visualisation macroscopique de l'exécution et permet donc de suivre l'exécution d'un programme dans ses grandes lignes. Elle permet notamment de déterminer l'opération (ou la procédure) dont l'exécution provoque le comportement erroné.

   Cette visualisation ne donne cependant pas la possibilité d'observer l'exécution dans ses détails, car elle ne montre pas les instructions exécutées.

3. **Instruction**

   Une observation au niveau des instructions du langage, ou au niveau de leur traduction en instruction de la machine hôte, offre une visualisation microscopique de l'exécution et permet de comprendre les plus intimes détails de celle−ci. Elle permet de suivre le déroulement de l'exécution de manière exhaustive et fournit donc une vue de l'application très descriptive qui facilite la correction des erreurs.

   Cependant, la quantité d'informations visualisées par cette observation devient rapidement trop importante dès que l'application dépasse la taille du simple exemple de programmation. Le programmeur se retrouve submergé par une foule de renseignements qui ne lui permettent pas d'appréhender le comportement global de son application.

Nous remarquons que ces trois niveaux d'observation rendent des services complémentaires pour la mise au point d'une application. Les conditions d'observation optimum sont donc réunies lorsque le programmeur dispose à la fois de l'observation des processus, des opérations et des instructions. Il peut alors choisir le niveau d'abstraction le mieux adapté aux besoins d'observation du moment.

III.1.1.3 Pourquoi les objets

Nous proposons d'utiliser pour mettre au point une application la même abstraction que celle utilisée lors de sa conception et lors de son exécution. Nous proposons donc que l'utilisateur travaille à la mise au point de son application en manipulant des objets, puisque l'objet est une entité structurante permettant un haut niveau d'abstraction des applications visées, en ce qui concerne leur conception et leur exécution.

L'observation de l'exécution basée sur les objets permet de voir les objets utilisés, et est le support tout naturel pour suivre le déroulement de l'exécution au niveau des opérations. L'utilisateur peut ainsi suivre l'enchaînement des opérations sur les objets, mais aussi voir la synchronisation des accès aux objets partagés, accéder à l'état des objets, etc.

L'objet permet une observation macroscopique de l'exécution qui joue un rôle central dans la mise au point. Il apporte une réponse au problème de la granularité de l'information observée en permettant d'observer une exécution dans ses grandes lignes, sans se perdre dans ses détails.

Le niveau microscopique de l'observation correspond à l'observation basée sur les instructions internes aux opérations. Cette observation ainsi que celle plus générale des activités doivent être disponibles pour compléter efficacement l'observation basée sur les objets que nous proposons.

III.1.2 Mise au point cyclique

La mise au point cyclique est la méthode de mise au point la plus utilisée à l'heure actuelle. Elle est bien ancrée dans les habitudes des programmeurs, et doit de ce fait être prise en compte lors de la conception d'un metteur au point.

Nous soulignons tout d'abord l'intérêt de cette méthode. Nous étudions ensuite les problèmes posés par son utilisation dans un environnement parallèle, réparti et à base d'objets persistants.

III.1.2.1 Intérêt de la mise au point cyclique

Le succès rencontré par la méthode cyclique de mise au point s'explique tout d'abord par le fait qu'elle permet de réduire progressivement l'espace de recherche d'une erreur, et facilite ainsi la découverte de la cause directe du comportement erroné d'un programme. Le programmeur peut, à partir d'une approche globale de l'exécution, affiner sa compréhension du problème pour finalement concentrer son attention sur une particularité de l'exécution.

De plus, lorsque cette cause est elle−même la conséquence d'erreurs en cascade, le programmeur utilise la même méthode pour remonter la chaîne des conséquences, trouver l'erreur initiale et la corriger.

III.1.2.2 Résoudre le non−déterminisme d'une exécution parallèle

Nous avons vu en section II.1.2.2 que le principal problème à résoudre dans la mise au point de programmes parallèles est celui du non−déterminisme de l'exécution de ces programmes. Ce phénomène rend difficile la correction des erreurs car les exécutions successives d'un programme peuvent avoir des comportements variables. On voit donc apparaître un facteur aléatoire dans la reproduction du comportement erroné de l'exécution.

Parmi les techniques proposées pour la mise au point de tels programmes, l'Instant Replay [49] présenté en section II.3.2.2 nous a semblé convenir le mieux à nos objectifs. Cette technique utilise un mécanisme d'exécution à deux phases : la première phase enregistre des informations sur l'exécution dans un historique qui est utilisé lors de la deuxième phase pour reproduire la même exécution.

Cette approche possède deux avantages importants :

- Elle permet de corriger un programme en situation réelle, puisqu'elle se base sur une exécution de l'application. L'utilisateur peut donc demander, à tout moment de l'exécution, un complément d'information, pour mieux comprendre les tenants et les aboutissants d'un comportement erroné.
- Elle permet d'utiliser la méthode classique de mise au point cyclique qui est d'un grand confort d'utilisation, puisqu'elle permet de reproduire une exécution.

L'utilisateur peut répéter la même exécution plusieurs fois de suite, sans être gêné par le non−déterminisme de son programme.

Il est à noter que plusieurs enregistrements de l'exécution de l'application peuvent se révéler nécessaires pour obtenir l'historique d'une exécution contenant une erreur aléatoire. Ce genre d'erreur est inhérent au non−déterminisme des applications parallèles et réparties. Par contre le plus difficile est fait lorsque l'historique désiré est obtenu, le programmeur peut alors prendre tout son temps pour corriger cette erreur.

Le principal inconvénient de cette méthode est lié à la perturbation engendrée par l'enregistrement de l'historique. Celle−ci doit être minimale pour que cette méthode soit utilisable (cf III.2.2). Le coût de la conservation d'information, essentiellement lié au coût du mécanisme de conservation utilisé, évolue aussi en fonction du volume d'information à conserver. Le mécanisme de collecte de l'information doit donc non seulement être efficace mais également conserver le minimum d'informations suffisant à la reproduction.

III.1.2.3 Tenir compte de la répartition de l'exécution

La répartition complique la reproduction d'une exécution. En effet une exécution répartie se déroule sur plusieurs sites, mettant en jeu différents facteurs propres à influencer le cours de cette exécution : les communications, les types de machines, la présence de certaines ressources sur ces sites, etc.

Pour obtenir le même comportement, les réexécutions d'un programme doivent suivre exactement le chemin emprunté à travers le réseau par l'exécution enregistrée. Ce cheminement est une caractéristique importante de l'exécution.

La phase d'enregistrement doit donc être capable de suivre les opérations à distance et d'enregistrer les portions d'exécution qui se déroulent sur les différents sites visités. La phase de reproduction doit alors reproduire l'exécution enregistrée sur les sites correspondants.

III.1.2.4 Tenir compte de la persistance des objets

L'exécution d'un programme est en grande partie dirigée par les données utilisées, et plus particulièrement par la partie persistante de ces données. Nous appelons dans la suite **contexte d'exécution** l'ensemble des données utilisées par une exécution.

La réexécution d'un programme nécessite d'assurer que le contexte d'exécution soit exactement le même entre deux exécutions. Les valeurs des données persistantes doivent donc être remises à leur valeur initiale avant de lancer une réexécution.

Dans un système traditionnel, la tâche de conservation et de réinitialisation du contexte est effectuée en général manuellement. L'ensemble des données concernées se limite aux variables internes du programme, qui sont réinitialisées par le cours de l'exécution, et aux fichiers manipulés par celui−ci, qui eux sont clairement identifiables et facilement réinitialisables.

Dans notre environnement, cette tâche de conservation et de réinitialisation du contexte ne peut s'effectuer que de façon automatique. Il nous semble, en effet, difficile et même hasardeux de laisser l'initialisation du contexte à la charge du programmeur, alors que celui−ci, dans un tel environnement, a tendance à ne plus faire la distinction entre ce qui est

temporaire et ce qui est persistant. Il y a deux raisons à cela : d'une part, l'ensemble des données persistantes utilisées par un programme est beaucoup plus riche et varié ; d'autre part, l'interface de ces données elle aussi est plus variée et surtout n'est pas limitée à un ensemble d'opérations prédéfinies, comme c'est le cas des fichiers.

III.2 Réexécution d'une application

L'objectif principal de notre travail est de permettre la mise au point cyclique des applications parallèles et réparties à base d'objets persistants. Pour ce faire, nous voulons utiliser un mécanisme de réexécution pour nous affranchir du non−déterminisme des exécutions parallèles.

Nous étudions dans la suite les principales caractéristiques d'un tel mécanisme, prenant en compte les particularités de notre environnement.

III.2.1 Fonctions d'une réexécution

Les fonctions de base de la reproduction d'une exécution en deux phases sont par définition : la création de l'historique de l'exécution et sa reproduction.

- La sauvegarde de l'historique s'effectue lors de la phase d'enregistrement durant laquelle une trace des événements de l'exécution est sauvegardée. Cette trace ne comporte qu'une certaine catégorie d'événements (cf III.2.3), considérés comme significatifs pour résoudre le non−déterminisme de l'exécution. Les données associées aux événements ne sont pas toutes conservées dans l'historique, pour réduire le temps et l'espace nécessaire à l'enregistrement : elles seront recalculées durant la deuxième phase.
- La reproduction de l'exécution a lieu lors de la deuxième phase, durant laquelle l'application est exécutée sous le contrôle direct d'un mécanisme de réexécution. Celui−ci se sert des informations conservées dans l'historique pour diriger les réexécutions successives et obtenir le même comportement de l'exécution.

Des fonctions spécifiques à l'environnement d'exécution peuvent être nécessaires pour reproduire une exécution. Ce sont celles qui prennent en compte les particularités de l'environnement d'exécution qui ont une influence sur la reproduction, comme par exemple la répartition de l'exécution et la persistance des objets.

III.2.2 Perturbation et efficacité

Le comportement de l'application mise au point peut être modifié par l'action intrusive du metteur au point. En effet, celui−ci va intervenir pendant l'exécution pour enregistrer son historique ou diriger sa réexécution, modifiant ainsi ses temps d'exécution. Ces modifications peuvent conduire en particulier à la disparition de certaines erreurs, dues à un changement des conditions d'exécution.

Cette perturbation a de graves conséquences lors de la phase d'enregistrement de l'historique. A ce moment là, plus le temps nécessaire à la création de l'historique est

important, plus le comportement de l'application risque d'être perturbé. Ainsi, des erreurs de synchronisation qui se produisent au cours d'une exécution normale peuvent ne plus apparaître lorsque l'exécution est tracée en vue de sa mise au point. Ce fonctionnement, au demeurant non satisfaisant, a la particularité d'exaspérer tout programmeur !

Toutes les applications ne sont pas sensibles à ce phénomène. Beaucoup de programmes échappent à ces méfaits, notamment les programmes séquentiels et les programmes parallèles composés d'activités strictement indépendantes. Dans ces cas, la perturbation de l'exécution se réduit à la modification des temps d'exécution et ne se traduit pas par des différences de comportement.

Les applications coopérantes, cibles principales d'un environnement comme le nôtre, sont au contraire très sensibles à cette perturbation. Les caractéristiques d'exécution des applications parallèles ou réparties en font déjà des applications non−déterministes : concurrence entre activités, charge des systèmes de communication, différence de vitesse entre processeurs, etc. L'action intrusive du metteur au point ne peut qu'amplifier le phénomène, en introduisant un facteur supplémentaire de perturbation.

Il est donc très important pour nous d'utiliser un mécanisme de collecte des informations le plus efficace possible (le moins perturbant).

III.2.3 Définition des événements

Les événements utilisés pour la reproduction d'une exécution peuvent être choisis à différents niveaux du modèle d'exécution. Ils peuvent concerner différentes entités qui participent à l'exécution comme les processus, les objets ou les instructions.

Nous avons choisi de baser la reproduction de l'exécution sur les événements concernant les objets en harmonie avec nos choix pour l'observation de l'exécution (cf III.1.1). Ces événements conservent suffisamment d'informations sur une exécution pour permettre sa reproduction précise, contrairement aux événements concernant les processus, et leur enregistrement ne provoque qu'une perturbation limitée de l'exécution, contrairement aux événements associés aux instructions.

Dans ce cadre, l'exécution d'une application est caractérisée par deux catégories d'événements qui sont conservés dans l'historique de l'exécution :

- La première catégorie correspond aux actions qui composent le(s) flot(s) d'exécution de l'application. Ces événements correspondent aux actions effectuées par l'exécution du code de l'application, c'est−à−dire les opérations appelées explicitement sur les objets. Les paramètres d'appel ne sont pas conservés, ils sont recalculés lors de la réexécution.
- La seconde catégorie caractérise les points de synchronisation entre les flots d'exécutions parallèles de l'application. Ces événements correspondent aux actions de synchronisation effectuées par le système de manière implicite lors de l'exécution de l'application, comme par exemple la synchronisation de l'accès aux objets. Ces événements déterminent le comportement des objets au cours de l'exécution et donc influent sur le comportement de l'application.

III.2.4 Ordonnancement des événements

L'historique d'une exécution conserve la nature des événements signalés, mais aussi l'ordre dans lequel ces événements se sont produits. Cet ordre est utilisé pour diriger les réexécutions, c'est−à−dire reproduire la même séquence des événements et ainsi obtenir le même comportement du programme.

Contrairement à une exécution séquentielle, une exécution parallèle et répartie ne se traduit pas naturellement par une suite totalement ordonnée d'événements, mais par un ensemble d'événements dont l'ordre n'est que partiel. Deux événements sont dits concurrents lorsque l'ordre de leur exécution n'a pas d'influence sur le reste de l'exécution.

Pour obtenir un ordonnancement total de ces événements, nous devons utiliser un référentiel de temps logique global, comme par exemple le mécanisme d'horloge logique vu en section II.2.2.5.

Un ordre total sur les événements est requis, par exemple, par les méthodes d'interactions directes (cf II.2.2.5). Celles−ci utilisent le référentiel de temps pour comparer l'ordre d'occurrence des événements d'une première exécution dans le but d'arrêter l'exécution dans un état global cohérent.

Par contre, la méthode de reproduction de l'exécution se satisfait d'un ordre partiel sur les événements. Pour reproduire une exécution, elle utilise uniquement la connaissance locale à chaque flot d'exécution de la suite des événements qui le composent. Les points d'arrêts globaux de l'exécution sont déterminés, lors de la phase de reproduction, en s'appuyant sur la connaissance de l'historique.

III.2.5 Équivalence des exécutions

Tout mécanisme de reproduction permet d'obtenir des exécutions équivalentes à celle enregistrée, au niveau des événements concernés. L'ordre des événements lors des réexécutions est donc le même que celui de l'exécution enregistrée. Dans notre cas, nous obtenons des réexécutions ayant le même comportement au niveau des opérations sur les objets, ce qui permet leur mise au point.

Cependant, les réexécutions ainsi obtenues peuvent ne pas être tout à fait identiques à l'exécution initiale. En effet, rien ne garantit la reproduction de l'ordre des opérations élémentaires qui composent ces événements, ni la reproduction du temps utilisé.

- D'une part, l'ordre d'exécution des instructions qui composent une opération sur un objet n'est pas garanti lors des réexécutions. Par exemple, les appels de deux opérations concurrentes sur un même objet seront toujours reproduits dans le même ordre, ce qui n'est pas forcément le cas des instructions composant le corps de ces opérations.
- D'autre part, nous ne tenons pas compte du facteur temps car nous ne nous intéressons pas à la mise au point d'applications temps réel. Dans notre environnement, le temps n'est pas un facteur déterminant.

L'étude de l'exécution des instructions du langage correspond à un niveau microscopique de la reproduction et nécessite l'utilisation d'un mécanisme complémentaire au nôtre dont le rôle est de reproduire le même ordre d'exécution des instructions à l'intérieur d'un objet.

III.2.6 Divergence des exécutions

La réexécution d'une application ne peut pas toujours suivre l'historique enregistré : dans certains cas les deux exécutions divergent. Notre mécanisme de reproduction s'en apercevra mais ne pourra pas corriger cette divergence, il pourra juste la signaler, sans en expliquer la cause.

L'utilisateur est à l'origine de cette divergence lorsque celui−ci modifie les données de son exécution (valeur des objets, code des classes exécutées, entrées/sorties, etc). Ce phénomène d'évolution de l'exécution est étudié ci−après en section III.2.7.

Dans le cas contraire, lorsque l'utilisateur n'a rien modifié, la divergence de la réexécution peut s'expliquer de deux manières différentes :

- Soit l'algorithme du programme est par nature non−déterministe. Dans ce cas, la méthode de reproduction d'une exécution ne peut pas s'appliquer. C'est à l'utilisateur d'intervenir et de modifier, corriger, son programme pour le rendre déterministe s'il veut le mettre au point avec notre méthode.
- Soit le programme contient une erreur qui ne s'est pas manifestée lors de l'exécution enregistrée et qui est provoquée par un ordre de réexécution des instructions différent. Ce phénomène traduit une erreur de synchronisation des opérations en cours d'exécution, que l'utilisateur peut alors corriger. L'apparition de ces erreurs est un effet de bord positif de la méthode de reproduction qui en fixant l'ordre des opérations laisse libre l'ordre des instructions.

Notre reproduction de l'exécution au niveau des objets nous permet de reproduire les conditions macroscopiques de l'occurrence d'un comportement erroné qui correspond à une situation bien particulière de l'exécution. L'observation de cette situation est la plupart du temps suffisante pour déterminer la cause de ce comportement erroné.

III.2.7 Évolution des exécutions

Pour terminer, il nous semble important d'indiquer une limitation des techniques de reproduction : elles permettent de reproduire plusieurs fois une exécution, mais ne prennent pas en compte son évolution liée à des modifications. Or la modification des données est une fonction très utile lors de la mise au point interactive, qu'il n'est pas concevable d'interdire.

Toute modification du code de l'application ou des données utilisées par l'exécution, comme par exemple l'état d'un objet ou les paramètres d'une opération, empêche la poursuite de la reproduction de celle−ci. En particulier, une modification des données intervenant au cours d'une réexécution définit une nouvelle suite de l'exécution qui a de fortes chances d'être différente de celle enregistrée.

Pour continuer la mise au point d'une exécution après une modification, nous proposons d'utiliser de façon transparente une exécution à deux phases. L'historique de la nouvelle suite de l'exécution est enregistré avant de permettre au programmeur de poursuivre la mise au point de son application en s'appuyant sur une nouvelle réexécution intégrant la modification.

L'ensemble des modifications que l'utilisateur effectue définit une arborescence d'historiques que le système de mise au point doit gérer. En effet, chaque modification d'une donnée définit un nouvel historique qui se greffe sur l'historique précédent, à l'endroit précis de l'exécution où a eu lieu cette modification.

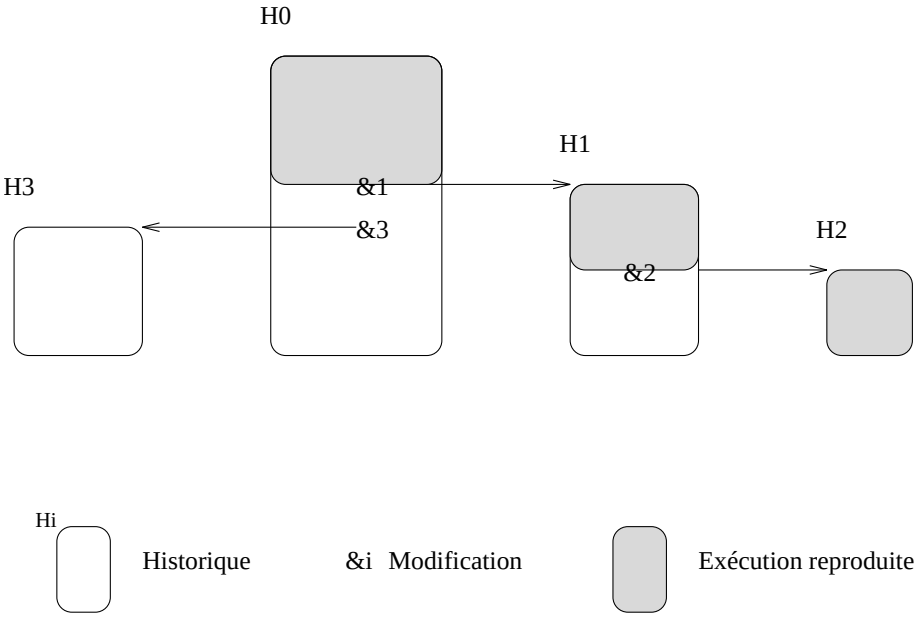


Fig. 3.1 : Arborescence d'historiques

La figure Fig. 3.1 montre un historique H0 d'une exécution qui après une première modification &1 a donné lieu à l'historique H1. Lui−même après la modification &2 a donné lieu à l'historique H2. L'exécution reproduite intégrant ces deux modifications suit les parties hachurées des historiques H0, H1 et H2. H3 est la suite de l'exécution après la modification &3 de H0 (sans &1).

III.3 Observation de l'application

La réexécution d'une application résoud les problèmes liés au non−déterminisme des applications parallèles et distribuées, en permettant de reproduire des exécutions équivalentes à celle enregistrée. Le programmeur peut donc reproduire à loisir une exécution erronée. C'est une première étape dans la réalisation de nos objectifs.

Une étape complémentaire consiste à lui offrir une observation de l'application qui lui permette une meilleure compréhension du comportement de son exécution. Cette observation doit l'amener à déterminer plus facilement et plus rapidement l'erreur qu'il veut corriger.

III.3.1 Conditions générale de l'observation

Nous présentons dans ce paragraphe les conditions que doit remplir à notre avis la partie observation d'un outil de mise au point moderne. Ces réflexions sont le résultat d'une étude plus complète que le lecteur peut trouver dans [4].

WYSIWYG (What You See Is What You Get)

L'observation de l'état du programme doit refléter l'état réel du programme, de la manière la plus fidèle possible. Cela signifie en particulier que lorsqu'une donnée est modifiée, sa représentation doit être mise à jour.

Convivialité

Les commandes les plus utilisées du metteur au point doivent être accessibles le plus simplement possible (utilisation de boutons, de menus, etc). L'utilisateur ne doit pas être obligé de mémoriser des commandes complexes. D'autre part, il doit avoir la possibilité de choisir son "style d'interaction" préféré : clavier, souris et bientôt la voix.

Filtrage

Il est très important que l'outil de mise au point permette à l'utilisateur de filtrer les informations observables pour ne voir que celles qui l'intéressent. Ceci est valable pour la visualisation des données manipulées par le programme, pour la visualisation du source du programme mais également pour la visualisation de son exécution. Par exemple, dans un environnement à objets, lors de l'observation de l'état d'un objet, il doit avoir la possibilité de cacher les variables héritées. De même lors de l'observation d'un flot d'exécution, il doit pouvoir cacher les opérations appartenant à une classe donnée.

D'une manière générale, la quantité d'informations produites à l'écran doit être minimisée pour éviter l'excès d'information.

Multi−fenêtres

L'outil de mise au point doit utiliser plusieurs fenêtres pour afficher ses sorties, pour que l'utilisateur puisse les différencier aisément. Dans le cas contraire, les sorties du programme sont mélangées avec celles de l'outil et quand le programme mis au point est multi−processus, les entrées/sorties des différents processus sont entremêlées et la mise au point devient très difficile.

Il doit au minimum utiliser :

- Une fenêtre pour visualiser le source.
- Une fenêtre de boutons pour les commandes les plus fréquemment utilisées.

- Une fenêtre pour dialoguer avec un interpréteur de commande en mode ligne.
- Une fenêtre pour indiquer l'état courant de l'outil.
- Des fenêtres spéciales pour visualiser les données complexes.
- Des fenêtres pour visualiser divers aspects de l'exécution du programme.

Si le programme mis au point est multi−processus, il faut séparer dans des fenêtres différentes les sorties des différents processus et offrir des mécanismes qui permettent de voir l'interaction entre les processus (communication, synchronisation) et de les désigner individuellement (consultation de leur état, par exemple).

Cependant, là encore, il faut trouver un compromis. En effet, s'il y a trop de fenêtres superposées à l'écran, on arrive au résultat opposé : l'utilisateur est noyé dans trop d'informations.

III.3.2 Types d'informations observables

Pour comprendre le comportement d'une exécution, différentes informations sont nécessaires :

- Les informations sur le(s) flot(s) d'exécution qui correspondent à la suite des actions effectuées pendant cette exécution. Elles permettent de connaître les entités qui interviennent dans l'exécution (processus, sites, mémoires virtuelles) et de suivre le comportement de l'application.
- Les données manipulées par l'application qui permettent de suivre l'évolution de l'état de l'application.
- Le code source de l'application qui permet de comprendre l'enchaînement des actions effectuées en fonction des valeurs des données. Ce code source de l'application constitue la représentation statique de l'application qui contient les erreurs à corriger.

Ces informations sont complémentaires, et il est indispensable de permettre à l'utilisateur de toutes les observer si l'on veut lui offrir une aide de qualité.

III.3.3 Observation de l'exécution

L'observation de l'exécution doit permettre de connaître les entités qui participent à l'exécution ainsi que leur comportement dynamique. Dans notre cas, l'observation doit donc montrer les activités créées et les objets utilisés, ainsi que les opérations exécutées sur ces objets.

III.3.3.1 Disponibilité des informations

La méthode de reproduction est très intéressante pour l'observation d'une exécution, car elle permet d'observer et d'étudier son comportement à tout moment de la réexécution (avant, pendant, ou après). Dès la fin de la phase d'enregistrement, elle rend disponible des informations sur l'exécution (par le biais de son historique) qui ne sont normalement connues,

dans un cadre classique, qu'après la fin de l'exécution ou au mieux découvertes au fur et à mesure du déroulement de l'application :

- Le programmeur peut ainsi se familiariser avec le comportement d'une exécution avant sa reproduction, ce qui lui permet de se faire une première idée sur la localisation de l'erreur qu'il cherche à corriger.
- Pendant les réexécutions, il a à sa disposition la connaissance non seulement de l'état courant de son exécution mais aussi du comportement passé et à venir de l'exécution. Son attention n'est plus focalisée sur la découverte de l'exécution et peut être utilisée de manière plus efficace à la localisation de l'erreur.

Ces informations sont obtenues à partir des événements caractéristiques contenus dans l'historique. L'ensemble des flots d'exécution peut être connu par un simple parcours de l'historique, alors qu'une analyse plus complète est nécessaire pour obtenir des renseignements tels que l'ensemble des objets utilisés, l'ensemble des objets partagés par deux activités, etc.

III.3.3.2 Vues complémentaires de l'exécution

Les informations conservées dans l'historique peuvent être utilisées pour offrir à l'utilisateur plusieurs vues complémentaires de l'exécution dans le but de faciliter la mise au point.

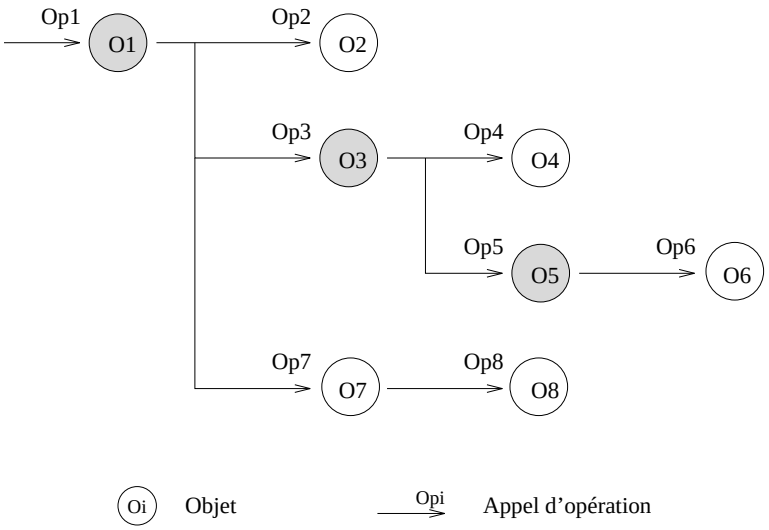


Fig. 3.2 : Observation du flot d'exécution

Un flot d'exécution peut être montré sous la forme d'un arbre d'appels d'opérations, dont une animation appropriée permet de suivre la progression de l'exécution. En montrant l'opération en cours d'exécution, l'utilisateur peut aisément suivre l'enchaînement des opérations et ainsi mieux comprendre le déroulement de l'exécution. La figure Fig. 3.2 montre un exemple d'observation d'un flot d'exécution sur lequel on peut voir la suite des opérations en cours d'exécution O1.Op1, O3.Op3, O5.Op5, connaître l'opération précédente O4.Op4 et la suivante O6.Op6.

L'observation de l'exécution doit aussi offrir des vues appropriées aux particularités de l'environnement telles que le parallélisme, la répartition, la persistance des objets, etc.

Cette observation doit être facilitée par des fonctions d'affichage puissantes qui doivent permettre le contrôle de l'animation, le filtrage et le regroupement des données, la désignation symbolique des données, la visualisation des piles d'appels, etc.

III.3.3.3 Support pour la mise au point

Les vues de l'exécution peuvent servir de support au contrôle de l'exécution. En permettant au programmeur de définir des points d'arrêts sur les opérations montrées dans un flot d'exécution, nous lui permettons de choisir l'endroit exact de l'arrêt de la réexécution. Les points d'arrêt ainsi définis gagnent en précision puisqu'ils ne concernent qu'une seule exécution d'une opération donnée sur laquelle la réexécution s'arrêtera directement, même si cette opération est exécutée plusieurs fois auparavant (cas d'appels multiples ou d'appels récursifs).

Nous pouvons aussi utiliser ces vues de l'exécution pour permettre à l'utilisateur de définir des états globaux cohérents de l'exécution, ce qui peut se révéler fort utile pour effectuer une réexécution partielle de l'application lorsque celle−ci prend beaucoup de temps. En désignant une opération dans un flot d'exécution, le programmeur désigne l'endroit à partir duquel il veut pouvoir reprendre la réexécution. La définition d'un état global cohérent de l'exécution est à la charge du système de mise au point.

III.3.4 Observation des données

Les structures de données jouent un rôle fondamental dans la programmation. Outre le fait qu'elles représentent un modèle d'organisation de données, de natures différentes suivant l'application, les structures de données représentent souvent des abstractions.

Il est donc primordial, en phase de mise au point, de pouvoir observer les données accédées par le programme à un instant donné, autrement dit de consulter l'état du programme à cet instant là : le caractère erroné du fonctionnement d'un programme se manifestant généralement par une valeur incohérente de cet état.

Cette observation doit permettre de suivre l'évolution des données utilisées par l'exécution de manière simple et efficace. Les données qui nous intéressent sont essentiellement les variables constituant l'état des objets. L'accès à la valeur de cet état doit aussi nous permettre de naviguer dans la composition des objets.

Nous présentons ci−dessous les conditions que doit remplir à notre avis l'observation des données. Ces réflexions sont le résultat d'une étude plus complète que le lecteur peut trouver dans [4].

Présentation

> L'outil de mise au point doit offrir plusieurs façons de présenter les données : la présentation classique (<nom> : valeur) bien sûr, mais aussi des présentations graphiques plus expressives. Il doit fournir des figures prédéfinies qui permettent de présenter les données sous une forme "analogique" : utilisation de tables, icônes,

barres, graphes, etc. Il serait également intéressant qu'il offre à l'utilisateur la possibilité de concevoir lui−même ses propres présentations.

Interaction

Les données affichées à l'écran doivent être accessibles par leur représentation externe. Par exemple, si une variable est visualisée sous forme d'un camembert, on devrait pouvoir changer sa valeur en modifiant à l'aide de la souris les différentes parts du camembert.

Affichage continu

Il arrive souvent lors d'une phase de mise au point que l'on veuille voir de façon continue la valeur d'une variable donnée. Dans les outils classiques, on procède généralement de la manière suivante : on exécute le programme au pas à pas et on demande à chaque fois au système d'afficher la valeur de la variable. Ceci peut être très fastidieux, notamment si la variable a un nom compliqué (champs de profondeur N d'une structure X, repérée par le pointeur Y...).

L'outil de mise au point doit donc offrir un mécanisme permettant de "lier" une variable, de telle sorte que sa présentation reste à l'écran et soit mise à jour aussi longtemps qu'elle appartient à l'environnement d'exécution. Ainsi par exemple, la représentation d'une variable locale sera visible aussi longtemps que la fonction correspondante sera active.

III.3.5 Observation du source

L'observation du code source de l'application donne accès à la structuration du programme et facilite ainsi la compréhension de l'exécution. De plus, c'est le support tout indiqué pour suivre le déroulement de l'exécution et agir sur elle.

Une des particularité des langages à objets est d'aboutir à des applications très structurées. Ces langages offrent une structuration très forte au niveau de la programmation en décomposant les applications en termes de classes qui définissent les opérations permises sur leurs instances. Les classes sont reliées entre elles par de nombreuses relations (héritage, composition, utilisation, instance, réalisation) qui sont définies lors de la phase de conception [5].

Pour faciliter la compréhension de la structure d'une application, il est important d'offrir un outil de navigation dans les sources permettant de les consulter et de suivre les relations qui existent entre eux. Cet outil d'observation des sources doit permettre de voir la définition des classes mais aussi de parcourir ces relations pour accéder simplement au composants désirés.

Le source peut aussi être utilisé pour suivre le comportement dynamique de l'exécution ainsi que pour son contrôle. Par un affichage approprié, l'utilisateur peut connaître l'opération en cours d'exécution et localiser celle qui est la cause du comportement erroné. De plus il peut contrôler le déroulement de l'exécution en définissant des points d'arrêt de l'exécution sur des opérations. Il accède ensuite à l'état des données utilisées en désignant les variables utilisées dans le source.

III.4 Organisation de ces services

Nous avons étudié les deux principaux services (réexécution et observation) d'un outil de mise au point évolué pour la correction d'applications parallèles et réparties à base d'objets persistants.

Nous présentons maintenant l'organisation de ces services dont une partie est réalisée par des outils existants avec lesquels le metteur au point coopère. L'outil de mise au point, dont nous présentons l'architecture réalise les autres fonctions qui lui sont propres.

La coopération entre ces outils s'effectue dans le cadre bien particulier d'un environnement de développement intégré dont nous soulignons les principales caractéristiques.

III.4.1 Coopération avec des outils existants

La mise au point est une activité qui prend de plus en plus de place dans la vie d'une application et accessoirement dans celle du programmeur. Les applications devenant de plus en plus complexes (elles sont parallèles, distribuées, utilisent des objets persistants, etc), leur mise au point se transforme en une tâche ardue.

Ces applications ont donc besoin d'un cadre beaucoup plus complet pour leur mise au point, qui permette d'accéder et de gérer de plus en plus d'informations, toutes celles nécessaires à leur mise au point. Ce cadre de mise au point ne peut être réalisé que par la coopération de nombreux outils dont le metteur au point est l'outil central : il assure les fonctions de base de la mise au point cyclique et d'observation de l'exécution et sous−traite à d'autres outils certaines tâches, essentiellement celles de consultation des sources et des données.

La réutilisation d'outils déjà connus du programmeur facilite son travail, tout en nous permettant de tirer parti de leur spécialisation : un éditeur permet de consulter et modifier les sources, un "browser" de sources permet de suivre les relations entre les composants de l'application, un "browser" d'instances permet de montrer les valeurs des objets, etc.

L'observation des informations plus spécifiques à la mise au point doit par contre être fournie au travers d'outils ad hoc.

III.4.2 Architecture de l'outil de mise au point

Le metteur au point est structuré en trois couches (cf Fig. 3.3) dont chacune réalise une fonction particulière :

- La couche basse (le noyau de réexécution) assure les conditions d'exécution particulières que réclame la mise au point cyclique des applications parallèles et réparties, à base d'objets persistants. Il assure la reproduction de l'exécution en enregistrant l'historique d'une exécution et en contrôlant les réexécutions successives pour produire des exécutions équivalentes. Il offre de plus les fonctions indispensables à la mise au point cyclique d'une application qui permettent au programmeur d'agir sur le déroulement de l'exécution (définition des points d'arrêts, accès aux objets, contrôle de l'exécution, etc).

- La couche intermédiaire s'occupe de l'observation de l'application, elle a en charge la visualisation du source, de l'exécution et des données.
- La couche haute correspond au frontal du metteur au point qui permet à l'utilisateur, par le biais d'une interface graphique conviviale, d'accéder facilement aux fonctions principales de l'outil. Le frontal établit les communications avec les autres outils en réponse aux requêtes de l'utilisateur.

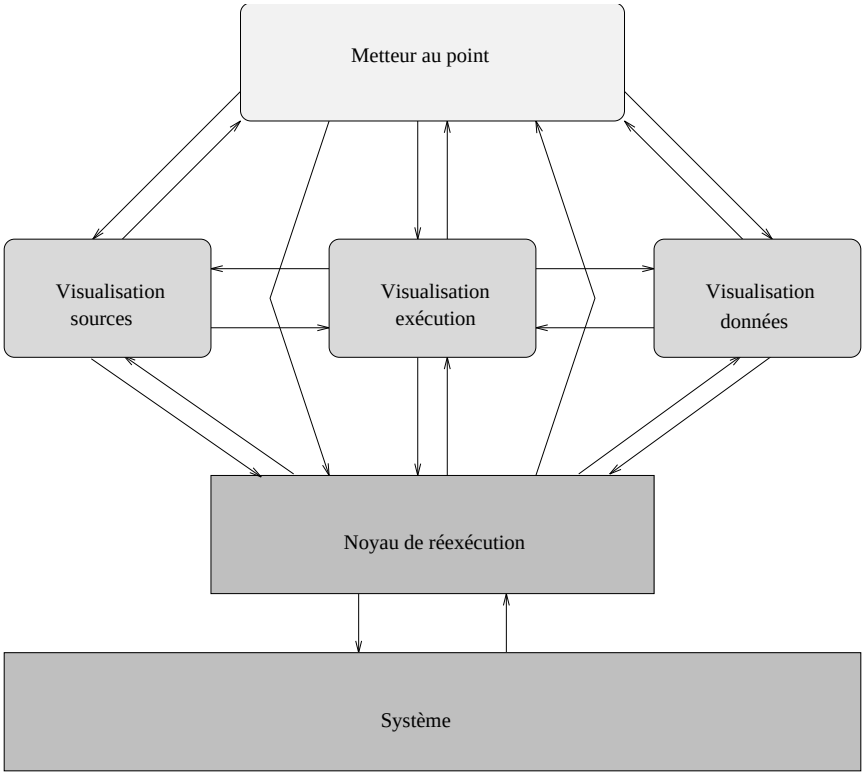


Fig. 3.3 : structuration du metteur au point

III.4.3 Cadre de la coopération

Un environnement de développement est destiné à aider les utilisateurs dans les différentes phases du développement de leurs applications. Il est en général composé d'un ensemble d'outils reliés par une interface commune qui sont utilisés tout au long de la chaîne de vie d'une application. L'édition, la compilation, la mise au point sont les principales phases prises en charge par cet environnement.

La caractéristique principale d'un environnement intégré est le fort taux de coopération qui existe entre les outils qui le composent : un outil peut utiliser les services d'un autre outil pour réaliser les fonctions dont il a besoin. Ainsi, lorsque l'utilisateur demande la visualisation d'un source à partir de l'outil de mise au point, celui−ci demande à l'éditeur d'afficher le source en question. Ces deux outils coopèrent lorsque l'utilisateur suit son exécution sur le source du programme : le metteur au point signale l'opération en cours d'exécution à l'éditeur qui la matérialise dans le source.

Chapitre IV

Un metteur au point pour Guide

Nous avons présenté au chapitre précédent notre approche de la mise au point parallèle et répartie dans un système à base d'objets, la structurant comme un ensemble de services dont l'objectif est de résoudre chacun des problèmes inhérents à ce type de mise au point. Nous avons par ailleurs proposé une architecture permettant d'assembler ces services pour en obtenir un outil très modulaire et tout aussi puissant. Nous reprenons dans ce chapitre nos propositions pour les appliquer de façon concrète et précise au système Guide.

Après la présentation du contexte de réalisation de notre travail, nous détaillons les caractéristiques du système Guide qui peuvent avoir des conséquences sur la mise au point d'une application. Nous analysons ensuite les problèmes posés par ce système pour la réalisation des services que nous avons proposés. Les solutions adoptées ont été validées par une mise en œuvre qui est décrite au chapitre V.

IV.1 Contexte de réalisation

Notre travail s'est effectué dans le cadre du projet Guide dont nous présentons tout d'abord les principales caractéristiques. Il nous a semblé ensuite intéressant, puisque notre propos est la mise au point d'applications dans le système Guide, de familiariser le lecteur avec le déroulement de l'exécution d'un programme dans ce système. Nous situons ensuite notre travail dans le cadre du projet Guide et plus particulièrement dans le cadre de l'environnement de développement de Guide.

IV.1.1 Présentation générale

Le projet Guide (Environnement Distribué et Intégré des Universités de Grenoble) est un projet de recherche qui a débuté en 1986 au sein du Laboratoire de Génie Informatique de l'IMAG (Institut d'Informatique et de Mathématiques Appliquées de Grenoble) et du Centre de Recherche Bull de Grenoble. Il a pour objectifs la conception et la réalisation d'un système à base d'objets pour le développement et le support d'applications réparties sur un réseau local.

Les applications pilotes visées par ce système font partie de domaines tels que le génie logiciel (développement, mise au point, maintenance de programmes) et la gestion de documents structurés (édition, archivage, courrier électronique), où la distribution et la coopération sont des mots clés.

Afin de favoriser le développement du projet et la valorisation des résultats, le Groupe Bull et les autorités de tutelle de l'IMAG ont décidé, en 1989, la création d'une Unité Mixte de

Recherches autour de l'équipe Guide. L'objectif de l'Unité Mixte est la poursuite et le développement du système Guide jusqu'au stade de prototype pré−industriel.

Une première réalisation du système Guide a eu lieu au−dessus du système Unix [9]. Un langage, le langage Guide, a été conçu pour tirer avantage des concepts propres au système. Sa mise en œuvre a permis d'écrire des applications représentatives pour la machine d'exécution Guide (plus de 150.000 lignes de programmes Guide).

Les travaux actuels liés au projet visent la réalisation d'un deuxième prototype du système, nommé Eliott [16], au−dessus du micro−noyau Mach [54], ainsi que l'étude d'un environnement de développement pour le langage Guide.

Une partie des travaux du projet Guide a été menée dans le cadre du projet ESPRIT COMANDOS (Construction and Management of Distributed Office Systems) [7].

IV.1.2 Caractéristiques principales

Nous présentons dans la suite les caractéristiques principales du système Guide, dont une description plus complète et détaillée se trouve dans [26]. Un des choix de conception les plus importants du système Guide a été l'adoption d'un modèle à objets comme structure de base du système ainsi que pour la programmation des applications. Ce choix permet d'unifier les concepts liés aux structures d'exécution et au schéma de conservation de l'information.

IV.1.2.1 Modèle d'objets

Le système est organisé autour de la notion d'**objet** [43] [60]. Un objet Guide, unité d'exécution et de conservation, est caractérisé par un **état** et un ensemble d'opérations ou procédures d'accès, appelées **méthodes**. Un objet ne peut être manipulé qu'au moyen des opérations qui lui sont associées.

Tout objet a une **classe** qui réalise un **type**. Ce type définit les signatures des opérations applicables à l'objet (nom de la méthode, nombre et type des paramètres et identification des signaux d'exception pouvant être levés par la méthode). La représentation de l'état de l'objet (type des variables) et le code des opérations sont définis par la classe de l'objet, qui est une réalisation particulière de son type.

Les constructeurs d'objets du système Guide sont les classes. A chaque classe est associée une opération de génération (dite aussi instanciation) qui permet de créer des objets dits **instances** ou exemplaires de cette classe. Ces objets ont tous le même format, et partagent le même programme pour leurs opérations. Une classe est elle−même un objet, une de ses opérations étant précisément la génération d'objets.

Les objets sont accédés dans le langage Guide [65] par des variables typées. Le type de l'objet n'est pas obligatoirement celui de la variable référence qui le manipule, mais doit être conforme. La relation de conformité est vérifiée à la compilation, pour tout ce qui est affectation et passage de paramètres.

Les types sont organisés hiérarchiquement avec héritage simple ; un type peut être défini comme spécialisation d'un autre type (sous−type). Une hiérarchie analogue existe entre les

classes (héritage de code). Il est par ailleurs possible de définir des types et des classes génériques.

La distinction entre type et classe est analogue à la séparation entre interface et réalisation dans les langages modulaires. Il y a toutefois deux différences importantes : la génération dynamique des instances d'une classe, et le fait qu'un même type puisse être réalisé par plusieurs classes différentes (ou aucune).

IV.1.2.2 Modèle d'exécution

Le modèle d'exécution du système Guide [10] [20] offre aux utilisateurs une machine virtuelle multi−site et multi−processus dans laquelle le parallélisme est apparent et la distribution cachée. Ce modèle est fondé sur la séparation des objets et des flots d'exécutions qui les accèdent, les objets Guide étant passifs.

L'unité d'exécution est appelée **domaine** et correspond conceptuellement à une machine virtuelle multi−processus pouvant s'étendre dynamiquement sur plusieurs sites. Un domaine est composé d'un ensemble d'objets et de processus séquentiels qui partagent ces objets. Ces derniers, appelés **activités**, exécutent en séquence des opérations sur les objets (appels de méthode) selon un schéma procédural : l'appel d'une opération d'un objet, qu'il soit local ou distant, est une opération synchrone. L'ensemble des objets qui composent un domaine (appelé **contexte**) évolue dynamiquement par liaison et déliaison des objets lors des appels de méthode effectués par les activités du domaine. Un objet peut être lié simultanément à plusieurs domaines, alors que les activités appartiennent à un seul domaine.

La figure Fig. 4.1 représente deux domaines *D1* et *D2*. Ces domaines partagent à l'instant considéré l'objet *O2* ; chacun d'eux possède en outre des objets qui lui sont propres.

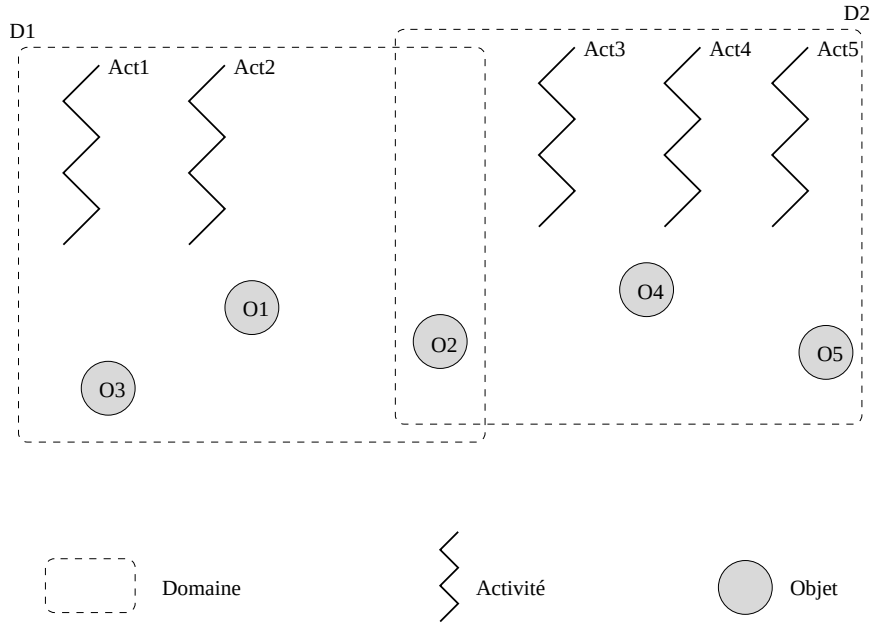


Fig. 4.1 : Domaines, activités et objets dans Guide

Un domaine est créé par une activité d'un autre domaine, l'opération de création spécifiant un objet initial et une opération sur cet objet : la création d'un domaine implique la création d'une activité initiale pour exécuter cet appel d'opération. Une fois créé, le domaine est autonome, mais l'activité qui l'a créé peut exercer un certain contrôle sur ce nouveau domaine par des opérations de destruction, d'arrêt ou de reprise.

Un objectif du système étant de dissimuler la répartition à ses utilisateurs, aucun lien a priori n'est imposé entre un domaine et un site d'exécution : un domaine peut s'étendre dynamiquement sur plusieurs sites (pour accéder à des objets distants) et un site peut être le siège de plusieurs domaines.

Le parallélisme au sein d'un domaine est introduit par une instruction du langage qui permet à une activité mère de créer plusieurs activités filles concurrentes, dont chacune exécute une suite d'opérations sur des objets. Le schéma d'exécution est synchrone. L'activité mère est suspendue tant qu'une condition portant sur la terminaison des activités fille n'est pas vérifiée. Celle−ci permet d'attendre sélectivement la terminaison (normale ou exceptionnelle) de tout ou partie des activités filles. Lorsque la condition est réalisée, les activités filles qui ne sont pas encore terminées sont détruites et l'activité mère reprend son exécution.

La communication entre activités est réalisée par partage d'objets. Elle s'effectue via les variables d'état de ces objets, protégées si nécessaire par un mécanisme de synchronisation (condition d'activation) associé aux opérations sur les objets [21]. Tous les objets sont potentiellement partageables. La communication entre activités peut être interne à un domaine, ou s'effectuer entre activités de domaines différents.

Afin de mettre en évidence les conditions considérées comme anormales dans le déroulement d'un programme, le modèle d'exécution fournit une notion d'exception [45] [44]. Une erreur, une panne ou la violation d'une contrainte d'intégrité sont des exemples d'exception. Il est possible de déclarer des exceptions propres à une application, de déclencher une exception et de définir des routines de traitement d'exceptions.

IV.1.2.3 Modèle de mémoire

Les objets dans Guide sont persistants, c'est−à−dire que leur durée de vie est indépendante de celles des procédures ou programmes qui les manipulent. A cet effet, chaque objet est désigné par une **référence**, qui permet de l'identifier de manière unique et de le localiser dans l'ensemble du système. Un objet ne peut être accédé que par l'intermédiaire de sa référence. Un ramasse−miettes [66] a la charge de libérer la place occupée en mémoire secondaire par les objets qui ne sont plus accessibles.

L'état d'un objet peut contenir des références à d'autres objets, ce qui permet de construire des structures complexes.

La gestion des objets met en œuvre une mémoire à deux niveaux, répartie sur l'ensemble des sites. Le niveau bas ou Mémoire Permanente d'Objets (**MPO**) est le support de conservation des objets. Elle est implantée sur l'ensemble des sites qui réalisent la fonction d'archivage. Le niveau haut, ou Mémoire Virtuelle d'Objets (**MVO**) est le support des objets liés dans les domaines, et fonctionne comme un cache vis−à−vis de la MPO.

IV.1.3 Exécution d'une application Guide

Nous détaillons ci−après le fonctionnement de l'exécution d'une application Guide pour que le lecteur puisse se familiariser avec ses caractéristiques et pour qu'il se rende compte de la difficulté de sa mise au point.

Au lancement d'une application, définie par un objet initial et une méthode sur cet objet, le système Guide crée un domaine dans lequel l'objet est chargé et crée une première activité pour exécuter cette méthode. Ce domaine et cette activité sont présents à ce moment là uniquement sur le site de lancement. L'exécution de cet appel de méthode peut nécessiter l'exécution d'autres appels de méthodes sur d'autres objets, en fonction du code de cette opération, créant ainsi un flot d'exécution.

La composition du contexte du domaine évolue en fonction des appels effectués par l'activité. Il y a liaison/déliaison dynamique d'un objet en début et en fin d'appel d'une méthode sur un objet.

Un objet ne peut être présent en mémoire à un instant donné que sur un seul site. En conséquence, lorsqu'une activité appelle une opération d'un objet qui n'est pas présent sur son site d'exécution, deux cas se présentent :

- Soit l'objet n'est chargé sur aucun site.

  Une image de l'objet est alors chargée dans la MVO du site demandeur et l'objet est lié localement au domaine. L'appel de méthode est effectué localement.

- Soit une image de l'objet est déjà chargée sur un autre site.

  Si l'objet n'est lié par aucun domaine sur ce site, il est déchargé pour être chargé dans la MVO du site demandeur. Nous assistons alors à une migration d'objet, et l'appel de méthode est aussi effectué localement.

  Si l'objet est déjà lié par un domaine, il ne peut pas être déplacé. L'activité qui veut utiliser cet objet va alors s'étendre dynamiquement sur ce site pour exécuter son opération. Pour cela, il y a création d'un représentant local de l'activité sur le site. L'extension d'activité provoque une extension de son domaine, si celui−ci ne s'est pas encore étendu sur ce site.

Un domaine est donc composé de plusieurs représentants locaux, un sur chaque site où ses activités se sont étendues. Une activité est également composée de plusieurs représentants locaux, un sur chaque site où elle s'est partiellement exécutée. Un seul de ces représentants est actif à un instant donné.

Une activité peut par ailleurs lancer de nouvelles activités en parallèle selon un schéma d'exécution synchrone : l'activité mère est suspendue en attente d'une condition de terminaison des activités filles. Une activité peut aussi lancer de nouveaux domaines selon un schéma d'exécution asynchrone : le domaine ainsi créé a une existence indépendante de celle de l'activité qui l'a créé.

Nous remarquons qu'une exécution dans le système Guide est relativement complexe : elle peut concerner plusieurs sites sur lesquels s'exécutent plusieurs activités dans plusieurs domaines, en utilisant une multitude d'objets. Les activités s'étendent de site en site pour accéder aux objets lorsque ceux−ci ne peuvent pas bouger. Dans le cas contraire, les objets

migrent à la rencontre des activités. De plus, les objets peuvent être partagés par plusieurs activités suivant les conditions de synchronisation définies sur les méthodes appelées.

Or le programmeur a une connaissance très simplifiée de son exécution qui est préjudiciable à l'activité de mise au point : il ne voit pas la répartition qui est cachée par le système Guide et ne connaît pas de manière exacte l'ensemble des objets/classes utilisés par son application. Pour lui son application effectue une suite d'appels de méthode et il ne sait pas si ces appels de méthode s'exécutent localement ou à distance, si son exécution met en œuvre des migrations d'objets ou des extensions d'activités. Le partage d'objet est également caché. Il existe donc un décalage entre la complexité de l'exécution et la connaissance qu'en a le programmeur.

Dans la suite, pour simplifier notre propos, nous considérons une exécution qui est constituée d'un seul domaine.

IV.1.4 L'environnement de développement

Notre étude sur la mise au point s'inscrit dans le cadre plus général de l'étude d'un environnement de développement intégré pour le langage Guide qui a été défini pour tirer avantage des concepts propres au langage Guide et à la plate−forme d'exécution. Nous présentons ci−dessous les principaux objectifs de cet environnement de développement, nommé Cybèle [12].

L'objectif général de l'environnement de développement Cybèle est d'aider l'utilisateur à développer des applications écrites en langage Guide à destination d'une plate−forme répartie telle que Guide ou Eliott. Cette aide sera fournie pour chacune des phases de développement du cycle de vie d'une application : conception, programmation, test et validation, ainsi que pour une phase supplémentaire de configuration, propre aux applications visées (structurées en termes d'objets, réparties et coopératives).

Un objectif de base de l'environnement est l'intégration des éléments qui le composent. Cette intégration devra se faire à trois niveaux : informations persistantes manipulées par ces éléments (sources, binaires, documentation, schémas de conception, etc), coopération entre outils (délégation de service, notification d'information, etc) et interface utilisateur. Chaque outil sera spécialisé dans une tâche particulière mais ils collaboreront pour réaliser des tâches plus complexes. L'environnement en entier sera vu comme une seule et unique application, composée certes de multiples éléments mais travaillant tous à l'unisson dans un seul et même but.

Un deuxième objectif de base de Cybèle est de tirer profit des concepts propres au langage. Ces concepts nous serviront autant au niveau de la structuration des informations communes que pour déterminer quels sont les outils et même les fonctions de ces outils qui sont les plus utiles pour le développement. Des informations sémantiques seront extraites des composants (essentiellement à la compilation) et exploitées par d'autres outils pour faciliter et simplifier les différents aspects du développement.

Un autre objectif de l'environnement est de rester ouvert. Il doit être possible d'échanger des informations avec l'extérieur (importation et exportation de composants) mais aussi et surtout d'intégrer des outils réalisés en dehors de l'environnement. Cette intégration ne sera

jamais immédiate mais nous essaierons de fournir des mécanismes facilitant sa mise en place.

IV.2 Étude du modèle d'exécution de Guide

IV.2.1 Analyse générale

Les caractéristiques principales du modèle d'exécution de Guide sont les suivantes :

Transparence de la répartition

Une exécution peut se dérouler sur plusieurs sites, en fonction des objets utilisés. Cette répartition est transparente au programmeur dans la plupart des cas. C'est en effet le système qui décide de continuer une exécution localement ou à distance, suivant la disponibilité des objets utilisés par l'application. Il est difficile de ce fait pour l'utilisateur de connaître l'état global de son exécution, un état qui est éclaté sur plusieurs sites et dont il ne connaît pas nécessairement la composition. En outre, la localisation et les conditions de liaison des objets pouvant varier d'une exécution à l'autre, le comportement d'une application peut lui aussi varier entre deux exécutions, rendant ces applications non−déterministes.

Parallélisme de l'exécution

Une application Guide peut être constituée par plusieurs activités qui se déroulent en parallèle et qui sont concurrentes vis à vis des mécanismes d'ordonnancement et de synchronisation. Les conditions qui régissent ces mécanismes pouvant varier entre deux exécutions d'une même application, rien ne garantit que son comportement pour un même jeu de données soit le même. Ces applications sont donc non−déterministes par nature.

Persistance des objets

Les objets Guide étant persistants, les modifications effectuées par une exécution sont rémanentes et peuvent changer le déroulement de l'exécution suivante.

Partage d'objets

Le système Guide permet et encourage même l'utilisation d'objets partagés, fournissant les mécanismes appropriés pour synchroniser leurs accès. Un objet peut donc parfaitement être utilisé par deux applications différentes à un moment donné, sans qu'elles s'en aperçoivent. Ce type de fonctionnement peut provoquer des variations dans le comportement d'une application, liées au problème de concurrence mais aussi et surtout aux modifications des valeurs des objets qui peuvent se produire.

IV.2.2 Conséquences sur la mise au point

Plusieurs des caractéristiques décrites en section IV.2.1 ont des conséquences directes sur la mise au point cyclique, car elles rendent non−déterministes certaines applications. Nous devons de ce fait les prendre en compte dans la réalisation de notre mécanisme de réexécution.

Nous analysons ci−après les éléments précis qui peuvent provoquer des différences entre deux exécutions successives. Cette analyse nous a permis de déterminer les événements caractéristiques d'une exécution Guide, présentés en section IV.2.3.

Exécution séquentielle

Dans le cas d'une exécution séquentielle, le flot d'exécution suivi par l'application dépend uniquement des données initiales et de la valeur des entrées effectuées durant l'exécution. Pour obtenir le même flot d'exécution, lors des réexécutions, nous devons donc garantir les même valeurs pour ces informations.

Exécution parallèle

L'utilisation des mêmes valeurs n'est pas suffisante pour reproduire une exécution parallèle. Pour le montrer, intéressons nous tout d'abord à une exécution parallèle dont les activités ne communiquent pas. Chacune des activités suit un flot d'exécution complètement indépendant de ceux des autres activités. Seule la condition de terminaison des activités lancées par un même CO_BEGIN peut provoquer des exécutions différentes, suivant les conditions d'exécution particulières de chacune (ordonnancement du système). Pour obtenir des réexécutions équivalentes, nous devons donc garantir aussi la même terminaison aux activités parallèles.

Dans le cas d'activités qui communiquent entre elles, il est en plus nécessaire de reproduire cette communication. Comme les activités ne communiquent en Guide qu'au travers d'objets partagés, il nous faut alors garantir le même ordre d'accès à ces objets pour obtenir le même comportement au niveau de la communication.

Exécution répartie

Une exécution répartie dans le système Guide est composée de portions d'exécution qui se déroulent sur différents sites et dont les interactions se limitent aux seules opérations d'extension d'activité et de migration d'objet. Pour obtenir la même répartition de l'exécution, nous devons donc reproduire ces même opérations.

Nous avons vu en section IV.1.3 que lors d'un appel distant, le choix entre l'une ou l'autre de ces opérations dépend de la disponibilité de l'objet à ce moment là : à savoir si l'objet est lié ou pas par un domaine. Nous devons donc reproduire, tout au long de l'exécution, le même ordre d'utilisation des objets par les différents sites, ce qui revient à reproduire le même ordre entre les demandes de chargement d'un objet et les liaisons et les déliaisons de cet objet.

Pour reproduire la même exécution répartie, il faut au préalable retrouver la localisation initiale des objets concernés.

Interventions externes

Le partage d'objets entre applications peut provoquer des variations dans le comportement d'une application. Pour reproduire une exécution nous devons d'une part reproduire les interventions externes à l'application mise au point qui ont eu lieu pendant cette exécution, et d'autre part isoler les reproductions pour qu'elles ne soient pas troublées par d'autres applications.

En résumé, pour reproduire une exécution Guide nous devons :

- garantir les même valeurs des données initiales (l'état des objets) et des entrées,
- garantir la même localisation initiale des objets,
- garantir la même terminaison des activités parallèles,
- garantir la même synchronisation des appels concurrents aux objets,
- garantir la même synchronisation entre les demandes de chargement et les liaisons et les déliaisons des objets,
- garantir la reproduction des interventions externes et l'isolement des reproductions.

Enfin l'édition de liens dynamique est de nature à rendre plus difficile la gestion et la manipulation des tables de symboles nécessaires à tout metteur au point. En effet la composition exacte de l'application ne sera connue que dynamiquement, nous obligeant à déterminer les tables nécessaires au fur et à mesure des éditions de liens. Par ailleurs les liaisons symbole/adresse mémoire ne pourront pas, elles non plus, être calculées statiquement. Les migrations d'objets ne font que rendre plus complexe cette situation.

Dans un premier temps, nous avons délibérément laissé de côté l'étude de la reproduction des paramètres de lancement de l'appel de l'application ainsi que la reproduction des entrées, sous leurs différentes formes (clavier, souris, etc). La reproduction de ces informations est à la charge de l'utilisateur qui en a la maîtrise. Il doit donc faire preuve de bonne volonté pour reproduire son exécution. La solution à ce problème consiste à conserver automatiquement pour chaque entrée, les données associées et à les retrouver lors de chaque réexécution. Ceci peut être réalisé en utilisant par exemple des classes d'entrées particulières (cf VII.3.1).

IV.2.3 Événements caractéristiques d'une exécution Guide

Il est extrêmement important pour la réexécution de bien identifier les événements qui caractérisent une exécution, car de leur choix dépend la fidélité de la reproduction. Ces événements sont propres à chaque système car ils dépendent du modèle d'exécution que celui−ci offre.

Nous avons analysé en section IV.2.2 l'incidence du modèle Guide sur la réexécution, analyse qui nous a permis d'identifier clairement les événements caractéristiques et donc nécessaires à toute réexécution. Nous présentons ci−après ces événements, en les structurant en fonction du rôle qu'ils jouent et en justifiant leur choix.

Appels de méthode

Ces événements permettent de connaître les appels de méthode effectués ainsi que les objets utilisés par l'application. Ils indiquent aussi la synchronisation des appels concurrents et permettent de la reproduire. Les événements que nous avons retenus sont :

- L'appel de méthode.
- Le retour de méthode.
- L'évaluation de la condition de synchronisation associée à une méthode.

Accès aux objets

Ces événements correspondent aux accès effectués par les activités à des objets. Ils permettent de reproduire les migrations d'objet et les extensions d'activité entre les sites. Les événements que nous avons choisi de conserver sont :

- La liaison d'un objet.
- La déliaison d'un objet.
- La demande de chargement d'un objet sur un site.
- La demande de déchargement d'un objet d'un site.

Création et disparition des entités

Ces événements concernent la création et la disparition (normale ou brutale) d'une entité. Celle−ci devra se reproduire au même moment lors des réexécutions. Ces événements sont :

- La création et la disparition d'un domaine.
- La création et la disparition d'une activité.
- La création et la disparition d'un objet (sa destruction).

IV.3 Comment enregistrer l'historique de l'exécution

L'enregistrement de l'historique est une opération importante, car d'elle dépend en grande partie la fidélité des reproductions qui en suivront. Nous avons de ce fait mené une étude sur les techniques d'enregistrement pouvant être utilisées dans le cadre de notre projet. Nous présentons dans les paragraphes qui suivent les approches étudiées, avant de nous pencher sur la solution que nous avons adoptée.

IV.3.1 Approches possibles

Nous avons présenté en section II.3.1.1 les différentes solutions apportées au problème de la récolte des informations d'une exécution d'une application : l'utilisation de matériel spécialisé, l'instrumentation du système, l'instrumentation des primitives d'une librairie, l'instrumentation du code objet et du code source du programme.

Nous avons constaté que toutes les solutions logicielles provoquent une perturbation de l'exécution qui est préjudiciable au bon déroulement du programme. Le temps nécessaire à la

sauvegarde des informations, aussi faible soit−il, introduit une perturbation et par conséquent peut modifier le comportement du programme.

La seule façon de ne pas perturber l'exécution du programme est d'utiliser du matériel spécialisé. Cependant, nous ne disposons pas de matériel spécialisé pour effectuer la collecte des informations d'une exécution et nous ne sommes pas intéressés par sa définition. En effet, nous pensons qu'affecter une machine particulière à la mise au point est une solution qui possède un inconvénient majeur : elle restreint la mise au point à de telles machines et devient alors un élément critique. Le programmeur peut développer ses programmes sur sa machine, mais ne peut pas les mettre au point si celle−ci n'est pas affectée à cette tâche ! Voilà qui est gênant !

L'instrumentation du programme, au niveau du code source ou du code objet, sont des solutions intéressantes pour suivre l'évolution de quelques variables lors d'une exécution mais ne sont pas adaptées pour effectuer une conservation systématique d'une trace complète d'une exécution, en vue de sa réexécution.

Nous préférons une solution système qui, par rapport à la solution matérielle, a l'avantage d'être disponible pour tous et, par rapport aux autres solutions, a l'avantage d'effectuer une conservation de l'historique de manière automatique sans que l'utilisateur ne soit mis à contribution. Cette solution permet de conserver l'historique complet d'une exécution sous la forme d'événements prédéfinis qui sont ceux qui vont permettre sa réexécution. Ces événements caractéristiques, que nous avons déterminés en section IV.2.3, sont reconnaissables par le système et peuvent donc être aisément signalés. De plus nous avons la maîtrise du système Guide et sommes en mesure d'effectuer les modifications du système qui sont nécessaires au signalement de ces événements et à leur conservation.

IV.3.2 Solution proposée : les processus Voyeur

Nous discutons ci−après les diverses formes que peut prendre un mécanisme système permettant de conserver automatiquement l'historique d'une exécution lorsque celle−ci s'exécute dans un mode particulier.

La façon la plus simple d'enregistrer l'historique d'une exécution est qu'il soit créé directement par les processus de l'application, qui se chargent à chaque apparition d'un événement de le conserver. Dans ce cas, l'exécution du programme est ralentie par le temps nécessaire à la conservation des événements, c'est−à−dire à leur écriture dans l'historique. Si de plus nous voulons appliquer un traitement à ces événements, par exemple réduire la taille des informations à conserver en les codant, l'exécution du programme se trouve ralentie du temps supplémentaire nécessaire à ce traitement.

Une autre solution pour générer l'historique d'une exécution est d'utiliser un processus différent de ceux mis en œuvre par l'exécution de l'application. Ces derniers se chargent uniquement d'identifier les événements, et c'est cet autre processus, appelé processus Voyeur, qui les récupère, les traite et donc crée l'historique de l'exécution. L'avantage de cette solution est que le processus Voyeur et les processus de l'exécution s'exécutent de façon concurrente, ce qui provoque une perturbation de l'exécution moins importante que la solution précédente.

De plus, cette solution nécessite peu de modifications du système, seul le signalement des événements est ajouté. Toute la partie traitement de ces événements, propre à la mise au point, est effectuée en dehors du système Guide.

Nous avons donc choisi d'utiliser cette dernière solution.

IV.3.3 Collecte globale des informations

L'organisation de la collecte des informations qui constituent l'historique dépend largement des particularités du système sous−jacent et, dans notre cas, essentiellement de la répartition et du parallélisme de l'exécution.

Le caractère réparti d'une exécution Guide nous place devant le choix de conserver l'historique de manière centralisée ou bien répartie :

- La solution centralisée implique une surcharge de la communication entre sites et surtout un phénomène de goulot d'étranglement sur le site de conservation, puisqu'elle nécessite le transport des événements vers le site de conservation.
- La solution répartie permet d'éviter ces inconvénients : elle conserve sur chaque site l'historique de la portion d'exécution qui a eu lieu sur ce site. Il semble par ailleurs naturel de profiter de la répartition de l'exécution pour conserver son historique.

D'autre part, l'exécution peut concerner plusieurs activités sur un même site du fait du parallélisme potentiel des applications Guide. Nous avons donc le choix entre conserver l'historique d'un site de manière séquentielle ou parallèle :

- La solution séquentielle nécessite une synchronisation entre les activités pour l'opération d'insertion d'un événement dans l'historique, ce qui implique une perturbation supplémentaire de l'exécution.
- La solution parallèle permet d'éviter cet inconvénient, car elle conserve l'historique du flot d'exécution de chaque activité de façon indépendante. Elle permet en outre de profiter du parallélisme de Guide pour effectuer cette tâche.

Nous avons choisi de collecter les informations de l'historique de manière parallèle et répartie. L'historique global de l'exécution sera donc constitué d'un ensemble d'historiques locaux qui eux mêmes regrouperont les historiques de leurs activités locales (cf Fig. 4.2).

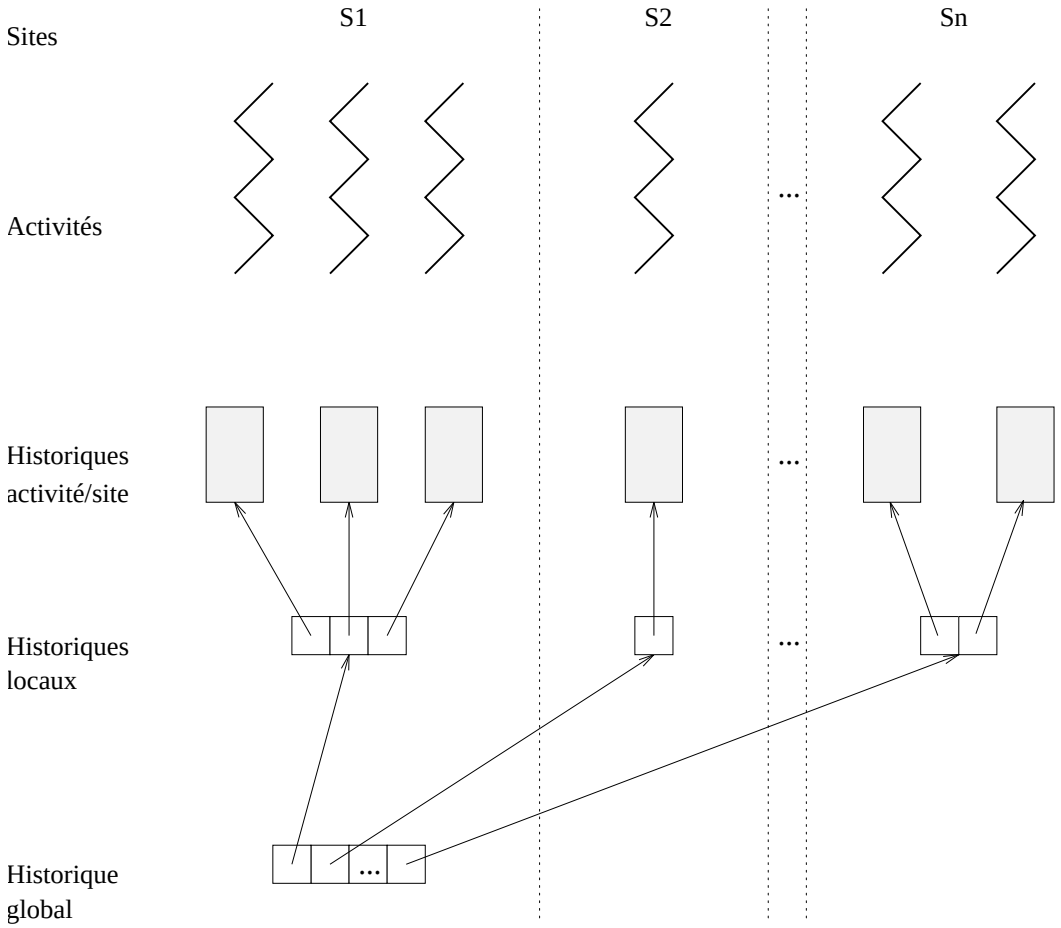


Fig. 4.2 : Organisation de l'historique d'une exécution

Nous avons expliqué en section IV.3.2 que pour conserver l'historique d'une exécution nous allions utiliser des processus Voyeurs. Pour obtenir un historique par activité et par site, la solution la plus simple consiste à associer un processus Voyeur à chaque représentant d'activité sur un site, celui−ci étant créé en même temps que ce représentant.

Cependant, cette solution coûte relativement chère en ressources, puisqu'elle double le nombre de processus. De plus, elle introduit une perturbation importante car la création d'un processus est une opération coûteuse.

Nous avons donc choisi d'utiliser un seul processus Voyeur pour conserver l'historique de tout un site. Ce processus se chargera alors de conserver les événements des activités de ce site dans des historiques différents (cf Fig. 4.3). Le premier processus Voyeur sera créé sur le site initial au lancement de l'application, les autres étant créés lors des extensions de l'application sur des nouveaux sites.

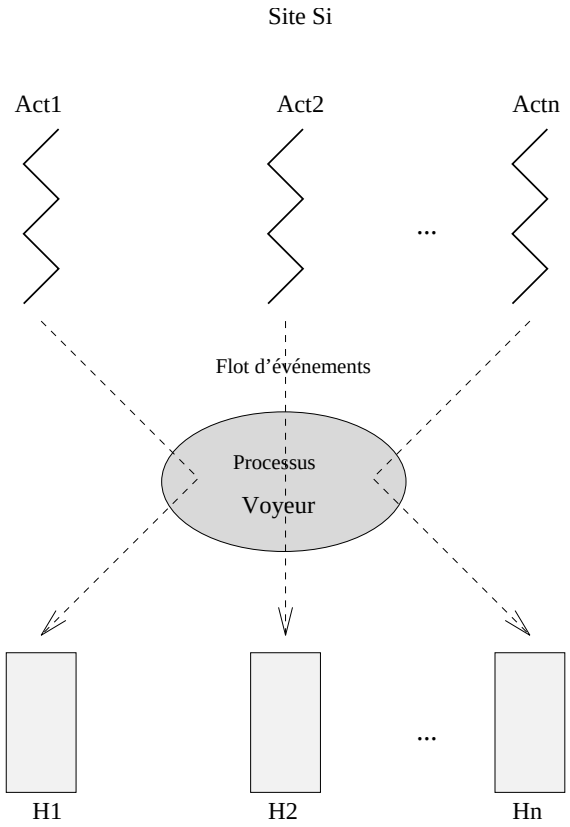


Fig. 4.3 : Conservation de l'historique sur un site

IV.4 Comment conserver le contexte initial de l'exécution

Nous avons vu en section III.1.2.4 l'importance de la réinitialisation automatique des objets persistants dans le cadre d'une réexécution. Cette réinitialisation peut être effectuée par le biais d'une copie du contexte initial de l'exécution qui est réinstallée avant chaque réexécution. Nous présentons dans la suite l'approche que nous avons suivie pour effectuer cette copie.

IV.4.1 Déterminer automatiquement le contexte initial

L'identification du contexte initial d'une exécution par l'utilisateur nous paraît difficilement réalisable pour deux raisons essentielles :

- Le modèle d'objets de Guide pousse l'utilisateur à utiliser un grand nombre d'objets persistants. Le contexte initial peut de ce fait être très volumineux.
- La composition exacte du contexte initial est difficile à cerner. Les relations complexes qui peuvent se nouer entre les objets à travers leurs références expliquent cette difficulté.

Demander cette identification à l'utilisateur nous semble donc irréaliste. L'identification automatique du contexte initial nous paraît plus intéressante, autant pour le confort de l'utilisateur que pour la bonne marche de l'outil de mise au point.

IV.4.2 Copie au vol du contexte initial

Nous proposons d'utiliser un mécanisme de copie d'objets qui effectue une copie des objets composant le contexte initial de cette exécution, et qui permet de retrouver une copie fraîche au début de chaque réexécution de ce programme.

La copie de ces objets ne peut s'effectuer qu'au cours de l'exécution. Elle ne peut pas en effet se faire avant l'exécution, car les objets constituant le contexte initial ne sont pas encore identifiés. Elle ne peut pas se faire non plus après l'exécution, car l'état de ces objets a pu être modifié par celle−ci.

Le contexte initial d'une exécution est donc déterminé et conservé de façon incrémentale, lors du premier accès aux objets, et se trouve complètement défini à la fin de l'exécution. L'édition de liens dynamique que Guide effectue peut nous être d'une grande utilité dans cette tâche. En effet, quand un objet est référencé pour la première fois, le système Guide détecte son absence et demande une liaison dynamique : le contexte de l'application va donc se constituer au fur et à mesure de ces défauts d'objets. Une façon simple de déterminer le contexte initial d'une application, et de le conserver, consiste à s'informer des défauts d'objets et à effectuer une copie de leur état au moment de la première liaison d'un objet par l'exécution.

IV.4.3 Informations à conserver

Pour minimiser le volume des informations à conserver, nous avons par ailleurs choisi de ne conserver que les informations strictement nécessaires à la reproduction de l'exécution. Nous effectuons donc une copie des objets sans prendre en compte leur composition (les objets référencés par un objet). C'est lors de l'accès à chaque objet référencé que l'on effectuera sa copie.

De plus, la copie au vol des objets ne concerne qu'un sous−ensemble du contexte d'exécution de l'application : il s'agit des objets utilisés mais non créés par celle−ci. Les autres, ceux qui sont créés au cours de cette exécution, seront recréés lors des exécutions successives. Il est donc inutile d'en conserver une copie.

Pour obtenir le même comportement lors des réexécutions, nous devons conserver, en plus de sa valeur, la nature répartie du contexte initial de l'exécution : le site du premier accès à l'objet.

IV.5 Comment isoler l'exécution

Nous avons vu en section IV.2.2 que le partage d'objets nécessite d'isoler l'exécution mise au point, pour se protéger des interventions externes. Nous discutons ci−après les modalité de la réalisation de cet isolement.

IV.5.1 Reproduction des interventions externes

La reproduction des interventions externes comporte au moins deux inconvénients majeurs :

- La détection des interventions externes nécessite des mécanismes capables de déterminer qu'une application accède à un objet, de l'identifier et surtout de se rendre compte de l'action qu'elle effectue sur l'objet. Un tel mécanisme n'est pas simple à réaliser. Il aurait par ailleurs l'inconvénient de gêner les applications qui ne sont pas en cours de mise au point.
- La reproduction des interventions externes peut elle aussi nécessiter des mécanismes complexes. Il faudra pouvoir recalculer au minimum la valeur des objets partagés avec l'application mise au point après chaque intervention.

Nous proposons d'empêcher ces interventions externes en isolant l'application mise au point lors de l'enregistrement comme lors de la reproduction d'une exécution.

IV.5.2 Délimiter l'application

L'isolement d'une application ne doit pas se faire aux dépens des applications coopérantes, dont la communication se fait au moyen d'objets partagés. Un exemple d'applications coopérantes est celui basé sur le schéma producteur/consommateur : une application (le producteur) produit des informations qu'elle stocke dans un objet, tandis qu'une seconde application (le consommateur) les récupère depuis ce même objet et les utilise dans son traitement. La mise au point isolée d'une de ces deux applications n'a pas de sens, car la communication entre producteur et consommateur ne sera plus possible.

Il est important, pour la mise au point des applications qui se servent d'objets partagés pour mettre en œuvre une coopération entre elles, de bien délimiter leurs frontières, avant d'avoir recours à toute forme d'isolement. Dans l'exemple précédent, les deux applications n'en font qu'une structurée à l'aide d'activités parallèles. Elles doivent par conséquent être mises au point comme un tout.

IV.5.3 Difficultés pour obtenir un isolement automatique

L'isolement de l'exécution mise au point doit se faire sans gêner le déroulement des autres applications : elle ne doit pas interdire l'accès aux objets qu'elle utilise (exemple : le catalogue). Nous avons donc besoin de deux versions d'un même objet : l'une est utilisée par l'application mise au point et l'autre est disponible pour les autres applications. Pour y parvenir, nous pouvons soit prendre des copies spécifiques à la mise au point de ces objets, soit faire en sorte que l'application mise au point soit la seule à connaître ces objets. Dans les deux cas, il n'y aura plus de partage.

La solution automatique, qui consiste à prendre une copie spécifique des objets utilisés, est difficilement réalisable avec la version actuelle du système Guide. La cohabitation de deux versions d'un même objet demande de profondes modifications au système.

Par contre, l'isolement est beaucoup plus simple à atteindre par l'utilisateur qui peut facilement exécuter son application dans un cadre particulier, où elle est la seule à connaître les objets qu'elle utilise.

Cette exécution en monde fermé correspond bien à l'étape de mise au point. Nous sommes par ailleurs convaincus que c'est ce qui est fait normalement par tout programmeur : tester son application en simulant les conditions réelles de son application ; rares sont les programmes mis au point dans les conditions réelles d'exploitation. La découverte d'une erreur dans un programme déjà livré à l'exploitation entraîne d'ailleurs un retour à l'étape de mise au point.

IV.6 Comment reproduire l'exécution

Nous effectuons la reproduction de l'exécution par une réexécution de l'application guidée par l'historique. Nous décrivons ci−après la façon dont nous obtenons une parfaite correspondance entre les différentes exécutions.

IV.6.1 Lancer une nouvelle exécution

Dans la phase de reproduction de l'exécution, nous devons produire une exécution équivalente à celle qui a eu lieu lors de la phase d'enregistrement.

Nous ne devons pas reproduire uniquement le comportement de l'application mise au point, mais aussi et surtout permettre la mise au point de l'application à partir d'une véritable exécution. Ainsi l'utilisateur peut obtenir toutes les informations qu'il veut, sans limitation sur leur nature ni sur leur validité.

Ce service ne peut pas être rendu par simple simulation de l'exécution puisque l'on reproduit uniquement le comportement de celle−ci. Les informations qui sont disponibles à ce moment là correspondent uniquement à celles conservées lors de la première exécution, ce qui n'est pas toujours suffisant. Il faudrait conserver une quantité d'informations considérable pour offrir à l'utilisateur un niveau d'information suffisant lors de la reproduction de l'exécution.

Nous avons donc choisi de reproduire une exécution à partir d'une nouvelle exécution de l'application que nous dirigeons, grâce aux informations contenues dans l'historique.

IV.6.2 Réinstaller son contexte initial

Pour que les réexécutions produisent le même comportement, elles doivent utiliser le contexte initial que nous avons conservé et suivre le cours de l'exécution décrit par l'historique.

Le contexte initial de l'exécution est réinstallé avant chaque réexécution : chaque objet est remis dans son état initial sur le site où il a été accédé pour la première fois par l'exécution.

IV.6.3 Contrôler et vérifier le déroulement de l'exécution

Nous nous servons des informations conservées dans l'historique pour diriger la nouvelle exécution et faire en sorte que les mêmes événements soient exécutés dans le même ordre. La réexécution effectue donc un pas à pas sur les événements conservés dans l'historique en vérifiant la correction de chaque pas.

Nous garantissons la reproduction correcte de l'exécution conservée, car nous arrêtons la réexécution si nous obtenons un comportement de l'exécution qui diverge de celui enregistré dans l'historique. Ce phénomène de divergence, qui se traduit par un événement incorrect, correspond à une erreur dans l'application mise au point (par exemple, un appel de méthode non synchronisé).

Nous donnons ci−après plus de détails sur le comportement des activités et du processus Voyeur lors de la reproduction (cf Fig. 4.4).

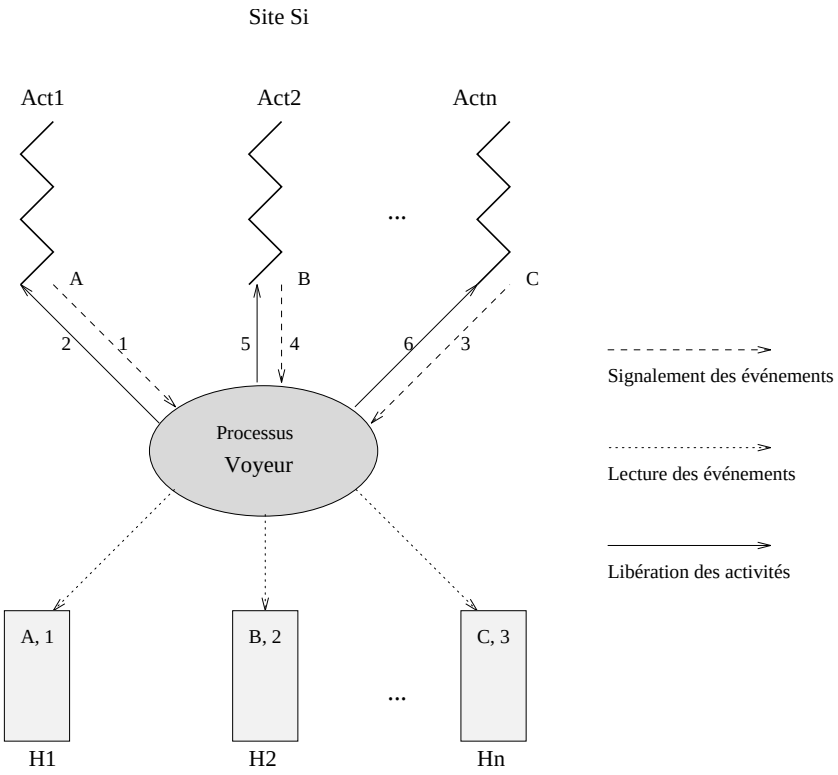


Fig. 4.4 : Reproduction de l'exécution sur un site

Lors de la réexécution, le signalement d'un événement ne signifie plus l'exécution de l'action associée, mais une demande d'exécution de cette action. L'activité, après avoir signalé un événement, attend (en se bloquant) la permission du processus Voyeur pour continuer.

C'est le processus Voyeur qui dirige la reproduction de l'exécution sur son site, et libère les activités prêtes suivant l'ordre conservé dans l'historique. Avant de libérer une activité, il vérifie la correction de l'événement qu'elle s'apprête à exécuter.

IV.7 Comment faciliter l'observation

Nous avons étudié, dans les sections précédentes, la reproduction d'une exécution dans le système Guide. Nous présentons maintenant l'étude de l'observation d'une application Guide.

Les résultats présentés dans cette section sont en partie le fruit d'un travail préliminaire sur la visualisation dans la mise au point que le lecteur intéressé peut trouver dans [4].

IV.7.1 Observation de l'exécution

Nous avons présenté en section III.3.3 l'intérêt d'un outil d'observation de l'exécution pour la mise au point d'une application. Nous étudions ci−après comment cette observation peut être réalisée dans le cadre de Guide.

Les entités que l'utilisateur est intéressé à voir lorsqu'il observe l'exécution d'une application Guide sont les domaines et les activités créés, les objets utilisés, ainsi que les appels de méthodes qui sont exécutés.

Nous avons vu en section III.3.3.1 que toutes ces informations sont contenues dans l'historique de l'exécution et qu'elles peuvent être connues par simple parcours de l'historique. Pour obtenir plus de renseignements sur cette exécution, une étude plus complète de l'historique est nécessaire. Cette étude permet de connaître par exemple : le comportement réparti de l'exécution (extension de domaine et d'activité, migration d'objet), le partage d'objets entre activités et la synchronisation des accès concurrents.

Nous proposons de profiter de la connaissance apportée par ces informations pour offrir à l'utilisateur plusieurs vues complémentaires, très utiles pour suivre l'exécution de son application. Chacune de ces vues permet l'observation de l'exécution suivant un aspect particulier.

IV.7.1.1 Vues basées sur les domaines et les activités

Nous proposons tout d'abord deux vues qui montrent l'exécution en se basant sur les domaines et les activités.

Vue globale des domaines et des activités

> Cette vue permet de connaître l'ensemble des domaines et des activités mis en œuvre par l'exécution. Elle met en évidence la filiation entre les domaines et les activités et montre ainsi la structuration de l'exécution en termes de domaines et d'activités.

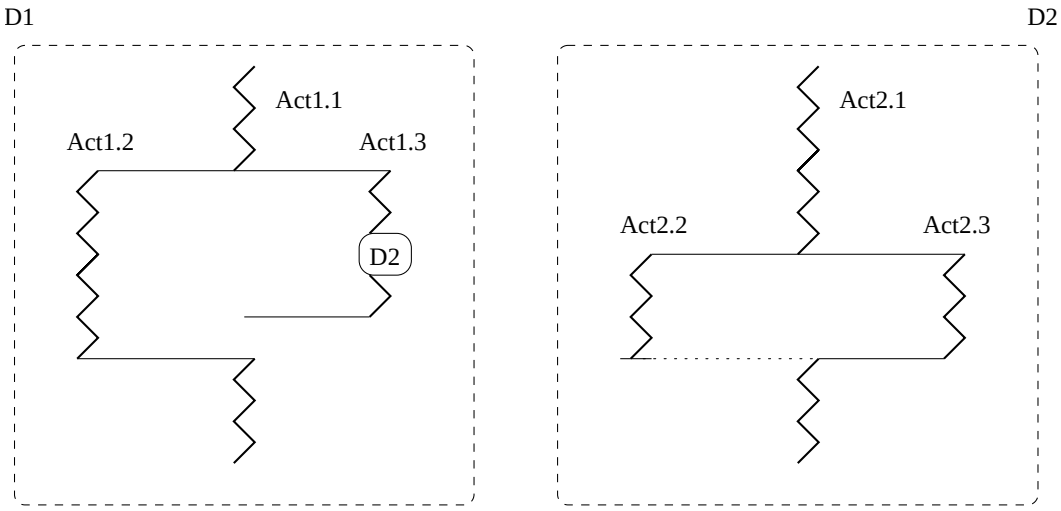


Fig. 4.5 : Vue globale des domaines et des activités

La figure Fig. 4.5 nous montre une exécution composée de deux domaines *D1* et *D2*. Le premier domaine *D1* contient initialement une première activité *Act1.1* qui crée par la suite deux activités filles *Act1.2* et *Act1.3*. A la fin de l'activité *Act1.2* l'activité mère reprend son exécution. On remarque qu'à ce moment l'activité *Act1.3* a aussi terminé. Le domaine *D2* est créé par l'activité *Act1.3* et est composé initialement de l'activité *Act2.1* qui crée par la suite deux activités filles *Act2.2* et *Act2.3*. A la fin de l'activité *Act2.3* l'activité *Act2.2* est tuée et l'activité mère reprend son exécution.

En selectionnant une activité parmi celles présentées, on peut accéder à la vue d'une activité.

Vue d'une activité

Cette vue permet de connaître le flot d'exécution suivi par une activité. Elle montre la suite des appels de méthode effectués et les objets utilisés.

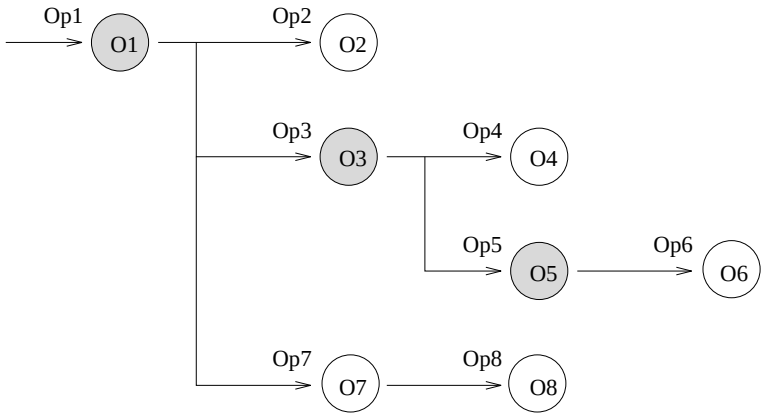


Fig. 4.6 : Vue d'une activité

La figure Fig. 4.6 montre le flot d'exécution suivi par une activité. On voit que l'appel de la méthode *Op1* sur l'objet *O1* provoque les appels successifs *O2.Op2*, *O3.Op3* et *O7.Op7*.

Une animation appropriée de ces deux vues permet à l'utilisateur de suivre le déroulement de son exécution avec les domaines et les activités comme centre d'intérêt, de façon générale avec la première et avec plus de détails avec la deuxième.

La répartition de l'exécution dans ces deux vues se manifeste par le phénomène d'extension des domaines et des activités. Elle peut être montrée ou cachée, suivant les désirs de l'utilisateur.

IV.7.1.2 Vues basées sur les objets

Une autre façon de voir l'exécution de l'application est de s'intéresser aux objets. Les deux vues que nous présentons ci−après montrent l'exécution en se basant sur les objets.

Vue globale des objets

Cette vue permet de connaître l'ensemble des objets utilisés par l'exécution. Elle met en évidence pour chaque objet l'ensemble des objets qu'il utilise et de ceux qui l'utilisent.

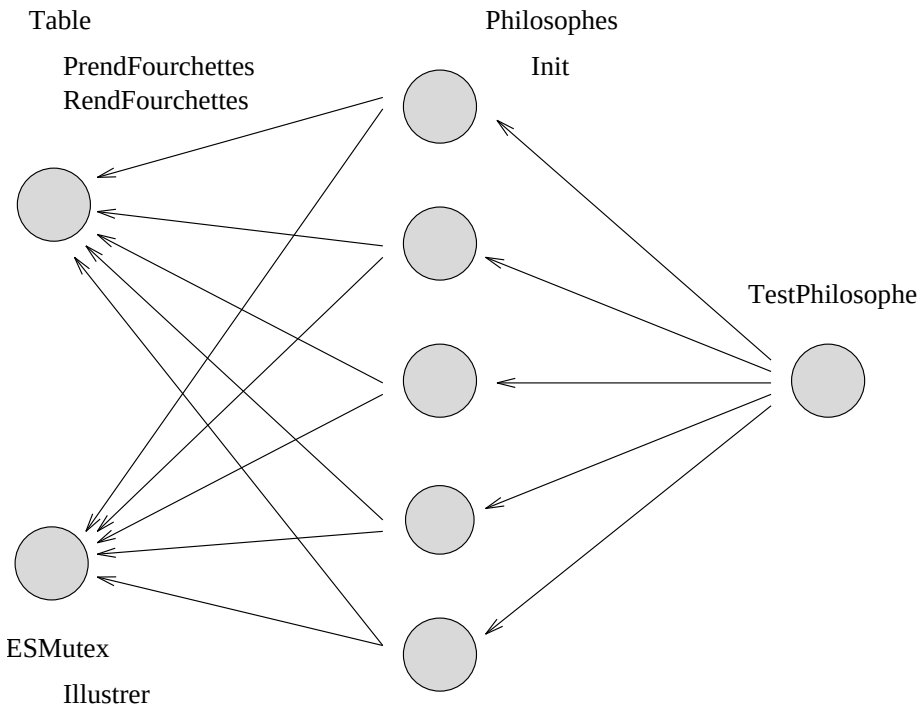


Fig. 4.7 : Vue globale des objets

La figure Fig. 4.7 montre l'ensemble des objets qui sont utilisés par une exécution. On remarque qu'un objet *Philosophe* utilise un objet *Table* par le biais des méthodes *Prend-Fourchettes* et *RendFourchettes* et un objet *ESMutex* par le biais de la méthode *Illustrer*.

En choisissant un objet parmi ceux présentés, on peut accéder à la vue d'un objet.

Vue d'un objet

Cette vue permet de connaître l'utilisation qui est faite d'un objet. Elle montre la suite des appels de méthode qui sont effectués par les différentes activités de l'exécution sur cet objet. Elle met en évidence le partage d'objets et la synchronisation des appels concurrents qui traduisent la communication entre les activités.

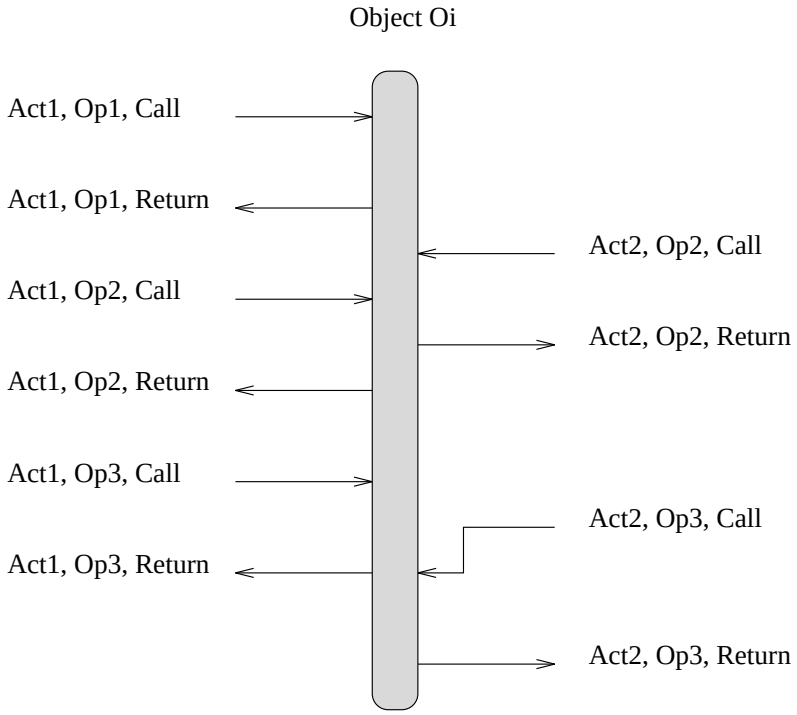


Fig. 4.8 : Vue d'un objet

La figure Fig. 4.8 montre la séquence des appels de méthodes qui sont effectués sur un objet. On remarque que les deux appels à la méthodes *Op2* effectués par les deux activités *Act1* et *Act2* s'exécutent en parallèle alors que l'appel à la méthode *Op3* effectué par l'activité *Act2* ne s'exécute que lorsque l'appel à la méthode *Op3* effectué par l'activité *Act1* se termine. On peut déduire de ceci que la condition de synchronisation associée à la méthode *Op3* est exclusive, ce qui n'est pas le cas de la méthode *Op2*.

Une animation appropriée de ces deux vues permet à l'utilisateur de suivre le déroulement de son exécution avec les objets comme centre d'intérêt, de façon générale avec la première et avec plus de détails avec la deuxième.

La répartition de l'exécution dans ces deux vues se manifeste par la localisation et la migration des objets. Elle peut être montrée ou cachée, suivant les désirs de l'utilisateur.

IV.7.2 Observation des données

Nous avons énoncé en section III.3.4 les critères que doit respecter un outil d'observation des données. Suivant ces critères nous avons conçu et développé un outil qui permet de visualiser de manière graphique l'état d'un objet Guide (voir la valeur de ses variables) et qui permet de naviguer dans sa composition (accéder aux objets référencés par cet objet). Nous présentons succinctement les principales caractéristiques de cet outil.

L'outil de visualisation et de navigation a été conçu pour être utilisable par n'importe quelle application Guide. En particulier, le metteur au point peut faire appel à lui pour visualiser les objets utilisés par un programme lors de sa mise au point. Cet outil se comporte comme un serveur auquel les autres outils sous−traitent l'affichage des objets. Ses principales fonctions sont la visualisation d'un objet, l'effacement d'un objet visualisé et le rafraîchissement de l'affichage d'un objet.

Cet outil offre aussi des fonctions qui facilitent la gestion de l'affichage de l'état d'un objet. L'utilisateur peut définir pour une classe donnée un niveau de visualisation et un filtre, qui seront pris en compte lors de la visualisation d'un objet instance de cette classe. Un filtre permet à l'utilisateur de choisir les variables qu'il veut visualiser, les autres étant cachées. Le niveau de visualisation permet à l'utilisateur de choisir le niveau de détail avec lequel il veut voir un objet. Ces fonctions d'affichage associées à chaque objet visualisé peuvent aussi être modifiées dynamiquement.

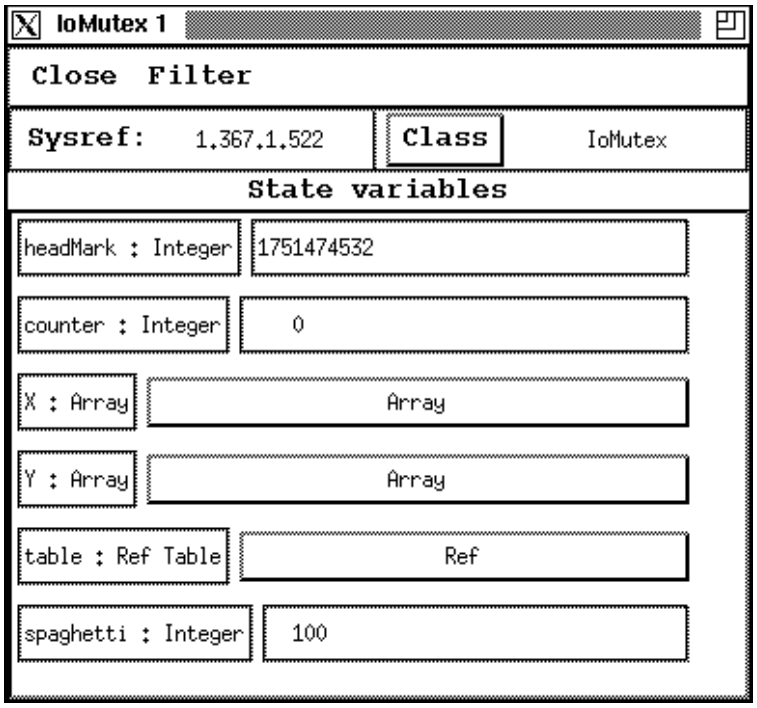


Fig. 4.9 : Observation de l'état d'un objet

En ce qui concerne l'affichage des variables, celles des types élémentaires sont représentées directement par leurs valeurs alors que les variables structurées (tableau, enregistrement, liste, référence, etc) sont représentées partiellement, en ne montrant qu'un seul

niveau d'imbrication à la fois. L'accès aux valeurs d'une variable dont la structure est complexe (un tableau d'enregistrements contenant des listes) se fait donc de manière progressive : l'utilisateur n'est pas submergé par l'affichage de tout son contenu, il le découvre de façon incrémentale.

Dans le cadre de la mise au point de programme, l'outil de visualisation graphique des objets permet d'observer l'état des objets utilisés. Il manque cependant une fonction importante pour la mise au point qui est la possibilité de modifier cet état.

IV.7.3 Observation des sources

Nous avons introduit en section III.3.5 les motivations pour un outil de navigation dans les sources qui sont d'une part l'accès au grand nombre de composants constituant le source d'une application écrite dans un langage à base d'objets et d'autre part le parcours des relations qui relient ces composants. Nous avons conçu et développé un tel outil pour le langage Guide dont nous présentons ci−après les principales caractéristiques.

Les sources des applications écrites dans le langage Guide se décomposent en types et classes. Le type définit l'interface d'accès pour une catégorie d'objets et une classe en définit une réalisation.

L'outil de navigation dans les sources permet d'une part de visualiser la définition des types et des classes et d'autre part de parcourir les nombreuses relations qui existent entre ces entités, qui sont les relations : sous−type, sous−classe, réalisation, référence et création d'instances [37].

La visualisation d'un type montre :

- son nom,
- la liste des signatures des méthodes qui sont applicables aux objets de ce type,
- la liste des exceptions que ce type est susceptible de signaler (mécanisme d'exception),

et permet d'accéder à :

- son super−type,
- la liste des sous−types de ce type,
- la liste des classes qui réalisent ce type,
- la liste des types qui référencent ce type (variable d'état visible, paramètre de méthode),
- la liste des classes qui référencent ce type (variable d'état, variable de travail),
- son source.

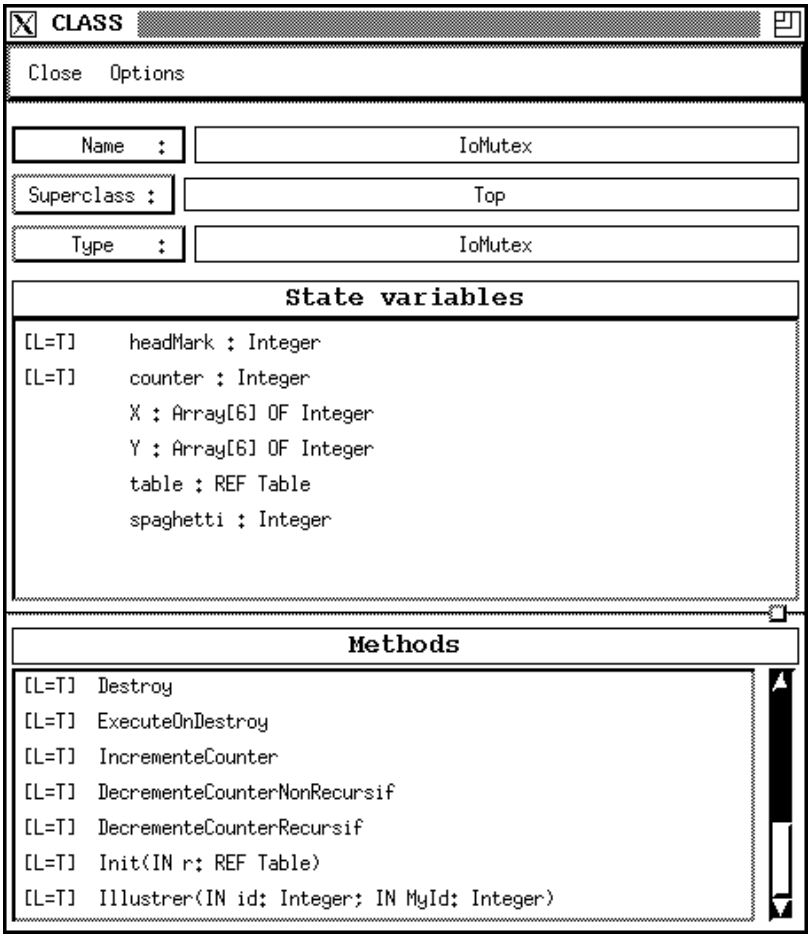


Fig. 4.10 : Observation d'une classe

La visualisation d'une classe montre les informations suivantes :

- son nom,
- la liste des variables définissant l'état des objets de cette classe,
- la liste des signatures des méthodes qui sont applicables aux objets de cette classe,
- la liste des procédures qui sont déclarées dans cette classe,

et permet d'accéder à :

- son type,
- sa super−classe,
- la liste des sous−classes de cette classe,
- la liste des classes qui créent des instances de cette classe,
- son source.

L'accès à ces informations facilite la compréhension de la structuration de l'application.

Chapitre V

Thésée : mise en œuvre du noyau de réexécution

Nous avons présenté au chapitre précédent la conception d'un outil pour la mise au point d'applications dans le cadre du système Guide. Nous avons notamment étudié les problèmes posés par la reproduction d'une exécution en Guide ainsi que ceux liés à son observation.

Nous présentons ci−après la mise en œuvre du noyau de réexécution Thésée qui est le composant central de l'outil de mise au point pour Guide.

V.1 Présentation générale de Thésée

Thésée est un héros grec, fils d'Egée et roi d'Athènes qui se distingua par la destruction de nombreux monstres dont le Minotaure. Le Minotaure est un monstre mi−homme, mi−taureau, enfermé dans le Labyrinthe de Crète construit par Dédale, qui se nourrissait d'un tribut annuel de chair humaine (sept jeunes filles et sept jeunes gens jetés en pâture au monstre). Thésée alla combattre ce monstre et le tua. Il put se guider dans le labyrinthe et parvenir au repaire du Minotaure grâce à un peloton de fil que lui avait remis Ariane, fille de Minos, qui s'était éprise de notre héros.

Thésée aujourd'hui est un outil de réexécution de programmes Guide. Il permet de reproduire une exécution en se servant de son historique (le fil d'Ariane) et facilite ainsi la découverte et la correction des erreurs d'exécution (la destruction du Minotaure dans son repaire). La correction de ces erreurs, qui sans cette aide est une tâche ardue (demandant de nombreuses souffrances), devient facile grâce à la puissance (la vaillance) de Thésée.

V.1.1 Les services offerts

Les services de base devant être offerts par un noyau de réexécution ont été présentés en section III.4.2. Nous détaillons ci−après ceux offerts par le noyau de réexécution Thésée.

- Enregistrer une exécution

  Thésée permet d'exécuter une application en conservant automatiquement un historique de l'exécution ainsi que son contexte initial.

  Ces deux fonctions réalisées à l'aide de mécanismes du système Guide sont étudiées en sections V.2 et V.3.

- Obtenir des renseignements sur cette exécution

  Thésée permet d'obtenir un grand nombre de renseignements sur l'exécution enregistrée. Ces informations, qui caractérisent le déroulement de l'exécution, sont notamment utilisées par les outils d'observation de l'exécution pour montrer le comportement statique de l'exécution (chemins parcourus, ressources utilisées).

  Ces informations sont rendues disponibles par une analyse de l'historique conservé qui est détaillée en section V.4.

- Reproduire l'exécution enregistrée

  Thésée permet de reproduire l'exécution enregistrée (cf V.5). Cette réexécution demande de réinstaller le contexte initial de l'exécution et s'effectue en lançant une nouvelle exécution de l'application qui est dirigée par les informations contenues dans l'historique. Nous pouvons distinguer essentiellement deux sous−services :

  * Suivre la progression de l'exécution

    Thésée permet de suivre la progression de l'exécution puisqu'il signale des événements caractéristiques de cette exécution. Ces informations sont notamment utilisées par des outils d'observation de l'exécution pour montrer le comportement dynamique de l'exécution.

  * Contrôler la réexécution

    Thésée offre les fonctions classiques de contrôle de l'exécution (définition de points d'arrêts, reprise de l'exécution, etc.) qui facilitent la recherche des erreurs.

V.1.2 Les clients de Thésée

Les clients du noyau de réexécution sont naturellement les outils qui participent à la mise au point d'une application. Nous avons vu en section III.4.1 que la mise au point d'un programme Guide est réalisée par la coopération de plusieurs outils dont le noyau de réexécution est l'outil central :

- Le frontal de mise au point s'adresse à Thésée pour lancer l'enregistrement de l'exécution, lancer la reproduction, contrôler la reproduction, etc.
- Les outils d'observation de l'exécution obtiennent de Thésée les informations dont ils ont besoin pour montrer le comportement de l'exécution (avant et pendant la réexécution).
- Les outils d'observation des données dialoguent avec Thésée pour montrer l'état des objets Guide utilisés.
- Les outils d'observation des sources coopèrent avec Thésée pour montrer le programme des types et des classes Guide utilisées.

Mais, l'utilisation du noyau de réexécution ne se limite pas au seul contexte de la mise au point. Beaucoup d'autres outils peuvent apprécier les services offerts par Thésée, notamment ceux qui s'intéressent à une exécution :

- Des outils d'observation d'une exécution, comme l'Observer [68] [38] peuvent utiliser les événements signalés par Thésée pour montrer le comportement dynamique d'une exécution.
- Des outils d'analyse de traces peuvent se servir de l'historique conservé comme information de base à leurs traitement. Pour certains, les informations contenues dans l'historique sont entièrement satisfaisantes. Pour d'autres, comme des outils d'analyse de performance ou d'analyse du parallélisme, il est nécessaire de modifier le noyau de réexécution pour ajouter certaines informations dans l'historique, telle que la notion de temps d'exécution.
- etc.

V.1.3 Décomposition en objets

Le noyau de réexécution Thésée est une application écrite en grande partie en langage Guide. Nous donnons ci−après la décomposition en objets Guide de Thésée.

V.1.3.1 L'objet Thésée

L'objet Thésée est le frontal du noyau de réexécution. C'est par cet objet que les clients du noyau de réexécution ont accès aux services présentés en section V.1.1.

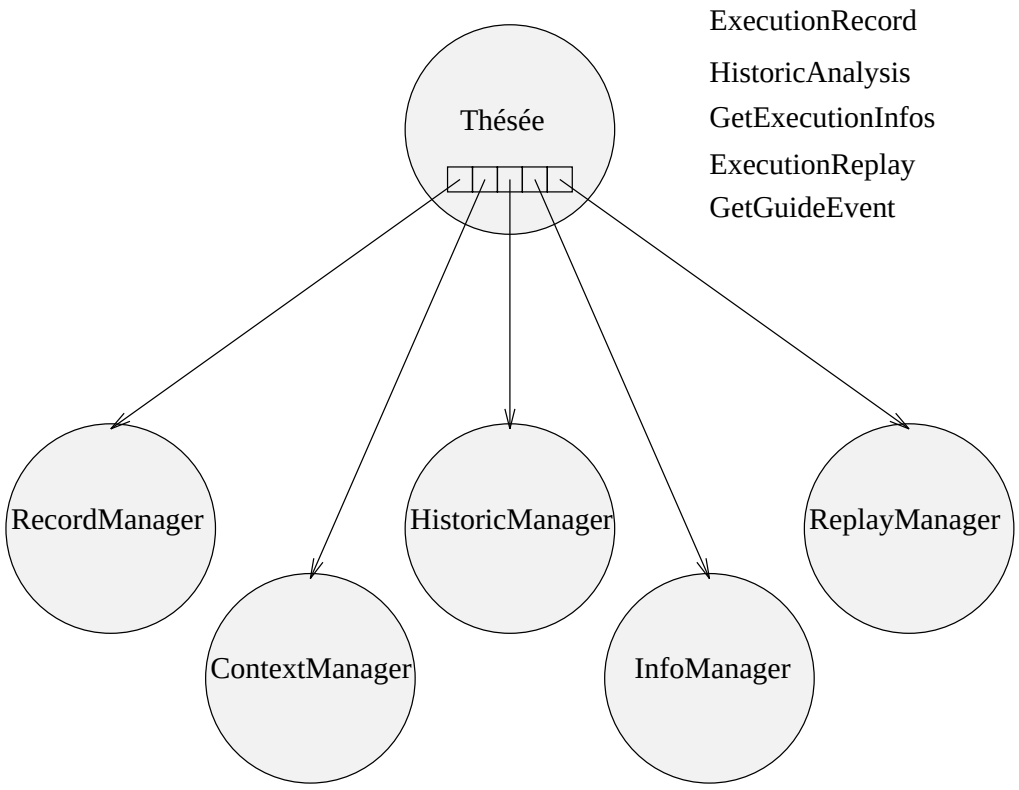


Fig. 5.1 : Décomposition en objets de Thésée

La figure Fig. 5.1 donne le nom des principales méthodes d'accès de Thésée et montre les objets qu'il référence :

- La méthode *ExecutionRecord* enregistre une exécution d'une application en se servant de l'objet *RecordManager*. A la fin de l'exécution de l'application, son historique et son contexte initial sont conservés de façon permanente par les objets *HistoricManager* et *ContextManager*.
- La méthode *HistoricAnalysis* effectue l'analyse de l'historique et conserve les résultats dans l'objet *InfoManager*. Ces informations sont rendues disponibles par la méthode *GetExecutionInfos*.
- La méthode *ExecutionReplay* reproduit l'exécution enregistrée : elle utilise pour cela l'objet *ReplayManager*.
- La méthode *GetGuideEvent* permet de récupérer les événements signalés par la réexécution et donc de suivre sa progression.

Nous décrivons ci−après chacun des objets référencés par Thésée.

V.1.3.2 L'objet RecordManager

L'objet *RecordManager* se charge de lancer une exécution d'une application en mode Trace et de conserver de façon permanente son historique et son contexte initial (cf Fig. 5.2) :

- La méthode *ExecutionRecord* lance une exécution en mode Trace, et enregistre son historique dans un ensemble de fichiers Unix : un par activité et par site visité (cf V.2).
- La méthode *HistoricSave* conserve ces fichiers historique dans des objets Guide, référencés par l'objet *HistoricManager*.
- La méthode *ContextIdentification* identifie les objets qui composent le contexte initial de cette exécution à partir des informations contenues dans l'historique (cf V.3.3), et conserve cette information dans des objets Guide, référencés par l'objet *ContextManager*.

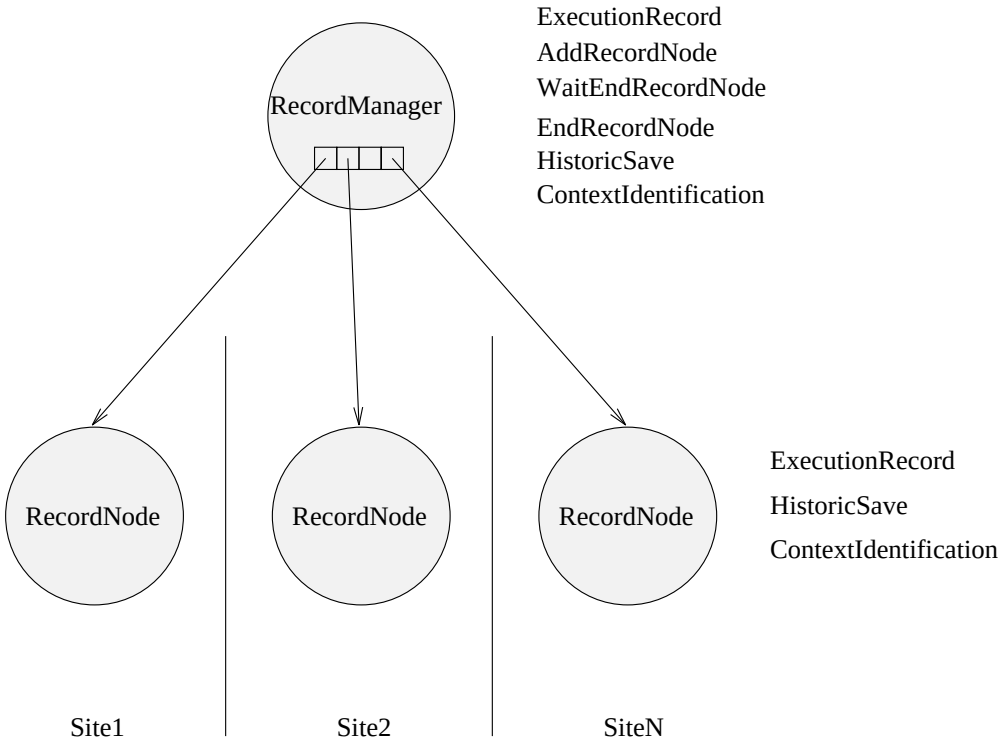


Fig. 5.2 : Décomposition en objets de RecordManager

L'objet *RecordManager* est composé d'objets *RecordNode*, un par site visité par l'exécution. Ces objets se chargent, sur chaque site, de l'enregistrement de l'exécution (méthode *ExecutionRecord*), ainsi que de la conservation permanente de l'historique (méthode *HistoricSave*) et du contexte initial (méthode *ContextIdentification*).

Le premier objet *RecordNode* est créé sur le site de lancement de l'exécution et doit attendre (méthode *WaitEndRecordNode*) la fin de l'enregistrement des exécutions sur les autres sites (méthode *EndRecordNode*) avant de se terminer. A chaque extension de domaine sur un nouveau site, un objet *RecordNode* est créé et enregistré dans l'objet *RecordManager* (méthode *AddRecordNode*).

V.1.3.3 L'objet HistoricManager

L'objet *HistoricManager* conserve de façon permanente l'historique de l'exécution et offre les fonctions d'accès aux événements qui le composent (la liste de ces événements est présentée en section V.2.4) ainsi que la fonction d'analyse de l'historique :

- La méthode *HistoricAnalysis* effectue l'analyse de l'historique et conserve le résultat dans l'objet *InfoManager*.
- Les méthodes *GetFirstEvent* et *GetNextEvent* permettent d'accéder séquentiellement aux événements de l'historique.

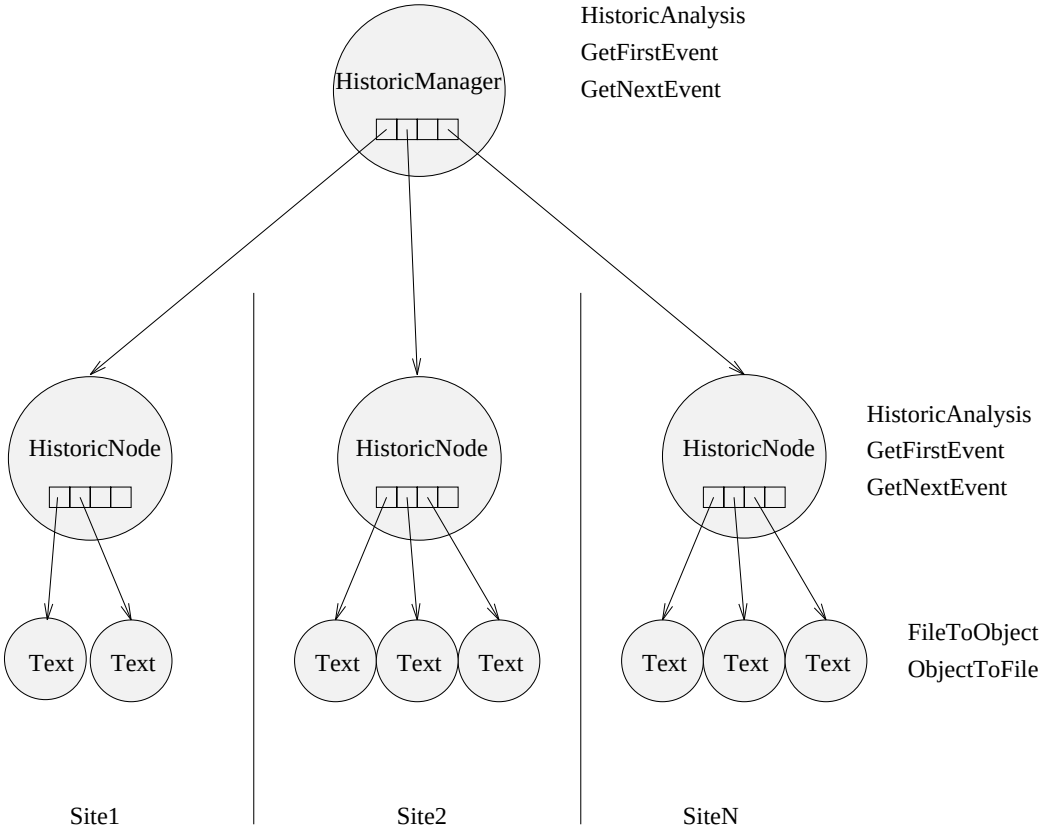


Fig. 5.3 : Décomposition en objets de HistoricManager

L'objet *HistoricManager* est composé d'objets *HistoricNode* qui conservent l'historique de la portion d'exécution du site qui les concerne. Ils se chargent également de l'analyse de l'historique (méthode *HistoricAnalysis*) et de l'accès aux événements (méthodes *GetFirstEvent* et *GetNextEvent*) de ce site.

L'historique de chaque activité sur un site est stocké dans un objet *Text*. Un objet *Text* constitue une interface d'accès Guide à un fichier Unix : il possède deux méthodes permettant de transférer le contenu d'un fichier dans son état et inversement.

V.1.3.4 L'objet ContextManager

L'objet *ContextManager* conserve de façon permanente l'identification des objets composant le contexte initial de l'exécution et offre les deux fonctions suivantes :

- La méthode *ContextAbort* remet les objets qui composent le contexte initial dans leur état initial (utilisée avant chaque réexécution et lorsque l'on veut enregistrer une nouvelle exécution se déroulant avec les mêmes données).
- La méthode *ContextCommit* valide l'état final obtenu par les objets utilisés par l'exécution : leur état initial devient celui atteint en fin d'exécution. Ceci permet d'enregistrer une nouvelle exécution prenant en compte les modifications effectuées par la précédente exécution ou réexécution.

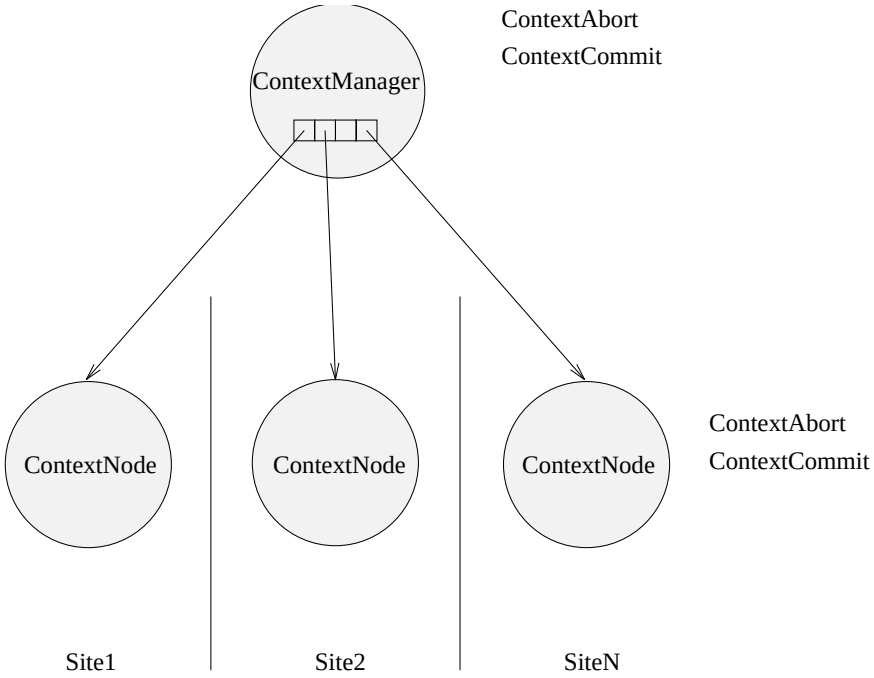


Fig. 5.4 : Décomposition en objets de ContextManager

L'objet *ContextManager* est composé d'un objet *ContextNode* par site qui conserve l'identification du contexte initial de la portion d'exécution qui le concerne et se charge sur chaque site : de réinstaller ce contexte initial (méthode *ContextAbort*) et de valider l'état final obtenu (méthode *ContextCommit*).

V.1.3.5 L'objet InfoManager

L'objet *InfoManager* conserve le résultat de l'analyse de l'historique et offre des fonctions d'accès à ces informations. Il permet notamment de connaître : les entités intervenant dans l'exécution (domaines et activités créés, objets utilisés), la répartition de cette exécution (extensions de domaines et d'activités, localisation et migration d'objets), etc.

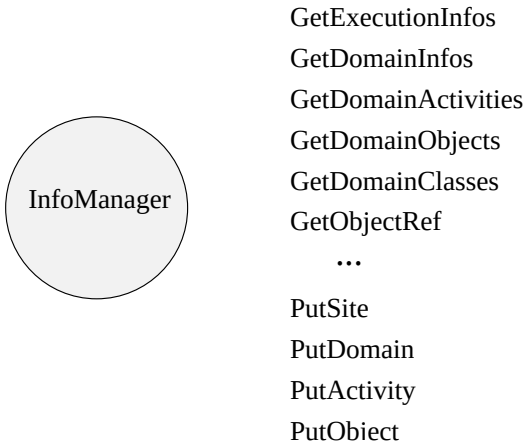


Fig. 5.5 : L'objet InfoManager

- La méthode *GetExecutionInfos* permet de connaître la liste des domaines mis en œuvre par cette exécution.
- La méthode *GetDomainInfos* permet d'obtenir plus de renseignements sur ces domaines : le domaine père, le nombre de domaines fils, le nombre d'activités créées, le nombre d'extensions effectuées, le nombre d'objets et de classes utilisés.
- Les méthodes *GetDomainActivities*, *GetDomainObjets*, *GetDomainClasses* permettent d'obtenir respectivement la liste des activités, des objets et des classes d'un domaine.
- La méthode *GetObjectRef* permet d'obtenir la référence d'un objet Guide.
- Les méthodes *PutSite*, *PutDomain*, *PutActivity*, *PutObject* sont utilisées lors de la phase d'analyse pour conserver ces renseignements.

V.1.3.6 L'objet ReplayManager

L'objet *ReplayManager* se charge de lancer une nouvelle exécution de l'application et de contrôler sa réexécution (méthode *ExecutionReplay*).

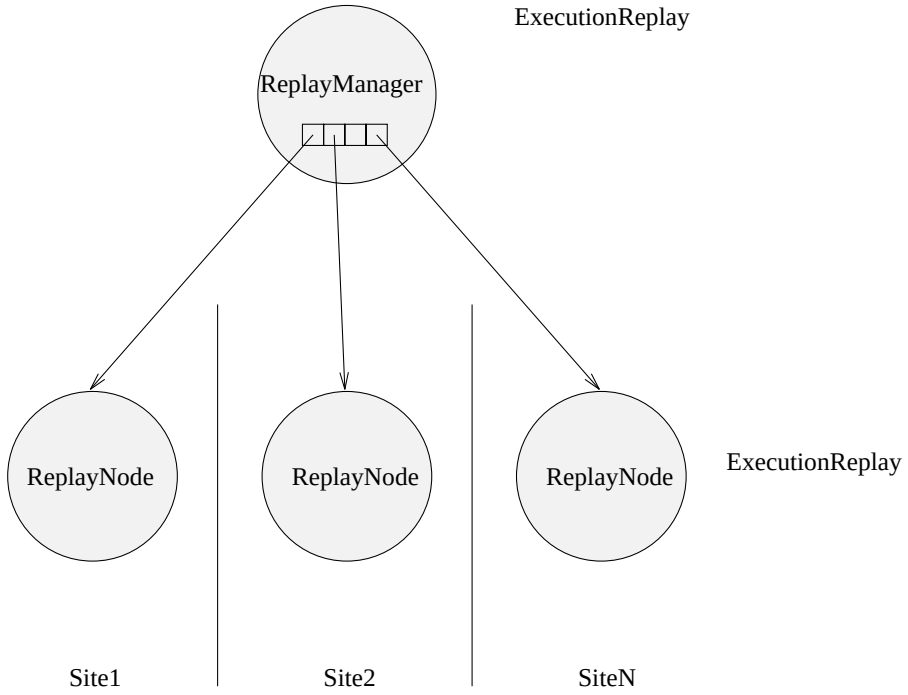


Fig. 5.6 : Décomposition en objets de ReplayManager

L'objet *ReplayManager* est composé d'un objet *ReplayNode* par site visité par l'exécution. Chacun de ces objets se charge de reproduire la portion d'exécution qui concerne son site (méthode *ExecutionReplay*). Pour ce faire, il se sert des informations contenues dans les historiques des activités pour diriger la réexécution. Le contexte initial de l'exécution doit auparavant être réinstallé.

V.2 Enregistrement de l'historique

Nous avons choisi en section IV.3 de conserver automatiquement l'historique d'une exécution Guide. Nous utilisons un mécanisme système pour enregistrer la trace d'une exécution, qui est effectif lorsque l'application s'exécute dans un mode particulier, le mode Trace. Dans ce mode, l'exécution signale des événements prédéfinis qui sont récupérés par un processus spécialisé (le processus Voyeur), qui se charge de les traiter et de les conserver. Cette conservation s'effectue sur chaque site visité, et aboutit à la création d'un historique par activité et par site. Nous décrivons ci−après la réalisation du mécanisme d'enregistrement d'une exécution Guide.

V.2.1 Création des processus Voyeur

L'historique de l'exécution sur un site est conservé par un seul processus Voyeur qui se charge d'enregistrer les événements signalés par les activités de ce site dans les historiques associés à ces activités.

La création des processus Voyeur ne peut se faire qu'en cours d'exécution, puisque l'ensemble des sites visités n'est pas connu à l'avance et peut varier d'une exécution à l'autre. Un premier processus Voyeur est créé sur le site initial d'exécution, au lancement de l'application en mode Trace. Les autres processus Voyeurs sont créés lors de l'extension d'un domaine de l'exécution sur un site non encore visité.

La figure Fig. 5.7 montre la création de processus Voyeur au moment du lancement de l'enregistrement d'une exécution, ainsi que lors d'une extension de domaine.

Le lancement de l'enregistrement d'une exécution s'effectue par appel de la méthode *ExecutionRecord* sur un objet *Thésée* :

- Cet appel crée un objet *RecordManager* et provoque l'exécution de façon asynchrone, donc dans un nouveau domaine (*D2*), de la méthode *ExecutionRecord* sur cet objet. Le client récupère donc le contrôle avant que l'enregistrement se soit terminé. Il a la possibilité de s'informer de la terminaison de l'enregistrement via l'objet *Thésée*.
- L'appel de méthode *RecordManager.ExecutionRecord* lance d'une part l'exécution de l'application en mode Trace (*Application.Méthode*) dans un nouveau domaine (*D3*) et crée d'autre part un objet *RecordNode* sur ce site et exécute la méthode *ExecutionRecord* de cet objet. Cet appel enregistre l'historique de la portion d'exécution qui se déroule sur ce site. Le processus Voyeur correspond à l'activité qui exécute l'appel de méthode *RecordNode.ExecutionRecord*.
- Lors d'une extension de domaine, le système Guide s'aperçoit que l'application s'exécute en mode Trace et crée un processus Voyeur sur ce site. Il crée pour cela un objet *RecordNode* et provoque l'exécution asynchrone de la méthode *ExecutionRecord* (*D4*) sur cet objet.
- L'enregistrement de l'exécution se termine lorsque tous les autres processus Voyeur ont terminé leur travail : la méthode *WaitEndRecordNode* réalise un rendez−vous entre tous les Voyeurs.

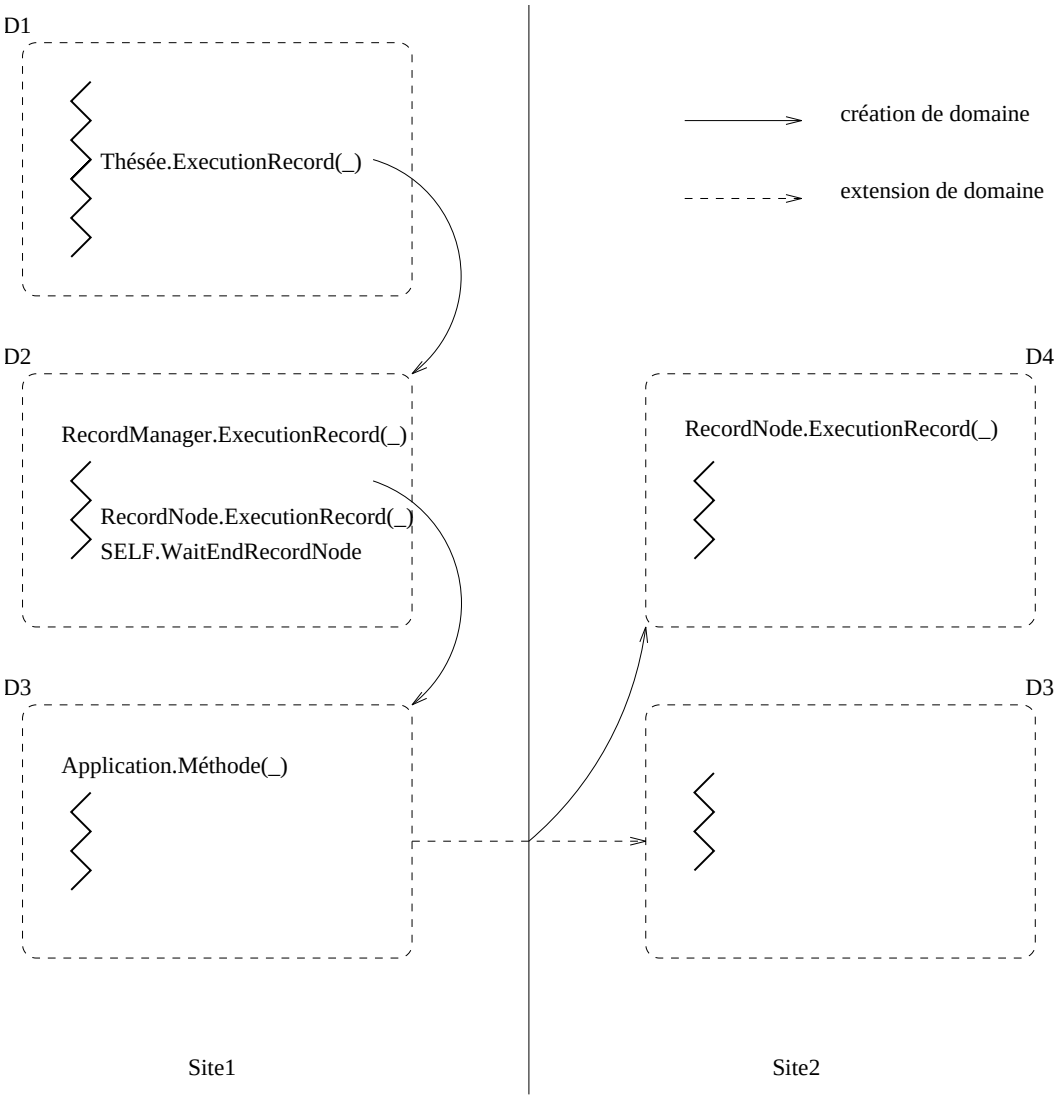


Fig. 5.7 : Création des Voyeurs lors de l'enregistrement d'une exécution

V.2.2 Étude de la communication sur un site

La communication asynchrone entre les activités de l'application qui s'exécutent sur un site et le processus Voyeur qui se charge de conserver les événements signalés par ces activités est un aspect primordial du noyau de réexécution.

Nous avons effectué une étude des différentes approches envisageables pour réaliser cette communication, dans le but de les comparer et de choisir la plus performante, celle qui provoque la plus faible perturbation de l'exécution (cf III.2.2).

V.2.2.1 Approches possibles

Nous présentons ci−après les différents supports de communication disponibles sur Unix et Guide pouvant être utilisés pour réaliser cette communication. Pour les comparer, nous considérons le cas d'une communication simplifiée entre une seule activité qui signale des

événements et le processus Voyeur qui récupère ces événements mais sans les traiter. Les solutions décrites ont été mises en œuvre et les résultats de cette étude sont présentés dans la section suivante.

Objet Guide

Le partage d'objet permet la communication entre les activités du système Guide.

Pour communiquer, l'activité et le processus Voyeur partagent un objet Guide, utilisé comme un tampon, dans lequel le processus Voyeur récupère les événements (méthode *Retirer*) qui ont été préalablement signalés par l'activité (méthode *Déposer*). Ces deux méthodes sont synchronisées pour garantir la cohérence des événements contenus dans l'objet. Les événements sont alors décrits par les paramètres des méthodes.

Sockets Unix

Les sockets Unix permettent d'établir une communication bidirectionnelle entre des processus n'ayant pas d'ancêtre commun. Il existe deux domaines d'adressage pour les sockets Unix : le domaine Unix pour la communication entre processus résidant sur la même machine et le domaine Internet (réseau) pour la communication entre processus situés éventuellement sur des machines différentes. Seuls les sockets de domaine Unix nous intéressent ici.

Pour communiquer, l'activité et le processus Voyeur créent chacun leur propre socket, puis l'activité connecte son socket avec celui du processus Voyeur. Chaque processus utilise son propre socket pour envoyer et recevoir des messages.

Queue de messages

La communication inter−processus par message s'effectue par échange de données, stockées dans le noyau Unix, sous la forme d'une file de messages. Chaque processus peut émettre des messages et en recevoir. La synchronisation de l'accès à ces informations est réalisée en interne par le système Unix.

Pour conserver les événements d'une exécution, le processus Voyeur crée une queue de messages Unix, dans laquelle il récupère les événements déposés par l'activité.

Mémoire partagée

La zone partagée (appelée segment de mémoire partagée) correspond à un espace d'adressage en mémoire virtuelle commun à chacun des processus et constitue le moyen le plus rapide d'échange de données entre processus. Mais il est nécessaire d'en synchroniser les accès pour garantir la cohérence des informations contenues.

Pour communiquer, le processus Voyeur crée un segment de mémoire partagée, puis l'attache à sa mémoire virtuelle pour lire les événements écrits par l'activité Guide, qui l'a préalablement elle aussi attaché.

Nous avons deux types de sémaphores à notre disposition, pour réaliser la synchronisation des accès à ce segment de mémoire partagée : les sémaphores Unix

et les sémaphores propres au système Guide, réalisés par un pilote Unix. Nous avons choisi d'utiliser ces derniers pour des raisons de performance.

Nous avons envisagé l'utilisation d'autres supports pour réaliser la communication entre l'exécution et le processus Voyeur (pipe, protocole de communication propre à Guide, etc), mais leur intérêt moindre ou leur inadéquation à notre problème nous ont incité à ne pas les prendre en compte dans notre étude. Il s'agit entre autres des pipes Unix et du protocole de communication utilisé en interne par le système Guide.

V.2.2.2 Comparaison de ces approches

Nous avons évalué les solutions décrites précédemment, dans le but d'identifier celle qui provoque la plus faible perturbation de l'exécution.

Pour déterminer le support de communication le plus efficace, nous avons considéré une communication simplifiée concernant une seule activité Guide et le processus Voyeur. Nous organiserons ensuite la communication globale, entre toutes les activités de l'exécution et le processus Voyeur, en fonction du support choisi et en prenant garde à ne pas dégrader ses performances. Il faudra revoir notre choix dans le cas contraire.

Pour comparer ces différentes réalisations, nous avons défini un état de référence que chacune d'elle doit vérifier : cet état est atteint lorsque le processus Voyeur dispose des informations qu'on lui a envoyé. Les phases de codage et de conservation des événements ne sont pas prises en compte, car elles ne font pas partie de cette communication.

Pour obtenir une mesure pertinente de la perturbation engendrée par cette communication, nous avons choisi de prendre l'appel de méthode comme opération de référence et de mesurer donc le temps UC de cet appel dans les différentes solutions. Il s'agit en effet de l'opération du système Guide la plus fréquente lors d'une exécution mais également la plus perturbée par cette communication : par appel de méthode, deux événements sont signalés, l'un au début de cet appel et l'autre au retour (cf V.2.4.1).

Nous avons tout d'abord étudié le cas d'une communication qui s'effectue événement par événement. Les mesures obtenues nous ont permis de déterminer les deux solutions les plus intéressantes (cf annexe A.1), à savoir celle utilisant une queue de messages et celle utilisant un segment de mémoire partagée allié à un sémaphore Guide.

Nous avons ensuite regroupé les événements de façon à diminuer le trafic de la communication et donc d'améliorer les performances des deux solutions retenues. Bien entendu, nous avons fait varier le nombre d'événements ainsi factorisés. Dans la solution utilisant une queue de messages, la taille des messages a été augmentée pour contenir plusieurs événements. Dans la solution utilisant la mémoire partagée, l'activité signale les événements un par un (les écrit dans le tampon), mais le processus Voyeur, bloqué sur un sémaphore Guide, est réveillé uniquement lorsque un certain nombre d'événements se trouvent dans le tampon. La situation du tampon plein ou de la queue de message pleine est gérée par une temporisation (attente active) : c'est à notre avis la façon la plus simple de résoudre ce problème en n'introduisant pas de coût supplémentaire en fonctionnement normal. Cette situation, à caractère exceptionnelle, doit par ailleurs être écartée par une initialisation adéquate des différents paramètres de cette communication (taille adaptée du tampon et du regroupement).

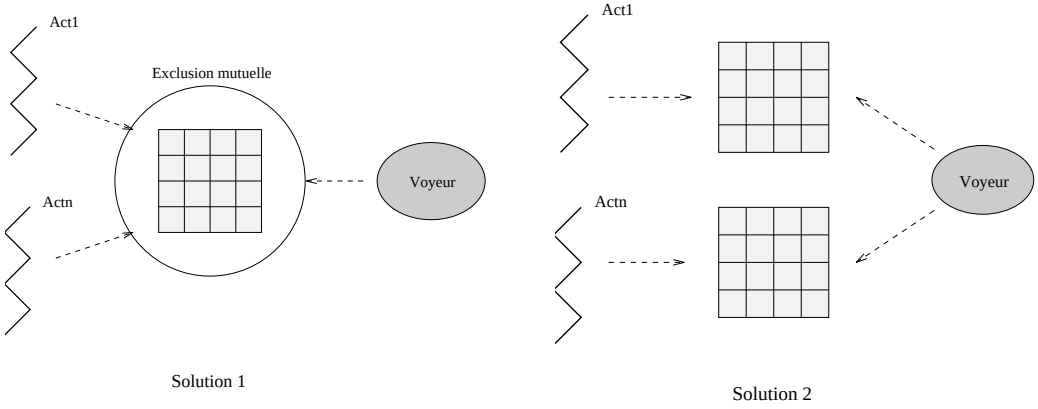
Nous pouvons observer dans le tableau ci−après, que la solution utilisant la mémoire partagée devient rapidement plus efficace dès que l'on regroupe les événements. C'est donc la solution que nous avons retenue comme support de notre communication.

temps UC en ms	Mémoire partagée Sémaphore Guide	Queue de Messages
appel sans paramètre	1.949	1.443
x10 appels	1.090	1.135
x20 appels	1.035	1.121
x30 appels	0.990	1.107

Les mesures du tableau ci−dessus correspondent au temps moyen pris par l'exécution d'un appel de méthode sans paramètre, sur un objet chargé localement (100000 appels). La taille des informations conservées pour chaque événement est de 96 octets. La machine utilisée est une station de travail Sun Sparc sous SunOs Release 4.1.1. Nous utilisons la version du système Guide V1.6. Le temps de référence (sans perturbation) d'un tel appel de méthode est de 0.789 ms.

V.2.2.3 Mise en œuvre de la communication

L'organisation de la communication peut prendre deux formes différentes suivant que l'on utilise un seul segment de mémoire partagée pour toutes les activités de ce site (solution 1) ou que l'on utilise un segment de mémoire partagée par activité (solution 2).



Solution 1

Solution 2

1. La première solution possède deux inconvénients majeurs. D'une part, elle nécessite un traitement supplémentaire de la part du processus Voyeur pour distinguer les événements appartenant à chaque activité. D'autre part, la performance de cette communication est pénalisée par l'utilisation de deux sémaphores : le premier sert à bloquer le processus Voyeur lorsque le tampon est vide et le deuxième réalise l'exclusion mutuelle des accès au tampon. En effet, pour assurer la cohérence des informations transmises, le tampon en mémoire partagée doit être accédé en

exclusion mutuelle par les activités qui déposent leurs événements et par le processus Voyeur qui le vide de temps en temps.

2. La deuxième solution facilite la conservation en parallèle des historiques des activités (conformément à nos désirs), et de plus économise l'utilisation d'un sémaphore. Nous n'avons plus besoin de protéger les accès au tampon par une exclusion mutuelle, car celui−ci est partagé uniquement par une activité et le processus Voyeur. La communication entre les deux peut être réalisée en utilisant un algorithme de *producteur/consommateur*, dans lequel le producteur est maître de la communication et contrôle l'activité du consommateur.

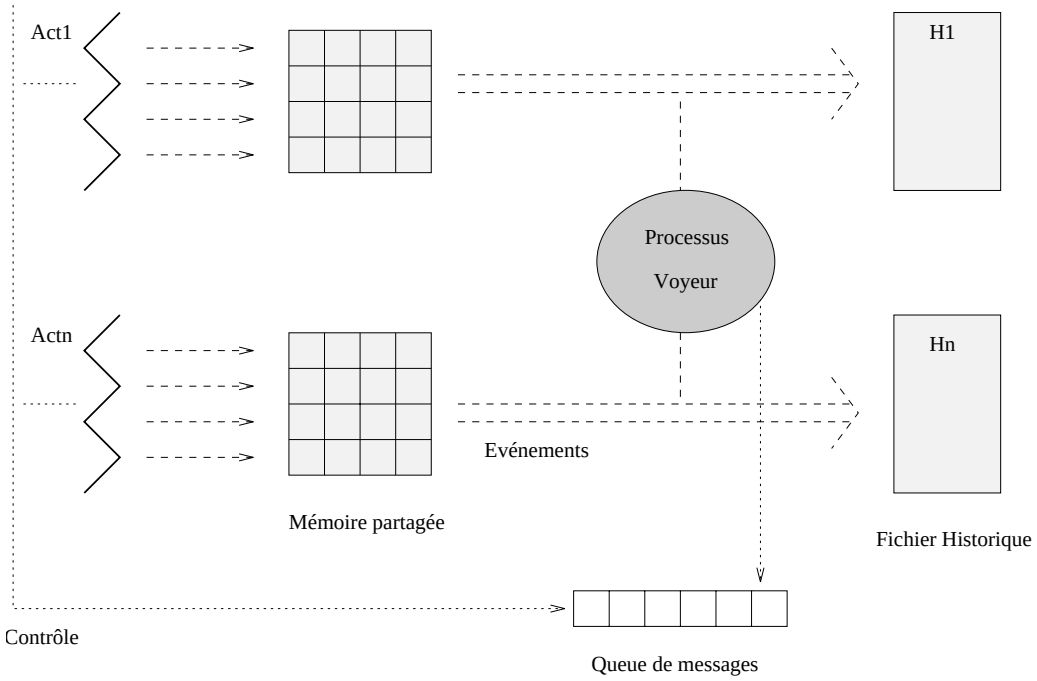


Fig. 5.8 : Architecture de la communication en mode Trace

Dans la réalisation que nous avons effectuée (cf Fig. 5.8), nous utilisons un segment de mémoire partagée par activité comme support pour la communication des événements, et une queue de messages Unix pour effectuer le contrôle de cette communication (phases de connexion, de transfert des événements et de déconnexion). Le processus Voyeur se bloque en réception sur la queue de message lorsqu'il a fini de vider un tampon (nous n'utilisons plus de sémaphore).

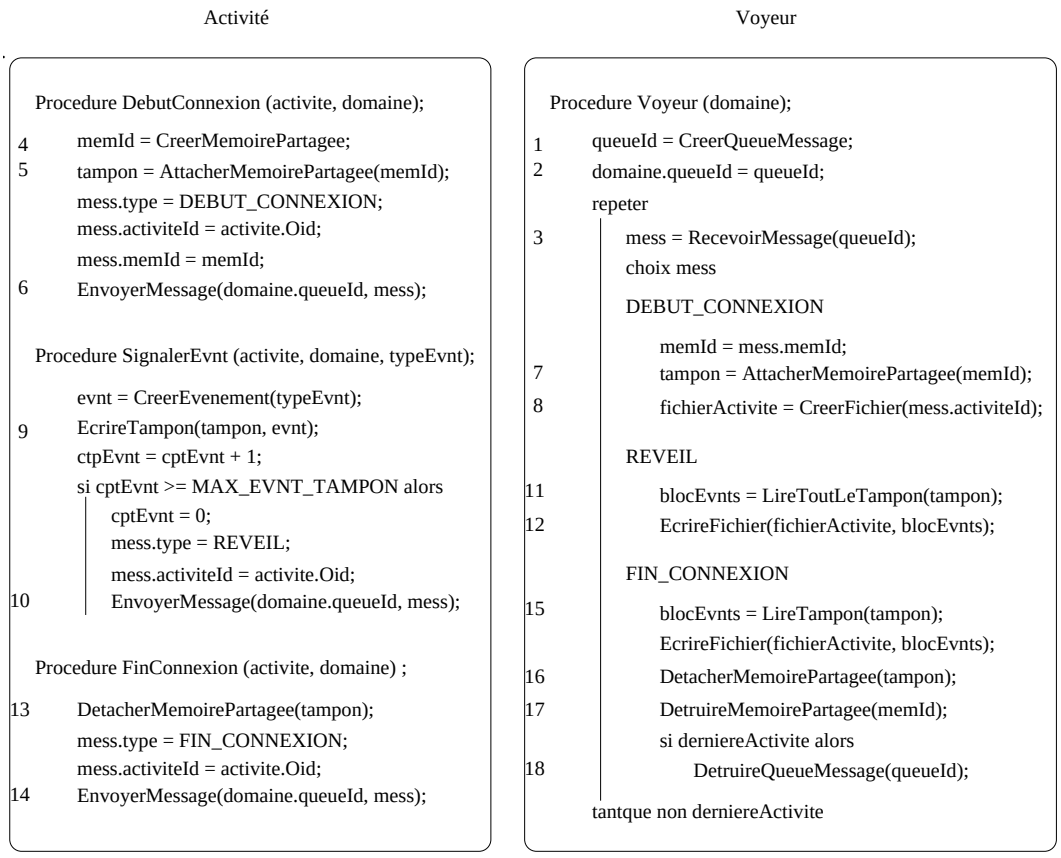


Fig. 5.9 : Protocole de la communication en mode Trace

Nous décrivons ci−après le protocole de la communication sur un site entre une activité qui s'exécute en mode Trace et le processus Voyeur (cf Fig. 5.9) :

- Création du processus Voyeur

  La queue de messages Unix est créée sur un site en même temps que le processus Voyeur (1). Pour que les activités de l'application mise au point puissent envoyer leurs messages de contrôle au processus Voyeur, l'identification de cette queue de messages est conservée dans une variable d'état du domaine de l'application (2). Le processus Voyeur se bloque alors en attente d'un message (3).

- Début de la connexion

  Un segment de mémoire partagée est créé et associé à chaque activité Guide lors de sa création (4 et 5). L'activité envoie alors un message contenant l'identification de ce segment de mémoire partagée (6) au processus Voyeur pour que celui−ci puisse l'attacher lui aussi à sa mémoire virtuelle (7). Le processus Voyeur en profite pour créer le fichier historique de l'activité (8) qui prend pour nom l'identificateur (Oid) de l'activité Guide concernée.

- Signalement d'un événement

  Le segment de mémoire partagée est géré comme un tampon circulaire dans lequel l'activité dépose les informations décrivant les événements qu'elle signale (9).

Lorsque la quantité d'informations contenues dans ce tampon atteint un certain seuil, elle envoie un message au processus Voyeur pour lui demander de le vider (10). Celui−ci, à la réception de ce message, récupère les événements contenus dans le tampon (11) et les stocke dans le fichier associé à cette activité (12).

- Fin de la connexion

  Lorsque l'activité se termine, elle détache le segment de mémoire partagée (13) et envoie un message de fin de connexion au processus Voyeur (14). Celui−ci commence alors par vider le tampon de l'activité (15) avant de le détacher (16) et de le libérer (17). La queue de messages est libérée lors de la terminaison de la dernière activité (18).

V.2.3 Ordonnancement de l'historique

Pour reproduire une exécution, nous devons connaître l'ordre d'exécution des événements caractéristiques sur l'ensemble des sites. Tous les événements caractéristiques sont concernés car ils sont potentiellement concurrents entre eux : concurrence entre les accès à un objet, concurrence entre les appels de méthode sur un même objet, concurrence entre les terminaisons d'activités parallèles.

Le caractère local du partage des objets de Guide permet heureusement de simplifier cette opération d'ordonnancement à un ordonnancement par site. En effet, les opérations concurrentes sur un même objet sont effectuées, à un instant donné, sur un seul site (les objets dans Guide ne sont pas dupliqués). Pour reproduire l'exécution, il nous suffit de traduire les interactions entre sites (extension d'activités et migrations d'objets) par des événements locaux sur les sites origine et destination.

L'historique global de l'exécution est donc partiellement ordonné, il est composé d'historiques locaux qui eux sont totalement ordonnés. Sur chaque site, nous utilisons un compteur d'événements propre à chaque exécution mise au point pour ordonner les événements des activités de cette exécution qui ont lieu sur ce site.

V.2.4 Événements conservés

Après avoir présenté le mécanisme de conservation de l'historique de l'exécution, nous nous intéressons dans la suite au contenu de cet historique : aux événements conservés et à leur codage.

V.2.4.1 Nature des événements

Nous avons classé en section IV.2.3 les événements caractéristiques d'une exécution Guide en trois familles : ceux permettant de connaître les appels de méthodes effectués ainsi que les objets utilisés, ceux permettant de connaître l'ordre d'accès des activités aux objets et ceux concernant la disparition des entités.

Nous donnons ci−après la liste exacte des événements conservés dans l'historique d'une exécution. Nous verrons plus loin (en sections V.3.3 et V.5.5) que nous avons été amenés à enrichir cette liste en définissant des événements supplémentaires pour déterminer le contexte initial de l'exécution ainsi que pour permettre la reproduction de l'exécution. Nous

précisons pour chaque événement les informations qui le définissent (la nature de l'événement et les paramètres associés) ainsi que le rôle qu'il occupe dans la reproduction de l'exécution.

Le premier groupe d'événements présenté permet de connaître la nature des appels de méthode effectués par une activité (appel synchronisé ou non, objet utilisé, méthode appelée), ainsi que l'ordre d'exécution des appels de méthode concurrents. La reproduction dans cet ordre permet d'obtenir la même synchronisation des appels de méthode sur les objets.

METHOD_CALL <objectId, methodId>

METHOD_RETURN <>

Ces événements indiquent le début et la fin d'un appel de méthode effectué par l'activité. Les paramètres associés à l'événement METHOD_CALL sont l'identification de l'objet et de la méthode concernés. L'événement METHOD_RETURN n'a pas besoin de paramètre puisqu'il indique la fin de l'appel de méthode en cours, qui est déjà complètement défini.

METHOD_SYNCHRO_CALL <objectId, methodId>

METHOD_SYNCHRO_RETURN <>

Ces événements indiquent le début et la fin d'un appel de méthode synchronisé, c'est−à−dire une méthode pour laquelle est définie une condition de synchronisation. Ces événements indiquent, en plus de l'appel, l'évaluation de cette condition et une modification des compteurs de synchronisation de la méthode. Les paramètres associés à ces événements sont les mêmes que pour un appel de méthode non synchronisé.

ACTIVITY_RESUME <>

Cet événement indique que l'activité qui était bloquée en attente sur la condition de synchronisation de la méthode en cours, reprend son exécution et évalue donc une nouvelle fois la condition de synchronisation de la méthode (celle−ci n'est pas obligatoirement passante).

Le deuxième groupe d'événements permet de connaître l'ordre d'accès des activités aux objets. Leur reproduction dans cet ordre permet de reproduire la concurrence d'accès des activités aux objets et donc d'obtenir la même répartition de l'exécution (migrations d'objet et extensions d'activité).

OBJECT_MVO_LINK <objectId>

OBJECT_UNLINK <>

Ces événements indiquent la liaison et la déliaison d'un objet en MVO, qui provoquent une modification de son compteur de liaison. Ces opérations se produisent lors des appels de méthode effectués par les activités d'un domaine. L'événement de liaison a comme paramètre l'identification de l'objet. L'événement de déliaison n'a pas de paramètre puisqu'il indique la déliaison du dernier objet lié.

OBJECT_END_UNLINK <objectId>

Cet événement se produit lors de la terminaison de l'activité (cf *ACTIVITY_KILLED*) et indique la déliaison d'un objet. Lorsqu'une activité se termine, elle libère (délie) tous les objets qu'elle possède et qui sont encore liés. Cet événement a comme

paramètre l'identification de l'objet, car il se produit en fin d'exécution et n'est pas directement rattaché à l'événement de liaison de l'objet.

OBJECT_MPO_GET <objectId>

Cet événement indique que l'activité veut effectuer la liaison d'un objet qui n'est pas présent en MVO, ce qui va provoquer son chargement. Cet événement est signalé sur le site d'exécution de l'activité.

OBJECT_GETDESC <objectId>

Cet événement indique la demande de chargement d'un objet à la MPO. Il est signalé sur le site de stockage de l'objet.

OBJECT_UNMAPPING <objectId>

Cet événement indique la demande de déchargement d'un objet de la MVO vers la MPO.

Le troisième groupe d'événements permet de connaître l'ordre de création et de disparition des entités participant à l'exécution. Leur reproduction dans cet ordre permet d'obtenir le même comportement de l'exécution lors de la création de domaines, de la création d'activités et lors de la création d'objets ainsi que lors de la terminaison de domaines, de la terminaison d'activités parallèles et lors de la destruction d'objet.

DOMAIN_CREATE <domainId>

DOMAIN_END <>

Ces événements indiquent la création et la destruction d'un domaine. L'événement de destruction n'a pas de paramètre, puisqu'il indique la fin du domaine de l'activité concernée.

ACTIVITY_KILLED <>

Cet événement indique que l'activité va mourir. Celle−ci n'est pas encore réellement terminée : elle doit notamment libérer les ressources qu'elle utilise (cf *OBJECT_ALL_UNLINK*) avant de se terminer effectivement.

ACTIVITY_CREATE <activityId>

ACTIVITY_END <>

Ces événements indiquent la création et la terminaison effective d'une activité.

ACTIVITY_KILL_ACTIVITY <activityId>

Cet événement indique que l'activité provoque la mort d'une autre activité. Il a comme paramètre l'identification de l'activité tuée (identification de l'objet activité).

OBJECT_CREATE <objectId>

OBJECT_DESTROY <objectId>

Ces événements indiquent la création et la destruction d'un objet.

V.2.4.2 Codage des événements

Nous avons vu précédemment que la réduction du trafic de la communication, obtenue par regroupement des événements, permet de diminuer la perturbation de l'exécution. Pour améliorer encore les performances de cette communication, nous nous sommes

naturellement intéressés au codage de ces événements, qui permet de diminuer le volume des informations échangées et stockées.

L'efficacité d'une technique de codage peut être évaluée suivant les deux critères suivants : la réduction de la place occupée par les informations et le temps de calcul nécessaire pour effectuer ce codage. Ces deux critères sont souvent corrélés : on comprend aisément que plus la place occupée est réduite, plus le codage est gourmand en temps de calcul.

Il s'agit de choisir, dans notre cas, le codage qui minimise la perturbation de l'exécution. Nous avons donc opté pour un codage simple qui n'effectue pas de transformation complexe des données. Celui−ci se contente de diminuer le volume des informations à conserver, en déterminant le minimum d'informations indispensable à la reproduction, et en les conservant sous une forme simplifiée.

Nous illustrons nos propos en présentant le codage employé pour conserver l'identification d'un objet et d'une méthode, le premier minimisant les informations à conserver, le second les simplifiant.

- Un objet est identifié au niveau du langage par une référence langage, qui est une structure composée entre autres de sa référence système. Cette dernière information est indispensable, si l'on veut accéder à l'objet, alors que le champ Oid de cette référence est suffisant pour l'identifier.

```
struct T_LgeRef {         /*référence langage*/
    T_SysRef mpoRef;      /*référence système*/
    short int objectLoc; /*index objet en MVO*/
    short int classLoc;  /*index classe en MVO*/
} T_LgeRef;
struct T_SysRef {   /*référence système*/
    int Oid;        /*identificateur unique*/
    int Loc;       /*localisation en MPO*/
} T_SysRef;
```

- Une méthode est identifiée au premier abord par sa signature qui se compose de son nom et du type de ses paramètres. Comme la conservation d'une chaîne de caractères nécessite un codage complexe, nous préférons identifier une méthode par son index (qui est unique) dans la table des définitions externes (TDE) de sa classe. Cette table, construite à la compilation, répertorie l'ensemble des points d'entrée ou méthodes définis pour une classe.

Le lecteur intéressé peut trouver le détail des informations conservées pour chaque événement en annexe A.2. A titre d'exemple, nous donnons ci−après les informations conservées pour les événements METHOD_CALL et METHOD_SYNCHRO_CALL.

```
struct T_Event {
    int eventCpt;    /*type evnt + compteur d'evnt*/
    int identifier; /*Oid de l'objet*/
    int index;       /*index de la méthode*/
};
```

V.3 Conservation du contexte initial

Nous avons choisi en section IV.4 de déterminer automatiquement le contexte initial d'une exécution en utilisant un mécanisme système de copie au vol des objets. Celui−ci effectue une copie des objets réutilisés (non créés par l'exécution tracée), lors de leur premier accès, et permet ainsi de retrouver facilement leur valeur initiale. Nous décrivons ci−après la réalisation de ce mécanisme.

V.3.1 Copie au vol paresseuse

La façon la plus simple de conserver l'état d'un objet est d'en prendre une copie. Nous obtenons ainsi deux objets dont les états sont identiques, mais dont les références sont différentes. Cette solution possède cependant deux inconvénients : d'une part, la copie d'un objet est une opération coûteuse, du fait qu'elle nécessite la création d'un nouvel objet et la recopie de son état, et d'autre part, elle nécessite de conserver des informations de correspondance entre les objets et leur copie, pour retrouver le contexte initial.

Nous préférons utiliser une autre solution, dite de copie paresseuse, qui tire parti du fonctionnement du système Guide pour réaliser la copie de manière indirecte. Dans cette solution, un objet peut être mis dans un mode mise au point qui lui permet de posséder deux états permanents en mémoire permanente : un état initial qui correspond à l'état de l'objet avant l'exécution, et un état courant, qui est celui réellement utilisé par l'exécution et donc sujet à évoluer (cf Fig. 5.10).

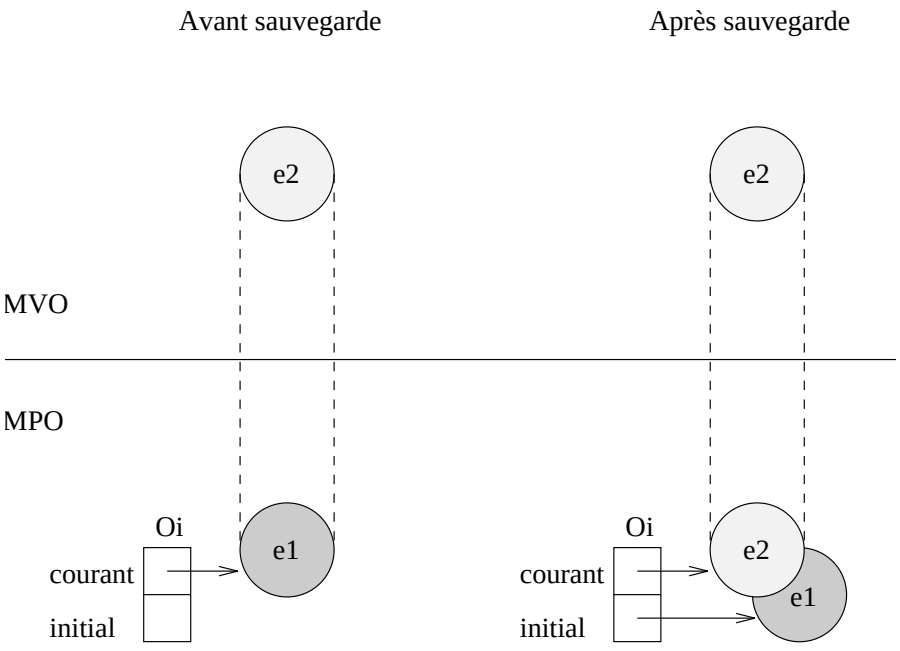


Fig. 5.10 : Création du double état d'un objet en mode mise au point

Nous détaillons ci−après le fonctionnement de notre mécanisme de copie au vol paresseuse (cf Fig. 5.11) :

- Lors du premier accès de l'application à un objet, nous le passons dans le mode mise au point (marqué **réutilisé**) pour mettre en marche le mécanisme de copie paresseuse. Si l'objet est déjà chargé en MVO, et que son état est différent de celui conservé en MPO, nous forçons auparavant sa sauvegarde pour avoir un état initial à jour.
- Nous profitons ensuite des sauvegardes d'objets, effectuées par le fonctionnement normal du système Guide, pour créer son double état. Lors de la sauvegarde d'un tel objet, le gestionnaire de la MPO n'écrase pas l'état courant de l'objet : il le conserve dans l'état initial et crée un nouvel état qui est promu état courant de l'objet.

```
Procedure MarquerObjetReutilise (objet);
    si non ObjetMarque(objet, REUTILISE) alors
        si ObjetCharge(objet) et ObjetModifie(objet) alors
            MPO_Sauvegarde(objet);
        MarquerObjet(objet, REUTILISE);


Procedure MPO_Sauvegarde (objet);
    si ObjetMarque(objet, REUTILISE) alors
        objet.etatInitial = objet.etatCourant;
    CopierMVOversMPO(objet);
```

Fig. 5.11 : Sauvegarde de l'état initial d'un objet réutilisé

Les sauvegardes d'objet sont provoquées par deux mécanismes du système Guide :

- Le déchargement d'un objet, à condition que son état ait été modifié. Ce déchargement intervient uniquement lorsqu'il y a migration de cet objet sur un autre site ou, de manière plus exceptionnelle, pour libérer de la place lorsque la MVO est saturée.
- Une activité asynchrone recopie régulièrement les objets qui ont été modifiés depuis leur chargement ou leur dernière sauvegarde.

V.3.2 Copie sélective

Le mécanisme de copie au vol ne s'applique pas à tous les objets accédés par l'exécution, mais uniquement aux objets réutilisés par l'application.

Pour limiter le nombre d'objets copiés, nous devons donc être capables de distinguer ces objets de ceux qui sont créés par l'application, et cela dès le premier accès. Pour ce faire, tout objet créé par une exécution en mode Trace se voit associer une marque distinctive, la marque **créé** (cf Fig. 5.12). Ainsi, lors de l'accès à un objet, si celui−ci ne possède aucune marque (ni créé, ni réutilisé), il s'agit alors d'un objet réutilisé qui est accédé pour la première fois et qui fait partie du contexte initial de cette exécution. Il est donc passé en mode mise au point.

```
Procedure MarquerObjetCree (objet);
    MarquerObjet(objet, CREE);

Procedure MarquerObjetReutilise (objet);
    si non ObjetMarque(objet, REUTILISE) et
       non ObjetMarque(objet, CREE) alors
        si ObjetCharge(objet) et ObjetModifie(objet) alors
            MPO_Sauvegarde(objet);
        MarquerObjet(objet, REUTILISE);
```

Fig. 5.12 : Distinction entre les objets créés et les objets réutilisés

V.3.3 Identification du contexte initial

Pour réinstaller le contexte initial d'une exécution, nous avons besoin de connaître sa composition exacte en termes d'objets, ainsi que la localisation précise de ces objets : en plus du site, elle doit mentionner si l'objet était présent en MVO ou en MPO lors de son premier accès. Nous devons en effet remettre les objets au même endroit (même site et même type de mémoire) pour obtenir la même séquence d'événements lors de la réexécution.

L'identification du contexte initial est déduite de l'historique de l'exécution, lors d'une phase d'analyse de celui−ci (cf V.3.4). Des événements spécifiques, décrits dans la suite de cette section, sont signalés lors du premier accès aux objets réutilisés, et nous permettent de conserver leur identification et leur localisation. La localisation de cet objet est conservée de manière implicite dans l'historique d'une activité : son site est celui de l'historique lui−même et son type de mémoire est fonction de la nature de l'événement signalé. Nous signalons en effet des événements différents suivant l'endroit (MVO ou MPO) où s'est effectué le premier accès à l'objet.

Nous présentons ci−après les événements que nous avons définis pour identifier le contexte initial d'une exécution. Ils sont signalés à la place des événements d'accès aux objets,

présentés en section V.2.4.1, mais uniquement lors de leur premier accès (cf Fig. 5.13). Nous conservons, pour chacun de ces événements, la référence système des objets concernés (l'Oid de l'objet n'est plus suffisant), car nous avons besoin par la suite d'accéder à ces objets pour retrouver leur état initial (cf V.3.4).

OBJECT_FIRST_MVO_LINK <objectId>

Cet événement indique la liaison d'un objet en MVO, qui est aussi le premier accès à un objet réutilisé.

OBJECT_FIRST_GETDESC <objectId>

Cet événement indique la demande de chargement d'un objet à la MPO, qui est aussi le premier accès à un objet réutilisé.

OBJECT_FIRST_UNMAPPING <objectId>

Cet événement indique la demande de déchargement de l'objet de la MVO vers la MPO, qui est aussi le premier accès à un objet réutilisé.

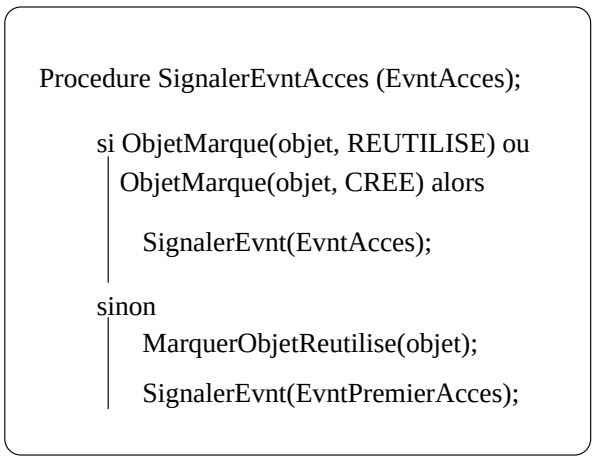


Fig. 5.13 : Signalement des événements d'accès à un objet

V.3.4 Validation des copies

A la fin de l'exécution, les objets marqués réutilisé peuvent se trouver dans l'un des trois états illustrés par la figure Fig. 5.14 :

1. L'état de l'objet n'a pas été modifié durant l'exécution (*O1*). Il n'est donc pas très difficile de retrouver son état initial.
2. L'état de l'objet chargé a évolué en MVO, mais l'objet ne possède pas de double état en MPO (*O2*). Les modifications subies par cet objet n'ont donc pas été répercutées en MPO par une sauvegarde. Son état initial correspond alors à celui conservé en MPO.
3. L'état de l'objet chargé a évolué en MVO et l'objet possède un double état en MPO (*O3*). Ce double état a été créé lors d'une sauvegarde de l'objet, qui a permis de conserver son état initial. Il est à noter que l'état de l'objet en MVO (*e3*) peut être

différent de celui conservé en MPO (*e2*), suivant si il a été modifié ou non depuis la dernière sauvegarde.

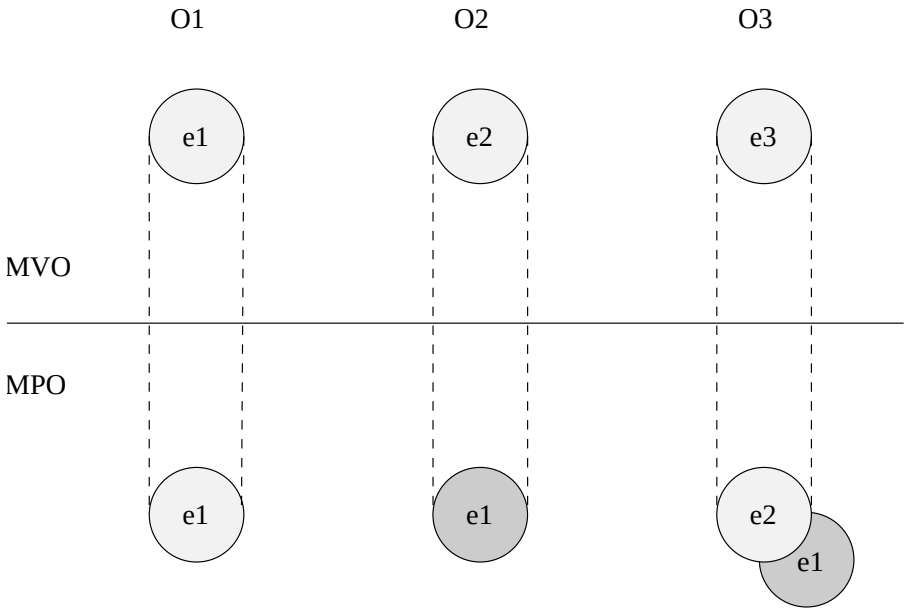


Fig. 5.14 : Les différents cas à la fin de l'exécution

Nous avons rajouté deux primitives à la gestion de la mémoire permanente de Guide (cf Fig. 5.15), qui permettent de choisir l'état que l'on veut conserver en fin d'exécution. La primitive *MPO_Abandon* permet de retrouver l'état initial d'un objet réutilisé, alors que la primitive *MPO_Validation* permet de valider l'état obtenu à la fin de l'exécution. Ces deux primitives enlèvent les marques mises sur les objets.

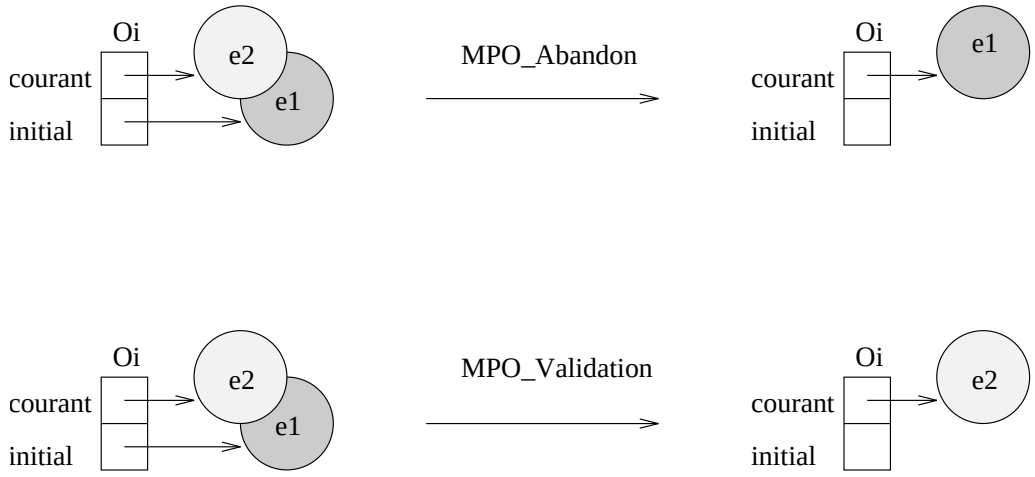


Fig. 5.15 : Choix de l'état à conserver

V.4 Analyse de l'historique

Les informations conservées dans l'historique renferment un grand nombre de renseignements sur l'exécution qui peuvent être extrêmement utiles pour les clients de Thésée, en dehors de l'utilisation qui en est faite en interne. La phase d'analyse a pour objectif d'extraire cette information de l'historique et de la rendre disponible.

V.4.1 Identifier le contexte initial de l'exécution

Nous avons vu en section V.3.3 que le contexte initial de l'exécution est identifié dans l'historique par des événements spécifiques, signalés lors du premier accès aux objets réutilisés. Ces événements permettent de connaître pour chaque objet : son identification, son site initial de localisation et sa localisation en mémoire (MPO ou MVO).

Un simple parcours de l'historique, effectué par la méthode *ContextIdentification* (cf V.1.3.2), permet de récupérer ces informations et de les conserver dans l'objet *ContextManager*. Cet objet (cf V.1.3.4) offre les fonctions nécessaires à la réinstallation du contexte initial de l'exécution (méthode *ContextAbort*), et à la validation de l'état obtenu en fin d'exécution (méthode *ContextCommit*).

La récupération du contexte initial s'effectue de manière automatique, dès la fin de l'enregistrement d'une exécution, et avant toute autre opération. Nous devons en effet réinstaller le contexte initial de l'exécution, dans le cas où l'utilisateur n'est pas satisfait de l'enregistrement obtenu, et qu'il veut enregistrer une nouvelle exécution.

V.4.2 Obtenir des renseignements sur l'exécution

Les événements conservés dans l'historique d'une exécution sont une riche source d'informations sur cette exécution.

Une analyse plus complète de l'historique, effectuée par la méthode *HistoricAnalysis* (cf V.1.3.3), permet de récupérer ces informations et de les conserver dans l'objet *InfoManager* (cf V.1.3.5). Cette analyse nous permet en particulier de déterminer les entités intervenant dans l'exécution (domaines, activités et objets), la répartition de cette exécution (extensions de domaines et d'activités, localisation et migration des objets), etc.

Cette phase d'analyse de l'historique est effectuée uniquement lorsque l'utilisateur est satisfait de l'exécution enregistrée et qu'il veut travailler avec. Les informations ainsi récupérées lui permettent d'observer son exécution et d'étudier son comportement dès la fin de l'enregistrement.

V.5 Reproduction de l'exécution

Nous avons choisi en section IV.6 de reproduire l'exécution enregistrée en lançant une nouvelle exécution de l'application que nous dirigeons grâce aux informations contenues dans son historique. Cette reproduction est effectuée grâce à un mécanisme système qui permet d'exécuter une application en mode Pas à pas. Dans ce mode, l'exécution signale des événements prédéfinis à un processus spécialisé (le processus Contrôleur), qui se charge de

la diriger pour que les actions associées aux événements s'exécutent dans le même ordre que celui conservé dans l'historique. Le processus spécialisé est appelé Contrôleur car, non seulement il regarde de près ce que fait chaque activité, mais en plus il contrôle leur exécution. Ce contrôle s'effectue sur chaque site visité, en utilisant les historiques des activités créés lors de la phase d'enregistrement. Nous décrivons ci−après la réalisation de la conduite de la réexécution.

V.5.1 Création des processus Contrôleur

Nous utilisons un processus Contrôleur par site pour reproduire la portion d'exécution qui s'est déroulée sur ce site : il dirige l'exécution des activités de ce site en utilisant les historiques créés lors de la phase d'enregistrement.

La création des processus Contrôleur, à l'inverse des processus Voyeur, peut se faire avant la reproduction de l'exécution puisque les sites où l'exécution va s'étendre sont connus. Un processus Contrôleur est donc créé sur chaque site participant à l'exécution lors du lancement de la reproduction.

Le lancement de la reproduction de l'exécution s'effectue par appel de la méthode *ExecutionReplay* sur un objet *Thésée* (cf V.1.3) :

- Cet appel crée un objet *ReplayManager* (uniquement lors de la première réexécution), et provoque l'exécution de façon asynchrone de la méthode *ExecutionReplay* sur cet objet. Cette méthode s'exécute dans un nouveau domaine pour la même raison que lors du lancement de l'enregistrement de l'exécution : le client ne se trouve donc pas bloqué du fait de la reproduction.
- L'appel de méthode *ReplayManager.ExecutionReplay* lance d'une part l'exécution de l'application dans le mode Pas à pas, dans un domaine qui lui est propre, et crée d'autre part un objet *ReplayNode* sur chaque site participant à l'exécution (uniquement lors de la première réexécution), et exécute la méthode *ExecutionReplay* sur l'objet *ReplayNode* de ce site.
- Lors d'une extension de domaine, le système Guide s'aperçoit que l'application s'exécute en mode Pas à pas et lance l'exécution d'un processus Contrôleur sur ce site : il provoque l'exécution asynchrone de la méthode *ExecutionReplay* sur l'objet *ReplayNode* de ce site.

V.5.2 Communication entre l'exécution sur un site et son contrôleur

Nous utilisons, pour effectuer cette communication, la même organisation de la communication que celle employée pour la création de l'historique (cf V.2.2.3). Nous utilisons donc un segment de mémoire partagée par activité, pour la communication des événements, et une queue de messages Unix, pour le contrôle de cette communication (phases de connexion, de transfert et de déconnexion).

Le protocole de communication de la reproduction de l'exécution est cependant différent (cf Fig. 5.16) : le signalement des événements est bloquant, et des messages de contrôle (en phase de transfert) sont envoyés à chaque signalement d'événement. Dans le mode Pas à pas,

une activité signale un événement en l'écrivant dans son segment de mémoire partagée et en envoyant un message de contrôle, de type Evnt, au processus Contrôleur ; elle se bloque ensuite sur son sémaphore privé, avant d'exécuter l'action correspondante à l'événement. Le processus Contrôleur, lorsqu'il reçoit ce message, lit l'événement dans le segment de mémoire partagée de l'activité, mais ne libère cette activité que lorsque son tour d'exécution est venu (cf V.5.3).

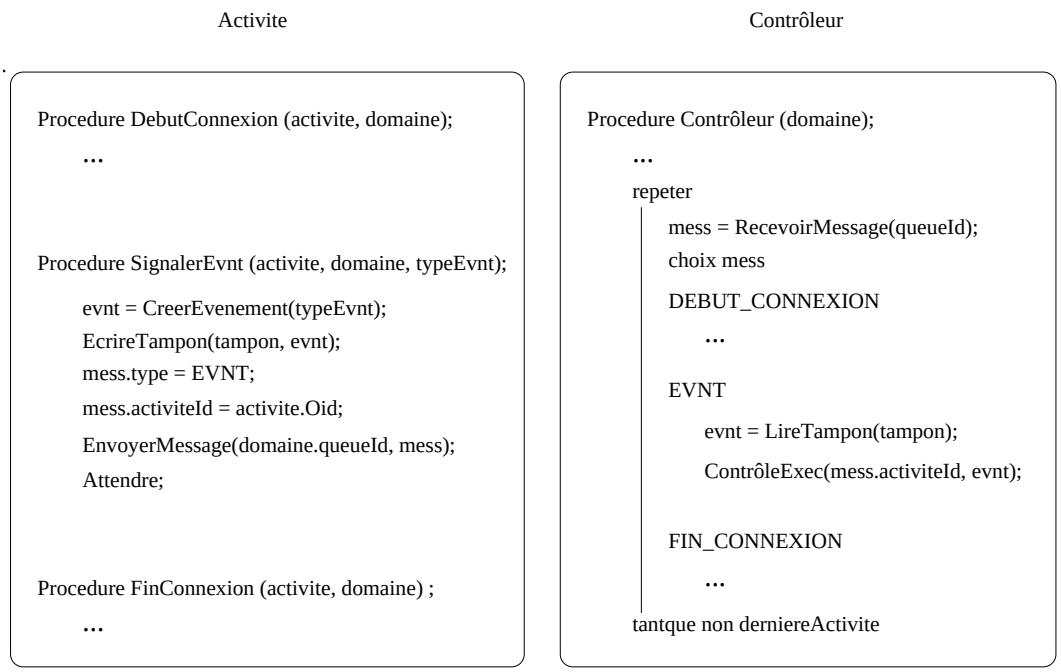


Fig. 5.16 : Protocole de la communication en mode Pas à pas

V.5.3 Contrôle de la réexécution

La reproduction de l'exécution est réalisée en reproduisant sur chaque site visité la portion d'exécution qui le concerne. Les interactions entre les sites (extensions de domaine et d'activité, et migrations d'objet) apparaissent sur chacun des sites comme des événements locaux et permettent de synchroniser les reproductions locales.

Sur chaque site, un processus Contrôleur se charge de diriger l'exécution des activités de la nouvelle exécution (cf Fig. 5.17). A partir des informations conservées dans les historiques locaux, le processus Contrôleur détermine le prochain événement à exécuter et donc la prochaine activité à libérer (*1, 5*). Si l'activité concernée est prête à exécuter cet événement (*2*), c'est−à−dire si elle a déjà signalé l'événement en question et attend donc son tour d'exécution, il la libère (*4*). Autrement, il attend son signalement pour effectuer cette opération. Avant de libérer une activité, le contrôleur vérifie que l'événement signalé est correct (*3*) (type de l'événement et paramètres équivalents à ceux de l'historique) ; dans le cas contraire, il interrompt la reproduction en signalant une erreur (*6*).

```
Procedure Controleur (domaine);
        ...
1       RechercherPremiereActivite(activiteCourante, evntHist);
        repeter
            mess = RecevoirMessage(queueId);
            choix mess
            DEBUT_CONNEXION
                ...
            EVNT
                evntExec = LireTampon(tampon);
                ActivitePrete(activiteId, evntExec);
2               tantque EstPreteActivite (activiteCourante) faire
3                   si EstEquivalent(evntExec, evntHist) alors
4                       LibererActivite(activiteCourante);
5                       RechercherProchaineActivite(activiteCourante, evntHist);
                    sinon
6                       ArreterReproduction(erreurEvntNonEquivalents);
            FIN_CONNEXION
                ...
        tantque non dernierActivite
```

Fig. 5.17 : Algorithme de la reproduction de l'exécution sur un site

V.5.4 Reproduction de la synchronisation de l'exécution

La libération des activités dans l'ordre conservé dans l'historique ne garantit pas leur exécution dans ce même ordre. En effet, comme nous ne contrôlons pas l'ordonnancement du système, il peut arriver que deux activités concurrentes libérées dans un certain ordre, s'exécutent et évaluent la synchronisation dans l'ordre inverse.

Pour nous assurer de l'ordre d'exécution de ces événements, nous signalons un événement supplémentaire (*SYNCHRO_OK*) après chaque modification de la synchronisation. Nous ne libérons la prochaine activité qu'à la réception de cet événement, qui indique le début de l'exécution de l'activité courante.

Nous illustrons ce fonctionnement particulier en prenant la reproduction du début d'un appel de méthode synchronisée comme exemple. La primitive du système Guide *Appel-MethodeSynchronisee* (cf Fig. 5.18) se place en situation d'exclusion mutuelle (*1*) pour modifier les compteurs associés à une méthode. L'événement *METHOD_SYNCHRO_-CALL* est signalé lors de l'enregistrement et lors de la reproduction (*2*) avant la modification effective de ces compteurs (*7*). Lors de la reproduction, l'activité se bloque (*4*) après le

signalement de cet événement, en attendant son tour d'exécution. Auparavant, elle libère l'exclusion mutuelle (*3*), qu'elle demandera de nouveau à son réveil (*5*) par le processus Contrôleur. Pour nous assurer de son exécution, nous signalons un événement supplémentaire *SYNCHRO_OK* immédiatement après la rentrée en section critique (*6*).

```
AppelMethodeSynchronisee (domaine, activite, objet, methode);
    ...
1   P(mutex);
    si modeExec = Trace ou modeExec = PasApas alors
2       SignalerEvnt (activite, domaine, OBJECT_METHOD_CALL);
        si modeExec = PasApas alors
3           V(mutex);
4           P(semaphorePrive);
5           P(mutex)
6           SignalerEvnt (activite, domaine, SYNCHRO_OK);
7   ModifierCompteurSynchro(objet, methode);
    V(mutex);
    AppelEffectfMethode
    ...
```

Fig. 5.18 : Reproduction de la synchronisation

L'événement *SYNCHRO_OK* (sans paramètre) est signalé lors de la réexécution, après chaque événement défini en V.2.4.1 traduisant une modification de synchronisation.

V.5.5 Compléments d'information sur l'exécution

Nous avons défini des événements supplémentaires, signalés uniquement lors de la réexécution, pour permettre à Thésée de connaître de façon plus précise le comportement de l'exécution. Thésée se sert de ces informations pour offrir à ses clients un suivi du déroulement de l'exécution plus complet (cf V.6.1). Ces événements ne sont pas nécessaires pour reproduire une exécution, et ne sont donc pas signalés lors de l'enregistrement de l'exécution.

Les événements présentés ci−après sont signalés uniquement lors de la réexécution :

ACTIVITY_EXTEND <siteId>

Cet événement indique l'extension de l'activité sur un site. Il a comme paramètre l'identification du site d'extension.

ACTIVITY_SUSPEND <>

Cet événement indique que l'activité se bloque sur la condition de synchronisation de la méthode en cours d'exécution. Il n'a pas de paramètre.

CLASS_LINK <classId>

Cet événement indique le chargement d'une classe. Il a comme paramètre l'identification de la classe.

Les événements présentés ci−après sont signalés à la place de l'événement *SYNCHRO-_OK*, qui est utilisé pour reproduire la synchronisation de l'exécution, dans les cas où l'exécution de l'action libérée apporte un supplément d'information sur le déroulement de l'exécution :

METHOD_EFFECTIVE_CALL <objectId, methodId>

METHOD_EFFECTIVE_RETURN <>

Ces événements sont signalés après les événements *METHOD_SYNCHRO_CALL* et *METHOD_SYNCHRO_RETURN* pour indiquer le début et la fin effectifs de l'exécution de la méthode appelée. Ils n'ont pas de paramètre.

OBJECT_UNMAPPED <>

OBJECT_NOT_UNMAPPED <>

Ces événements sont signalés après un événement *OBJECT_UNMAPPING* (demande de déchargement) pour indiquer le déchargement effectif ou non de l'objet. Ils n'ont pas de paramètre.

V.6 Fonctions de mise au point

Le noyau de réexécution offre un certain nombre de fonctions de base pour le contrôle de la progression de l'exécution : il s'agit principalement des fonctions de suivi et d'arrêt de l'exécution.

V.6.1 Suivi du déroulement de l'exécution

Thésée permet de suivre la progression de la réexécution par l'intermédiaire d'événements qu'il rend disponibles, à travers la méthode *GetGuideEvent* (cf V.1.3.1). Ces événements correspondent à ceux utilisés en interne par le noyau de réexécution, porteurs d'information pertinente sur l'exécution.

Thésée propose différents niveaux d'observation de l'exécution, qui traduisent une vision plus ou moins complète du modèle d'exécution de Guide. Au niveau le plus complet, tous les événements utilisés en interne par Thésée sont disponibles et permettent d'observer le comportement du système Guide dans ses moindres détails.

Ces événements sont déposés, par les processus Contrôleur, dans la file d'événements gérée par l'objet Thésée. Un processus Contrôleur rend disponible un événement après avoir vérifié qu'il est correct et avant de libérer l'activité concernée (cf V.6.2).

V.6.2 Arrêt de l'exécution

Le noyau de réexécution Thésée permet d'arrêter la progression de la réexécution en définissant des points d'arrêts sur les événements observables.

Chaque processus Contrôleur, après avoir déposé l'événement qui est prêt à être exécuté dans la file des événements de l'objet Thésée, regarde si celui−ci fait l'objet d'une demande d'arrêt de l'exécution. Si ce n'est pas le cas, il libère l'activité courante et continue la reproduction de l'exécution.

Nous envisageons de définir trois modèles d'arrêt global de l'exécution :

1. Arrêt immédiat.

   Dans ce cas, l'arrêt de l'exécution d'une activité provoque l'arrêt de toute l'exécution.

2. Arrêt dépendant

   Dans ce cas, l'arrêt de toute l'exécution se fera en fonction des dépendances entre les différentes activités et l'activité consernée par le point d'arrêt.

3. Arrêt causal

   Dans ce cas, l'arrêt de l'exécution vérifie la relation de causalité entre les événements. Cet état de l'exécution est déterminé à partir des informations conservées dans l'historique.

Chapitre VI

Évaluation

Nous présentons dans ce chapitre une évaluation de notre travail. Nous présentons tout d'abord les avantages et les inconvénients de notre méthode de reproduction d'une exécution. Nous évaluons ensuite Thésée, le noyau de réexécution d'applications Guide, que nous avons réalisé. Nous mentionnons enfin les évolutions souhaitables de Thésée, nécessaires pour devenir un véritable outil de mise au point, ainsi que les utilisations potentielles de Thésée.

VI.1 Évaluation de notre méthode de reproduction d'une exécution

VI.1.1 Avantages

Le principal intérêt de la méthode de reproduction d'une exécution est d'apporter une solution au non−déterminisme des applications parallèles et réparties. Cette méthode garantit un comportement identique lors des réexécutions, au niveau des événements considérés, et rend possible une mise au point cyclique de ces applications (cf II.3).

La mise au point s'effectue alors dans des conditions particulières, où le comportement de l'exécution à corriger est déjà connu (à travers son historique), et peut être présenté à l'utilisateur sous la forme de vues de l'exécution, qui facilitent la localisation et la correction des erreurs (cf III.3.3).

Notre méthode de reproduction de l'exécution, qui reproduit le comportement d'une exécution au niveau des objets, nous permet d'offrir à l'utilisateur une observation de l'exécution à un haut niveau d'abstraction, puisque basée sur les objets. Celui−ci peut ainsi mettre au point ses applications avec le même niveau d'abstraction que celui utilisé lors de leur conception. L'objet permet une observation macroscopique de l'exécution qui joue un rôle central dans la mise au point. Il apporte une réponse au problème de la granularité de l'information observée en permettant d'observer une exécution dans ses grandes lignes, sans se perdre dans ses détails (cf III.1.1).

Notre méthode de reproduction provoque une perturbation minimale de l'exécution. Le type de reproduction utilisé (dirigé par le contrôle) est celui qui introduit la plus faible perturbation de l'exécution (cf II.3.2.2). De plus, les informations conservées sont minimisées : l'historique ne conserve que les événements liés aux objets (cf III.2.3) et le contexte initial ne conserve que les objets réutilisés (cf IV.4).

La reproduction de l'exécution s'effectue de manière automatique, sans que le programmeur ait à intervenir. L'enregistrement d'une exécution (historique et contexte initial) et les réexécutions sont réalisées de manière automatique par Thésée.

Les informations contenues dans l'historique peuvent être utilisées pour offrir des vues de l'exécution appropriées aux particularités de l'environnement d'exécution telles que le parallélisme, la répartition, la persistance des objets, etc (cf IV.7.1). Ces vues de l'exécution peuvent servir de support au contrôle de l'exécution.

L'organisation du metteur au point que nous proposons est réalisée par la coopération de nombreux outils spécifiques, et possède l'avantage d'être modulaire (cf III.4). L'outil de mise au point n'est pas conçu comme un outil monolithique, figé dans une configuration donnée, mais peut évoluer et peut être enrichi par le remplacement d'une de ses parties ou par l'adjonction de nouvelles entités.

Les outils spécifiques au contexte de la mise au point que nous avons introduits (noyau de réexécution, observation de l'exécution, etc) peuvent être réutilisés dans un autre contexte puisqu'ils ont été conçus comme des outils indépendants (cf III.4.2). Le noyau de réexécution, par exemple, peut être utilisé par tout outil s'intéressant à l'exécution d'une application (cf VI.3.2).

VI.1.2 Inconvénients

Le principal inconvénient de la méthode de reproduction d'une exécution est lié au fait que l'utilisateur doit au préalable enregistrer une exécution erronée, pour pouvoir la corriger. Ce fonctionnement peut être considéré comme fastidieux par certains. En contrepartie, une fois ce stade atteint, l'utilisateur a tout son temps pour localiser et corriger l'erreur mise en évidence par cette exécution.

L'utilisateur doit aussi apprendre à manipuler l'arborescence d'historiques (cf III.2.7), gérée par l'outil de mise au point, qui est la conséquence des modifications qu'il a apportées à une exécution. Chaque modification du code ou des données utilisés produit un nouvel historique, qui se greffe sur l'historique précédent à l'endroit précis de l'exécution où s'est effectuée la modification. L'utilisateur peut rencontrer quelques difficultés à retrouver son chemin parmi la succession de ses modifications, bien qu'une représentation adéquate puisse lui être d'un grand secours.

Notre méthode de reproduction de l'exécution, qui reproduit le comportement d'une exécution au niveau des objets, ne garantit pas le même comportement de l'exécution à l'intérieur des objets. La réexécution n'est pas fidèle et peut donc diverger de celle enregistrée, provoquant alors l'arrêt de la réexécution (cf III.2.6). Cette divergence, causée par un ordre de réexécution différent des instructions à l'intérieur d'un objet, met en évidence une erreur différente de celle recherchée, liée à la synchronisation des opérations en cours d'exécution sur l'objet incriminé.

Pour corriger une erreur, l'utilisateur ne dispose pas actuellement d'informations sur le comportement de l'exécution à l'intérieur d'un objet, et doit donc corriger cette erreur sans cette aide. Il dispose quand même d'une bonne connaissance des conditions d'occurrence de

l'erreur puisqu'il connaît l'objet impliqué et les conditions d'exécution qui provoquent l'erreur (les appels de méthode en cours).

VI.2 Évaluation du noyau de réexécution Thésée

VI.2.1 Utilisation de Thésée

Thésée est actuellement utilisé par un outil de visualisation de l'exécution en cours de développement au sein de notre laboratoire [81]. Cet outil a pour but d'offrir différentes vues de l'exécution, en se servant des informations que Thésée rend disponible, via l'historique de l'exécution. Cette première expérience montre l'intérêt de disposer de telles informations et la pertinence des renseignements qu'elles permettent d'obtenir.

VI.2.2 Réalisation de Thésée

Thésée est écrit en grande partie en langage Guide. Seules les primitives liées à la communication des événements (signalement et récolte) entre le système Guide et Thésée sont écrites en langage C.

- Thésée est constitué de 13 types et de 11 classes Guide qui représentent 8000 lignes de code Guide, dont 400 lignes de code C.
- Le signalement des événements correspond à une primitive de 400 lignes de code C, dont l'appel a été ajouté à différents endroits du système.

L'utilisation du langage Guide a facilité la gestion du parallélisme, du partage et de la synchronisation nécessaire à Thésée. Nous utilisons aussi la persistance et les exceptions de Guide

VI.2.3 Mesures et performances

Nous évaluons ci−après notre mécanisme d'enregistrement et de reproduction d'une exécution Guide. Nous présentons des mesures de perturbations de l'exécution et de taille de l'historique conservé dans le cas de programmes de références et d'applications complètes.

VI.2.3.1 Programmes de références

Le tableau ci−après présente la perturbation introduite par notre mécanisme sur les deux principaux appels de base du système Guide, qui sont la création d'objet et l'appel de méthode.

Appels de base	Guide		Thésée enregistrement %	Taille Historique Koctets	Evénements Nb
	Temps en ms	Ecart−type sur 20 essais			
Création d'objet (10000 créations)	3.946	0.023	5.5	360.9	50079
Appel de méthode (100000 appels)	0.653	0.002	13.9	2400.9	400150

La machine utilisée est une station de travail Sun Sparc sous SunOs Release 4.1.1. Nous utilisons la version du système Guide V1.6. Le tampon en mémoire partagée a une taille de 90 Koctets et est vidé tous les 30 Koctets par un processus Voyeur.

Les mesures obtenues traduisent de manière fidèle la perturbation introduite par notre mécanisme de réexécution. Elles prennent en compte le temps de sauvegarde du tampon par le processus Voyeur : le programme calculant le temps d'une création d'objet remplit 4 fois le tampon et demande 12 réveils du processus Voyeur. Le programme calculant le temps d'un appel de méthode remplit 26.6 fois le tampon et demande 80 réveils du processus Voyeur.

La perturbation mesurée sur un appel de méthode est relativement proche de la limite des 10% considérée comme acceptable. Elle traduit la perturbation maximum introduite par notre mécanisme. L'appel de méthode est en effet l'opération la plus perturbée du modèle d'exécution : pour chaque appel de méthode, nous conservons deux événements (appel et retour), ainsi que les événements de liaison et de déliaison de l'objet concerné. Ces 4 événements occupent 24 octets dont la sauvegarde pénalise le temps (relativement faible) d'un appel de méthode. De plus, nous avons considéré le cas le plus défavorable en utilisant une méthode vide (ne contenant pas d'instructions) et non synchronisée (ne possédant pas de clause de contrôle associée).

La perturbation mesurée sur une création d'objet est beaucoup plus faible. La mesure présentée comprend, en plus de la création proprement dite d'un objet, l'appel de la méthode d'initialisation de cet objet (InitOnNew) ajoutée par le compilateur. Une création provoque donc la sauvegarde de 5 événements occupant 36 octets.

VI.2.3.2 Applications

Nous présentons ci−après les résultats obtenus avec des applications qui sont des programmes modestes de démonstration du système Guide.

L'application des tours de Hanoï est une application séquentielle qui réalise le transfert de rondelles (10) d'un piquet à un autre (en utilisant un troisième piquet).

L'application des Philosophes qui mangent est une application parallèle qui utilise des objets partagés et des conditions de synchronisation pour l'accès à ces objets.

Applications	Guide		Thésée enregistrement %	Thésée reproduction
	Temps en s	Ecart−type sur 20 essais		
Hanoï	21.341	0.145	13.7	X4
Philosophes	145.531	1.42	1.2	X5

Applications	Taille Historique Koctets	Evénements Nb	Appels de méthode Nb	Objets créés Nb
Hanoï	201.5	33562	8386	22
Philosophes	70.5	11822	2816	12

La perturbation ajoutée par l'enregistrement de l'historique est inférieure dans tous les cas à 13.9%. Elle est de beaucoup inférieure à cette limite dans le cas des applications parallèles utilisant des objets partagés avec de la synchronisation (1.2% pour l'application Philosophes), qui sont les applications cybles du noyau de réexécution Thésée. Elle est plus proche de cette limite (13.7% pour l'application Hanoï) dans le cas d'applications séquentielles effectuant peu de calculs.

La reproduction d'une exécution multiplie par un facteur de 4 ou 5 le temps d'exécution. Ceci a comme unique conséquence de pouvoir faire naître un sentiment d'impatience chez l'utilisateur. Nous proposons des solutions pour remédier à cet inconvénient dans la section suivante.

Nous faisons remarquer que ces résultats sont obtenus avec notre premier prototype, et qu'ils peuvent donc être améliorés dans les versions futures.

VI.3 Perspectives

Une version industrielle de Thésée va être réalisée dans le cadre du projet Oode de Bull. Le projet Oode a pour objectif de réaliser une version industrielle du projet Guide, à partir de la version Eliott du système. L'approche que nous proposons a été retenue pour effectuer la mise au point des applications Oode. L'outil de mise au point qui va être développé sera l'application de démonstration phare de ce projet.

VI.3.1 Évolutions de Thésée

La version actuelle de Thésée ne correspond pas aux exigences d'un véritable outil de mise au point. Il manque en effet des fonctions indispensables à la mise au point qui n'ont pas été prises en compte par notre travail : elles ne faisaient pas partie de nos objectifs et n'ont pas été abordées par manque de temps. Elle peuvent faire l'objet de travaux futurs à l'occasion de la nouvelle réalisation de Thésée sur la version Eliott du système :

- Conserver les entrées effectuées par l'exécution

  Il est nécessaire de conserver de manière automatique les entrées effectuées par une exécution, pour pouvoir les retrouver lors de la reproduction, sans les demander à chaque fois à l'utilisateur (cf IV.2.2).

  La conservation de ces entrées peut être effectuée par utilisation de classes d'entrées particulières, dont le code des opérations a été modifié en conséquence.

- Isolation de l'exécution

  Le partage d'objet nécessite d'isoler l'exécution de l'application mise au point pour éviter qu'elle soit troublée par une autre exécution (cf IV.5).

  L'isolement de l'exécution n'a pas été automatisée dans Thésée et a été laissé à la charge de l'utilisateur. Un isolement automatique est difficile à obtenir avec la version actuelle du système Guide, alors qu'il peut être réalisé de manière assez simple par l'utilisateur. Ce choix peut être remis en cause lors d'une prochaine réalisation de Thésée sur le système Eliott.

- Offrir une mise au point symbolique

  Il est nécessaire de donner à l'utilisateur la possibilité de désigner de façon symbolique les entités (classes, objets, variables, etc) qui participent à l'exécution de son application (cf II.2.2).

  Ce mode de désignation demande la modification du compilateur qui doit produire les tables de symboles associées aux classes. Il faut aussi disposer d'un mécanisme qui puisse les retrouver lors de l'exécution (lors de la liaison dynamique d'une classe) et permette de les utiliser.

- Permettre les modifications d'objets

  Il est nécessaire de donner à l'utilisateur la possibilité de modifier les données et le code utilisés par son application. Toute modification crée une suite de l'exécution différente, pour laquelle est conservée un historique (cf III.2.7).

  La création de cet historique peut être effectuée en relâchant le mode de reproduction après une modification, et en passant en mode enregistrement. Il faut ajouter un mécanisme de gestion de ces historiques qui permette de les relier entre eux et de les enchaîner lors d'une reproduction prenant en compte plusieurs modifications.

- Permettre la prise de photographies de l'exécution

  Il est nécessaire de donner à l'utilisateur la possibilité de définir des photographies de l'exécution à partir de l'historique. Ces photographies permettent en effet de diminuer le temps pris par la réexécution pour arriver à un endroit précis du programme (cf II.3.1.2). Nous n'envisageons pas leur utilisation pendant la phase d'enregistrement, car le coût de ce mécanisme est important et anéantirait tous les efforts effectués jusqu'à présent pour diminuer la perturbation de l'exécution.

  Les photographies de l'exécution peuvent être réalisées à la demande de l'utilisateur lors d'une reproduction et être associées à un point d'arrêt de

l'exécution. Il faut alors disposer d'un mécanisme de création et de réinstallation d'une photographie.

- Reproduire l'exécution des instructions du langage

  Il est nécessaire de reproduire l'exécution des instructions internes à un objet pour faciliter la localisation et la correction d'une erreur. Cette reproduction n'a pas été prise en compte lors de la première exécution pour deux raisons : elle introduirait une perturbation de l'exécution très importante, de plus son utilisation s'avérerait intéressante uniquement pour quelques objets, ceux suspectés d'être en rapport avec l'erreur recherchée (cf III.2.3).

  La reproduction de l'exécution interne à un objet est un mécanisme complémentaire au nôtre qui enregistre l'ordre d'exécution des instructions internes aux objets, que l'utilisateur a sélectionnés. Cet enregistrement s'effectue lors d'une réexécution et peut être répété jusqu'à obtention du comportement attendu. Cette reproduction demande d'utiliser une version particulière des classes. Le compilateur doit en effet instrumenter le code produit pour conserver dans l'historique l'ordre des accès aux variables d'état de l'objet.

VI.3.2 Les utilisations possibles de Thésée

Les utilisations du noyau de réexécution Thésée sont multiples. Dans le cadre de la mise au point, nous avons déjà proposé différentes vues de l'exécution qui tirent parti des informations conservées par Thésée (cf IV.7.1). Nous présentons ci−après différentes utilisations du noyau de réexécution qui peuvent demander le cas échéant quelques modifications.

- Observation

  Thésée peut être utilisé par des outils d'observation des programmes, pour observer le comportement d'une exécution.

- Test de programmes

  Thésée peut être utilisé par des outils de test de programmes, qui permettent, à partir d'une exécution enregistrée, de modifier l'ordre d'occurrence d'événements concurrents pour tester différentes variations d'une exécution.

- Analyse de performances

  Thésée peut être utilisé par des outils d'analyse de performance, qui permettent de connaître le temps d'exécution des différents composants d'une exécution, pour optimiser le temps d'exécution des programmes.

- Analyse du parallélisme

  Thésée peut être utilisé par des outils d'analyse du parallélisme, qui permettent de connaître le comportement d'une exécution, pour optimiser le parallélisme des programmes.

Chapitre VII

Conclusion

VII.1 Rappel des objectifs

L'objectif de notre travail était d'offrir une aide à la mise au point des applications parallèles, réparties et à base d'objets persistants, permettant une mise au point cyclique et offrant une observation de l'exécution à un haut niveau d'abstraction.

La méthode cyclique de mise au point ne peut pas être utilisée directement avec autant de succès pour des applications parallèles et réparties que pour des programmes séquentiels :

- Le non−déterminisme de ce type d'exécution et sa sensibilité à toute perturbation se traduisent par un comportement variable de l'exécution, ce qui rend très difficile la correction des erreurs liées aux conditions d'exécution.
- Les délais de communication entre les sites et l'utilisation de référentiels de temps différents ne permettent plus d'obtenir un arrêt instantané de l'exécution dans un état global cohérent.
- La notion de pas de l'exécution est elle aussi remise en cause, et doit être redéfinie.

VII.2 Travail réalisé

La reproduction d'une exécution définit un cadre intéressant pour la mise au point cyclique :

- Elle apporte une solution au non−déterminisme des exécutions parallèles et réparties, en garantissant un comportement identique de l'exécution mise au point, au niveau des événements considérés, lors des exécutions successives d'une application.
- Elle rend disponible des informations qualitatives et quantitatives sur l'exécution (par le biais de l'historique des événements) qui ne sont connues dans un cadre classique qu'après la fin de l'exécution ou, au mieux, découvertes au fur et à mesure du déroulement de l'application.
- La conservation de l'historique peut cependant provoquer une perturbation importante de l'exécution et changer le cours de l'exécution au point que le comportement erroné attendu ne se reproduise plus.
- L'enregistrement d'une exécution erronée peut nécessiter plusieurs essais, ce qui peut être considéré comme fastidieux par certains. En contrepartie, le programmeur

a alors tout son temps pour localiser et corriger l'erreur mise en évidence par cette exécution.

- La reproduction n'est plus possible après une modification des données ou du code de l'application. La nouvelle suite de l'exécution doit alors être enregistrée pour être ensuite reproduite. Le nouvel historique se raccorde au précédent à l'endroit précis de l'exécution où s'est effectuée cette modification.

Pour faciliter la mise au point cyclique des applications parallèles et réparties à base d'objets persistants, nous avons choisi d'utiliser la méthode de reproduction dirigée par le contrôle :

- Elle minimise la perturbation de l'exécution introduite par l'enregistrement de l'historique, en conservant l'ordre d'occurrence des opérations de communication et de synchronisation et non la valeur de celles−ci.
- Le déterminisme des réexécutions est alors assuré en garantissant la même séquence d'occurrence des événements, ce qui conduit à un comportement observable identique.
- Sa mise en œuvre est cependant complexe car elle nécessite un ordonnancement des événements de l'application et demande un contrôle de la réexécution complète de l'application.

L'utilisation de cette méthode dans un environnement comme le nôtre demande quelques aménagements :

- La répartition de l'exécution doit être reproduite à l'identique lors des réexécutions : le chemin emprunté à travers les différents sites est une caractéristique importante de l'exécution mise au point.
- La nature persistante des informations nécessite la conservation des données initiales utilisées par l'exécution considérée (son contexte initial) pour pouvoir les retrouver avant chaque réexécution.

Nous avons choisi de baser la reproduction de l'exécution sur les objets :

- Le programmeur peut ainsi mettre au point ses applications avec le même niveau d'abstraction que celui utilisé lors de leur conception.
- L'objet apporte une réponse au problème de la granularité de l'information observée, en permettant d'observer une exécution dans ses grandes lignes, sans se perdre dans ses détails. Cette observation macroscopique de l'exécution joue un rôle central dans la mise au point.
- Une reproduction de l'exécution au niveau des objets ne provoque qu'une perturbation limitée de l'exécution. Le volume des informations conservées dans l'historique est sans commune mesure avec celui qui serait nécessaire pour effectuer une reproduction au niveau des instructions.
- Les réexécutions obtenues sont équivalentes : elles ont le même comportement au niveau des opérations sur les objets. Elles ne sont pas identiques car l'ordre d'exécution des instructions qui composent une opération n'est pas garanti.

- La réexécution peut donc diverger de celle enregistrée. La reproduction est alors arrêtée. Cette divergence met en évidence une erreur liée à la concurrence d'exécution des opérations en cours sur l'objet incriminé.

La reproduction de l'exécution interne à un objet n'a pas été prise en compte lors de la première exécution car elle introduirait une perturbation de l'exécution très importante. De plus, son utilisation ne s'avérerait intéressante que pour quelques objets, ceux suspectés d'être en rapport avec l'erreur recherchée. Cette fonction, complémentaire de celle que nous proposons, doit être disponible lors des réexécutions pour être appliquée aux objets sélectionnés par l'utilisateur.

Les informations conservées dans l'historique peuvent être utilisées pour offrir à l'utilisateur plusieurs vues complémentaires de l'exécution dont le but de faciliter la mise au point :

- Ces vues doivent être appropriées aux particularités de l'environnement telles que le parallélisme, la répartition, la persistance des objets, etc.
- Ces vues doivent disposer de fonctions d'affichage puissantes facilitant le contrôle de l'animation, le filtrage et le regroupement des données, etc.
- Ces vues peuvent servir de support au contrôle de l'exécution. En permettant au programmeur de définir des points d'arrêts sur les opérations montrées dans un flot d'exécution, nous lui permettons de choisir l'endroit exact de l'arrêt de la réexécution. Les points d'arrêt ainsi définis gagnent en précision.

Le metteur au point que nous proposons est réalisé par la coopération de plusieurs outils spécifiques, et possède l'avantage d'être modulaire. Ce n'est pas un outil monolithique, figé dans une configuration donnée. Il peut au contraire évoluer et être enrichi par l'adjonction de nouvelles entités ou le remplacement d'un de ses composants.

Les outils spécifiques au contexte de la mise au point que nous avons introduits (noyau de réexécution, vues de l'exécution, etc) peuvent être réutilisés dans un autre contexte puisqu'ils ont été conçus comme des outils indépendants.

Nous avons appliqué les résultats de l'étude précédente au système Guide. Nous avons réalisé un noyau de réexécution, Thésée, qui permet la reproduction d'une application Guide au niveau des opérations sur les objets :

- Les événements conservés dans l'historique correspondent aux événements du modèle d'exécution de Guide. Les appels de méthode permettent de connaître les appels effectués et les objets utilisés, ainsi que la synchronisation des appels concurrents. Les accès aux objets permettent de reproduire les migrations d'objets et les extensions d'activités entre les sites. La disparition des objets doit se reproduire au même moment lors des réexécutions.
- L'historique d'une exécution est conservé de manière automatique par le système Guide lorsque l'application s'exécute dans un mode particulier (le mode Trace). La conservation de l'historique est effectuée par des processus spécialisés Voyeur, différents de ceux mis en œuvre par l'application, qui récupèrent les événements signalés par le système Guide.

- L'historique de l'exécution est créé de manière parallèle et répartie et est constitué d'un historique par site, qui regroupe les historiques des activités locales. Sur chaque site, un processus Voyeur se charge de conserver les événements des activités de ce site dans des historiques différents : un historique par activité. Le premier processus Voyeur est créé sur le site initial au lancement de l'application, les autres étant créés lors de l'extension de l'application sur de nouveaux sites.
- Le contexte initial d'une exécution est conservé de manière automatique par un mécanisme de copie d'objets qui permet de retrouver une copie fraîche au début de chaque réexécution. Ce mécanisme effectue une copie au vol des objets utilisés par l'exécution au premier accès (liaison dynamique).
- Le volume d'informations à conserver est minimisé : seuls les objets réutilisés (et non créés) sont concernés, de plus la copie des objets ne prend pas en compte la composition des objets (copie à un niveau). Pour obtenir le même comportement lors des réexécutions, nous devons conserver, en plus de sa valeur, la localisation du contexte initial de l'exécution.
- La reproduction d'une exécution est effectuée en lançant une nouvelle exécution qui est dirigée par les informations conservées dans l'historique pour que les mêmes événements soient exécutés dans le même ordre. Auparavant, le contexte initial est réinstallé : chaque objet réutilisé est remis dans son état initial, sur son site initial (celui de son premier accès). Chaque processus Voyeur reproduit l'exécution sur son site, en libérant les activités qui sont prêtes à s'exécuter en suivant l'ordre conservé dans l'historique.
- Nous garantissons la reproduction correcte de l'exécution conservée, car nous arrêtons la réexécution si nous obtenons un comportement de l'exécution qui diverge de celui enregistré dans l'historique. Ce phénomène de divergence, qui se traduit par un événement incorrect, correspond à une erreur dans l'application mise au point (par exemple, un appel de méthode non synchronisé).
- Nous proposons plusieurs vues complémentaires d'une exécution Guide, qui permettent de suivre le cours de l'exécution suivant différents points d'intérêt (vue globale des domaines et des activités, vue d'une activité, vue globale des objets et vue d'un objet).

Thésée est écrit en grande partie en langage Guide (8000 lignes), seules les primitives liées à la communication des événements entre le système Guide et Thésée sont écrites en langage C (400 lignes). L'utilisation du langage Guide a facilité la gestion du parallélisme, du partage et de la synchronisation nécessaire à Thésée. Nous utilisons aussi la persistance et les exception de Guide

VII.3 Résultats obtenus

Nous avons montré l'intérêt de la méthode de reproduction d'un exécution pour effectuer la mise au point des applications parallèles et réparties. Cette méthode apporte une solution au non−déterminisme des applications parallèles et réparties et permet une mise au point

cyclique de ces applications. La mise au point s'effectue alors dans un contexte particulier où le comportement de l'exécution à corriger est déjà connu et peut être observé à l'aide de vues de l'exécution adaptées aux particularités de l'environnement d'exécution.

Nous avons montré l'intérêt d'une reproduction basée sur les objets. Ce choix permet de limiter la perturbation de l'exécution introduite par le metteur au point. De plus, la mise au point d'une application s'effectue alors avec le même niveau d'abstraction que celui utilisé lors de sa conception.

Nous avons réalisé un noyau de réexécution, Thésée, qui se charge de manière automatique de l'enregistrement et de la reproduction d'une exécution Guide. La perturbation de l'exécution due à l'enregistrement de l'historique, dans le cas d'applications parallèles utilisant des objets partagés et de la synchronisation, est très faible : de l'ordre de 1.2%.

Thésée est actuellement utilisé par un outil de visualisation de l'exécution en cours de développement au sein de notre laboratoire. Cet outil a pour but d'offrir différentes vues de l'exécution, en se servant des informations que Thésée rend disponible via l'historique de l'exécution. Cette première expérience montre l'intérêt de disposer de telles informations et la pertinence des renseignements qu'elles permettent d'obtenir.

Une version industrielle de Thésée va être réalisée dans le cadre du projet Oode de Bull. Le projet Oode a pour objectif de réaliser une version industrielle du projet Guide, à partir de la version Eliott du système. L'approche que nous proposons a été retenue pour effectuer la mise au point des applications Oode. L'outil de mise au point qui va être développé va de plus devenir l'application de démonstration phare de ce projet.

VII.4 Perspectives

La version actuelle de Thésée ne correspond pas aux exigences d'un véritable outil de mise au point. Il manque en effet des fonctions indispensables à la mise au point qui n'ont pas été prises en compte par notre travail : elles ne faisaient pas partie de nos objectifs et n'ont pas été abordées par manque de temps. Elle peuvent faire l'objet de travaux futurs à l'occasion de la nouvelle réalisation de Thésée sur la version Eliott du système :

- La conservation automatique des entrées effectuées par une exécution apporterait un supplément de confort, et éliminerait une source d'erreur, en permettant de retrouver ces informations lors de la reproduction sans les demander à l'utilisateur.
- L'isolement automatique de l'exécution garantirait que l'exécution mise au point n'est pas troublée par une autre exécution.
- La mise au point symbolique de l'exécution faciliterait la désignation des entités qui participent à l'exécution (classes, objets, variables, etc).
- La modification des objets (code et données), indispensable à la mise au point, apporterait de la souplesse à la méthode de reproduction en permettant d'explorer de nouvelles suites de l'exécution.

- La prise de photographies de l'exécution permettrait de diminuer le temps pris par la réexécution pour arriver à un endroit précis du programme, et pourrait être réalisée à la demande de l'utilisateur lors d'une reproduction.
- La reproduction de l'exécution des instructions du langage, appliquée à certains objets de l'exécution, faciliterait la localisation et la correction des erreurs.

Un noyau de réexécution d'applications, tel que Thésée, est un support très intéressant pour effectuer des recherches dans différents domaines qui s'intéressent à l'exécution d'une application :

- Il peut être utilisé pour faciliter l'observation de l'exécution. Les informations conservées dans l'historique permettent en effet d'offrir des vues de l'exécution qui facilitent la compréhension de son comportement.

  Nous avons proposé dans le cadre de Guide différentes vues complémentaires d'une exécution qui tirent parti des informations conservées dans l'historique. Cette proposition a donné suite, au sein de notre laboratoire, à un travail qui vise à exploiter au mieux la connaissance globale de l'exécution, en proposant de nouvelles vues et en étudiant la présentation des informations.
- Il peut être très utile pour tester un programme. Il permet en effet de disposer d'une exécution, dont on peut modifier l'ordre d'occurrence des événements concurrents, pour tester différentes variations d'une exécution.
- Il peut être utilisé pour analyser les performances d'une application. Le temps d'exécution des différents composants d'une exécution peut être calculé lors de la reproduction, et permettre ainsi d'optimiser le temps d'exécution des programmes.
- Il peut être très utile pour analyser le parallélisme d'une application. Il permet en effet de connaître le comportement d'une exécution dans ses détails, et facilite ainsi l'optimisation du parallélisme des programmes.

Annexe A

Historique d’une exécution

A.1 Étude de la conservation de l’historique sur un site

Nous voulons conserver la partie de l’historique de l’exécution d’une application qui concerne un site. Nous avons à notre disposition plusieurs solutions pour réaliser la communication entre l’application qui s’exécute dans le mode *Trace* et le processus *Unix Voyeur* : objet Guide, sockets, queue de messages et mémoire partagée. Nous voulons que cette communication perturbe le moins possible l’exécution de l’application. Nous allons donc mesurer les performances de chacune, pour retenir la plus efficace.

A.1.1 Cadre de l’étude

Pour faciliter la comparaison des différentes réalisations possibles, nous considérons le cas d’une communication simplifiée qui comprend : une activité du système *Guide*, qui produit des événements, et un processus *Unix Voyeur*, qui les récupère et les traite. Le traitement effectué par le processus Unix *Voyeur* (codage, conservation) n’est pas abordé ici.

Pour effectuer nos mesures, nous utilisons les informations utilisées par l’outil d’observation d’une exécution *Guide*, l’*Observer* [68], dans sa communication avec le système *Guide* pour suivre l’exécution d’une application. Les messages utilisés dans cette communication ont une taille de 96 octets et contiennent les informations suivantes :

```
struct Message {
long
   /* type de l’événement */
    int event;
    /* référence du domaine */
    T_LgeRef domain;
    /* référence de l’activité */
    T_LgeRef activite;
    /* référence de l’objet */
    T_LgeRef object;
    /* numéro du site */
    unsigned int site;
    /* nom de la méthode */
    T_string calledMethodName;
} Message;
```

```
struct T_LgeRef {          /*référence langage*/
    T_SysRef mpoRef;      /*référence système*/
    short int objectLoc; /*index objet en MVO*/
    short int classLoc;  /*index classe en MVO*/
} T_LgeRef;
struct T_SysRef {   /*référence système*/
    int Oid;        /*identificateur unique*/
    int Loc;       /*localisation en MPO*/
} T_SysRef;
```

A.1.2 Mesures et analyse des résultats

Nous avons choisi d'évaluer les performances de ces différentes réalisations en utilisant le temps pris par un appel de méthode comme référence. L'appel de méthode est l'appel au système *Guide* qui est le plus fréquent lors de l'exécution d'une application. Sa mesure est donc significative de la perturbation de l'exécution engendrée par la communication entre l'application qui s'exécute dans le mode *Trace* et le processus *Unix Voyeur*. De plus, pour chaque appel de méthode effectué, deux événements sont signalés : un au moment de l'appel de la méthode et un autre au moment du retour de cet appel de méthode. Tous les autres appels au système *Guide* ne provoquent qu'un seul événement. L'appel de méthode est donc l'appel au système *Guide* qui est le plus perturbé par la communication rajoutée, sa mesure est donc pertinente pour l'évaluation de l'efficacité du mécanisme utilisé.

La mesure du temps mis par l'appel de méthode, est calculé en exécutant un certain nombre de fois (100000 fois) le même appel de méthode sur le même objet et en divisant le temps total par le dit nombre. L'objet utilisé est local, et la méthode est sans paramètre. La machine utilisée est une station de travail Sun Sparc sous SunOs Release 4.1.1. Nous utilisons la version V1.6 du système Guide.

Les tableaux ci−dessous présentent le temps UC mis par un appel de méthode dans les différentes réalisations.

temps UC en ms	Guide	Fichier Unix	Objet Guide
appel sans paramètre	0.789	1.779	2.933

temps UC en ms	Sockets domaine Unix	Queue de message	Mémoire partagée	
			Sémaphore Unix	Sémaphore Guide
appel sans paramètre	1.688	1.443	2.069	1.949

La première mesure du premier tableau correspond au coût de l'appel de méthode dans le système *Guide* sans aucune modification. C'est la mesure de référence qui va nous permettre d'évaluer le coût engendré par l'ajout de la communication entre l'application en mode *Trace* et le processus *Unix Voyeur*.

La deuxième mesure de ce tableau correspond au coût de l'appel de méthode dans le cas où le flot d'exécution de l'application conserve lui−même les événements dans un fichier *Unix*. Cette mesure a été effectuée uniquement pour évaluer le gain apporté par l'utilisation d'un processus différent pour conserver l'historique de l'exécution d'une application.

La dernière mesure de ce tableau correspond à l'utilisation d'un objet *Guide* comme support de communication entre l'exécution de l'application et le processus *Unix Voyeur*. Cette solution s'avère être très coûteuse, elle est même moins efficace que la conservation directe de l'historique dans un fichier *Unix*. Son coût s'explique par le fait que deux événements sont signalés lors de chaque appel de méthode sur un objet, l'appel et le retour. Or le signalement des événements est aussi réalisé à l'aide d'appels de méthode. On a donc en tout trois appels de méthode exécutés pour chaque appel initial. De plus, le système *Guide* effectue un traitement supplémentaire lors de l'appel de méthode initial qui permet d'interrompre cet appel pour en exécuter un autre, celui correspondant au signalement de l'appel.

Le deuxième tableau montre les résultats obtenus dans les réalisation utilisant les sockets Unix, une queue de messages et la mémoire partagée avec des sémaphores Unix et avec des sémaphores Guide. La solution utilisant une queue de messages *Unix* est la plus performante de toutes, elle double quand de même le coût de l'appel de méthode.

Les résultats obtenus pour les différentes réalisations sont loin d'être satisfaisants. La perturbation engendrée par la communication entre l'application qui s'exécute en mode *Trace* et le processus *Unix Voyeur*, même dans le cas le plus efficace, est encore trop importante pour être acceptable.

Nous avons ensuite essayé d'améliorer les résultats obtenus, en regroupant les événements (pour les transmettre par paquets) de façon à diminuer le trafic de la communication. Nous avons alors choisi de comparer les performances obtenues par les deux solutions les plus intéressantes : l'utilisation d'une queue de message, qui est la solution la plus efficace, et l'utilisation de la mémoire partagée, qui est la seule solution dont les performances puissent être améliorées de manières sensibles. Cette solution est en effet pénalisée par l'emploi de sémaphores, et une gestion efficace de la zone partagée doit permettre d'obtenir de meilleurs résultats.

A.2 Informations conservées dans l'historique

A.2.1 Codage des informations

Nous utilisons une conservation de l'historique par activité du système *Guide*, nous n'avons donc pas besoin de conserver pour chaque événement l'identification de l'activité *Guide* concernée. Par contre, il faut rajouter à chaque événement la valeur du compteur d'événements ainsi que le type de l'événement. Les autres informations définissant les événements peuvent être simplifiées comme suit :

- Un identificateur d'objet (domaine, classe, instance) est une référence du système *Guide*, dont seule la partie *Oid* (identificateur d'objet) nous intéresse. L'*Oid* d'un

objet *Guide* ne permet pas son accès, dans ce cas la référence système est nécessaire, mais c'est un identificateur unique qui permet de le différencier des autres objets. Ce qui est suffisant pour notre usage.

- Une méthode est identifiée par son nom, qui est une chaîne de 46 caractères. Utiliser le nom de la méthode comme identification prend trop de place. On préfère utiliser un index pour identifier la méthode.

  Nous avons deux façons d'obtenir cet index :

  - Il peut être calculé. Dans ce cas l'activité *Guide* conserve les noms des méthodes que l'application utilise dans une table qui lui est propre. A chaque nouveau nom de méthode elle associe un numéro d'entrée (hash−code). Elle utilise ce numéro pour identifier cette méthode. A la fin de l'exécution de l'application, elle transmet cette table au processus *Unix Voyeur*, pour qu'il puisse retrouver les noms des méthodes à partir des index.
  - Il peut être récupéré à partir du système *Guide*. Le système *Guide* identifie une méthode par son numéro d'entrée dans la table (*TDE*) qui contient toutes les méthodes définies pour une classe donnée. L'index d'une méthode est donc propre à une classe.

  Nous avons choisi la deuxième solution car elle ne nécessite pas de traitement supplémentaire. Par contre elle est plus liée au système *Guide.*

Le codage utilisé est le suivant :

- Le type de l'événement est conservé sur un octet.
- La valeur du compteur d'événements est conservée sur trois octets (en complément du type).
- L'Oid d'un objet, ainsi que le Loc, sont conservés sur quatre octets.
- Le numéro de site est conservé sur un octet.
- L'index de la méthode est conservée sur deux octets.

A.2.2 Liste des événements

Nous présentons ci−après la liste des événements utilisés par Thésée ainsi que les informations qui leurs sont associées. Nous distinguons ceux qui sont conservés dans l'historique (H), des autres, signalés uniquement lors de la reproduction.

DOMAIN_END (H),

ACTIVITY_SUSPEND,

ACTIVITY_RESUME (H),

ACTIVITY_KILLED (H),

ACTIVITY_END (H),

OBJECT_UNLINK (H),

OBJECT_UNMAPPED,

OBJECT_NOT_UNMAPPED,

METHOD_EFFECTIVE_CALL,

METHOD_EFFECTIVE_RETURN,

METHOD_RETURN (H),

METHOD_SYNCHRO_RETURN (H),

SYNCHRO_OK :

```
struct T_mess1 {
    int eventCpt;    /*type evnt + compteur d'evnt*/
} T_mess1;
```

DOMAIN_CREATE (H),

ACTIVITY_CREAT (H),

ACTIVITY_EXTEND,

ACTIVITY_KILL_ACTIVITY (H),

OBJECT_MPO_GET (H),

OBJECT_GETDESC (H),

OBJECT_MVO_LINK (H),

OBJECT_END_UNLINK (H),

OBJECT_UNMAPPING (H),

OBJECT_DESTROY (H),

CLASS_LINK :

```
struct T_mess2 {
    int eventCpt;    /*type evnt + compteur evnt*/
    int identifier;  /*Oid objet/numéro site*/
} T_mess2;
```

OBJECT_FIRST_GETDESC (H),

OBJECT_FIRST_MVO_LINK (H),

OBJECT_FIRST_UNMAPPING (H),

OBJECT_CREATED_LINK_EVENT (H),

METHOD_CALL (H),

METHOD_SYNCHRO_CALL (H) :

```
struct T_mess3 {
    int eventCpt;    /*type evnt + compteur d'evnt*/
    int identifier;  /*Oid objet*/
    int index;       /*index méthode/Loc objet*/
} T_mess3;
```

Bibliographie

[1] E. Adams, S.S. Muchnick, ''Dbxtool: A Window−Based Symbolic Debugger for Sun Workstations'', *Software Practice and Experience*, 16(7), July 1986.

[2] H. Agrawal, R.A. DeMillo, E.H. Spafford, ''An Execution−Backtracking Approach to Debugging'', *IEEE Software*, pp. 21−26, May 1991.

[3] H. Agrawal, J.R. Horgan, ''Dynamic Program Slicing'', *ACM SIGPLAN Notices*, pp. 246−256, June 1990.

[4] J.F. Andriamalala, *Application des techniques de visualisation à la mise au point*, (DEA Informatique), Institut National Polytechnique de Grenoble, Bull−IMAG, 2 av de Vignate, 38610 Gières, Septembre 1991.

[5] R. Amouroux, *Aide graphique à la conception orienté−objet dans un environnement intégré*, (DEA Informatique), Institut National Polytechnique de Grenoble, Bull−IMAG, 2 av de Vignate, 38610 Gières, Septembre 1992.

[6] W. Appelbe, C. McDowell, ''Developping Multitasking Programs'', *Proc. of Hawaï International Conf. on System Sciences IEEE*, pp. 94−101, january 1988.

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

[8] Z. Aral, I. Gertner, ''High−Level Debugging of Distributed Systems'', *SIGPLAN NOTICES*, 24(1), January 1989.

[9] M.J. Bach, *The Design of the Unix Operating System*, Prentice−Hall International Editions, 1986.

[10] 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, G. Vandôme, ''Architecture and Implementation of Guide, an Object−Oriented Distributed System'', *Computing Systems*, 4(1), pp. 31−67, 1991.

[11] P.C. Bates, J.C. Wileden, V.R. Lesser, ''A debugging tool for distributed systems'', *Second Annual Phœnix Conf. on Computers and Communications*, pp. 311−315, 1983.

[12] F. Boyer, H. Jamrozik, E. Lenormand, M. Santana, *Une introduction à l'environnement de développement Cybèle*, (noteE05), Bull−IMAG, Z.I. de Mayencin, 2 av de Vignate, 38610 Gières, Juin 1992.

[13] G. Brooks. G.J. Hansen, S. Simmons, ''A new Approach to Debugging Optimized

Code'', *ACM SIGPLAN'92 Conf. on Programmining language Design and Implementation*, 27(7), pp. 1−11, July 1992.

[14] K.M. Chandy, L. Lamport, ''Distributed snapshots: Determining global states of distributed systems'', *ACM Transactions on Computer System*, 3(1), pp. 63−75, February 1985.

[15] W.H. Cheung, J.P. Black, E. Manning, ''A Framework for Distributed Debugging'', *IEEE Software*, pp. 106−115, January 1990.

[16] P−Y. Chevalier et D. Hagimont, S. Krakowiak, X. Rousset de Pina, ''System support for shared objects'', *5th European SIGOPS Workshop*, Septembre 1992.

[17] J.D. Choi, B.P. Miller, R.H.B. Netzer, ''Techniques for Debugging Parallel Programs with Flowback Analysis'', *ACM Transaction on Programming Languages and Systems*, 13(4), pp. 491−530, October 1991.

[18] R. Cooper, ''Pilgrim : A debugger for Distributed Systems'', *Proc. of the 7th Int. Conf. on Distributed Computing Systems*, IEEE, pp. 458−465, 1987.

[19] R. Curtis, L. Wittie, ''BugNet: A Debugging System for Parallel Programming Environments'', *Proc. of the 3rd int. Conf. on Distributed Computing Systems*, Hollywood, October 1982.

[20] D. Decouchant, A. Duda, A. Freyssinet, E. Paire, M. Riveill, X. Rousset de Pina, G. Vandôme, ''Guide : an Implementation of the Comandos Object−Oriented architecture on Unix'', *Proc. European Unix Users Group (EUUG) Autumn Conference*, Lisbon, october 1988.

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

[22] *Dynix Pdbx Parallel Debugger User's Manual*, Sequent Corp., 1986.

[23] S. Eichholz, F. Abstreiter, ''Runtime Observation of Parallel Programs'', *CONPAR'88*, Manchester, September 1988.

[24] S. Feldmann, C. Brown, ''IGOR: A System for Program Debugging via Reversible Execution'', *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1), pp. 112−123, January 1989.

[25] C. Fidge, ''Timestamps in Message−Passing Systems that Preserve the Partial Ordering'', *Proc. of the 11th Australian Computer Science Conference*, Univ. of Queensland, February 1988.

[26] A. Freyssinet, *Architecture et Réalisation d'un Système réparti à Objets*, Thèse de doctorat, Université Joseph Fourier Grenoble, Juillet 1991.

[27] J. Gait, ‘‘A debugger for concurrent programs’’, *Software Practice and Experience*, 15(6), pp. 539−554, June 1985.

[28] H. Garcia−Molina, F. Germano, W.H. Kohler, ‘‘Debugging a distributed computing system’’, *IEEE Transaction on Software Engineering*, 10(2), pp. 210−219, March 1984.

[29] G.S. Goldszmidt, S. Yemini, ‘‘High−Level Language Debugging for Concurrent Programs’’, *ACM Transactions on Computer Systems*, 8(4), November 1990.

[30] J. Griffin, *Parallel Debugging System User’s Guide*, Technical Report Los Alamos National Laboratory, 1987.

[31] D. Haban, ‘‘DTM−A Method for Testing Distributed Systems’’, *6th Symposium on Reliability in Distributed Software an Database Systems*, Williamsburg, Virginia, March 1987.

[32] P.K. Harter, D.M. Heimbigner, R. King, ‘‘IDD: An Interactive distributed debugger’’, *Proc. of the 5th Int. Conf. on Distributed Computing Systems*, pp. 498−506, May 1985.

[33] U. Hölzle, C. Chambers, D. Ungar, ‘‘Debugging Optimized Code with Dynamic Deoptimization’’, *ACM SIGPLAN’92 Conf. on Programmining language Design and Implementation*, 27(7), pp. 32−43, July 1992.

[34] A. Hough, J. Cuny, ‘‘Belvedere: Prototype of a pattern−oriented debugger for hyghly parallel computation’’, *Proc.of the 1987 Int. Conf. on Parallel Processing*, 1987.

[35] W. Hseush, ‘‘DataPath Debugging : Data Oriented Debugging for a Concurent Programming Langage’’, *SIGPLAN NOTICES*, 24(1), January 1989.

[36] M. Hurfin, N. Plouzeau, M. Raynal, ‘‘A debugging tool for distributed Estelle programs’’, *Computer and Communication*, (à paraître).

[37] H. Jamrozik, *Gestionnaire de types et de classes dans un langage à objets, Application au système Guide*, (DEA Informatique), Universite Joseph Fourier, Institut National Polytechnique de Grenoble, Bull−IMAG, 2 av de Vignate, 38610 Gières, Juin 1989.

[38] H. Jamrozik, C. Roisin, M. Santana, ‘‘A Graphical Debugger for Object−Oriented Distributed Programs’’, *Tcchnology of Object−Oriented Languages and Systems (TOOLS) U.S.A 91*, pp. 117−128, July 1991.

[39] H. Jamrozik, M. Santana, *Mise au point dans Guide: Problèmes et Perspectives*, (noteE01), Bull−IMAG, Z.I. de Mayencin, 2 av de Vignate, 38610 Gières, Mars 1991.

[40] J.M. Jezequel, ‘‘Building a Global Time on Parallel Machines’’, *Proc. 3rd Int. Workshop on Distributed Algorithms*, Nice, September 1989.

[41] S. Jones, R.H. Barkan, L.D. Wittie, ‘‘Bugnet: A Real−Time Distributed Debugging System’’, *Proc. of the 6th Int. Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, pp. 56−65, March 1987.

[42] J. Joyce, G. Lomow, K. Slind, B. Unger, ‘‘Monitoring Distributed Systems’’, *ACM Transaction on Computer System*, 5(2), pp. 121−150, May 1987.

[43] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, 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.

[44] S. Lacourte, ‘‘Exception 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.

[45] S. Lacourte, *Exception dans les langages à objets*, Thèse de doctorat, Université Joseph Fourier Grenoble, Juillet 1991.

[46] L. Lamport, ‘‘Time, Clocks and Ordering of Events in a Distributed System’’, *Communication of the ACM*, 21(7), pp. 558−565, July 1978.

[47] J.R. Larus, ‘‘Abstract Execution : A Technique for Efficiently Tracing Programs’’, *Software Practice and Experience*, 20(12), pp. 1241−1258, December 1990.

[48] C.H. LeDoux, D.S. Parker, ‘‘Saving traces for Ada debugging’’, *In Ada In Use, Proc. of the Ada Int. Conf*, pp. 87−108, 1985.

[49] T.J. Leblanc, J.M. Mellor−Crummey, ‘‘Debugging Parallel Programs with Instant Replay’’, *IEEE Transactions on Computers*, C−36(4), pp. 471−482, April 1987.

[50] E. Leu, A. Schiper, ‘‘Techniques de déverminage pour programmes parallèles’’, *Techniques et Science Informatiques*, 10(1), Janvier 1991.

[51] E. Leu, A. Schiper, A. Zramdini, ‘‘Execution Replay on Distributed Memory Architectures’’, *Proc. on the 2’nd IEEE Symp. on Parallel and Distributed Processing*, Dallas, pp. 106−112, December 1990.

[52] J.E. Lumpp, T.L. Casavant, H.J. Siegle, D.C. Marinescu, ‘‘Specification and Identification of Events for Debugging and Performance Monitoring of Distributed Multiprocessor Systems’’, *Proc. of the 10th Int. Conf. on Distributed Systems*, pp. 476−483, June 1990.

[53] P.C. Lewis, D. Reed, ‘‘Debugging Shared Memory Parallel Programs’’, *SIGPLAN NOTICES*, 24(1), January 1989.

[54] *Mach Kernel Interface Manual*, Dept of Computer Science, Carnegie−mellon University, Pittsburgh, PA 15213 (USA), January 1990.

[55] A.D. Malony, D.A. Reed, J.W. Arendt, D. Grabas, R.A. Aydt, B.K. Totty, ‘‘An

Integrated Performance Data Collection Analysis and Visualisation System'', *Proc. of the 4th Conf. on Hypercube Concurrent Computers and Applications*, pp. 229−236, 1989.

[56] M. Mansouri−Samani, M. Sloman, *Monitoring Distributed Systems*, (DOC92/23), imperial College of Science Technology and Medecine, Department of Computing, London SW7 2BZ, UK, September 1992.

[57] K. Marzullo, R. Cooper, M.D. Wood, K.P. Birman, ''Tools for Distributed Application Management'', *IEEE Computer*, pp. 42−51, August 1991.

[58] F. Mattern, ''Virtual Time and Global States of Distributed Systems'', *Proc. Tenth Int. Conf. on Distributed Algorithm*, pp. 215−226, Noth−Holland 1989.

[59] C.E. McDowell, D.P. Helmbold, ''Debugging Concurrent Programs'', *ACM Computing Surveys*, 21(4), pp. 593−622, December 1989.

[60] M. Meysembourg−Männlein, *Modèle et Langage à Objets pour le Programmation d'Applications Réparties*, Thèse de doctorat, Université Joseph Fourier Grenoble, Juillet 1989.

[61] B.P. Miller, J.D. Choi, ''Breakpoints and halting in distributed programs'', *Proc. of the 8th Int. Conf. on Distributed Computing Systems*, pp. 316−323, June 1988.

[62] B.P. Miller, J.D. Choi, ''A mechanism for efficient debugging of parallel programs'', *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1), pp. 141−150, January 1989.

[63] P. Moukeli, *Les Principales Approches de Conception des Débogueurs pour les Langages Parallèles*, (90−05), Laboratoire LIP Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07 france, Janvier 1990.

[64] *NeWs Preliminary Technical Overview*, Sun MicroSystems, 1986.

[65] H. Nguyen Van, M. Riveill, C. Roisin, *Manuel du langage Guide*, Bull_Imag, Z.I. de Mayencin, 2 av de Vignate, 38610 Gières, Décembre 1990.

[66] H. Nguyen Van, *Compilation et Environnement d'Exécution d'un Langage à base d'Ojets*, Thèse de doctorat, Université Joseph Fourier Grenoble, Février 1991.

[67] D. Pan, M. Linton, ''Supporting Reverse Execution for Parallel Programs'', *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1), pp. 124−129, January 1989.

[68] C. Roisin, M. Santana, *The Observer: A tool for observing Object−Oriented Distributed Applications*, (91−08), Bull−Imag, 2 av de Vignate, 38610 Gières, Janvier 1991.

[69] J.F. Roos, L. Courtrai, JF. Méhaut, ''Execution Replay of Parallel Programs'', *Proc of the Euromicro Workshop on Parallel and Distributed Processing*, pp. 429−434, January 1993.

[70] R. Rubin, L. Rudolph, D. Zernik, ''Debugging Parallel Programs in Parallel'', *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*,

24(1), pp. 216−225, January 1989.

[71] M. Salome, *Mise au point de programmes parellèles en langages d'acteurs : l'environnement de Plasma II*, Thèse de doctorat, Université Paul Sabatier de Toulouse, Septembre 1991.

[72] R.W. Scheifler, J. Gettys, ''The x window system'', *ACM Transactions on Graphics*, 5(1), April 1986.

[73] N. Shahmehri, P. Fritzson, *Algoritmic Debugging for Imperative Langages with Side effects*, (89−49), Institutionen för datavetenskap Universietet i Linköping och Tekniska Högskolan, Linköping, Sweden, November 1989.

[74] E.Y. Shapiro, *Algoritmic program debugging*, MIT Press, 1983.

[75] R. Snodgrass, ''A relational approach to minitoring complex systems'', *ACM Transactionon Computer Systems*, 6(2), pp. 158−196, May 1988.

[76] D. Socha,M.L. Bailey, D. Notkin, ''Voyeur : Graphical Views of Parallel Programs '', *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1), pp. 206−215, January 1989.

[77] R. Stallman, *The Gnu Debugger*, Tecnical Report, Free Software Foundation, Inc, 675 Mass.Avenue, Cambridge, ma, 02139, USA, 1986.

[78] J.M. Stone, ''Debugging concurrent processes: a case study'', *Poc. of the SIGPLAN'88 Conf. on Programming Langage Design and Implementation*, 23(7), pp. 145−152, July 1988.

[79] H. Tokuda, ''A Real−Time Monitor for a Distributed Real−Time Operating System'', *Proc. of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, 24(1), pp. 68−77, January 1989.

[80] C. Valot, *A Survey of distributed debugging techniques*, Note technique SOR−94, Projrt SOR − INRIA Rocquemcourt, Septembre 1990.

[81] JY. Vion−Dury, *Etude et Réalisation d'un Metteur au Point Graphique pour Applications Réparties à Objets*, Mémoire CNAM Ingénieur, (à parraître), Décembre 1993.

[82] M. Young, R. Taylor, ''Combining Static Concurrency Analysis with Symbolic Execution'', *IEE Transactios on Software Engineering*, 14(10), October 1988.

Chapitre I

Introduction

Chapitre II

État de l'art de la mise au point

Chapitre III

Définition d'un système de mise au point évolué

Chapitre IV

Un metteur au point pour Guide

Chapitre V

Thésée : mise en œuvre du noyau de réexécution

Chapitre VI

Évaluation

Chapitre VII

Conclusion

Annexe A

Historique d'une exécution