Coordination entre outils dans un environnement intégré de développement de logiciels

Fabienne Boyer

THESE

présentée par

Fabienne Boyer

pour obtenir le titre de

Docteur de l'Université
Joseph Fourier − Grenoble I

(arrêté ministériel du 23 novembre 1988)

Spécialité : Informatique

Coordination entre outils dans un environnement intégré de développement de logiciels

Thèse Soutenue devant la commission d'examen le : 8 février 1994

Roland Balter	Président
Daniel Herman	Rapporteur
Bernard Lang	Rapporteur
Sacha Krakowiak	Directeur de thèse
Jacques Mossière	Examinateur
Miguel Santana	Examinateur

Thèse préparée au sein du Laboratoire Unité Mixte Bull−IMAG

Coordination entre outils dans un environnement intégré de développement de logiciels

Résumé

Cette thèse propose un mécanisme de coordination entre outils pour un environnement intégré de développement de logiciels. Le rôle d'un tel environnement est d'accroître la productivité des développeurs et d'améliorer la qualité du logiciel développé en intégrant les composants de l'environnement. Le mécanisme que nous proposons intégre les outils de développement, en permettant à ceux−ci d'échanger des informations pour agir de manière cohérente et homogène. Nous qualifions ces échanges de *coordinations*.

Ce mécanisme se fonde sur un modèle de coordination mis en œuvre par un langage nommé Indra qui comprend des parties déclaratives et impératives. Il présente les caractéristiques suivantes.

Pour faciliter l'évolution des composants de l'environnement (outils et coordinations), il permet d'une part d'exprimer les coordinations de manière modulaire, *en dehors du code des outils*. Il permet d'autre part d'exprimer *explicitement* l'évolution dynamique des coordinations, en s'inspirant du concept d'automate d'états fini.

Afin de pouvoir coordonner des outils qui présentent des interfaces de coordination indépendantes les unes des autres, il définit un espace global de coordinations au travers duquel sont exprimées les liaisons entre les interfaces.

Enfin, pour gérer la forte évolution dynamique des outils actifs, il permet de désigner les outils sans connaître leur identité, en fournissant une désignation associative fondée sur le concept d'arbre attribué.

Ce mécanisme a été réalisé au dessus du système réparti et orienté objet Guide.

Mots−clés

Intégration d'applications, bus de messages, interopérabilité, communications asynchrones, désignation, arbres attribués, automate d'états.

Abstract

This thesis proposes a coordination mechanism for tools of an integrated software development environment. The objective of integrated environments is to increase developer's productivity and improve the quality of the developed software through integrating the environment components. The mechanism that we propose integrates development tools, by allowing them to exchange information in order to execute in a coherent and homogeneous way. We call these exchanges *coordinations*.

The features of our mechanism are provided through a specific language called Indra, which includes both declarative and imperative parts. It presents the following characteristics.

To ease the evolution of the environment components (tools and coordinations), il allows to express coordinations in a modular way, *ouside the tools code*. It also allows to express the dynamic evolution of coordinations *explicitly*, through the concept of finite state machine.

To allow to coordinate independently developped tools, which present coordination interfaces that do not match, it provides a global space of coordinations through which the links between the different interfaces are expressed.

Finally, to deal with the dynamic evolution of active tools, it allows to select tools without knowing their identity, by providing an associative naming service based on the attributed tree concept.

This mechanism has been implemented on top of the Guide object oriented and distributed system.

Key−words

Application integration, message bus, interoperability, asynchronous communications, attributed trees, finite state machines.

Je tiens à remercier

Monsieur Roland Balter, Professeur à l'Université Joseph Fourier et Directeur du laboratoire Bull−IMAG, qui m'a fait l'honneur de présider le jury de cette thèse. Je le remercie également pour ses encouragements et pour la motivation qu'il sait insuffler à chaque membre de son équipe.

Monsieur Daniel Herman, Professeur à l'Université de Rennes I, et Monsieur Bernard Lang, Directeur de Recherche à l'INRIA, qui ont accepté d'être rapporteurs de mon travail.

Monsieur Sacha Krakowiak, Professeur à l'Université Joseph Fourier et responsable du projet Guide, pour m'avoir acceptée dans son équipe et pour la confiance qu'il m'a accordée durant ces trois années de travail.

Monsieur Jacques Mossière, Professeur à l'Institut National Polytechnique de Grenoble, pour l'aide précieuse qu'il m'a apportée sur la fin de cette thèse.

Monsieur Miguel Santana, Docteur de l'Université Joseph Fourier et Ingénieur Bull, pour l'encadrement de ce travail, pour ses conseils pertinents, et pour son soutien qui a été permanent.

Monsieur Rémy Amouroux, pour les discussions fructueuses que nous avons eues.

La Société Bull, qui a financé mon travail.

Je tiens également à remercier très sincèrement l'ensemble des personnes du laboratoire Bull−IMAG, grâce auxquelles j'ai pu réaliser ce travail dans une ambiance exceptionnelle.

Introduction

.1 Le contexte

Parallèlement à l'essor des technologies relatives aux ordinateurs, apparaissent des applications informatiques de taille, de complexité et de coûts de plus en plus élevés. Les avancées technologiques informatiques transforment en effet les besoins des individus, en offrant de nouvelles possibilités toujours plus attractives et plus prisées par ces derniers.

Afin de faire face à l'accroissement des facteurs taille, complexité et coût des applications, on cherche à assister au mieux la production des logiciels informatiques. Pour ce faire, on fournit des outils qui déchargent les producteurs de logiciels d'un ensemble de tâches fastidieuses, contraignantes et sources de situation d'échec.

Un *environnement de développement de logiciel* a précisément pour rôle de fournir cette assistance. Afin de pourvoir à une demande toujours croissante, la recherche se rapportant au domaine des environnements de développement constitue un axe important, et une multitude de produits et de techniques sont apparus dans les dernières années.

Les diverses techniques apparues en vue d'améliorer l'assistance fournie n'en restent pas moins complémentaires, apportant chacune une plus−value souvent spécifique. Exploiter au mieux l'abondance des techniques devient dès lors un objectif primordial, qui se concrétise par l'intégration de ces techniques au sein d'un même environnement de développement. C'est précisément le rôle des environnements de développement qualifiés d'*intégrés*.

Cependant, les possibilités grandissantes apportées par l'amélioration des matériels informatiques diversifient les demandes : les logiciels requis sont plus spécifiques et nécessitent une assistance pratiquement ad−hoc. Or, concevoir et réaliser un environnement de développement requiert un effort considérable, pouvant rapidement devenir disproportionné en rapport au coût du logiciel à développer. On tend donc à dissocier la partie de l'outillage fourni que l'on peut considérer comme *générique* de la partie *spécifique* au logiciel à développer. Cette tendance constitue un principe fondamental dirigeant la conception d'environnements de développement intégrés.

Enfin, dès lors que la durée du développement d'un logiciel peut être importante, se mesurant en années, il devient nécessaire de fournir un environnement de développement

capable d'évoluer. D'une part, car de nouvelles techniques peuvent apparaître sur le marché et qu'il est très appréciable de pouvoir les incorporer dans l'environnement. D'autre part, parce que la longue durée du développement permet difficilement de définir la partie générique de l'environnement de développement une fois pour toutes.

Intégration, généricité et capacité d'évolution représentent les tendances essentielles qui régissent la construction d'environnements de développement intégrés.

Trois composantes sont couramment distinguées dans un environnement de développement : les données, les outils, et le procédé de développement.

Les outils sont les composants au travers desquels les développeurs construisent, manipulent et traitent les données qui vont progressivement constituer le logiciel développé. Le procédé de développement dénote la démarche adoptée vis−à−vis de l'utilisation des outils et de leur application sur les données.

Les techniques apparues sur le marché et permettant d'améliorer la qualité d'un environnement de développement s'attachent généralement à l'une des trois composantes citées. Certaines visent à fournir des fonctions de gestion des données avec un haut degré de qualité, permettant de décharger les développeurs d'un ensemble de tâches de gestion de ces données. D'autres ont pour objet d'améliorer la qualité des outils fournis, et d'autres encore tendent à diriger les actions des développeurs en fonction d'un procédé de développement défini, assurant un grand nombre de fonctions de gestion de projet (coordination des développeurs, gestion d'échéances, etc.).

Deux autres composantes, quelque peu indépendantes des précédentes, sont également à considérer : l'interface fournie par l'environnement de développement à ses utilisateurs, et le support d'exécution sous−jacent, commun à l'ensemble des composantes.

Chacune des composantes citées précédemment représente un *axe d'intégration* soumis aux contraintes d'évolution et de généricité.

.2 Le sujet

C'est à un axe d'intégration particulier que nous nous intéressons dans cette thèse, appelé *Intégration du contrôle*. Son objectif est d'améliorer la qualité des outils fournis, en permettant à ceux−ci d'échanger des informations pour agir de manière homogène et coordonnée.

Ces échanges, que nous qualifions de *coordinations*, sont de deux types principaux : les requêtes, qui permettent à un outil de demander un service à un autre outil et qui présentent un caractère obligatoire, et les notifications qui ont un caractère informatif, permettant à un outil de diffuser une information vers un groupe d'outils.

Les problèmes majeurs soulevés par la conception et la réalisation d'un support pour l'intégration du contrôle sont les suivants.

1. Le critère d'évolution cité ci−dessus implique de faciliter l'évolution des outils et des coordinations (ajout, retrait, modification).

2. Le critère de généricité implique d'une part de permettre aux outils d'être écrits dans des langages de programmation distincts. Il implique d'autre part d'autoriser les outils à présenter des interfaces de coordination indépendantes les unes des autres.
3. Il faut prendre en compte le fait que l'ensemble des outils qui sont actifs à un instant donné varie dynamiquement : les activations et désactivations d'outils prennent place au gré des développeurs. La désignation des outils intervenant dans une coordination ne peut donc être basée sur leur identification unique.

4. Le comportement d'un outil actif vis−à−vis de ses coordinations peut évoluer dynamiquement, l'outil pouvant décider, à tout instant de son exécution, de ne plus participer aux mêmes coordinations.

Nous proposons un mécanisme qui apporte des éléments de solution aux problèmes identifiés ci−dessus. Ce mécanisme se fonde sur un modèle de coordination qui présente les caractéristiques suivantes.

1. Du point de vue des structures logiques de contrôle, deux abstractions sont utilisées : l'*événement* au travers d'un bus de messages, et l'*appel de méthode*. Ceci permet de définir les outils en termes d'interfaces abstraites et les coordinations en termes de liaisons entre ces interfaces. Les coordinations peuvent ainsi être exprimées explicitement et modulairement, en dehors des programmes des outils.

2. Par la définition d'un espace de coordination global, l'hétérogénéité des outils est gérée par des traductions appliquées entre les éléments sources d'hétérogénéité et cet espace global.

3. L'état dynamique de l'environnement (organisation des outils actifs) est modélisé sous la forme d'un arbre attribué dont la définition est paramétrable. Un nœud non feuille de cet arbre représente un *groupe* d'outils actifs, et une feuille représente un outil actif. Cette modélisation permet de désigner les outils de manière associative, au travers de leurs attributs et de leur position.

4. La définition des coordinations d'un outil prend la forme de la définition d'un automate d'états fini, les transitions entre états modélisant l'évolution dynamique des coordinations dans lesquelles un outil actif est susceptible d'intervenir.

Nous avons choisi de rendre accessibles les fonctions de gestion des coordinations fournies par ce modèle au travers d'un langage déclaratif nommé Indra. Un environnement d'exécution pour ce langage a été prototypé au dessus du système d'exploitation réparti et à base d'objets Guide. L'ensemble a donné lieu à une expérimentation mettant en jeu quelques développeurs et six outils.

.3 Le cadre du travail

Cette thèse s'est déroulée au sein du projet Guide (Grenoble Universities Integrated Distributed Environment), lancé en 1986 comme un projet commun au Laboratoire de Génie Informatique de l'IMAG et au Centre de Recherche Bull et poursuivi depuis 1990 dans l'Unité Mixte Bull−IMAG.

L'objectif du projet Guide [6] est d'explorer le domaine des langages et systèmes pour applications réparties. En particulier, le développement d'une plate−forme d'exécution pour applications orientées objet, réparties et coopératives, et la conception d'un langage orienté objet (appelé langage Guide) donnant accès aux fonctions fournies par cette plate−forme ont constitué les axes de travail majeurs.

Les applications visées par ce système sont l'ensemble des applications faisant intervenir répartition, partage et coopération, et s'exécutant sur un réseau local de machines hétérogènes. On peut notamment citer les applications relatives au génie logiciel, l'édition coopérative de documents, etc.

Un premier prototypage de la plate−forme Guide a pris place au dessus du système Unix et a permis de valider les choix de base. Les travaux actuels visent d'une part la réalisation d'un deuxième prototype au dessus du micro−noyau Mach 3.0 [1] et d'autre part la réalisation d'un environnement de développement pour les applications écrites en langage Guide. Le travail de cette thèse prend place au niveau de la réalisation de cet environnement de développement, nommé Cybèle.

Une partie des travaux du projet Guide a été menée dans le cadre du projet ESPRIT COMANDOS (Construction and Management of Distributed Office Systems) [23].

.4 L'organisation de la thèse

Chapitre I. Ce chapitre présente tout d'abord le domaine des environnements de développement et définit la terminologie employée tout au long de ce document.

Il introduit ensuite la notion d'environnement de développement intégré, en présentant les divers axes d'intégration qui participent à l'élaboration d'un tel environnement.

La finalité de ce chapitre est de préciser le rôle et la position de l'*intégration du contrôle*, axe auquel nous nous intéressons dans cette thèse.

Chapitre II. Ce chapitre se concentre sur l'intégration du contrôle. Après avoir défini la fonction essentielle de cet axe d'intégration, les caractéristiques qui engendrent les difficultés posées sont énoncées.

La prise en compte de ces caractéristiques nous permet d'établir les critères de qualité d'un mécanisme d'intégration du contrôle. Diverses approches, tirées de travaux existants, sont alors comparées au regard de ces critères. Insuffisances ou facteurs sujets à améliorations sont soulignés.

Les Chapitres III, IV, V et VI présentent le résultat de notre travail.

Chapitre III. Ce chapitre présente le modèle que nous avons conçu pour intégrer les outils de développement du point de vue du contrôle. Après avoir évoqué notre démarche et nos objectifs, nous décrivons les concepts clés de ce modèle au travers du langage qui le met en œuvre, nommé Indra.

Cette présentation est concrétisée dans le **Chapitre IV**, par une illustration par l'exemple de l'utilisation du langage Indra.

Chapitre V. Ce chapitre décrit l'environnement d'exécution du langage Indra. Après avoir présenté nos objectifs, les principaux composants et les algorithmes clés de cet environnement sont présentés.

Chapitre VI. Ce chapitre fournit une évaluation de nos propositions et présente l'état actuel de nos travaux. Le modèle de coordination proposé ainsi que son environnement d'exécution sont évalués et les résultats obtenus par une expérimentation sont rapportés.

Conclusion. En conclusion générale, sont tout d'abord résumés les principaux objectifs qui ont dicté la réalisation de notre travail, ainsi que la démarche que nous avons adoptée. Les aspects de notre proposition que nous considérons comme originaux sont rappelés, de même que l'évaluation du travail réalisé. Enfin, nous énonçons un ensemble de perspectives qui constituent un prolongement de notre travail.

Annexes. Quatre annexes sont fournies à la fin de ce document.

L'annexe A présente la grammaire du langage Indra.

L'annexe B rapporte la gestion des cas d'erreurs et des retours d'informations fournis par le langage Indra.

L'annexe C offre des exemples d'utilisation du langage Indra pour des outils que nous avons intégrés et coordonnés dans l'environnement de développement Cybèle.

Enfin l'annexe D présente les éléments clés qui réalisent l'adéquation du système Guide comme support de l'environnement d'exécution du langage Indra.

Chapitre I

Environnements de développement intégrés

Ce chapitre présente le contexte au sein duquel s'inscrit notre travail.

La première section introduit les différents axes intervenant dans les environnements de développement intégrés, et précise la terminologie utilisée dans ce document .

Les sections suivantes présentent les objectifs, les problèmes et les approches relatives aux axes d'intégration que nous jugeons liés à l'intégration du contrôle, axe auquel nous nous intéressons dans cette thèse. Une discussion sur le rôle et sur la position de notre travail termine ce chapitre.

I.1 Environnements de développement intégrés

Cette section se base sur une vision très simplifiée des environnements de développement, mais dont nous pensons qu'elle est suffisante pour introduire ce domaine.

I.1.1 Environnements de développement

Un environnement de développement a pour rôle d'assister les développeurs(1) dans leur activité de production d'une application informatique, en fournissant un ensemble de mécanismes matériels et logiciels.

Le nombre de développeurs peut être plus ou moins grand, en rapport avec le facteur "taille" supporté par l'environnement de développement. Trois catégories de tailles sont couramment différenciées dans la littérature : *l'individu, le groupe, l'entreprise* [5][69].

La catégorie *individu* comprend les environnements "mono−développeur". La catégorie *groupe* dénote les environnements qui supportent et gèrent plusieurs développeurs travaillant à un tâche commune. En général, ces développeurs travaillent sur des machines connectées par un réseau local. Enfin la catégorie *entreprise* englobe les environnements composés d'un ensemble d'environnements de catégorie *groupe*. L'état actuel des travaux se situe essentiellement au niveau *groupe*.

---

(1) Le terme *développeur* dénote l'ensemble des personnes impliquées dans le développement d'une application : analystes, concepteurs, programmeurs, testeurs, etc.

La finalité de l'activité de développement est la production d'un ensemble de données nécessaires à l'exploitation de l'application par des utilisateurs, ainsi qu'à la maintenance de cette application.

De prime abord, nous disons que cette activité met en jeu un ensemble de *développeurs*, un ensemble d'*outils* et un ensemble de *données*. Les outils sont des programmes exécutables. Ils fournissent des *services* qui effectuent des traitements sur les données et qui permettent aux développeurs de produire, progressivement, les données finales.

Outre les données et les services (outils), on distingue une troisième composante qui intervient dans le développement d'une application, appelée *activité de développement*. Celle−ci spécifie comment (dans quel ordre, à quel moment, etc.) sont appliqués les services sur les données. Elle se décompose naturellement en sous−activités, reliées hiérarchiquement par des relations de précédence. Chaque *activité* du graphe de décomposition définit un objectif à atteindre vis à vis des données.

Seuls certains environnements fournissent explicitement la notion d'activité.

Définitions

Les ***données*** représentent l'ensemble des informations véhiculées lors du développement d'un logiciel. Exemples : programmes sources, documentations, plannings, programmes exécutables, relations de dépendances entre programmes sources, propriétés, contraintes, etc.

Un ***service*** est un traitement agissant sur une ou plusieurs données. Un ***outil*** est un programme exécutable qui fournit un ou plusieurs services. Exemples : un éditeur peut fournir les services éditer, visualiser, etc.

Une ***activité*** représente une phase de travail qui met en jeu des données et des services afin de produire un résultat défini. Exemples : le développement d'un composant donné, la validation d'un composant donné.

Reprenant une modélisation des environnements de développement proposée dans [69], les données sont des *structures* sur lesquelles agissent des *mécanismes* qui sont les services, selon des *règles* définies par les activités.

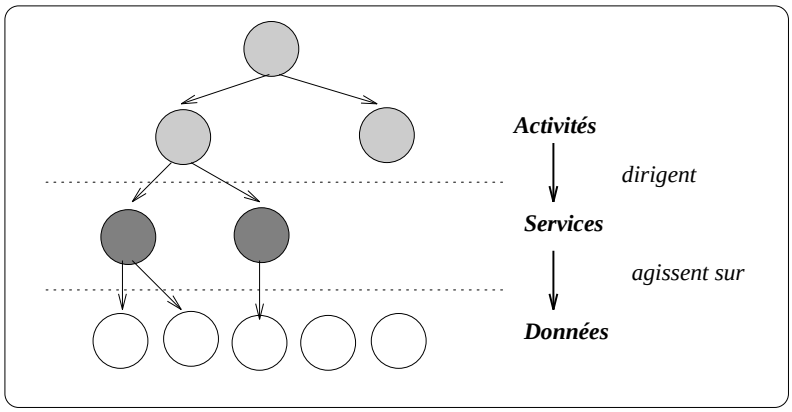


Fig. 1.1 : Relations entre activités, services et données

I.1.2 Objectifs des environnements de développement *intégrés*

L'objectif des environnements de développement *intégrés* est d'accroître la productivité des développeurs et d'améliorer la qualité du logiciel développé en intégrant les composants de l'environnement [16]. Parmi les composants concernés, citons les données, les services, les activités.

Le besoin d'intégration est apparu suite à une émergence considérable d'outils éparpillés et de techniques partielles, couvrant différents aspects du développement avec d'éventuels recouvrements [79]. Dans cette situation, les développeurs doivent passer d'une technique à l'autre, d'un outil à l'autre, sans contrôle du bien−fondé de leurs actions. L'intégration s'est alors avérée un moyen de ne pas remettre en cause ces différents travaux éparpillés, la motivation étant tout au contraire de les exploiter au sein d'une architecture homogène. Dans le même temps, l'intégration est aussi apparue attractive comme support pour la réalisation d'environnements de niveaux *groupe* et *entreprise*. Nous ne nous étendons pas plus sur l'origine des besoins d'intégration, le lecteur intéressé trouvant ces informations dans diverses études [79] [117] [120].

Du point de vue de la perception des développeurs, nous retenons trois aspects ciblés par l'intégration. D'une part, on vise à décharger ceux−ci d'un ensemble de tâches et de contraintes qui deviennent du ressort de l'environnement. D'autre part, on tend à guider et contrôler leurs actions, pour éviter qu'ils entreprennent un travail inapproprié, voué à l'échec. Enfin, on cherche à leur fournir des services homogènes, complets et s'exécutant de manière cohérente.

Les techniques utilisées pour l'intégration différencient cinq axes, liés chacun à un niveau conceptuel distinct [117] [120]. Les fonctions qui doivent être offertes par chacun de ces axes pour fournir un environnement intégré sont succinctement décrites ci−après.

1. **Intégration au niveau de la plate−forme**

   L'ensemble des structures d'exécution impliquées dans l'environnement doivent avoir pour support un système d'exploitation réparti commun, fournissant un ensemble de fonctions de base (gestion de l'hétérogénéité, de la répartition, etc).

2. **Intégration au niveau de la présentation**

   L'uniformité est ici le mot clé : l'interface homme−machine fournie aux développeurs doit être uniforme, tant au niveau de la présentation que de la manipulation.

3. **Intégration au niveau des données**

   Le support de gestion des données ne doit pas se limiter à assurer leur stockage. Il doit prendre à sa charge un ensemble de tâches qui relèvent des données pour libérer les outils et les développeurs de celles−ci. En particulier, il doit permettre à un outil d'accéder directement à une donnée produite par un autre outil.

4. **Intégration au niveau du contrôle**

   Le support de gestion des outils doit fournir des mécanismes permettant à ceux−ci de communiquer et d'interagir afin d'offrir des services plus uniformes, plus complets et plus cohérents.

5. **Intégration au niveau du processus de développement**

   Le support de gestion du processus de développement doit permettre de diriger, guider et contrôler les actions des développeurs selon un processus préalablement défini. A ce niveau d'intégration, le concept d'activité est généralement représenté explicitement au sein de l'environnement.

I.1.3 Contraintes des environnements de développement intégrés

Les environnements intégrés sont soumis à deux contraintes majeures : la généricité et la capacité d'évolution.

La généricité veut qu'un même environnement puisse être utilisé pour développer des applications de domaines différents, en raison du coût trés élevé de construction d'un environnement. Une importante partie des fonctions d'un environnement de développement restent requises, quel que soit le domaine d'application visé. La tendance est par conséquent de concevoir des *noyaux* d'environnements intégrant ces fonctions communes et permettant de définir les fonctions plus spécifiques. En quelque sorte, le noyau doit être *paramétré* pour

définir un environnement de développement complet. Ces noyaux portent le nom de *structures d'accueil*(2).

La paramétrisation est généralement triple, portant sur les données, les services et les activités. Pour chacun de ces niveaux conceptuels, leur *définition* constitue l'élément paramétrable :

Définitions

Données, services et outils possèdent chacun une ***définition*** et un ensemble (éventuellement vide) d' ***instances***.

La *définition* des données constitue une modélisation de ces dernières. Les *instances* sont les données produites, conformes à la modélisation exprimée dans la définition.

La *définition* des services, de même que la définition des activités, constitue une représentation opérationnelle de ces derniers. L'activation d'une définition de service (resp. d'activité) crée une *instance* de service (resp. d'activité).

Alors que la généricité veut que certains composants de l'environnement soient paramétrables, la capacité d'évolution concerne la possibilité qu'ont ces composants d'évoluer après qu'ils aient été définis. L'évolution peut être intra−composant (modification d'un composant), ou extra−composant (ajout / retrait de composants). Les différents types d'évolution sont décrits dans [108].

Le besoin d'évolution résulte principalement de deux causes. Premièrement, en raison des facteurs *durée* et *taille* du développement qui peuvent prendre des proportions importantes, il est difficile de définir les composants paramétrés de l'environnement une fois pour toutes. Deuxièmement, de nouveaux outils ou de nouvelles techniques peuvent apparaître sur le marché et être susceptibles d'accroître la productivité des développeurs. La possibilité d'incorporer ces nouveautés dans l'environnement est donc très appréciable.

Définitions

Un environnement ***générique*** est un méta−environnement qui permet de paramétrer les données, les services et les activités.

Un environnement ***évolutif*** permet de modifier les données, les services et les activités, en cours de développement.

(2) Le terme *Structure d'accueil* est couramment associé au terme *Integrated Project Support Environment (IPSE)* en anglais.

I.1.4 En résumé

Le dilemme suivant résume les contraintes essentielles des environnements de développement intégrés :

L'environnement (au sens de la structure d'accueil) doit être assez générique, afin d'être utilisable pour le développement d'applications très diverses. Il doit dans le même temps être assez spécialisable, pour fournir un support de qualité pour le développement d'une application particulière. Enfin, il doit également être capable d'évoluer pour s'adapter aux techniques nouvelles et aux besoins nouveaux des développeurs.

Du fait de la grande diversité du domaine, résultant de préoccupations, objectifs et approches différentes, la présentation donnée dans cette introduction est restée à un degré de généralité élevé. Ce degré reflète par ailleurs le niveau d'abstraction auquel se place la définition de l'architecture fonctionnelle d'intégration ECMA/TC33 [44], tentative de standardisation dans ce domaine (Fig. 1.2).

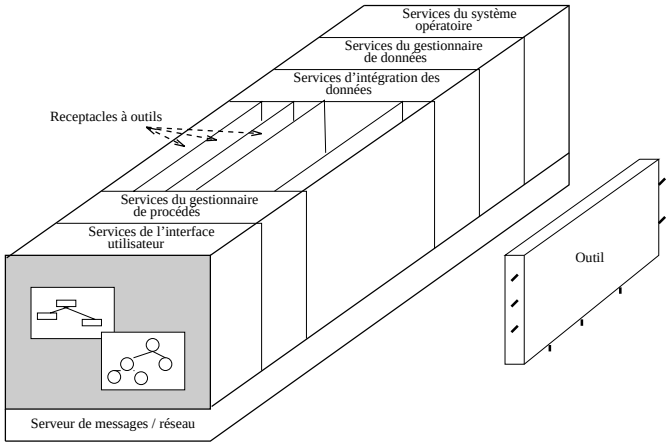


Fig. 1.2 : Modèle d'architecture de l'ECMA/TC33, dit "Toaster Model"

I.2 Intégration des données

Cette section présente les principaux objectifs (I.2.1), problèmes (I.2.2.), et approches (I.2.3) liés à l'intégration des données. Nous terminons sur une discussion en I.2.4.

I.2.1 Objectifs

L'objectif de l'intégration des données est de faire prendre en charge par un support spécifique, des fonctions de gestion des données qui seraient du ressort des développeurs et des outils en l'absence de ce support.

La définition des données doit être un paramètre de configuration de l'environnement (cf. I.1). Un langage de définition est donc requis, ainsi qu'un support pour gérer ces définitions et ces données. Langage et support sont dans la plupart des cas fortement couplés, et constituent ce que l'on appelle un *Système de gestion d'objets* (SGO)[(3)(4)].

Les fonctions requises au niveau d'un SGO sont décrites dans [89]. Nous jugeons les suivantes comme essentielles.

- Assurer la **persistance** des données et de leur définition.
- Gérer l'**évolution** des données via un mécanisme de versions.
- Maintenir la **cohérence** des données.
- Gérer les **accès multiples** aux données.
- Rendre les données accessibles de manière **répartie**.
- Fournir des mécanismes de **tolérance aux pannes**.

Notons que deux approches sont possibles quant à la relation entre le SGO et les outils de l'environnement de développement. Dans la première, les outils sont considérés comme clients du SGO ; le modèle fourni par le SGO est statique. Ceci constitue la majorité des cas. Une autre approche consiste à intégrer au niveau du SGO la définition et la gestion des données et des outils [59][102]; le modèle fourni est dans ce cas dynamique et si les fonctions requises sont plus étendues, les problèmes majeurs restent globalement similaires.

I.2.2 Problèmes

Les principaux problèmes posés s'attachent au *modèle de données* du SGO. Ils proviennent de propriétés qui sont difficiles à satisfaire, certaines étant quelque peu antagonistes. Les propriétés suivantes nous semblent essentielles.

1. Puissance d'expression :

   Le SGO doit permettre de définir les données de manière fine et précise.

---

(3) Object Management System (OMS) en anglais.

(4) Le terme *objet* est considéré ici comme l'unité de donnée rendue identifiable et accessible par le SGO, sans impliquer nécessairement de similitudes avec les objets du monde orienté−objet.

2. Ouverture :

   Le SGO ne doit pas restreindre l'environnement de développement vis−à−vis de l'intégration de nouveaux outils. Pour ce faire, il doit fournir des représentations des données(5) adaptée à celles utilisées par les outils. Il doit également fournir une interface multi−langages pour supporter des outils écrits dans des langages différents.

3. Evolution :

   La définition des données doit pouvoir évoluer, de même que les données elles−mêmes.

4. Performance :

   Ce critère est important car il peut à lui seul remettre en cause certains choix. C'est la raison pour laquelle nous le citons.

I.2.3 Approches

I.2.3.1 Approches basées sur un système de gestion de fichiers

Beaucoup d'approches ont utilisé un système de gestion de fichiers, notamment le SGF d'Unix (Unix programmer's Workbench, Cedar, Object Center, etc.) en tant que modèle de définition des données et implémentation du SGO. Les fonctions fournies sont à la base réduites à la gestion de la persistance et des contrôles d'accès. Elles peuvent être étendues à la répartition dans le cas des systèmes de gestion de fichiers répartis (NFS, AFS, DFS)[103] et à la gestion de versions (ODE) [77].

Les données sont représentées sous la forme de suites de caractères ASCII. En conséquence, chaque outil réalise généralement une transformation de la suite de caractères en une représentation interne structurée et inversement. Ceci entraîne une grande redondance au niveau du travail effectué par les outils. De surcroît, le SGF est passif et il ne peut donc gérer la cohérence de données dont les valeurs sont interdépendantes. Ceci constitue un inconvénient *majeur*.

En revanche, l'évolution des données est simplifiée par la non structuration de celles−ci. Par ailleurs, et ceci constitue certainement l'avantage le plus prisé, la plupart des outils utilisent des fichiers pour stocker les données qu'ils manipulent. En ce sens, l'environnement construit est très ouvert.

I.2.3.2 Approches basées sur un modèle relationnel

Une deuxième approche a été d'utiliser une base de données de type relationnel (Postgres

(5) Nous nous plaçeons ici au niveau de la représentation des données fournie par le SGO à ses clients, et qui peut être différente de la représentation des données utilisée en interne (i.e en mémoire centrale) par le SGO.

[109], Exodus [26]) comme support de gestion et modèle de définition des données. Persistance, distribution, déclencheurs, mécanismes de tolérance aux pannes et gestion des accès multiples sont des fonctions souvent offertes par les bases relationnelles.

Données et associations entre données de valeurs interdépendantes sont représentées de manière fine sous forme de relations. Les mécanismes de déclencheurs, rendant le modèle actif, permettent de faire gérer la cohérence des données et de leurs associations par le SGO. De plus, les modèles relationnels fournissent des mécanismes puissants de vues [36], ce qui permet de fournir différentes représentations des données aux clients.

Une vision relationnelle est toutefois imposée à tout client. Ceci limite l'ajout d'outil. Par ailleurs, l'évolution des données se limite à une évolution additive. Notons enfin que la définition des données à base de relations reste difficile, la complexité de ces dernières requiert qu'une certaine richesse sémantique soit fournie par le modèle lorsque celui−ci implique une description de grain fin. La représentation de données non interprétables par la base (par exemple, du code exécutable) doit être réalisée de manière ad hoc. Finalement, l'impossibilité d'identifier des données indépendamment de leurs valeurs constitue un obstacle important.

I.2.3.3 Approches basées sur un modèle Entité−Relation−Attributs

Les modèles de type Entités−Relations−Attributs (ERA) [29] ont été adoptés par divers projets et produits avec beaucoup de succès, même si certains problèmes restent ouverts. Plusieurs normes se basent sur ce modèle : la norme ANSI IRDS (Information Resource Dictionnary System [81], la norme CAIS (Common APSEs Interface Set) [21] et son évolution CAIS−A [22], la norme ECMA PCTE (Portable Common Tool Environment) et son évolution PCTE+ [13].

Les produits tels que le "repository" d'IBM (AD/Cycle) [61], l'atelier de génie logiciel CASE d'ORACLE s'appuient sur l'IRDS ; le produit PCTE Emeraude [55] s'aligne sur la norme PCTE. Nous nous limitons ci−après au modèle ERA défini par la norme PCTE / PCTE+ que nous jugeons représentatif.

PCTE fournit un grand nombre de fonctions pour la définition et la gestion des données : persistance, distribution, gestion d'accès concurrents, mécanismes de tolérance aux pannes, gestion de versions.

La définition des données repose sur le concept d'*entité*. Une entité peut posséder des attributs et un contenu non interprétable par PCTE (de type *fichier*, représenté sous la forme d'une suite de caractères). Les attributs peuvent être origine ou destination d'un ensemble de relations ; une relation lie deux entités et peut posséder des attributs.

Avec ce modèle, les données sont représentées par des entités et les associations entre données de valeurs interdépendantes par des relations. Le modèle étant actif, la cohérence de ces dernières peut être assurée par le SGO.

Comme pour les bases relationnelles, l'évolution de la définition des données est limitée à une évolution additive.

Un mécanisme de vues permet de définir des schémas spécifiques aux différents clients. Les vues ne diffèrent néanmoins que par leur structuration ; une donnée aura le même type (entier par exemple) dans chaque vue dont elle est un composant. Outre le concept de vue, l'interface d'accès à une entité PCTE est équivalente à l'interface d'accès à un fichier. Ceci accroît l'ouverture supportée.

Bien que l'adéquation de ce modèle avec l'intégration des données semble supérieure à celle du modèle relationnel, la représentation et la gestion des données à un niveau de grain fin n'est pas procurée de manière réellement satisfaisante [45] [94] . Comment par exemple définir des relations entre des données de grain *inférieur* à l'*entité* ? Ces relations sont dans ce cas exprimées au travers des attributs des entités englobantes. Ceci complique d'une part la tâche de définition des données et renvoie d'autre part la tâche de maintien de la cohérence des données intra−entité aux outils et développeurs.

I.2.3.4 Approches basées sur un modèle orienté−objet

Les modèles orientés objet tels que les bases de données orientées objet (citons Gemstone [80], O2 [7]) et en particulier ceux prévus pour gérer les objets d'un environnement de développement (ATIS [12], OOTIS [59], OROS [102]) sont prometteurs car le concept d'objet autorise une granularité variable. A la différence du concept d'entité de PCTE, l'objet intègre attributs et contenu, le contenu étant ici *structuré et interprétable* par le SGO.

Les mécanismes fournis et les propriétés assurées varient selon les modèles orientés objet et leurs mises en œuvre. Il est encore difficile de trouver l'ensemble des fonctions appréciables pour l'intégration des données au sein d'un même modèle : persistance, typage, héritage, vues, déclencheurs, distribution, synchronisation, mécanismes de tolérance aux pannes, sécurité, etc. L'élaboration d'un tel modèle est néanmoins d'actualité.

Il y a deux différences essentielles avec les modèles ERA. La première est la possibilité de représenter des associations entre des données de grain fin (intra−entité) et de faire gérer la cohérence de ces données par le SGO (au travers de déclencheurs). L'autre est d'encapsuler des méthodes de manipulation des données avec la définition de celles−ci. Ceci autorise une certaine souplesse pour, par exemple, définir le code des déclencheurs.

I.2.4 Discussion

Deux aspects semblent bien refléter les préoccupations actuelles quant à l'intégration des données : le pouvoir d'expression et l'ouverture supportée par le modèle de données [34][94].

Le pouvoir d'expression est d'autant plus fort que le modèle de données autorise la définition d'objets de grain gros *et* fin, de manière uniforme. Les modèles orientés objets sont à cet égard très attractifs. En outre, le modèle de données doit permettre de définir des caractéristiques sémantiques (expression de propriétés, relations, contraintes, etc), prises en charge au travers de déclencheurs (modèle actif).

En contre−partie d'une modélisation fine des données, le besoin de disposer d'objets composés s'accentue et il faut permettre à un client de manipuler un objet composé de manière unitaire. Par ailleurs la composition ne doit pas être nécessairement statique (c.a.d. la même pour toute opération). Ceci implique de fournir un mécanisme de vue puissant, compliquant fortement la gestion des objets composés et en particulier des contrôles d'accès.

Face au problème de l'ouverture, un débat semble s'ouvrir sur le choix d'une architecture mono ou multi modèles [34][94].

Une architecture multi modèles permet de disposer de modèles de donnée plus spécialisés. L'ouverture est accrue car pour intégrer un nouvel outil, il suffit que la représentation des données qu'il utilise soit exprimable dans l'un des modèles supportés par l'architecture. Par contre, il est difficile d'exprimer des relations entre les données de modèles différents, et donc de maintenir la cohérence de celles−ci.

L'architecture mono−modèle semble rester la plus désirable [94]. L'ouverture repose sur la définition de "traductions" entre la représentation des données utilisées par un client, et la représentation de celles−ci au niveau du SGO. Ces traductions peuvent être exprimées sous la forme de vues. L'ouverture fournie est néanmoins limitée car l'expression des traductions n'est pas toujours possible : aucun modèle de définition canonique n'a encore été établi. Aussi, la définition d'un modèle à la fois générique et expressif constitue un axe de recherche très actif qui rallie par ailleurs le domaine des environnements de développement au domaine de l'interopérabilité [122] (cf. II.4).

I.3 Intégration du contrôle

Cette section introduit l'intégration du contrôle de manière informelle en décrivant son rôle et en évoquant l'existant. Une description plus approfondie de la problématique, s'appuyant sur la présente introduction, fait l'objet du chapitre II.

Les sections suivantes présentent successivement les objectifs attendus de cet axe d'intégration (I.3.1), les principaux problèmes (I.3.2), et les approches existantes (I.3.3). Une discussion termine cette section en I.3.4.

I.3.1 Objectifs

L'objectif de l'intégration du contrôle est de permettre aux outils de l'environnement de communiquer entre eux et de faire varier leur comportement en fonction des informations recueillies au travers de ces communications [120].

De manière générale, deux principaux types d'échanges entre les outils sont requis :

- la notification d'information,
- la demande de service.

La notification d'information permet à un outil d'émettre une information vers d'autres outils. La demande de service permet à un outil d'utiliser un service offert par un autre outil.

A la différence de la notification d'information qui est une communication de type *un−à−plusieurs*, la demande de service est généralement une communication de type *un−à−un*.

Dans la suite, nous qualifions ces échanges de *coordinations*. Nous employons les termes *notification* et *requête* pour désigner respectivement les échanges de type notification d'information et demande de service. Enfin, nous qualifions de *mécanisme de coordination* le mécanisme qui permet de définir et de supporter les coordinations.

Les coordinations permettent d'accroître la productivité des développeurs en fournissant à ces derniers un environnement uniforme, cohérent et mieux contrôlé.

1. Un environnement uniforme :

   (a) Il est courant que différents outils aient besoin de services similaires. Par exemple, un service de visualisation d'un programme peut être requis par un metteur au point et par un gestionnaire de références croisées. Si ceux−ci partagent ce service de visualisation, alors les développeurs auront une vision plus uniforme de l'environnement car, à but identique, l'interface homme−machine du service réalisant le but sera unique.

   (b) Une même donnée est couramment présentée graphiquement aux développeurs au travers de différents outils. Par exemple, un programme peut être présenté sous forme iconique par un "browser" et sous forme textuelle par un éditeur. Si le développeur peut demander le service "mise au point" sur ce programme aussi bien depuis le "browser" que depuis l'éditeur, alors l'uniformité de l'environnement est accrue.

2. Un environnement cohérent :

   (a) Les services fournis aux développeurs ne peuvent être invoqués à n'importe quel instant et dans n'importe quel ordre sans entraîner un risque d'incohérence. Si les outils peuvent connaître les actions qui sont (ou ont été) réalisées par d'autres outils, alors ils peuvent décider si l'activation d'un service est correcte. Dans un environnement comprenant de multiples développeurs activant un grand nombre de services, la possibilité de libérer ceux−ci de la gestion de la cohérence de leurs actions est très attractive.

   (b) D'autres types de décisions peuvent être prises, toujours à des fins de cohérence. Par exemple, lorsqu'un outil qui contrôle les services exécutés dans une session est fermé par un développeur, alors il demande à tous les autres services actifs de la session de se terminer.

3. Un environnement mieux contrôlé :

   Il est possible de construire des services spécifiques, chargés de contrôler ou de récolter des informations sur le travail des développeurs. Par exemple, le résultat d'un service de validation d'un composant peut être transmis à un outil spécifique, chargé d'informer un chef de projet de l'état d'avancement des travaux.

I.3.2 Problèmes

Le problème essentiel est de fournir un mécanisme de coordination combinant les propriétés suivantes.

1. Ouverture :

   Il doit être possible de coordonner des outils de provenance diverses, ayant été développés indépendamment les uns des autres.

2. Evolution des outils et des coordinations:

   L'ajout, le retrait ou la modification des outils et de leurs coordinations doivent pouvoir être réalisés facilement.

3. Flexibilité :

   L'évolution *dynamique* des outils actifs présents dans l'environnement et des coordinations prenant place entre ces outils doit être autorisée et gérée.

   Par exemple, à tout moment et au gré des développeurs, un nouvel outil doit pouvoir être activé et un outil actif désactivé. Par ailleurs, un outil actif doit pouvoir, à tout instant de son exécution, décider de ne plus participer aux mêmes coordinations [53].

4. Puissance :

   Le mécanisme de coordination est puissant s'il répond aux besoins des outils. Notamment, outre le fait de fournir des coordinations de type requête et notification et divers modes de communication, il doit permettre de désigner judicieusement les outils intervenant dans une coordination.

5. Performance :

   Ce critère n'est pas primordial, mais suffisamment important pour remettre en cause le choix de certaines solutions.

Notons que le critère de performance se rapporte à l'implémentation du mécanisme de coordination, à l'inverse des autres critères qui se rattachent essentiellement aux fonctions fournies par celui−ci.

I.3.3 Approches

Beaucoup de réalisations de mécanismes de coordination entre outils existent, dont la plupart constituent des produits industriels. Citons Field [99], SoftBench [25], ToolTalk [112], DEC FUSE EnCASE [40], Koala [11] , le mécanisme proposé dans le cadre du projet Eureka Software Factory ESF [47], PACT [35] et le mécanisme proposé par Clément dans [31].

Tous ces mécanismes sont fondés sur la notion de *bus de messages*, initialement proposée dans le projet Field et définie ci−après (I.3.3.1). Nous décrivons ensuite les particularités relatives à certains mécanismes.

Les implémentations de ces mécanismes ne sont pas rapportées car elles sont souvent similaires, ayant pour support le système Unix.

I.3.3.1 Qu'est ce qu'un Bus de messages ?

Un bus de messages [(6)]est un service de diffusion sélective de messages. Il route des messages entre un ensemble de clients qui peut varier dynamiquement.

Le routage repose sur la technique de filtrage (*pattern−matching*), appliquée entre le contenu du message à transmettre et des formats précisés par les clients potentiellement récepteurs.

Les formats peuvent évoluer dynamiquement : un client peut, à tout instant de son exécution, modifier les formats qui définissent les messages qu'il accepte.

I.3.3.2 Field

Dans le projet Field [98][99] [100] développé à l'Université Brown vers 1987, le bus de messages gère deux types de messages : les requêtes et les notification, telles que définies en I.3.1.

Les messages sont constitués d'une chaîne de caractères de longueur arbitraire, respectant un certain format :

Requête : nom/classe du récepteur, nom de la commande, arguments.

Notification : nom de l'émetteur, nom de l'information, arguments.

Deux modes de communication sont fournis : le mode bloquant (synchrone) et le mode non bloquant (asynchrone). Le premier est utilisé pour les requêtes, le deuxième pour les notifications.

---

(6) Un bus de messages est également appelé *Bus logiciel* ou encore *Bus applicatif*.

I.3.3.3 Softbench

SoftBench [25][64] est un produit développé et industrialisé par Hewlett−Packard comprenant : un mécanisme de coordination entre outils, un support pour la répartition, un gestionnaire d'interface homme−machine et un ensemble d'outils de développement constituant un environnement minimal.

Les particularités relatives au bus de messages de SoftBench (appelé BMS (Broadcast Message Server)) sont les suivantes.

- Trois types de messages sont fournis : requête, notification de succès et notification d'erreur. Quel que soit le type de message, le format est identique et comprend l'identité de l'émetteur, un identifiant unique de message, le type du message, la classe du récepteur, le nom de la commande ou de l'information, le contexte de l'émetteur (données sur lesquelles il travaille) et les arguments du message.
- Un message de type requête peut, si aucun outil actif n'est disposé à réaliser le service demandé au moment de la demande, entraîner l'activation automatique d'un outil adéquat chargé de réaliser ce service.
- Tout outil est associé à une classe d'outils qui définit une interface minimum en termes des services offerts. Par exemple, l'interface définie par la classe metteur au point comprend les services *poser−point−d'arrêt*, *retirer−point−d'arrêt*, *continuer*, etc.
- Un "encapsulateur" permet d'intégrer dans l'environnement des outils non prévus pour utiliser l'interface fournie par SoftBench et dont on dispose uniquement de l'exécutable. Cet encapsulateur lie émissions et réceptions de messages aux entrées/sorties de l'outil.
- SoftBench permet à un outil de spécifier des associations *(nom−message, callback)*, où *callback* désigne la procédure à exécuter lorsqu'un message de nom *nom−message* est reçu. Une primitive qui attend et traite les messages reçus, et selon les besoins de l'outil, traite en parallèle l'arrivée des événements X/Motif est également fournie.

I.3.3.4 Koala

Le Koala bus [11] a été développé au sein du projet Koala, à Sophia Antipolis. L'une de ses particularités est qu'il permet à l'utilisateur d'étendre les fonctions de communication fournies par le bus de messages, grâce à une partie interprétée écrite en langage Lisp.

Les points essentiels du Koala bus sont les suivants.

- Il permet d'émettre des messages de type requête, notification et réponse à une requête. Tout message émis peut l'être avec demande ou non de réponse, en mode synchrone ou asynchrone.

- Le routage des messages repose sur l'identité des clients (identité fournie par le Koala bus) ou sur leur appartenance à des groupes. Tout client peut envoyer un message vers un ou plusieurs groupes, créer un ou plusieurs groupes et s'abonner à un ou plusieurs groupes en tant qu'émetteur et/ou récepteur. L'abonnement en tant que récepteur précise si le client agit comme gestionnaire du groupe (c'est à dire qu'il est capable de servir les requêtes qui seront adressées à ce groupe) et/ou observateur (c'est dire qu'il recevra les messages, mais à titre d'information seulement). Les groupes sont désignés par des expressions régulières.
- Un client peut recevoir des messages d'information provenant du bus et concernant les activités des autres clients : abonnement à un groupe, désabonnement à un groupe, etc.
- De même que SoftBench, le Koala bus permet à un outil de spécifier des associations *(nom_message, callback)*, où *callback* désigne la procédure à exécuter lorsqu'un message de nom *nom_message* est reçu.

I.3.3.5 DEC FUSE EnCASE

DEC FUSE EnCASE [40] est un produit industrialisé par la société DEC, fournissant un mécanisme de coordination entre outils et un ensemble d'outils de développement constituant un environnement minimal.

Les particularités relatives à DEC FUSE EnCASE sont les suivantes.

- L'expression des coordinations d'un outil est déclarative et séparée du code de l'outil. Un langage spécifique, nommé Tool Integration Langage (TIL), permet de spécifier quels messages sont transmissibles et recevables par un outil. Des primitives fournies par une bibliothèque permettent, depuis le code d'un outil, d'émettre ou de recevoir un message dont les caractéristiques auront été spécifiées en TIL.
- Pour exprimer l'évolution dynamique des coordinations d'un outil, la définition TIL d'un outil est découpée en états de coordination. Chaque état spécifie l'ensemble des messages qui sont transmissibles ou recevables par l'outil lorsque, lors de son exécution, il se trouve dans cet état. Le changement d'état est exprimé dans le code d'un outil.

I.3.4 Discussion

La majorité des mécanismes d'intégration du contrôle se fondent sur la solution "bus de messages" car son mécanisme de liaison dynamique est avantageux pour respecter les critères d'évolution et de flexibilité.

Ainsi, l'ajout d'un outil qui veut recevoir une notification ou une requête donnée n'implique pas de modifier les autres outils. A l'égard de la flexibilité, le choix du récepteur d'une coordination étant décidé dynamiquement, un outil peut à tout moment de son exécution décider de ne plus participer aux mêmes coordinations. De même, des outils peuvent être activés et désactivés à tout instant.

Actuellement, les solutions proposées ou recherchées tendent à augmenter la solution "bus de messages" initialement proposée dans Field pour mieux satisfaire les critères de qualité requis.

Pour faciliter la compréhension des coordinations déjà existantes lors de l'ajout, du retrait ou de la modification d'un outil, on cherche à séparer la définition des coordinations du code des outils. C'est la direction prise par DEC FUSE EnCase, PACT et Clément dans [31].

Pour accroître l'ouverture, SoftBench permet de coordonner de manière minimale un outil dont le code n'est pas modifiable, en liant ses entrées/sorties à un outil spécial appelé *encapsulateur*.

Par ailleurs, des solutions qui permettent aux outils d'échanger des messages structurés même si ceux−ci sont écrits dans des langages de programmation différents (et sont donc hétérogènes du point de vue de leur modèle de données) commencent à être considérées [35].

Pour augmenter la puissance, on cherche à fournir des moyens de désignation des outils qui permettent de sélectionner les émetteurs et les récepteurs de coordinations en fonction de diverses caractéristiques (leur fonction dans SoftBench et ToolTalk, leur appartenance à des groupes dans Koala). Le besoin d'utiliser d'autres caractéristiques, telle que la notion d'utilisateur (développeur dans notre cas), commence à apparaître [4].

Il faut noter que la solution "bus de messages" fournit un niveau d'intégration qualifié de *faible (ou léger)* [4], à l'opposé d'autres techniques d'intégration qui procurent un niveau d'intégration qualifié de *fort*, couramment basées sur une liaison statique des composants coordonnés. L'avantage principal de ces dernières est la rapidité d'exécution offerte. Leur inconvénient est qu'elles ôtent toute autonomie aux composants. Elles sont généralement utilisées pour coordonner des composants de grain fin (fonction, procédure, etc.), à la différence des techniques d'intégration faible qui sont utilisées pour coordonner des composants de gros grain : les outils représentent des applications [58].

La recherche de solutions permettant de fournir un mécanisme homogène qui autorise à la fois une intégration forte et faible est d'actualité [31][60]. Nous pensons toutefois que les problèmes et les solutions relatives à ces deux types d'intégration sont différents car chacun est soumis à des contraintes spécifiques (performances d'exécution, autonomie des composants). Chaque type d'intégration représente un domaine de recherche encore ouvert. Dans ce document, nous nous concentrons sur les techniques d'intégration du contrôle qualifiées de faible.

I.4 Intégration du processus de développement

Cette section présente les principaux objectifs (I.4.1), problèmes (I.4.2) et approches (I.4.3) se rattachant à l'intégration du processus de développement. Une courte discussion est fournie en guise de conclusion en I.4.4.

Le domaine correspondant à cet axe étant très vaste, la présentation donnée est très succincte.

I.4.1 Objectifs

L'objectif de l'intégration du processus de développement est d'assister, guider et contrôler les développeurs dans leur tâche de production de logiciel, selon une méthodologie préalablement définie(7).

Assistance, guidage et contrôle reposent sur l'exploitation d'une description du processus de développement. Le facteur *généricité* de l'environnement (cf. I.1.3) veut que cette description soit paramétrable. Un langage de description du processus de développement, de même qu'un environnement d'exécution ou d'interprétation pour ce langage sont requis.

Différents aspects peuvent être exprimés au travers d'un tel langage. Nous les regroupons en trois catégories partiellement dépendantes les unes des autres.

1. Description des tâches de développement :

   Le processus de développement est décomposé en tâches de grain plus ou moins fin (selon les modèles de description), liées par des flots de contrôle et de données. Certaines d'entre−elles peuvent être manuelles (faisant intervenir le développeur), d'autres automatiques.

2. Description d'aspects liés à la gestion de projet :

   Définition des développeurs, rôles, dates, coûts, disponibilité, etc. ainsi que des relations entre ces concepts, et entre ces concepts et les tâches de développement.

3. Description d'aspects liés à la coopération entre développeurs :

   Description des notifications émises aux développeurs afin de les tenir informés des actions des autres, description des protocoles de partage des ressources, etc.

---

(7) Il faut différencier ces environnements de ceux qualifiés d'Environnements de gestion du développement dont l'objectif est d'offrir des fonctions de gestion de l'état d'avancement du développement au travers d'outils spécifiques (agendas, etc), mais de manière indépendante de l'environnement de développement. Aucune liaison automatique n'est faite avec l'environnement de développement.

I.4.2 Problèmes

L'intégration du processus de développement se heurte à un problème majeur : la difficulté de compréhension du processus de développement et par suite d'expression de ce dernier.

Il s'agit en effet de décrire le comportement des agents intervenant dans ce processus (données, services, activités, développeurs, équipes, etc). Le nombre d'agents peut être grand et ceux−ci peuvent être très divers. De plus, les aspects à décrire peuvent différer selon que l'on se place au niveau de l'individu, du groupe, ou de l'entreprise [69] (cf. I.1).

Même à une échelle réduite, la description des tâches de développement est difficile. Le descripteur peut choisir une approche "impérative", dans laquelle il va décrire le graphe d'ordonnancement des tâches autorisées. Il est alors contraint de s'arrêter à un grain de tâche assez gros s'il ne veut pas ôter toute flexibilité aux développeurs. S'il adopte une approche "événementielle" dans laquelle il décrit les situations ou les états non autorisés, ou encore des enchaînements d'opérations partiels, alors grain fin et flexibilité peuvent être plus facilement offerts mais il est plus difficile d'avoir une vision globale du processus de développement.

Comprendre et spécifier des processus de développement de logiciel reste un domaine de recherche actif [92]. Certains auteurs remettent même en cause le bien−fondé des environnements dirigés par le processus de développement, faute d'être capable de décrire ce dernier [86]. D'autres considèrent que si seule une compréhension partielle du processus de développement peut être exigée, alors c'est au mécanisme d'intégration du processus de développement de s'adapter. Ainsi les contraintes qui semblent majeures sont les suivantes :

- le modèle de description doit autoriser une description *partielle* du processus de développement,
- l'affinement et la modification d'une description doivent pouvoir être réalisés en cours de développement,
- les situations "exceptionnelles", dans lesquelles un ou plusieurs développeurs doivent réaliser des actions contradictoires avec la description du processus de développement doivent être gérées.

I.4.3 Approches

Nous nous limitons à présenter les approches les plus courantes quant aux modèles de description du processus de développement. Le débat qui semble rester le plus ardu concerne en effet le choix de ce modèle.

I.4.3.1 Critères de comparaison

Nous caractérisons différents modèles de description du processus de développement selon deux aspects.

Premièrement, quels paradigmes sont utilisés par le modèle : procédural logique, fonctionnel, règles de production, plans d'action, objet, réseaux de Pétri ? Nous caractérisons ceux−ci par la nature de l'approche qu'ils fournissent pour décrire les tâches et leurs ordonnancements : impérative ou déclarative.

Deuxièmement, le modèle permet−il de décrire :

1. **Le comportement des données**(8). Par exemple, le fait que toute *modification* d'un programme source est suivie par la compilation de ce programme source.
2. **Le comportement des services.** Par exemple, le fait que toute *compilation* d'un code source est suivie par la mise au point du code si aucune erreur de compilation ne s'est produite.
3. **Le comportement des activités**, le concept d'activité étant fourni *explicitement* par le modèle. Par exemple, le fait que toute activité de codage est suivie d'une activité de validation, quels que soient les services qui composent l'activité de codage. A ce niveau, se pose la question de savoir comment est exprimé le lien entre activités et services.

I.4.3.2 Modèles de description du processus de développement

MARVEL [9][10] est un exemple d'approche déclarative, fondée sur des règles de production avec chaînage avant et arrière. Les enchaînements sont décrits sous la forme de *précondition, action, effets*, la précondition pouvant porter sur les services (sur le résultat d'un service par exemple) ou sur les données (sur la modification de la valeur d'une donnée par exemple). Les règles sont de diverses catégories qui expriment chacune une certaine sémantique. MARVEL exploite cette sémantique pour augmenter la coopération entre les développeurs, en autorisant un plus haut degré de partage des données.

Dans ARCADIA [71][114], une approche déclarative est également fournie au travers du langage APPL/A qui permet de spécifier des enchaînements de services au travers de règles de type ECA(9). La partie événement porte sur les modifications des données. APPL/A, qui est un sur−ensemble du langage ADA, permet également de décrire des enchaînements d'activités sous la forme d'un programme ADA.

Le projet SPADE [8] adopte une approche impérative basée sur les réseaux de Pétri de haut niveau (10). Un langage (nommé SLANG) permet de spécifier comment les services sont enchaînés et quelles sont les données communiquées entre les

---

(8) Données, services et activités sont utilisés dans le sens défini dans le chapitre I.

(9) Une règle ECA comprend trois parties : Evénement, Condition, Action. L'événement correspond à l'activateur de la règle. La condition se place ensuite comme un filtre préalable à l'exécution de l'action.

(10) Les réseaux de Pétri de haut niveau sont des réseaux de Pétri simples auxquels on a ajouté certaines extensions. Par exemple, les transitions peuvent avoir des priorités, des prédicats.

services. Les transitions du réseau de Pétri permettent d'activer des services automatiquement ou manuellement et le marquage des places s'effectue en fonction des résultats des services activés. L'état d'avancement du processus de développement est donné par ce marquage. Le langage permet aussi d'exprimer certains aspects de gestion de projet, en associant des contraintes de temps aux transitions.

Dans MELMAC [38], les réseaux de Pétri sont également utilisés mais ils servent à décrire l'enchaînement des activités et non pas l'enchaînement des services. La description d'une activité en termes des services qui la composent est réalisée sous la forme d'un programme.

Enfin, on trouve des approches multi−paradigmes, combinant approche déclarative et impérative [24][63]. Par exemple dans [63], les réseaux de Pétri de haut niveau sont utilisés pour décrire les enchaînements d'activité, le détail des activités étant spécifié au niveau des transitions en termes de services et de règles de production. Ces règles ne sont pas globales comme dans MARVEL car leur activation est attachée aux transitions.

Dans [24], les réseaux de Pétri sont également utilisés pour décrire les enchaînements d'activités. Le lien entre une activité et les services est exprimé par le concept d'espace de travail (WorkContext) dans lequel sont spécifiés quels sont les services que le développeur, en fonction de son rôle, est autorisé à activer et sur quelles données.

I.4.4 Discussion

Initialement, les approches impératives ont été opposées aux approches déclaratives. Actuellement, la combinaison des deux approches paraît prometteuse et est expérimentée dans divers projets [56][63].

Certaines descriptions (telles que des automatisations de services, du style *compile −> debug*) sont plus faciles à spécifier par une approche déclarative. Une approche impérative impliquerait en effet de spécifier cet enchaînement de services à chaque endroit de la description où le service *compile* est référencé.

Par contre l'approche déclarative ne fournit pas la notion d'état courant du processus de développement comme le fournit l'approche impérative. Cette notion est intéressante car elle permet de spécifier des actions relatives à l'état d'avancement (par exemple, lorsque l'activité de codage *Ai* est terminée, le chef de projet doit en être informé).

Par ailleurs, une approche déclarative ou impérative "plate" (sans notion de hiérarchie) rend très difficile l'expression du processus de développement dans lequel la notion de décomposition est intrinsèque.

Les approches combinées qui permettent de décrire le comportement de tâches de grain plus ou moins fin et selon des paradigmes différents (règles, procédures) nous semblent les plus intéressantes. Par exemple, dans [63] l'approche impérative est utilisée pour décrire le comportement des activités et l'approche déclarative pour décrire le

comportement des services. Notons que la possibilité de décrire le comportement de tâches de grain important (notamment les *activités*) facilite en outre la description de certains aspects relatifs à la gestion de projet (affectation des développeurs aux activités par exemple). Les inconvénients des approches "combinées" restent leur complexité de vérification et d'interprétation.

I.5 Discussion générale

La présentation des trois axes d'intégration (données, contrôle et processus de développement) effectuée dans ce chapitre permet de cibler le rôle de l'intégration du contrôle dans un environnement de développement intégré.

En guide de conclusion, nous donnons notre point de vue sur les liens qui existent entre ces trois axes d'intégration et sur la pertinence de disposer de chacun d'entre eux au sein d'un même environnement.

Intégration des données et du contrôle

Outre la gestion du stockage des données, l'intégration des données vise à permettre aux outils de partager des données, alors que l'intégration du contrôle a pour objet de permettre aux outils de partager des informations de coordination. Les outils ont−ils besoin des deux formes d'intégration ?

Certains ont opté pour un environnement uniquement basé sur l'intégration des données, au travers d'une base de données commune. C'est le cas de l'AD/Cycle d'IBM. D'autres ont choisi les fonctions offertes par l'intégration du contrôle comme seul moyen de communication entre outils, ces derniers disposant de leur propre base de données [46]. La tendance actuelle semble toutefois préconiser une approche combinant intégration des données et du contrôle [87] [115] [116] [117].

Nous pensons également que les deux formes d'intégration sont nécessaires, car elles répondent à des besoins distincts. Le partage de données entre outils paraît indispensable dans un environnement de taille groupe, dans lequel les développeurs partagent fortement les données qu'ils produisent.

Le partage d'informations de contrôle vient en complément du partage de données, car il permet aux outils d'agir de manière cohérente même si leurs actions n'ont aucun effet sur les données. C'est par exemple le cas pour la terminaison d'une session de travail : la fermeture du gestionnaire de la session provoque la fermeture de tous les outils qui s'exécutent dans la session.

En résumé, l'intégration des données permet d'assurer une cohérence donnée−donnée, alors que l'intégration du contrôle permet d'assurer une cohérence outil−outil.

Intégration du processus de développement

La différence essentielle entre les fonctions fournies par l'intégration du processus de développement et celles fournies par l'intégration des données et du contrôle est que la première, de notre point de vue, doit manipuler des concepts tels que les rôles, les échéances, les activités de développement.

Le concept d'activité de développement n'est néanmoins pas indépendant des données et des services. Au niveau le plus bas, il exprime une série d'actions portant sur les données et les services (déclenchements automatiques de services suivant l'état des données ou d'après les services précédemment activés).

Nous pensons que cette dépendance ne remet pas en question l'utilité des fonctions fournies par les deux autres axes d'intégration. En effet, on veut pouvoir exprimer des automatisations indépendantes de tout processus de développement, permettant de maintenir la cohérence des données ou de coordonner des services.

Aussi, l'intégration du processus doit permettre d'exprimer les liens entre les activités et les données et services *en s'appuyant* sur les fonctions fournies par ces deux autres axes (et non en fournissant ces fonctions).

Un problème subsiste néanmoins : celui de fournir un moyen qui permette de distinguer, au niveau des données et des services, l'expression des automatisations qui relèvent de propriétés intrinsèques aux données et aux services, de celles qui relèvent du processus de développement.

Chapitre II

Intégration du contrôle : position du problème

L'objectif de l'intégration du contrôle est de permettre aux outils d'échanger des informations, de manière à ce que les actions des uns entraînent des réactions de la part des autres. Nous qualifions ces échanges de coordinations.

Ce chapitre se concentre sur les techniques d'intégration qualifiées de *légères* [4], dans lesquelles les outils à coordonner sont de *gros grain* (i.e représentent des applications autonomes).

Après avoir défini une coordination (II.1), les caractéristiques qui engendrent les problèmes majeurs sont énoncées (II.2). Les critères de qualité d'un mécanisme d'intégration du contrôle sont ensuite déterminés et permettent de comparer différentes solutions existantes.

II.1 Définition d'une coordination

Précisons tout d'abord la terminologie employée :

- un *agent* dénote un programme exécutable ; un *agent actif* est un agent en cours d'exécution,
- un *événement* (ci−après noté A.e) dénote l'atteinte d'un état donné (e) par un agent actif donné (A),
- une *réaction* (ci−après notée B.r) est une méthode donnée (r) qui peut être exécutée par un agent actif donné (B).

Définition

Une coordination est une association entre un événement A.e et un ensemble de réactions $B_1.r_1, .., B_n.r_n$, dont certaines sont à exécuter lorsque A.e se produit.

À cette association est attaché un type (t) qui définit le choix des réactions à exécuter, et un mode (m) qui définit le mode de communication liant l'agent A aux agents qui exécutent les réactions.

Nous notons :

$$A.e \xrightarrow{t,m} (B_1.r_1, .. , B_n.r_n)$$

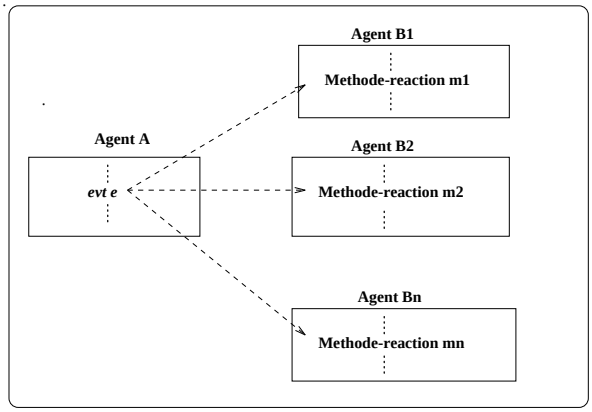


Fig. 2.1 : Illustration d'une coordination

Les types de coordination courants sont appelés requête et notification. Une requête spécifie qu'une et une seule réaction doit être exécutée à la suite de la production d'un évenement, à la différence d'une notification qui peut entraîner l'exécution d'un nombre quelquonque de réactions(1).

Dans la suite, nous employons les termes d'*émetteur* pour l'agent qui produit l'événement et de *récepteur* pour les agents qui exécutent les réactions.

II.2 Caractérisation du contexte

Les problèmes posés découlent du *contexte* dans lequel le mécanisme de coordination s'inscrit. Ce contexte est un environnement de développement et les agents à coordonner sont des outils de développement. Les caractéristiques de ce contexte sont d'une part dimensionnelles (ou quantitatives), et d'autre part qualitatives.

Les caractéristiques dimensionnelles sont les suivantes.

1. **Nombre d'outils**

   Le nombre d'outils critique est le nombre moyen d'outils *actifs*. L'intégration du contrôle doit idéalement être considérée dans un environnement de taille *entreprise*. Nous choisissons néanmoins de nous placer au niveau inférieur *groupe* (cf. I), car les problèmes y sont déjà importants. A ce niveau, l'environnement peut supporter plusieurs dizaines d'utilisateurs qui travaillent en parallèle. Le nombre moyen d'outils actifs est donc important car un utilisateur utilise en moyenne une dizaine d'outils actifs.

2. **Fréquence des coordinations**

   Comme précédemment, l'intégration du contrôle devrait idéalement supporter plusieurs niveaux de fréquence (gros, moyen et fin), si les outils à coordonner peuvent

(1) Types et modes sont définis de manière précise en II.3.2.4.

varier d'un grain fin (procédure, méthode) à un grain beaucoup plus gros (application).

En nous plaçant dans un environnement uniquement composé d'outils de gros grain, l'intervalle entre des coordinations engendrées par *un* outil actif se situe entre la seconde et la minute (ceci correspond aux niveaux moyen/grand selon une échelle proposée à cet égard dans [4]).

L'impact des caractéristiques dimensionnelles porte essentiellement sur la mise en œuvre d'un mécanisme de coordination, à l'inverse de celui des des caractéristiques qualitatives, énoncées ci−après, qui porte essentiellement sur le modèle de coordination.

1. **L'évolution des composants de l'environnement**

   L'évolution des composants dénote :

   a) l'évolution des outils : ajout, retrait, remplacement d'outils dans l'environnement,

   b) l'évolution des coordinations : ajout, retrait, modification des coordinations entre les outils

2. **L'évolution des composants actifs de l'environnement**

   L'évolution des composants actifs dénote :

   a) l'évolution des outils actifs :

      des activations et désactivations d'outils peuvent prendre place à tout moment, au gré des développeurs,

   b) l'évolution des coordinations actives [(2)]:

      les coordinations actives sont celles qui sont susceptibles de se produire à l'instant courant. Elles évoluent non seulement par l'évolution des outils actifs, mais également par le fait que le comportement d'un outil actif vis−à−vis de ses coordinations est susceptible d'évoluer dynamiquement.

      Par exemple, les coordinations dans lesquelles un éditeur est susceptible d'intervenir durant son exécution peuvent changer lorsque celui−ci change de rôle (passe par exemple d'un rôle d'édition à un rôle de visualisation pour un metteur au point). Un compilateur peut, à certains moments, accepter de participer à des coordinations en offrant la méthode *compiler* comme réaction, et à d'autres moment ne pas accepter d'y participer en raison d'une surcharge de travail.

      Enfin, on peut vouloir permettre aux développeurs de configurer dynamiquement des automatisations du type *(quand un programme est sauvé dans la base, compiler le programme)*. Cette configuration dynamique entraîne une évolution dynamique des coordinations actives.

---

(2) Dans la suite, nous employons le terme *évolution dynamique des coordinations*.

3. **L'indépendance entre agents**

   Les outils récupérés sur le marché proviennent souvent de sources distinctes et ont généralement été conçus pour fonctionner de manière autonome vis−à−vis des autres outils. En conséquence :

   a) ils peuvent être écrits dans des langages différents et s'exécuter sur des plate−formes différentes,

   b) ils ne partagent pas forcément un vocabulaire commun au niveau des éléments intervenant dans une coordination. Par exemple une réaction de nom r1 pour un premier outil peut être sémantiquement équivalente à une réaction de nom r2 pour un autre outil.

4. **L'asynchronisme des coordinations**

   Les outils de développement sont pour la plupart écrits de manière événementielle : leur comportement de même que leur activation et désactivation sont dirigés par les actions de l'utilisateur. Les coordinations sont donc de nature asynchrone : les actions de l'utilisateur peuvent *à tout moment* provoquer des coordinations.

Notre problème est donc de fournir un moyen d'expression et un support d'exécution pour des coordinations asynchrones, qui prennent place entre des agents indépendants, dans un environnement à forte évolution des agents et des coordinations et à forte évolution des agents actifs et des coordinations actives.

Ceci implique de :

- concevoir un modèle de coordination qui tient compte des caractéristiques précédemment introduites ; cet aspect est traité en II.3,
- fournir un support (environnement d'exécution) pour le modèle de coordination conçu ; cet aspect n'est pas abordé dans ce chapitre(3),
- intégrer le modèle de coordination conçu et son support avec les différents modèles de programmation et supports des agents (Fig. 2.2) ; cet aspect est traité en II.4.

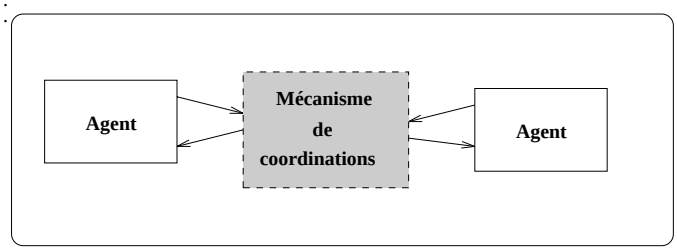


Fig. 2.2 : Un mécanisme de coordination permet à des agents hétérogènes d'interopérer selon des règles précises.

(3) Le lecteur pourra comprendre les principaux problèmes posés par cet aspect dans le chapitre V, qui présente l'environnement d'exécution du mécanisme de coordination que nous avons conçu.

II.3 Modèles de coordination

Après avoir défini la fonction générale d'un modèle de coordination (II.3.1), ses critères de qualité sont établis (II.3.2). Différents modèles sont ensuite comparés au regard de ces critères (II.3.3).

II.3.1 Fonction d'un modèle de coordination

Reprenant la définition d'une coordination énoncée en II.1, un modèle de coordination doit permettre de définir un environnement ENV : <E, R, T, M, C> avec :

E : un ensemble d'événements,

R : un ensemble des réactions,

T : un ensemble de types de coordination,

M : un ensemble de modes de communication,

C : un ensemble de coordinations liant E, R, T et M.

Son adéquation pour l'intégration du contrôle dépend :

- de la manière dont il permet d'exprimer :
  - les ensembles E et R,
  - l'ensemble C,
  - l'événement et les réactions intervenant dans une coordination,
- des types et des modes de coordination qu'il offre.

II.3.2 Critères requis pour l'intégration du contrôle

II.3.2.1 Expression des ensembles des événements (E) et des réactions (R)

Les critères requis résultent du besoin d'évolution des composants de l'environnement : il faut faciliter le travail de l'administrateur lorsqu'il doit intégrer, retirer ou remplacer un agent dans l'environnement.

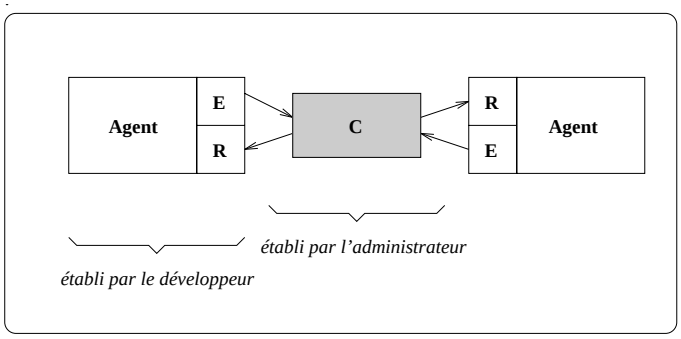
La personne qui développe un agent connaît les événements et les réactions susceptibles d'être utilisés pour coordonner l'agent qu'elle développe.

Il faut donc lui fournir des fonctions, insérables dans le code de l'agent, permettant d'émettre un événement ou d'exécuter une réaction. Il faut également lui fournir un moyen pour exprimer *explicitement* l'interface de coordination d'un agent (événements transmissibles et réactions exécutables)(4).

---

(4) Pour l'instant, nous disons que pour chaque agent, événements et réactions se décrivent par leur nom et par leurs paramètres formels.

Le développeur d'un agent ne connaît pas forcément les environnements dans lesquels son agent sera intégré. La tâche d'établissement des coordinations qui vont lier un agent à d'autres dans un environnement donné relève de l'administrateur de cet environnement (cf. figure ci−après). Si ce dernier ne dispose pas d'une description *explicite* des événements transmissibles et des réactions exécutables de l'agent, il doit les déterminer à partir de son code.



Spécification d'informations relatives aux coordinations.

II.3.2.2 Expression de l'ensemble des coordinations (C)

Les critères requis proviennent du besoin d'évolution des composants de l'environnement.

Le modèle doit permettre d'exprimer les coordinations de manière explicite, en dehors des programmes exécutables des agents. Ceci facilite la compréhension des coordinations et leur maintenance.

Il doit en outre permettre de modulariser l'expression des coordinations. Le grain d'évolution de l'environnement étant l'agent, nous pensons qu'une modularité prenant ce grain comme unité de décomposition et non la *coordination,* est adéquate.

Cette modularité doit être telle que, lors de l'ajout (resp. du retrait) d'un agent A dans l'environnement, seule l'expression des coordinations de A soit à définir ou à modifier pour coordonner (resp. "dé−coordonner") A avec les autres agents de l'environnement.

L'évolution des coordinations actives agit également au niveau *agent*. En particulier, on peut considérer qu'un agent, durant son exécution, passe par différents *états de coordination* car son comportement vis−à−vis des coordinations dans lesquelles il est susceptible d'intervenir peut varier dynamiquement. Le modèle doit permettre de décrire *explicitement* ces différents états.

II.3.2.3 Désignation des événements et réactions intervenant dans les coordinations

La désignation d'un événement (A.e) ou d'une réaction (A.m) comprend deux parties : l'une qui désigne l'agent actif intervenant (A) et l'autre qui désigne l'événement "local" (e) ou la réaction "locale" (m) de cet agent.

Les critères requis pour désigner les événements et les réactions intervenant dans les coordinations concernent :

1. **Les moyens qui permettent de désigner les agents actifs**

   Afin de mieux comprendre nos préoccupations, voici une situation concrète, reflet du contexte dans lequel nous nous plaçons.

   Prenons une entreprise au sein de laquelle différents employés exercent différentes fonctions et ont besoin de se coordonner. Un employé peut consulter son supérieur hiérarchique pour obtenir une information qui n'est pas de son ressort. Un directeur peut demander à sa secrétaire de réaliser un certain travail, par exemple de taper une lettre.

   Dans ces deux cas, un employé désigne un travail (taper une lettre) et un autre employé ("ma" secrétaire) pour réaliser ce travail.

   Supposons que cette entreprise soit soumise à une très forte évolution des personnes employées. Un employé a deux solutions pour désigner un autre employé. Soit il spécifie son identité (nom), soit il spécifie sa fonction et sa position dans l'organisation. Dans ce deuxième cas, la désignation d'un employé s'appuie sur la désignation de ses caractéristiques (fonction, position) au sein de l'organisation. Alors que les identités des employés évoluent fortement, leurs caractéristiques restent identiques.

   Un environnement de développement comporte de fortes similarités avec la situation précédemment énoncée.

   Premièrement, il existe une organisation des agents actifs en termes de différentes caractéristiques comme par exemple leur fonction (éditeur, metteur au point, etc.), leur localisation géographique (par exemple la session dans laquelle ils travaillent), leur contexte de travail (notamment les données sur lesquelles ils travaillent), etc.

   Deuxièmement, les agents actifs ont besoin de se désigner au travers de ces caractéristiques. Par exemple, un outil qui demande un service peut spécifier que l'outil qui va exécuter ce service doit être placé dans la même session que lui (parce que ceux−ci partagent un contexte de travail nécessaire à la bonne réalisation du service). Un outil peut également spécifier qu'il n'accepte de réagir à un événement donné que s'il provient d'un outil travaillant dans la même session que lui et ayant la fonction de metteur au point.

Troisièmement, les agents actifs sont soumis à une très forte évolution (activations et désactivations d'outils).

La désignation des agents actifs ne peut uniquement s'appuyer sur leur identité car cela impliquerait, pour toute coordination, de gérer leur présence et absence.

Il est donc judicieux de permettre de désigner les agents au travers de leurs caractéristiques (position, fonction).

2. **Les moyens qui permettent de désigner un événement local ou une réaction locale**

Les critères requis proviennent de l'indépendance entre agents, de l'évolution des composants et de l'évolution des composants actifs.

Les agents ayant été développés indépendamment les uns des autres, ils peuvent utiliser des noms d'événements (resp. de réactions) différents bien que sémantiquement équivalents. L'équivalence sémantique signifie ici que l'un peut remplacer l'autre une coordination.

Si deux agents fournissent des événements ou des réactions sémantiquement équivalents, on doit pouvoir remplacer l'un par l'autre sans modifier les agents de l'environnement ni l'espression de leurs coordinations. De même, dynamiquement, un agent actif doit pouvoir remplacer un autre agent actif sémantiquement équivalent.

De ce fait, il est utile que le modèle permette de désigner les événements et les réactions intervenant dans une coordination sans avoir à connaître leurs définitions locales (noms et paramètres), par exemple par des définitions globales qui représentent leur sémantique.

II.3.2.4 Ensembles des types (T) et des modes (M) de coordination

1. **Ensemble T**

Les besoins des outils vis à vis des types de coordination sont assez bien ciblés : les deux types définis ci−après sont requis.

Etant donné une coordination liant un événement donné à des réactions *r1,..,rn* :

- t = *requête* signifie que une et une seule (au hasard) des réactions dénotées par *r1, .. , rn* doit être exécutée lorsque l'événement *e* se produit,

- t = *notification* signifie que l'ensemble des réactions dénotées par *r1,..,rn* doivent être exécutées lorsque l'événement *e* se produit.

Notons que le type requête est particulièrement adapté lorsque les moyens de désignation remplissent les critères énoncés précédemment (cf II.3.2.3) : une désignation peut en effet sélectionner plusieurs réactions.

2. **Ensemble M**

Trois modes de communication nous semblent pertinents : *synchrone, asynchrone, et asynchrone avec accusé de réaction*(5).

Le mode synchrone permet à l'agent qui émet un événement d'attendre que les exécutions des réactions associées se terminent : cet agent peut ainsi recevoir des résultats des exécutions des réactions(6).

Avec le mode asynchrone, l'agent qui émet un événement ne dispose d'aucune information sur les exécutions des réactions.

Enfin, le mode asynchrone avec accusé de réaction permet à l'agent qui émet un événement d'attendre que les exécutions des réactions commencent, sans en attendre les terminaisons.

Alors que l'utilité des deux premiers modes est universellement reconnue, la pertinence du troisième constitue un avis personnel. Il nous paraît intéressant de permettre à un agent d'être sûr que l'événement qu'il émet est bien suivi de l'exécution d'une ou plusieurs (selon le type de la coordination) réactions. C'est notamment le cas lorsqu'un agent demande un service sans avoir besoin d'attendre la terminaison de l'exécution de ce service.

3. **La prise en compte du temps**

Une coordination, lorsqu'elle est instanciée, entraîne une communication entre des agents. Cette communication définit une relation de causalité [97] entre l'événement déclencheur (soit *e*) et le lancement de l'exécution des réactions entrant en jeu dans la coordination (soit *r1,..,rn*). Nous notons *e −> (r1,..,rn)*.

Par ailleurs, il faut aussi supposer que lorsque, dans un même agent, un événement *e* se produit après (relativement au temps) le lancement de l'exécution d'une réaction *r*, alors il peut exister une relation de causalité *(r −> e)* liant la production de l'événement *e* au lancement de l'exécution de la réaction *r*.

---

(5) Ce terme, par opposition au terme classique d'accusé de réception, permet de savoir si la notification ou requête a non seulement été reçue, mais également *acceptée* par un ou plusieurs agents.

(6) Dans le cas d'une notification, ce mode bloque l'émetteur jusqu'à ce que tous les outils recevant la notification (ceux−ci étant déterminés dynamiquement) aient finis d'exécuter leur réaction.

Par déduction, ces relations définissent un ordre causal et partiel sur le déclenchement des réactions. Nous pensons que cet ordre doit être respecté par le modèle de coordination, c'est à dire qu'il doit être assuré que la situation *(r2 −> r1)* ne peut se produire dès lors qu'une relation de causalité établit *(r1 −> r2)*.

Des outils chargés de transmettre des messages entre les développeurs ne peuvent en effet faire parvenir à un développeur *D* un message *m2* envoyé par *D2*, puis un message *m1* envoyé par *D1* si l'envoi de *m2* par *D2* a été causé par la réception de *m1* par *D2*. Pour ces outils, la réception d'un message est ici un *événement*, dont la *réaction* exécutée est l'affichage du message reçu au développeur.

II.3.2.5 Récapitulatif

En résumé, le modèle de coordination doit :

1. permettre d'exprimer les deux composants suivant de manière modulaire et en dehors du code des agents (cf. II.3.2.2) :
   - les événements susceptibles d'être émis et les réactions susceptibles d'être exécutées par les agents,
   - les coordinations,
2. permettre d'exprimer l'évolution dynamique des coordination explicitement et en dehors du code des agents (cf. II.3.2.2),
3. permettre de désigner les agents intervenant dans une coordination par leurs caractéristiques (cf. II.3.2.3),
4. permettre de désigner les événements et réactions intervenant dans une coordination par des définitions globales (cf. II.3.2.3),
5. fournir deux types de coordinations : les requêtes et les notifications (cf. II.3.2.4),
6. fournir trois modes de communication : synchrone, asynchrone et asynchrone avec accusé de réaction (cf. II.3.2.4),
7. respecter l'ordonnancement causal établi par les coordinations(7) (cf.II.3.2.4).

Comme nous pourrons le voir dans la suite de ce chapitre, certaines de ces critères sont liés. Notamment, la modularité d'expression des coordinations est liée au respect du critère 4.

II.3.3 Comparaison de différents modèles de coordination

Cette section compare différents modèles de coordination au regard des critères de qualité établis dans la section précédente.

---

(7) Ne disposant d'aucune information à cet égard sur les modèles de coordination existants, nous ne parlerons plus de cet aspect dans le reste de ce chapitre.

En s'inspirant d'une étude sur ce domaine [110], trois modèles sont distingués. Chacun utilise de manière spécifique l'une ou les deux des abstractions suivantes : l'appel de méthode et l'événement.

Nous supposons que les abstractions utilisées par chaque modèle sont fournies par (ou accessibles depuis) les modèles de programmation des agents. Le problème d'intégration du modèle de coordination avec les modèles de programmation des agents n'est abordé qu'en II.4.

Pour comparer les modèles, prenons l'exemple suivant qui servira de référence. On doit définir les coordinations dans lesquelles intervient un compilateur. Celui−ci doit être coordonné avec un éditeur de manière à ce que lorsque le développeur sélectionne une erreur de compilation, la ligne comportant cette erreur soit automatiquement visualisée dans un éditeur. Deux sélections d'erreur consécutives doivent être "visualisées" dans le même éditeur, si ce dernier est encore actif lors de la deuxième sélection. S'il ne l'est plus, un nouvel éditeur doit automatiquement être activé.

On veut donc exprimer la coordination suivante : *compiler.select_err → editor.visu_ligne*, avec les contraintes énoncées sur le choix de l'agent éditeur.

II.3.3.1 Modèle basé sur l'appel de méthode

Sont ici analysés les modèles avec lesquels l'administrateur décrit les coordinations au travers d'appels de procédure [8](dans le domaine des langages procéduraux), ou d'appels de méthode [9] (dans le domaine des langages à objets). L'appel de méthode étant l'un des moyens les plus simples et les plus directs pour exprimer des coordinations, il paraît nécessaire de préciser pourquoi il ne répond pas directement à nos besoins.

La coordination de l'exemple choisi s'exprime dans l'agent compilateur (C), par un test sur la présence de l'agent éditeur (E) suivi d'un appel de méthode vers cet éditeur (Fig. 2.3).

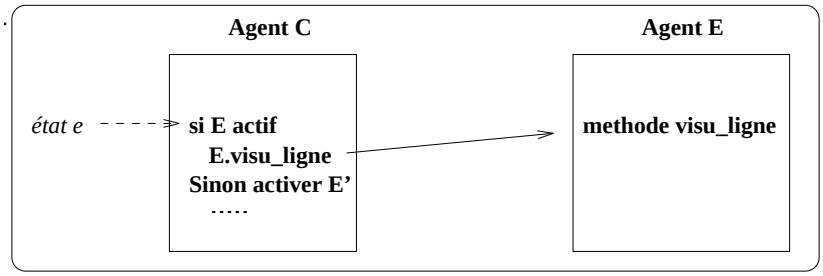


Fig. 2.3 : Expression de compiler.select_err → editor.visu_line au moyen de l'appel de méthode

(8) Le terme appel de procédure englobe ici l'appel de procédure local et l'appel de procédure distant (RPC [85]). L'appel local doit être considéré comme un cas particulier de l'appel distant.

(9) La distinction entre langages procéduraux et langages à objets n'est pas pertinente à ce niveau.

Ce modèle pose les problèmes majeurs suivants :

1. il implique que l'administrateur désigne l'agent appelé par son identité,
2. il implique que l'administrateur désigne la réaction à exécuter par sa définition locale (*visu_ligne*),
3. l'expression des coordinations n'est pas séparée des programmes exécutables des agents,
4. l'expression des coordinations n'est pas modulaire, étant entièrement exprimée dans les agents "appelants".

Le problème 1 paraît le plus important, remettant en cause l'adéquation de ce modèle dans un environnement à forte évolution dynamique des agents [31][111].

II.3.3.2 Modèle basé sur l'événement

Le mécanisme d'événement permet à un agent d'annoncer un événement auquel d'autres agents peuvent avoir attaché des réactions qui sont alors à exécuter [111] (Fig. 2.4).

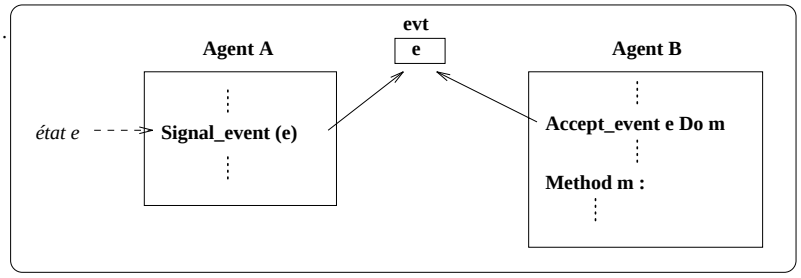


Fig. 2.4 : Expression de A.e → B.m au moyen d'un événement

Plusieurs mécanismes supportant la notion d'événement ont été conçus dans divers domaines (bases de données, intelligence artificielle, conception assistée par ordinateur, environnement de développement, etc). Nous nous concentrons sur ceux conçus pour l'intégration du contrôle qui possèdent certaines spécificités.

Ces mécanismes, qualifiés de *bus de messages*, ont été présentés en I.3. Ils permettent de désigner les agents intervenant dans une coordination au travers d'attributs (par exemple l'attribut *class* dans l'exemple de la figure Fig. 2.5). Les liaisons entre les attributs et les agents sont évaluées dynamiquement, lors des émissions d'événements.

Dans l'exemple, l'émission de *select_err* entraîne l'évaluation de l'ensemble des agents de classe *editor* qui acceptent de recevoir l'événement *select_err* en provenance d'un outil de classe *compiler*. Cette évaluation donne l'ensemble des réactions qui peuvent être

exécutées. Si le type de la coordination (donné par l'émetteur) est *notification*, toutes ces réactions sont exécutées. Si le type est *requête*, une seule réaction choisie au hasard est exécutée.

Dans ce dernier cas, pour gérer la situation dans laquelle aucune réaction ne peut être exécutée, certains mécanismes permettent d'activer automatiquement un agent adéquat (E' dans l'exemple).

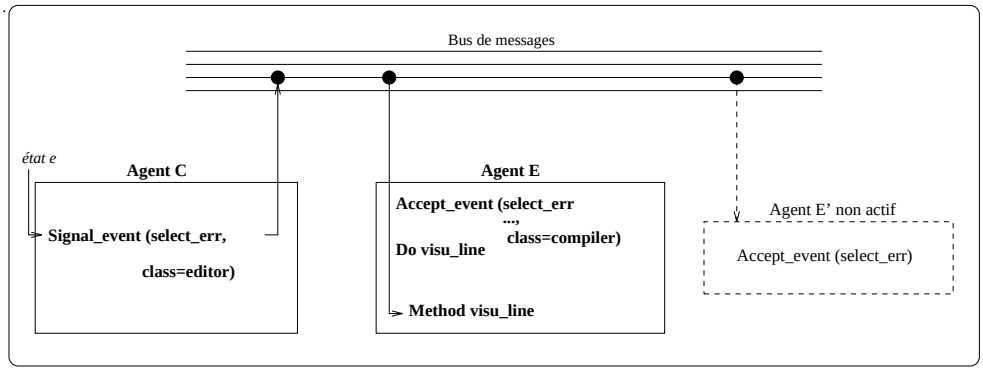


Fig. 2.5 : Expression de compiler.select_err → editor.visu_line au moyen d'un bus de messages

Les avantages procurés par ces mécanismes par rapport aux critères établis en II.3.2 sont essentiellement les suivants.

1. Il est possible de désigner les agents intervenant dans une coordination autrement que par leur identité, par des attributs qui peuvent représenter leurs caractéristiques.

   Les types d'attributs diffèrent selon les mécanismes.

   On trouve par exemple un filtrage sur la *classe* d'un agent (éditeur, metteur au point, etc.) (ToolTalk [112], SoftBench [25]), sur son *identité* (process−id) (ToolTalk), sur sa *localisation* (session dans laquelle il travaille) (ToolTalk), sur la *machine* sur laquelle il s'exécute (SoftBench), et enfin sur les *données* sur lesquelles il travaille (nom de fichier) (SoftBench, ToolTalk).

   Alors qu'avec SoftBench et ToolTalk, la désignation des agents s'appuie sur des attributs spécifiques aux environnements de développement (notion de session, notion de donnée), le Koala bus [11] fournit un mécanisme de désignation plus générique qui s'appuie sur le concept de groupe (cf. I.3.3.4).

2. En permettant à un émetteur de désigner ses récepteurs et inversement, on obtient une certaine modularité d'expression des coordinations. Par exemple, le fait que le

compilateur "veuille" que la réaction soit exécutée par un éditeur est exprimé au niveau du compilateur.

3. Ces mécanismes fournissent les types requête et notification.

4. Les types de communication fournis sont les modes synchrones et asynchrones. Toutefois, aucun mécanisme, à notre connaissance, ne fournit le mode asynchrone avec accusé de réaction.

Les points faibles que nous notons à l'égard de ces mécanismes sont les suivants.

1. Les compositions des ensembles des événements E et des réactions R ne sont pas explicites : aucun modèle de déclaration d'interface d'un agent en termes d'événements et de réactions n'est fourni.

2. L'expression des coordinations, et par suite de leur évolution dynamique(10) est mélangée aux programmes exécutables des agents.

3. La modularité d'expression des coordinations fournie n'est pas satisfaisante : les associations entre événements et réactions sont exprimées au niveau des agents "appelés". L'ajout d'un appelant qui émet *selection_err* à la place de de *select_err* implique de modifier l'appelé (ou les appelés).

4. La désignation de l'événement qui intervient dans une coordination utilise le nom et les paramètres définis dans le code de l'agent qui émet cet évenement (*select_err*).

5. Les fonctions de désignation des agents ne semblent pas assez puissantes.

   Les fonctions à base d'attributs fournies par SoftBench et ToolTalk ne permettent pas de différencier deux éditeurs travaillant dans une même session et sur une même donnée autrement que par leur identification interne (process−id). Ceci peut poser problème : dans notre exemple, si deux éditeurs sont actifs, l'un ayant un rôle de visualisation en association avec un metteur au point et l'autre ayant un rôle d'édition, comment assurer que deux sélections consécutives d'erreurs engendrent deux coordinations reçues par le même éditeur ?

   Les fonctions à base de groupes fourni par le Koala bus ne permettent pas de désigner des agents au travers d'opérateurs appliqués sur plusieurs groupes

---

(10) Des primitives (*UnAccept_event*) permettent à un agent de modifier dynamiquement les coordinations dans lesquelles il peut intervenir en tant que récepteur.

(11) Notons que ces communications, au niveau de l'implémentation, peuvent correspondre à des appels de procédures locaux.

(intersection, union, etc.). Seul *un* groupe peut être désigné. Ceci implique de disposer d'un groupe par agent ou par ensemble d'agents désignables. Le nombre d'agents pouvant être grand, l'établissement des différents groupes devient complexe.

Si certains des points faibles énoncés ci−dessus peuvent être résolus en modifiant le mécanisme de coordination, d'autres (2, 3 et 4) découlent du modèle et non du mécanisme le mettant en œuvre. Aussi, l'adéquation du modèle à base d'événements est dégradée par ces inconvénients.

II.3.3.3 Modèle mixte

Les modèles mixte sont ceux au travers desquels l'administrateur dispose de l'appel de méthode *et* de l'événement (tel que défini dans la section précédente) pour exprimer des coordinations.

Le modèle mixte qui semble le plus adéquat utilise des agents intermédiaires (qualifiés de *coordinateurs*) pour l'expression et pour la gestion des coordinations. A chaque agent est associé un coordinateur (Fig. 2.6). Les communications entre coordinateurs sont exprimées au travers d'événements, celles entre un coordinateur et l'agent qui lui est associé au travers d'appels de méthode.

La coordination de notre exemple s'exprime comme suit avec ce modèle :

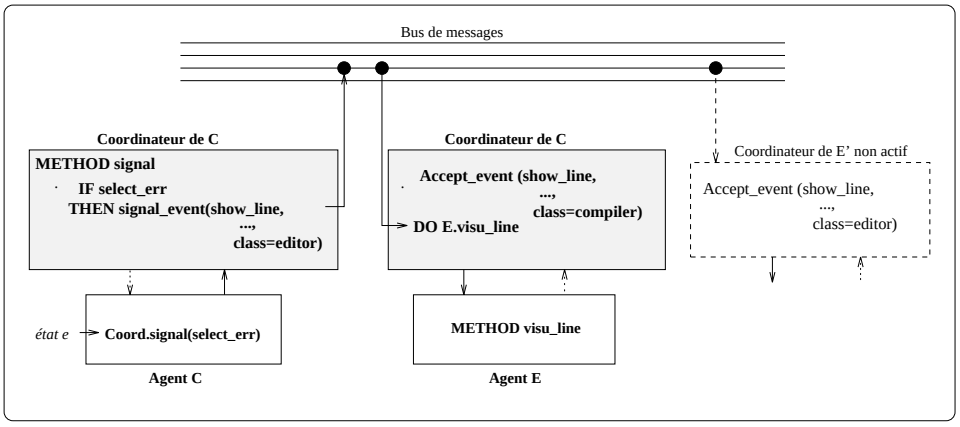


Fig. 2.6 : Expression de compiler.select_err → editor.visu_line au moyen du modèle mixte

La particularité essentielle de ce modèle est qu'il *dissocie* les événements et réactions des coordinations, en fournissant un espace global et explicite de coordinations, dans lequel les coordinations sont caractérisées par :

- un nom (*show_line* dans la figure Fig. 2.6),
- des paramètres formels (non représentés dans la figure Fig. 2.6),
- un type (non représenté dans la figure Fig. 2.6),
- un mode (non représenté dans la figure Fig. 2.6),

Les événements et réactions qui interviennent dans une coordination sont définis et modifiables dynamiquement, au travers d'opération de liaisons et déliaisons exprimées dans les coordinateurs.

Une liaison (événement ⇔ coordination) peut désigner les agents récepteurs de la coordination au travers d'attributs. De même, une liaison (réaction ⇔ coordination) peut désigner les agents émetteurs de la coordination au travers d'attributs.

Différents mécanismes de coordination entre outils, bien que n'exploitant pas tous les possibilités offertes par ce modèle, peuvent s'y rattacher : DEC Fuse EnCase [40], et le mécanisme proposé par Clément dans [31].

Outre les avantages procurés par le modèle à base d'événement que l'on retrouve dans certains de ces mécanismes, ils fournissent les avantages additionnels suivants :

1. Les compositions des ensembles des événements E et des réactions R sont explicites [31][40].
2. L'espace global de coordination permet de désigner événements et réactions au travers de définitions globales (*show_line* dans l'exemple*)*.
3. Cet espace global permet également de modulariser l'expression des coordinations. Dans chaque coordinateur sont définies les coordinations dans lesquelles l'outil associé est susceptible d'intervenir en tant qu'émetteur ou récepteur. Ainsi l'ajout d'un compilateur qui émet l'événement *selection_err* à la place de *select_err* implique uniquement de définir ou modifier le coordinateur de ce compilateur.
4. Les coordinations sont exprimées en dehors du code des agents, dans les coordinateurs.
5. Dans [40], l'évolution dynamique des coordinations est exprimée explicitement : un coordinateur est décrit sous la forme d'états de coordination. Chaque état définit quelles coordinations sont susceptibles d'être émises ou reçues par le coordinateur lorsque, dynamiquement, il se trouve dans cet état.

Ils présentent toutefois les inconvénients suivants.

1. Dans [40], les changements d'état sont exprimés dans les programmes exécutables des agents.
2. Les moyens de désignation à base d'attributs présentent les insuffisances énoncées pour le modèle à base d'événement.

Ces inconvénients peuvent être levés car ils ne sont pas imposés par ce modèle que nous jugeons, de ce fait, très avantageux.

En contrepartie, ce modèle est plus lourd que les précédents pour l'administrateur. Celui−ci doit écrire (avec un langage fourni à cet effet) le code des coordinateurs, définir les interfaces des agents en termes d'événements et réactions, et spécifier les appels liant un agent et son coordinateur.

II.3.3.4 Conclusion

L'objectif de cette section était de présenter les problèmes rencontrés dans l'élaboration d'un mécanisme de coordination pour les outils d'un environnement de développement. Différents modèles de coordination ont été évalués en fonction de critères préalablement établis.

Le modèle basé sur l'appel de méthode implique de désigner les agents intervenant dans une coordination par leur identité. Ceci, entre autre, remet en cause son adéquation au sein d'un environnement à forte évolution dynamique des agents.

Le modèle à base d'événements lève cet inconvénient, car les agents sont désignés au travers d'attributs. Par contre, l'expression des coordinations et de leur évolution dynamique est éparpillée dans les programmes exécutables des agents et n'est pas modulaire. En outre, la désignation des événements et réactions intervenant dans une coordination utilise leur définition locale. La compréhension et la maintenance des coordinations sont donc difficiles. Au sein d'un environnement à forte évolution des agents (ajout / retrait d'outils) et des coordinations, cet inconvénient est rédhibitoire.

Enfin, le modèle mixte présenté en II.3.3.3, qui fait usage du mécanisme d'événement et du mécanisme d'appel de méthode, semble permettre de lever les inconvénients posés par les modèles précédents, aux dépens d'une plus grande lourdeur d'expression des coordinations.

II.4 Interopérabilité des agents

Les agents à coordonner peuvent être écrits dans des langages différents et s'exécuter sur des systèmes et machines différents. Le mécanisme de coordination agit donc également comme un mécanisme d'interopérabilité(12) entre agents hétérogènes.

Seuls les éléments jugés nécessaires pour comprendre les problèmes posés sont décrits dans cette section. Le lecteur intéressé trouvera des informations plus complètes dans [121] [122].

Après avoir définit la fonction principale d'un mécanisme d'interopérabilité (II.4.1), nous nous plaçons dans le contexte de l'intégration du contrôle et présentons les besoins et les approches adoptées (II.4.2).

---

(12) Le terme interopérabilité dénote ici tout mécanisme permettant à des agents hétérogènes de communiquer, sans présupposer de la nature des communications autorisées.

II.4.1 Fonction d'un mécanisme d'interopérabilité

Un mécanisme d'interopérabilité doit permettre la communication de flots de contrôle et de données entre des agents écrits dans des langages de programmation différents et s'exécutant éventuellement sur des supports (systèmes, machines) différents. Il effectue des traductions entre les supports et langages des agents, par rapport à des règles préalablement définies.

Ci−après sont succinctement décrits trois principaux types de différences qui sont à prendre en compte au niveau des agents : les langages de programmation, les modèles de données et les modèles d'exécution.

1. **Langages de programmation**

   Pour permettre à un agent de communiquer avec un autre agent, il est nécessaire de fournir des fonctions de communication. Ces fonctions doivent être offertes à tout agent dans son langage de programmation.

2. **Modèles de données**

   - Divergences de représentation des données :

     Il s'agit de gérer les différences de représentation (ordonnancement des bits d'un entier, précision des nombres flottants, etc) d'un même type de base. Ces différences sont couramment dues à l'hétérogénéité des supports des agents (machines, systèmes d'exploitation, systèmes de communication).

   - Divergences des types de base :

     Il s'agit d'établir des traductions entre les types de base des modèles de données. Par exemple, comment est traduit le type *booléen* du langage Guide dans le langage C. Ces traductions sont indépendantes des schémas de données utilisés par les agents qu'elles lient.

   - Divergences entre les constructeurs :

     Il s'agit d'établir les règles de correspondance entre les constructeurs fournis par les modèles de données. Par exemple, quelle est la traduction du type record du langage Guide dans le langage C. Comme précédemment, ces traductions sont indépendantes des schémas de données utilisés par les agents qu'elles lient.

   - Divergences sémantiques :

     Il s'agit de gérer des correspondances entre des schémas de données utilisés par différents agents. Par exemple, pour un premier agent, un livre va être composé d'un nom d'auteur, d'une date et d'un contenu. Pour un autre agent, un livre va seulement contenir un nom d'auteur et un contenu.

Les règles de traduction sont ici spécifiques aux schémas de donnée des agents liés.

Selon les besoins, les divergences décrites ci−après ne sont pas forcément toutes gérées par un mécanisme d'interopérabilité.

Alors que les différences de niveau *représentation des données* ont fait l'objet de travaux très foisonnants, la gestion des autres types de différences constitue un axe de recherche plus récent et encore ouvert. On peut distinguer les approches selon qu'un modèle ou schéma global est utilisé ou non [101]. Dans un premier cas, des règles de traduction sont définies par couple de modèles de données (et éventuellement de schémas de données) à relier.

Dans un deuxième cas qui couple moins fortement les agents interopérants, les règles de traduction sont définies entre chaque modèle de données et un modèle global de référence, (voire entre chaque schéma de données et un schéma global) [105]. La difficulté repose ici sur le choix du modèle global, qui doit être aussi générique que possible.

Cette deuxième approche a été adoptée par l'Object Request Broker (ORB), issu de l'Object Management Group (OMG) [37], qui est un mécanisme permettant à des objets hétérogènes d'interagir au travers d'appels de méthodes. Le modèle global est mis en œuvre par le langage nommé IDL (Interface Description Language).

3. **Modèles d'exécution**

Le modèle d'exécution dénote les constructeurs qui permettent de spécifier le schéma d'exécution d'un agent. Définir des traductions entre des modèles d'exécution représente un domaine de recherche très complexe. Par exemple, comment traduire la notion de procédure au niveau d'un modèle d'exécution basé sur l'inférence de règles ?

La simplification couramment adoptée suppose que l'appel de méthode(13) est une abstraction fournie par l'ensemble des modèles d'exécution des agents, en tant que constructeur de base à partir duquel d'autres constructions plus élaborées peuvent être synthétisées [122]. L'interopérabilité entre agents repose alors sur l'appel de méthodes par certains agents envers d'autres agents. C'est le cas de l'ORB. L'appel de méthode est également l'abstraction choisie par la majorité des mécanismes d'intégration du contrôle.

La simplification précédente ne résout néanmoins pas tous les problèmes, comme expliqué ci−après.

(13) Rappelons que nous considérons l'appel de procédure comme un cas particulier de l'appel de méthode : l'identification de l'agent appelé étant implicite.

Il faut d'une part effectuer la liaison entre la désignation de la méthode à exécuter dans l'agent appelant et la désignation de celle−ci dans l'agent appelé.

Une solution simple oblige l'appelant à connaître le nom utilisé par l'appelé. Il n'est alors plus possible de développer les agents indépendamment les uns des autres : une concertation préalable définissant un espace commun de noms de méthode est nécessaire. Cette approche est celle de la spécification CORBA de l'ORB.

Une autre approche est de fournir un mécanisme d'indirection qui permet de définir les liaisons entre noms de méthode a posteriori par rapport au développement des agents. C'est la direction suivie par le mécanisme d'intégration du contrôle de Clément[31].

Il faut d'autre part mettre en place le flot d'exécution qui doit réaliser l'appel. Or les modèles qui offrent l'appel de méthode comme abstraction peuvent varier (modèles à objets passifs, modèles à objets actifs, modèles client/serveur, etc.). L'approche classique consiste à choisir le modèle client−serveur comme modèle de référence car il se prête bien à la mise en place d'interopérations entre agents. Pour les autres modèles, le mécanisme d'interopérabilité se charge d'utiliser les abstractions qu'ils fournissent pour simuler un modèle client−serveur.

II.4.2 Fonctions requises pour l'intégration du contrôle

L'intégration du contrôle nécessite tout d'abord de fournir des fonctions de coordination dans les différents langages de programmation des outils.

Au niveau des modèles de données, les mécanismes d'intégration du contrôle ne gèrent généralement que les divergences de niveau représentation. Nous jugeons utile de gérer les divergences de niveau représentation, types de base et constructeurs (cf. II.4.1) pour permettre aux outils d'échanger directement des données de structure simple sans se soucier de leurs modèles de données respectifs. Les données de structure complexe sont généralement stockées dans la base de données de l'environnement, et leur échange revient à un échange d'identificateur de donnée. L'accès à ces données par deux outils hétérogènes doit normalement être pris en charge par la base de données.

Enfin, au niveau des divergences portant sur les modèles d'exécution, nous adoptons la position courante qui suppose que chaque outil est capable de réaliser un appel de méthode. Les mécanismes de coordination fournissent ainsi des primitives qui permettent à un agent écrit dans un modèle donné de recevoir et de réaliser un appel de méthode venant d'autres agents. Par exemple, les agents qui sont écrits selon le modèle d'exécution événementiel de X/Motif disposent d'une primitive qui permet d'ajouter, au niveau des événements recevables et traitables, des événements de coordination émis par d'autres agents et devant déclencher des exécutions de méthodes locales [11][25].

II.5 Conclusion

Ce chapitre a précisé les problèmes soulevés par l'intégration du contrôle et analysé les solutions proposées pour résoudre ces problèmes.

L'objectif de cet axe d'intégration a été défini comme étant de permettre à des agents de se coordonner de manière à ce que les actions des uns entraînent des réactions de la part des autres.

Nous nous sommes placés dans un environnement de niveau groupe dans lequel les agents sont des outils de gros grain. Nous avons mis en évidence les caractéristiques qui engendrent les problèmes essentiels. Celles−ci sont : une forte évolution des agents et des coordinations (ajouts / retraits d'outils et de coordinations), une forte évolution des agents actifs et des coordinations actives (activations / désactivations d'outils et de coordinations), et l'indépendance entre agents (cf. II.2).

Nous avons choisi de diviser problèmes et solutions en deux parties, l'une concernant les modèles de coordination et l'autre l'interopérabilité. La première partie est influencée par l'ensemble des caractéristiques précitées. L'interopérabilité est en revanche due à l'indépendance des agents.

Nous rappelons ci−après les critères essentiels, énoncés en II.3.2.5 et II.4.2, qui doivent être remplis par un mécanisme d'intégration du contrôle. Celui−ci doit :

1. permettre d'exprimer les deux composants suivants de manière modulaire et en dehors du code des agents (cf. II.3.2.2) :
   - les événements susceptibles d'être émis et les réactions susceptibles d'être exécutées par les agents,
   - les coordinations,
2. permettre d'exprimer l'évolution dynamique des coordinations explicitement et en dehors du code des agents,
3. permettre de désigner les agents au travers de leurs caractéristiques (leur fonction, la session dans laquelle ils s'exécutent, etc.),
4. permettre de désigner les événements et réactions intervenant dans une coordination par des définitions globales,
5. permettre aux agents d'échanger des données de structures simples, même si leurs modèles de données respectifs divergent,
6. fournir des coordinations de type requête et notification,
7. fournir les modes de coordination synchrone, asynchrone, asynchrone avec accusé de réaction,
8. respecter l'ordonnancement causal établi par les coordinations.

L'état actuel est représenté par deux techniques [5] : les techniques à base de bus de messages du type de Field et les techniques d'interopérabilité du type de l'Object Request Broker (ORB) [20].

Les secondes s'attachent essentiellement à remplir le critère 6. Elles impliquent de désigner les agents intervenant dans une coordination par leur identité et ne répondent donc pas à nos besoins.

Les techniques de bus de messages et en particulier celles basées sur le modèle de coordination mixte répondent mieux à nos besoins. Les critères essentiels 3, 4 et 6 restent toutefois pas ou mal remplis :

1. l'expression de l'évolution dynamique des coordination reste mélangée au code des agents,
2. les fonctions de désignation des agents paraissent insuffisantes (cf.II.3.3.2),
3. aucune facilité n'est fournie aux agents qui ont des modèles de données divergents pour échanger des données.

Ces aspects constituent la motivation de notre travail.

Chapitre III

Proposition d'un modèle et d'un langage de description de l'intégration du contrôle

Nous proposons un mécanisme de coordination qui comprend un langage (nommé Indra) de gestion des coordinations, ainsi qu'un environnement d'exécution pour ce langage.

Le langage Indra se base sur le modèle de coordination mixte présenté en II. Du point de vue des structures logiques de contrôle, ce modèle permet de définir les outils en termes d'interfaces abstraites et les coordinations en termes de liaisons entre ces interfaces. L'expression des coordinations est ainsi modulaire et séparée du code des outils.

Pour gérer l'évolution dynamique des coordinations et des outils actifs, ce langage fournit deux concepts additionnels fondés d'une part sur la notion d'automate d'états fini et d'autre part sur la représentation de l'organisation des outils selon un arbre attribué.

La section III.1 présente notre cadre de travail, les objectifs qui ont dirigé notre proposition ainsi que les principes que nous avons adoptés. Les sections III.2 et III.3 présentent ensuite les aspects clés du langage Indra.

L'environnement d'exécution du langage Indra est présenté dans le chapitre V.

III.1 Introduction

III.1.1 Cadre de travail

L'environnement de développement Cybèle constitue notre cadre de travail. Il a pour objectif d'assister les développeurs d'applications écrites en langage Guide [75] et s'exécutant sur la plateforme d'exécution de même nom [6]. Les hypothèses de base ayant dirigé sa conception et sa réalisation définissent les caractéristiques suivantes.

Caractéristiques dimensionnelles et fonctionnelles :

- Cybèle doit être un environnement de moyenne envergure (de niveau *groupe*, cf. I), supportant un ensemble de développeurs travaillant sur un *réseau local* de stations de travail éventuellement hétérogènes.
- Le support fourni doit s'attacher à l'ensemble des phases de développement du cycle de vie d'une application : conception, programmation, validation, configuration, maintenance.
- Les développeurs doivent pouvoir utiliser des outils de développement existants, largement acceptés par le marché (par exemple, l'éditeur *emacs*, le metteur au point *dbx*, etc.), ou constituant des standards.

Choix de base de conception et de réalisation :

- L'environnement Cybèle a pour support une structure d'accueil qui doit fournir les cinq axes d'intégration classiques présentés dans le chapitre I.

  Pour l'instant, seule l'intégration au niveau de la plate−forme, des données et du contrôle a été traitée.
- L'intégration des données s'appuie sur un support existant, (version Emeraude V12 de PCTE industrialisée par le Gie Emeraude [55]).
- L'intégration du contrôle fait l'objet de travaux propres, car les problèmes qui restent ouverts nous ont semblé intéressants.
- La plate−forme support de l'environnement est composée du système Unix et du système Guide.

  Le système Unix permet de supporter beaucoup d'outils de développement déjà existants et de disposer de PCTE comme support d'intégration des données.

  Disposer aussi du système Guide permet d'exécuter les applications développées au sein de l'environnement. De plus, Guide fournit des fonctions de gestion de la distribution, de la persistance, du partage d'objets, qui sont très utiles pour la réalisation des divers axes d'intégration que nous voulons mettre en place à court ou long terme.

L'architecture conceptuelle de l'environnement de développement Cybèle est illustrée par la figure Fig. 3.1. Les éléments représentés en gris foncé dénotent les axes d'intégration que nous voulons fournir à court terme, contrairement à ceux représentés en gris clair dont nous visons la conception et la réalisation à plus long terme seulement.

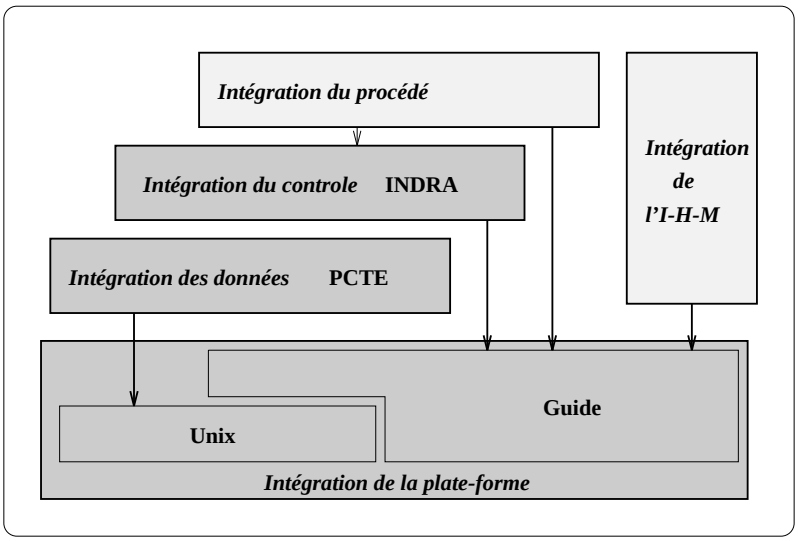


Fig. 3.1 : Architecture logicielle de l'environnement Cybèle

III.1.2 Objectifs et principes adoptés pour l'intégration du contrôle

Nos objectifs sont de satisfaire les critères de qualité énoncés en II.5 d'un mécanisme d'intégration du contrôle s'insérant dans un environnement de niveau groupe composé d'outils de gros grain.

Nous mettons l'accent sur les critères, toujours énoncés en II.5, qui nous ont paru mal remplis par les solutions existantes. Pour ce faire, nous adoptons les principes suivants.

1. Le modèle de coordination choisi est, à la base, le **modèle de coordination mixte** (Fig. 3.2) présenté en II.3.3.3. Celui−ci est en effet le plus apte à satisfaire les objectifs fixés, au vu des critiques émises en II.3.3.

2. Pour permettre de décrire explicitement l'évolution dynamique des coordinations d'un outil, notre modèle s'inspire du **concept d'automate d'états fini**.

3. Pour désigner les outils, notre modèle se fonde sur le **concept d'arbre attribué**. Les outils actifs sont placés dans un arbre attribué et pour les désigner, on peut utiliser leurs attributs et leur position dans cet arbre.

4. Enfin pour permettre aux outils ayant des modèles de donnée divergents d'échanger des données de structure simple (cf II.3.1), un **modèle de donnée global** est fourni. Le lien entre ce modèle et le modèle d'un outil est effectué au niveau de la Bibliothèque de fonctions de coordination (Fig. 3.2).

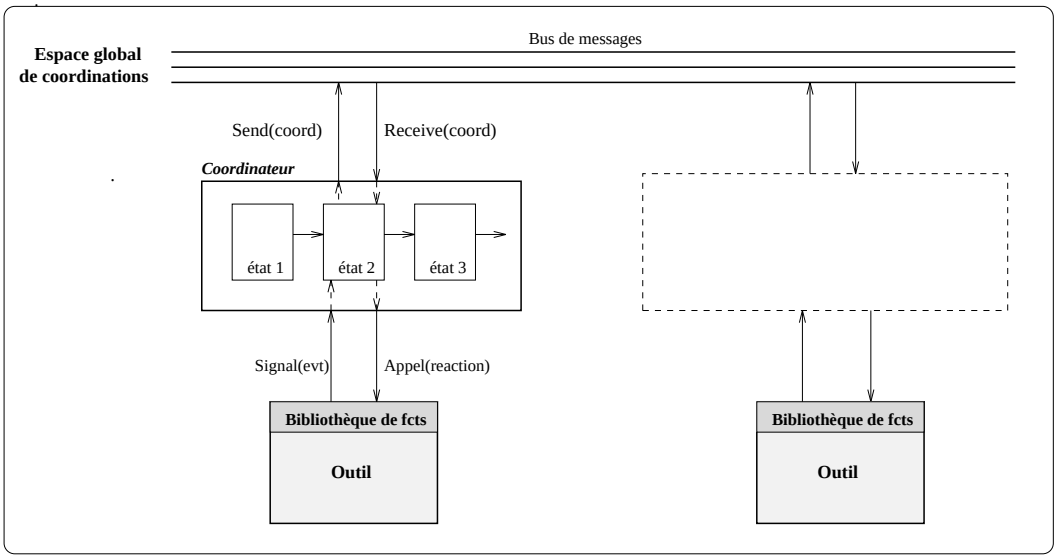


Fig. 3.2 : Modèle de coordination proposé.

Pour des raisons de temps, l'aspect 4 n'a pas été traité en profondeur. Le modèle de données choisi est celui du langage Guide [83]. A terme, nous jugeons intéressant d'adopter le modèle du langage IDL de Corba, défini spécifiquement pour permettre la communication de données entre agents hétérogènes.

La suite du chapitre présente principalement les aspects 2 et 3 que nous jugeons essentiels.

III.2 Description Indra de la coordination des outils

La description Indra de la coordination d'un outil comprend essentiellement :

1. la description de son ***interface de coordination*** (III.2.1).
2. la description de ses coordinations et de leur évolution dynamique. Nous appelons cette partie la *description des* ***états de coordination*** *de l'outil*(1) (III.2.2).

Parmi les différents états pouvant être associés à un outil, se trouve un état particulier appelé *état d'activation*, dans lequel les coordinations recevables entraînent l'activation de l'outil. La notion d'état d'activation fait l'objet de la section III.2.3.

Les descriptions des états de coordination et d'activation d'un outil font intervenir des mécanismes qui modélisent le comportement *dynamique* de l'outil vis à vis de ses coordinations. La section III.2.4 précise le modèle d'exécution associé à ces mécanismes.

(1) Le lien entre ces descriptions et l'outil est réalisé au travers du *nom* de l'outil, que celui−ci précise lorsqu'il commence à s'exécuter.

III.2.1 Description de l'interface de coordination d'un outil

L'interface de coordination d'un outil définit les réactions exécutables et les événements transmissibles de l'outil.

Outre les rôles affectés à cette description énoncés en II.3.2.1, elle est un support pour la vérification des descriptions des coordinations, permettant de vérifier que les événements et réactions sont correctement utilisés.

Un exemple simplifié de description de l'interface de coordination d'un outil est donné ci−après.

```
TOOL_COORD_ITF OF emacs
ASYNC_REACT   edit (IN file_name: String);
              visualize (IN file_name: String);
              ...
SYNC_REACT    show_line (IN file_name:String,IN no_line:Integer);
              ...
REQ_EVT       ask_compilation (IN file_name: String);
              ask_debug (IN file_name: String);
              ...
NOTIF_EVT     load_file (IN file_name: String);
              save_file (IN file_name: String);
              ...
END
```

A la différence d'une réaction définie par ASYNC_REACT, une réaction définie par SYNC_REACT signifie que l'outil notifie le mécanisme de coordination lorsqu'il a fini d'exécuter la réaction. Ceci permet à l'administrateur de savoir si une requête qui met en jeu une réaction donnée peut être ou non exécutée en mode synchrone (cf. II.3.2.4).

III.2.2 Description des états de coordination d'un outil

Le comportement d'un outil vis à vis de ses coordinations est décrit sous la forme d'un automate d'états de coordination. Un état de coordination est identifié par un *nom* et définit essentiellement :

1. L'ensemble des coordinations dans lesquelles l'outil peut intervenir lorsque, durant son exécution, il se trouve dans cet état. A un moment donné de son exécution, un outil ne peut donc se trouver que dans un seul état de coordination.
2. Quelles coordinations, parmi celles autorisées dans l'état, font passer l'outil à un autre état.

Ces deux points sont traités en III.2.2.2.

Pour permettre de manipuler et conserver des informations sur les coordinations dans lesquelles un outil actif est déjà intervenu, des variables peuvent également être déclarées et manipulées dans les états de coordination. Cet aspect est traité en III.2.2.1.

Pour faciliter le travail de l'administrateur, un mécanisme d'héritage permettant de définir un état de coordination en réutilisant et surchargeant les définitions d'un autre état a été conçu.

III.2.2.1 Manipulation de variables dans un état de coordination

Du point de vue statique :

La déclaration d'une variable dans un état de coordination définit le nom et le type de la variable. Les types autorisés sont ceux du langage Guide.

Ces variables peuvent être manipulées dans les *parties opératives* associées au traitement des coordinations définies dans ces états (voir ci−après, III.2.2.2). La désignation d'une variable fait intervenir le nom de l'état dans lequel elle est déclarée et son nom local : *<nom état>.<nom local>*.

Du point de vue dynamique :

Pour tout outil actif, les variables déclarées dans les états dans lesquels il est susceptible de se trouver ont une une durée de vie égale à sa durée d'exécution (cf. Fig. 3.3) et une visibilité globale à l'ensemble de ces états.

Par exemple, avant de faire passer un outil O de l'état nommé E_1 à l'état nommé E_2, on peut affecter la variable nommée v_2 déclarée dans E2 avec la valeur de la variable nommée v_1 déclarée dans E1, par l'instruction suivante : $E_2.v_2 := E_1.v_1$

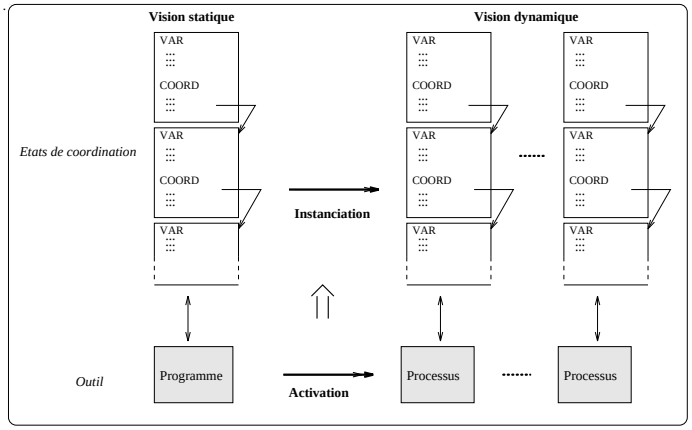


Fig. 3.3 : Chaque outil actif possède ses propres instances d'états de coordination

III.2.2.2 Définitions des coordinations dans un état de coordination

La définition d'une coordination comprend deux parties : une partie déclarative et une partie opérative. La partie déclarative spécifie la cause qui entraîne l'exécution de la partie opérative associée.

Cette cause peut être la signalisation d'un événement par l'outil ou la réception d'une coordination (de type requête ou notification(2)) en provenance des autres outils, comme illustré par les flèches en gras dans la figure Fig. 3.4.

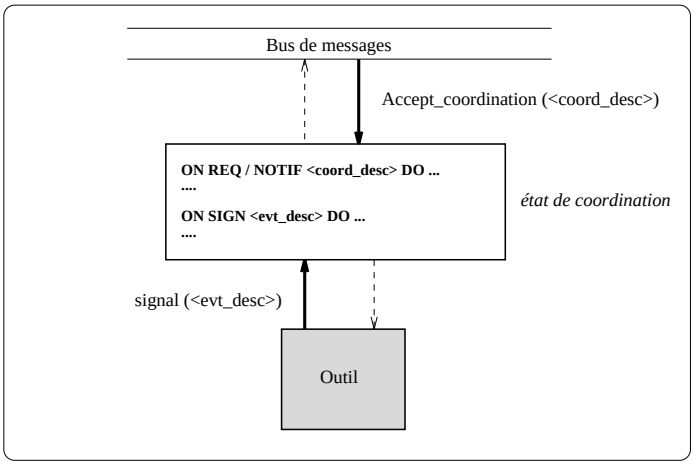


Fig. 3.4 : Principe de description des coordinations dans un état de coordination

Une partie déclarative exprimée sous la forme `ON SIGN <evt_desc>` caractérise un événement signalé par l'outil comme montré ci−après. La clause `WHERE`, permettant de spécifier des conditions portant sur les valeurs des paramètres effectifs, est optionnelle.

```
ON SIGN <evt_name> (arg1: t1, .. , argk: tk)
   WHERE   (arg1 = val1) AND ... (argk = valk)
```

De même une partie déclarative exprimée sous la forme `ON REQ <coord_desc>` ou `ON NOTIF <coord_desc>` caractérise une coordination recevable par l'outil comme montré ci−après. Les clauses `WHERE` et `FROM` (cette dernière permettant de désigner les émetteurs potentiels de la coordination) sont optionnelles.

```
ON [REQ/NOTIF] <coord_name> (arg1 : t1, .. , argk : tk)
   WHERE   (arg1 = val1) AND ... (argk = valk)
   FROM    ...(3)
```

Toute partie opérative s'exprime sous la forme `DO <operative_part>` et peut comprendre diverses instructions, toutes optionnelles, permettant :

---

(2) Les types et modes de coordination fournis sont présentés en III.2.2.3.

(3) La désignation des outils est traitée en III.3.

- d'émettre une requête ou une notification vers les autres outils. Nom, mode, paramètres et désignation des récepteurs potentiels (cf. III.3) doivent être spécifiés,
- de manipuler (tester et affecter) des variables qui auront été déclarées dans l'état de coordination courant ou dans un autre état de coordination associé à l'outil[?], avec les valeurs de ces mêmes variables, ou bien avec les valeurs des arguments effectifs du signal émis ou de la requête ou notification reçue, ou encore avec les valeurs des retours d'information sur une requête ou notification émise précédemment (cf. Annexe B),
- de changer d'état de coordination,
- d'appeler une méthode de l'outil, cette méthode ayant été déclarée comme réaction dans la description de l'interface de coordination de l'outil.

```
ON ...
DO  [REQ/NOTIF] <coord_name> (arg1: t1, .. ,argk: tk)  TO ...
    <var_affectation>
    CHANGE_CS  <cs_name>
    CALL  <react_name> (arg1', .. , argn')
```

Pour des raisons d'homogénéité, les instructions autorisées dans les parties opératives associées à la signalisation d'un événement et à la réception d'une coordination sont identiques. Néanmoins, la réception d'une coordination provoque généralement l'appel d'une méthode réaction. Inversement, la signalisation d'un événement provoque généralement l'émission d'une coordination.

Les instructions autorisées dans une partie opérative doivent être placées dans l'ordre dans lequel elles sont énumérées ci−dessus. Les raisons qui nous ont poussé à imposer cette contrainte sont exposées en III.2.4.

L'exemple ci−après décrit une coordination qui pourrait être placée dans un état de coordination associé à un outil de mise au point. Il est spécifié que lorsque l'outil signale l'événement `ask_visualize` avec le paramètre effectif `file_type` ayant pour valeur `SRC`, alors dans l'ordre :

- une requête en mode asynchrone avec accusé est émise, vers un éditeur travaillant dans la même session que le metteur au point,
- l'outil change d'état de coordination et passe dans l'état `cs_with_visualization`.

```
ON  SIGN ask_visualize (
              IN file_name: String, IN file_type: Integer)
    WHERE     file_type = SRC
DO  REQ_ASYNC_AR visualize_src (file_name: String)
```

```
TO   [un éditeur dans ma session, cf. III.3]
CHANGE_CS   cs_with_visualization
```

De manière identique, l'exemple ci−après pourrait s'appliquer à un outil d'édition. Il est spécifié que l'outil est intéressé à recevoir une requête de visualisation d'un fichier source si l'émetteur de cette requête est un metteur au point travaillant dans la même session que lui. Lorsqu'un tel événement est effectivement reçu, alors dans l'ordre :

- la variable `current_file_name`, définie dans l'état de coordination `cs_visualization`, prend la valeur du paramètre `file_name`,
- l'outil change d'état de coordination et passe dans l'état `cs_visualization`,
- l'outil exécute la méthode réaction `visualize (file_name)`.

```
ON   REQ visualize_src (IN file_name : String)
     FROM [un metteur au point dans ma session, cf. III.3]
DO   cs_visualization.current_file_name := file_name;
     CHANGE_CS   cs_visualization
     CALL  visualize (file_name)
```

III.2.2.3 Types et modes de coordinations

Le langage Indra fournit deux types de coordination : les requêtes et les notifications, définis en II.3.2.4. Pour ces deux types, le choix du ou des récepteurs d'une coordination est dicté par (Fig. 3.5):

- la désignation des récepteurs potentiels donnée par l'émetteur,
- la désignation des émetteurs potentiels spécifiée par tout récepteur.

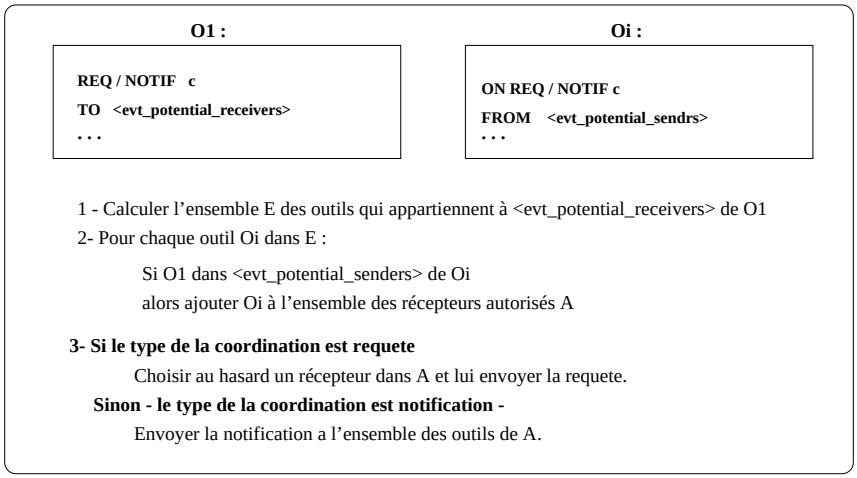


Fig. 3.5 : Algorithme de choix des récepteurs d'une coordination

Les modes de coordination offerts sont au nombre de trois : synchrone, asynchrone, asynchrone avec accusé de réaction (définis en II.3.2.4)[(4)].

III.2.3 Description de l'état d'activation d'un outil

Deux aspects sont à considérer quant à l'activation d'un outil :

- quelle situation est susceptible d'entraîner l'activation d'un outil ?
- quel état de coordination initial est affecté à un outil activé ?

Pour fournir un mécanisme homogène, l'activation d'un outil est également exprimée par un état de coordination (appelé *état d'activation*) qui décrit les coordinations qui sont susceptibles d'entraîner l'activation de l'outil, et ce sous la forme classique :

```
ON  <declarative_part>  DO  <operative_part>
```

Dans les parties déclaratives, seules des réceptions de coordinations de type requête peuvent être spécifiées. Cette restriction est temporaire. Elle a permis de simplifier certains aspects de niveau conceptuels du service de désignation des outils (présenté en III.3).

Les parties opératives doivent contenir une instruction de changement d'état. Ceci permet d'assurer que l'outil, une fois activé, ne se trouve pas dans son état d'activation. Elles doivent également spécifier la commande d'activation de l'outil. Notons enfin qu'un appel de méthode (méthode *réaction*) peut aussi être spécifié.

Ci−après se trouve un exemple simplifié de description d'état d'activation pour un outil de nom `emacs`. Il est spécifié que celui−ci peut être activé pour servir deux types de requêtes, l'une de nom `edit` et l'autre de nom `show_line`.

Pour servir la requête `show_line`, l'éditeur doit exécuter la méthode réaction `show_line` après avoir été activé par la commande `edit_file`. Par contre, pour servir la requête `edit_file`, la commande d'activation de l'éditeur est suffisante.

```
TOOL_ACT_STATE OF emacs
  ON REQ    edit (IN file_name: String)
     WHERE  [conditions sur les valeurs des arguments]
     FROM   [désignation des émetteurs potentiels, cf. III.3]
  DO COMMAND  'lemacs − edit <file_name>'
     CHANGE_CS   cs_edit

  ON REQ   show_line(IN file_name:String, IN no_line:Integer)
```

(4) Au niveau du prototype réalisé, nous avons toutefois simplifié notre tâche en limitant l'indépendance entre types et modes. Les modes synchrone et asynchrone avec accusé de réaction ne sont autorisés qu'avec le type requête. Le mode asynchrone n'est autorisé qu'avec le type notification.

```
      WHERE [conditions sur les valeurs des arguments]
      FROM  [désignation des émetteurs potentiels, cf. III.3]
   DO COMMAND  'lemacs − visualize <file_name>'
      CHANGE_CS   cs_visualize
      CALL   show_line  (no_line)
End emacs.
```

III.2.4 Modèle d'exécution

Dans un état de coordination sont spécifiés un ensemble de couples (partie déclarative, partie opérative). La partie déclarative définit une *condition* qui porte sur un *évenement*(5) : évenement signalé par l'outil associé à l'état, ou réception d'une notification ou d'une requête en provenance d'un autre outil.

Les sections III.2.4.1, III.2.4.2 et III.2.4.3 précisent les règles qui régissent l'évaluation des parties déclaratives, ainsi que les exécutions des parties opératives. Ces règles sont également celles appliquées aux état d'activation.

La section III.2.4.4 aborde le problème classique des interblocages.

III.2.4.1 Priorités

Dans un état de coordination, l'ordre de définition des parties déclaratives définit une priorité décroissante de celles−ci. Lorsqu'un même événement satisfait plusieurs parties déclaratives d'un même état, seule la partie opérative associée à la partie déclarative la plus prioritaire peut être exécutée. Ce choix nous a paru en accord avec les besoins des outils.

III.2.4.2 Ordre de traitement des événements

Cette section précise l'ordre dans lequel sont traités les événements qui sont reçus par un outil actif donné. Pour faciliter l'expression des coordinations, nous choisissons de traiter les événements séquentiellement, de manière à fournir un modèle d'exécution déterministe : la même séquence d'événements reproduit le même comportement de l'outil vis à vis de ses coordinations.

Considérons tout d'abord le traitement des événements de type *requête* et *notification* qui sont *asynchrones*.

Pour un outil actif donné O, ces événements sont traités de manière atomique, dans leur ordre d'arrivée. Le traitement d'un événement *e* consiste en trois étapes :

1. Evaluer les parties déclaratives définies dans l'état de coordination courant de O pour déterminer quelle est la partie opérative à exécuter (cette partie opérative étant nulle si l'événement ne satisfait aucune partie déclarative). Dans la suite, nous notons cette étape *Eval (e)*.

---

(5) Dans cette section, le terme événement est employé dans un sens général.

2. Exécuter la partie opérative déterminée par *Eval(e).* Nous notons cette étape *P_op (e).*
3. Si un appel à une méthode réaction est spécifié à la fin de la partie opérative, exécuter cette méthode. Nous notons cette étape *React (e).*

Un outil O qui reçoit dans l'ordre les événements (e_1,..,e_n) (Fig. 3.6) exécute *séquentiellement* les actions suivantes :

$$Eval(e_1) \text{ -> } P\_op(e_1) \text{ -> } React(e_1) \text{ ..... -> } Eval(e_n) \text{ -> } P\_op(e_n) \text{-> } React(e_n)$$

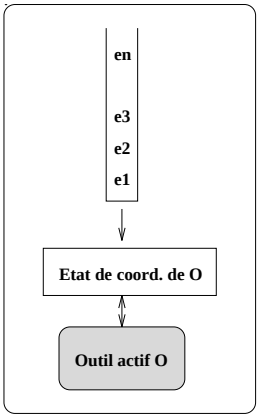


Fig. 3.6 : Les événements reçus sont mémorisés et traités séquentiellement

Considérons maintenant le fait que l'outil O peut également signaler des événements qui provoquent l'exécution de parties opératives définies dans son état de coordination courant. Dans la figure Fig. 3.7 par exemple, l'outil signale l'événement *e* durant l'exécution de la méthode réaction *m*.

Ces événements ont la particularité d'être synchrones par rapport aux événements précédents : ils interviennent lors de l'exécution d'une méthode réaction. Ces événements sont donc traités séquentiellement. Ceci signifie que toute étape *React* peut se se traduire en un ensemble fini de séquences : *Eval -> P_op -> React*.

Notons qu'avec ce modèle, l'exclusivité d'exécution des parties opératives n'est garantie qu'en considérant que l'exécution d'une partie opérative contenant un appel de méthode réaction prend fin dès que l'exécution de cette méthode est commencée. Cet aspect justifie l'ordre imposant que l'instruction d'appel à une méthode réaction soit la dernière instruction d'une partie opérative.

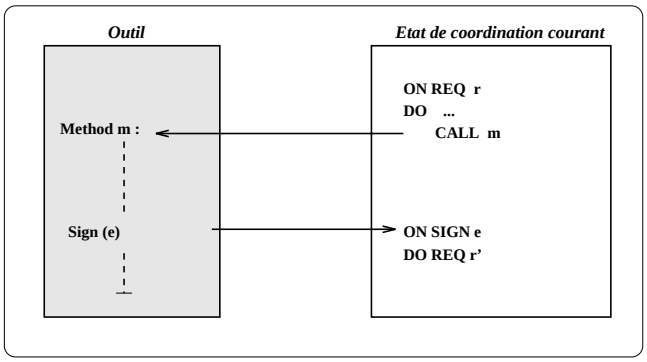


Fig. 3.7 : L'exécution d'une méthode réaction (m) peut entraîner l'émission d'événements (e)

Le modèle d'exécution que nous proposons, déterministe, implique des contraintes au niveau du traitement des événements par *un* outil actif donné. En revanche, aucune contrainte n'est engendrée sur un ensemble d'outils actifs : tout outil actif traite ses événements indépendamment des autres outils actifs.

III.2.4.3 Ordonnancement

Nous voulons que les parties opératives des coordinations soient exécutées dans un ordre conforme à l'ordre causal établi par les *émissions* d'événement (cf. II.5).

Pour ce faire, il suffit d'assurer que tout outil *reçoit* des événements dans un ordre conforme à l'ordre causal car la réception d'un événement et l'exécution de la partie opérative associée sont atomiques.

L'ordre assuré au niveau *modèle* repose donc sur l'ordre assuré au niveau *mise en œuvre*. Cet aspect est traité dans le chapitre V.

III.2.4.4 Interblocages

Le modèle de coordination choisi ne prévient pas la présence d'interblocages. Un outil qui émet une requête avec attente ne peut durant son attente servir une autre requête (Fig. 3.8).

Dans notre contexte, les situations d'interblocage ne peuvent être considérées comme une erreur de *spécification* des coordinations car ces dernières ont la propriété d'être asynchrones (cf. II.1). Il est donc important de gérer au mieux ces situations.

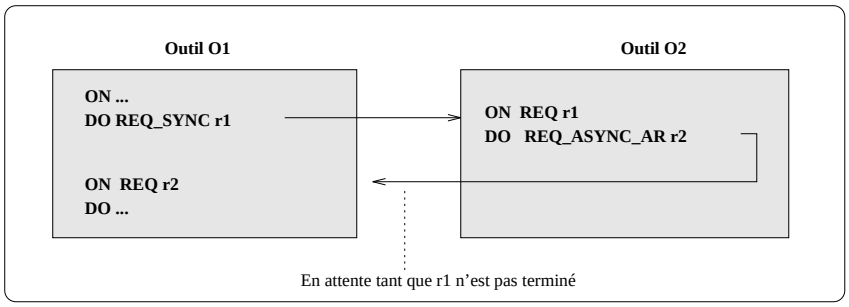


Fig. 3.8 : Interblocages provoqués par l'utilisation du mode synchrone, et par l'atomicité d'exécution des coordinations d'un même outil actif

Un interblocage est provoqué lorsqu'un ensemble d'outils émettent des coordinations (requêtes/notifications) en modes synchrone ou asynchrone avec accusé de réaction et qu'un cycle se forme entre les émetteurs et les récepteurs de ces coordinations, comme noté ci−après :

$$O_1.\text{émission}\,(c_1, O_2) \rightarrow O_2\text{émission}\,(c_2, O_3) \rightarrow \ldots \rightarrow O_k.\text{émission}\,(c_k, O_1)$$

Si chaque outil attend que la coordination émise soit reçue (en mode asynchrone avec accusé de réaction) ou traitée (en mode synchrone) par son récepteur, alors il y a interblocage.

Deux aspects sont à considérer : la détection des interblocages et leur résolution.

Une solution de détection simple et couramment adoptée consiste à rendre le contrôle à l'émetteur au bout d'un temps limite fixé lorsqu'aucun retour n'est reçu de la part d'un récepteur. Cette solution ne vérifie pas que l'attente de l'émetteur est bien provoquée par un interblocage. Une solution plus complexe augmente la précédente par l'activation automatique, lorsqu'un émetteur est bloqué depuis un temps fixé, d'un agent qui va détecter si ce client se trouve dans un cycle bloquant.

Pour des raisons de temps, nous avons opté pour la solution la plus simple et n'avons pas traité la résolution des interblocages.

Remarque Dans les cas d'interblocages provoqués par l'utilisation du mode asynchrone avec accusé de réaction, la résolution de ces conflits est *envisageable*. Ce mode de communication étant beaucoup plus utilisé que le mode synchrone(6), cette possibilité nous paraît importante.

Cette résolution repose sur le fait que, dans le cycle formé, au moins deux outils sont bloqués par une émission successive de coordinations en mode asynchrone avec accusé de réaction.

---

(6) Cette affirmation résulte d'une expérimentation que nous avons réalisé dans un environnement comprenant six outils.

Ainsi dans la figure Fig. 3.9, l'émission de c_2 ne peut être provoquée par la réception de c_1 (c_1 ayant été émis en mode asynchrone avec accusé de réaction, sa réception par O_2 aurait libéré O_1). Aucune relation de causalité n'existe donc entre l'émission de c_1 et l'émission de c_2. Il est alors envisageable de "faire revenir en arrière" O_2, de le faire recevoir et traiter c_1, et de le faire ensuite ré−exécuter le traitement qui l'a conduit à envoyer c_2. L'émission d'une coordination ne pouvant être que la première instruction de toute partie opérative, ce "retour arrière" ne constitue pas une opération complexe.

En revanche, lorsqu'un interblocage ne comprend pas, dans le cycle formé, deux émissions successives en mode asynchrone avec accusé de réaction alors le "retour arrière" s'avère très difficilement réalisable. Il demande en effet d'être capable de revenir en arrière dans une méthode réaction exécutée par un outil.

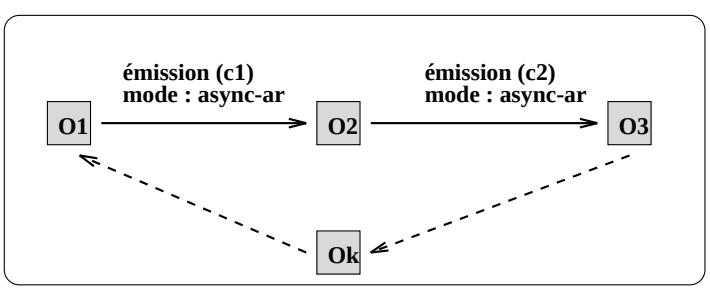


Fig. 3.9 : Situation d'interblocage résoluble

En conclusion, il est possible de résoudre certaines situations d'interblocage au niveau de la mise en œuvre du mécanisme de coordination. Cette possibilité ne présume cependant pas du tout de la faisabilité : l'impact sur les performances d'exécution peut être tel que cette résolution n'est plus envisageable.

Dans tous les cas, si la résolution des situations d'interblocage les plus courantes peuvent être résolues, d'autres ne peuvent l'être. Il est alors nécessaire de faire remonter au niveau du langage des moyens de gestion de ces dernières. Ce manque d'homogénéité au niveau de la gestion des interblocages constitue un inconvénient non négligeable, mais qui peut trouver sa contre−partie s'il permet de gérer les cas d'interblocage les plus courants.

III.3 Désignation des outils

Cette section présente le service qui permet de désigner les outils intervenant dans une coordination.

La section III.3.1 dresse les besoins et argumente les principes qui régissent notre solution. Les sections suivantes décrivent le service proposé.

III.3.1 Objectifs et principes

Le service de désignation doit permettre à un outil de sélectionner les récepteurs d'une coordination qu'il émet et de sélectionner les coordinations qu'il reçoit en fonction de leurs émetteurs.

Un critère essentiel, énoncé en II.5 (point 3), veut que ce service permette de désigner les agents au travers de caractéristiques qui reflètent leur organisation (leur fonction, la session dans laquelle ils s'exécutent, etc.).

Nous fournissons la notion d'*attribut* pour représenter les caractéristiques inhérentes aux outils, indépendantes de leur position dans l'organisation. La simple notion d'attribut fournit en effet un espace de désignation plat dans lequel les regroupements d'outils (par session, par développeur, par équipe de développeur, etc.) sont difficiles à représenter.

Pour représenter l'organisation des outils, nous permettons de placer ceux−ci dans un *arbre* et d'utiliser pour les désigner, outre leurs attributs, leur position dans cet arbre.

Par simplicité, la position d'un outil dans l'arbre et la valeur de ses attributs sont décidés lors de son activation et ne varient plus jusqu'à sa désactivation.

Pour permettre aux outils d'utiliser des informations qui varient dynamiquement pour se désigner, un mécanisme additionnel basé sur la notion de *relation* est fourni. Dynamiquement, des relations peuvent être établies ou supprimées entre les outils.

Enfin, nous jugeons utile de permettre de désigner un outil déjà actif, ou non déjà actif, ou indifféremment déjà actif ou non.

Par exemple, lorsqu'à la demande d'un utilisateur, un gestionnaire de session doit activer un outil de mise au point, le gestionnaire de session émet une requête de mise au point et spécifie qu'elle doit être servie par un outil à activer (c.a.d non encore actif).

Dans d'autres cas, un outil doit pouvoir émettre une requête sans se préoccuper du fait que l'outil qui va servir cette requête est déjà actif ou non. Par exemple, un "browser" d'erreurs qui affiche le résultat d'une compilation va, lorsque l'utilisateur sélectionne une erreur, émettre une requête de visualisation de la ligne concernée dans le code source vers un éditeur. Si un éditeur déjà actif et acceptant la requête existe, alors la requête sera dirigée vers celui−ci. Sinon, un nouvel éditeur sera automatiquement activé.

Finalement, un outil doit pouvoir émettre une requête vers un outil déjà actif exclusivement.

Nous adoptons donc les principes suivants.

1. L'organisation hiérarchique des outils ainsi que leurs attributs sont définis par un **schéma** qui spécifie la structure de l'organisation. Parce qu'il serait trop rigide d'imposer une organisation unique, la définition du schéma est un **paramètre**.

2. Le formalisme de définition du schéma est fondé sur le concept d'**arbre attribué**. Les nœuds de l'arbre représentent des éléments de regroupement d'outils et les feuilles représentent des outils. Des attributs peuvent être associés à tout nœud, c'est à dire non seulement aux outils mais aussi aux éléments de regroupement d'outils.
3. La désignation d'outils au sein d'une organisation donnée consiste à sélectionner un ou plusieurs sous−arbres dans l'arbre représentant cette organisation. Les outils désignés sont les feuilles des sous−arbres sélectionnés. Ces outils peuvent être **déjà actifs ou non**. Entre les feuilles de l'arbre, des **relations** peuvent être dynamiquement placées et enlevés. La sélection d'un ou plusieurs sous−arbres consiste à spécifier les chemins d'accès à ces sous−arbres, en utilisant **les arcs de l'arbre, les types de nœuds, les valeurs des attributs des nœuds et les relations**.

III.3.2 Définition du schéma d'organisation des outils

Le schéma d'une organisation est défini sous la forme d'un arbre de nœuds typés.

Un type de nœud défini un ensemble d'attributs et le domaine de valeurs de ceux−ci. Pour assurer des propriétés minimales, des types de nœuds prédéfinis ainsi qu'un mécanisme de sous−typage sont fournis par le service de désignation.

Un type de nœud T_1 sous−type d'un type de nœud T_2 récupère l'ensemble des attributs définis dans T_1. De nouveaux attributs peuvent être définis dans T_2 et le domaine de valeurs de l'ensemble des attributs de T_2 peut être redéfini.

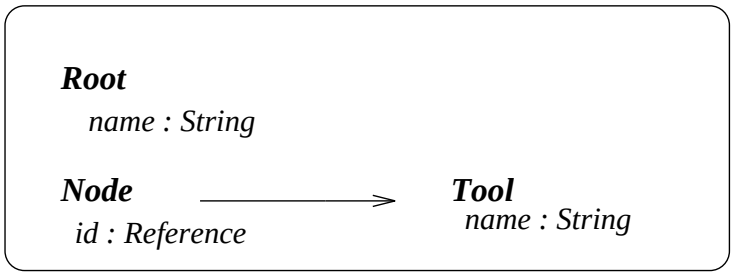


Fig. 3.10 : Types de nœud prédéfinis

Les types de nœud *Node*, *Root* et *Tool* sont prédéfinis. Tout nœud de l'arbre qui n'est ni feuille ni racine est obligatoirement sous−type du type *Node*. Les feuilles doivent être sous−type du type *Tool* et la racine sous−type du type *Root.*

Le type *Node* impose comme attribut un identificateur de type référence (identificateur unique pour un nœud d'un arbre). Le domaine de valeurs de cet attribut est prédéfini avec le symbole *, signifiant dans ce cas l'ensemble des entiers(7). Cet attribut pourra être utilisé pour désigner précisément des outils.

(7) Le symbole * dénote l'ensemble des valeurs associées à un type.

Le type *Tool*, sous−type du type *Node*, impose comme attribut un nom de type chaîne de caractère. Ce nom représente le nom de l'outil (emacs, gdb, thésee, etc.). C'est au travers de ce nom qu'est fait le lien avec la description de la coordination de l'outil.

Enfin le type *Root* impose comme attribut un nom, qui doit permettre de désigner de manière symbolique la racine de l'organisation.

La figure Fig. 3.11 illustre un exemple de définition du schéma d'organisation des outils. Les types de nœuds prédéfinis sont en italique, ceux définis par l'administrateur sont en romain. Dans cette organisation, les outils sont divisés en deux catégories : ceux de développement et ceux d'administration. Les outils de développement sont regroupés en *sessions*, elles−mêmes regroupées en *développeurs*, eux−mêmes à leur tour regroupés en *équipes de développement*. L'organisation des outils d'administration est quelque peu similaire, à la différence que ceux−ci ne sont pas regroupés en *session* mais directement en *administrateurs*.

Il est important de remarquer que si l'organisation définie utilise des concepts tels que la session, les utilisateurs, les équipes, ces concepts n'ont aucune signification sémantique autre qu'éléments de regroupement d'outils pour le service de désignation.

L'organisation illustrée montre que les outils de développement possèdent pour attributs un nom et une classe, ceux d'administration ne possèdent qu'un nom. La définition des domaines de valeurs des attributs (voir au bas de la figure) donne entre autres les couples de noms et classes d'outils de développement autorisés. Ainsi par exemple, lorsque un outil émet une requête vers un outil de classe *editor*, le service de désignation choisit au hasard un outil de nom *vi* ou *emacs*.

Pour mieux illustrer les types de schémas qu'il est possible de définir, deux autres exemples sont donnés.

Le premier exemple (cf. Fig. 3.12) définit un schéma qui, bien que différent du schéma précédent, offre des possibilités de désignation assez proches. Les nœuds *Dev_Team, Adm_Team, Dev, Adm* et *Session* ont été omis, et remplacés par des attributs affectés aux nœuds *User* et *Any_Tool*. Notons toutefois que le schéma précédent permettait de restreindre automatiquement les types d'outils pouvant être activés sous les nœuds de types *Dev* et *Adm* (en fonction de leurs noms et classes). Cette limitation n'existe plus avec le schéma présent.

Le dernier schéma (cf. Fig. 3.13) organise les outils de manière très simple, en les regroupant en *session*. Un outil peut désigner d'autres outils par leurs attributs et par la session dans laquelle ils travaillent. Ce schéma peut être utilisé dans un environnement mono−utilisateur.

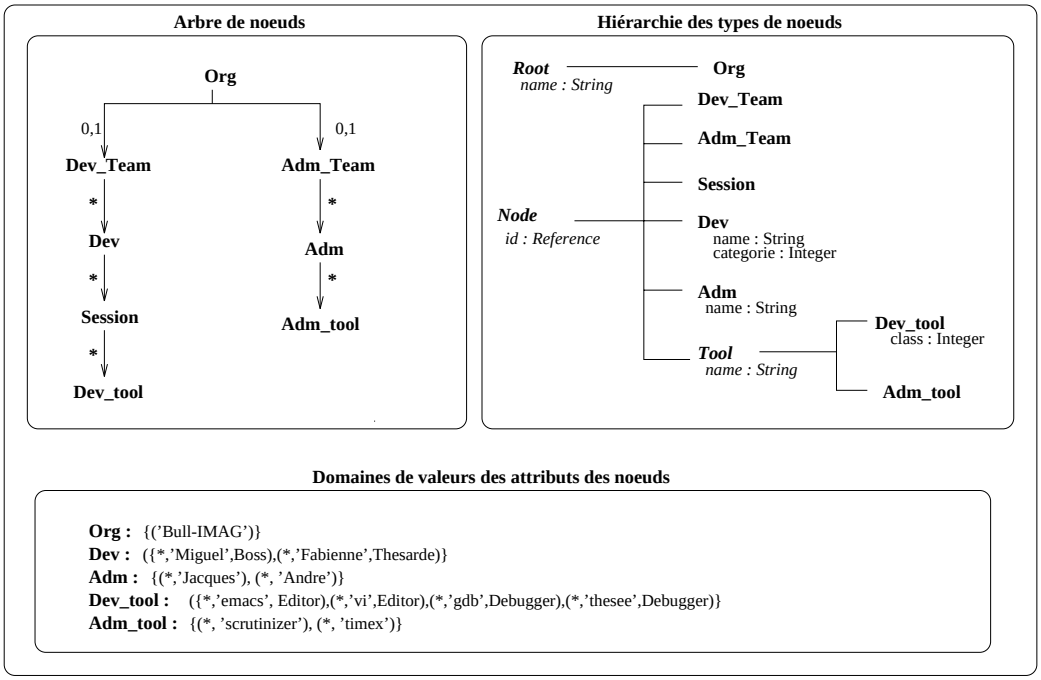


Fig. 3.11 : Premier exemple de schéma d'organisation des outils

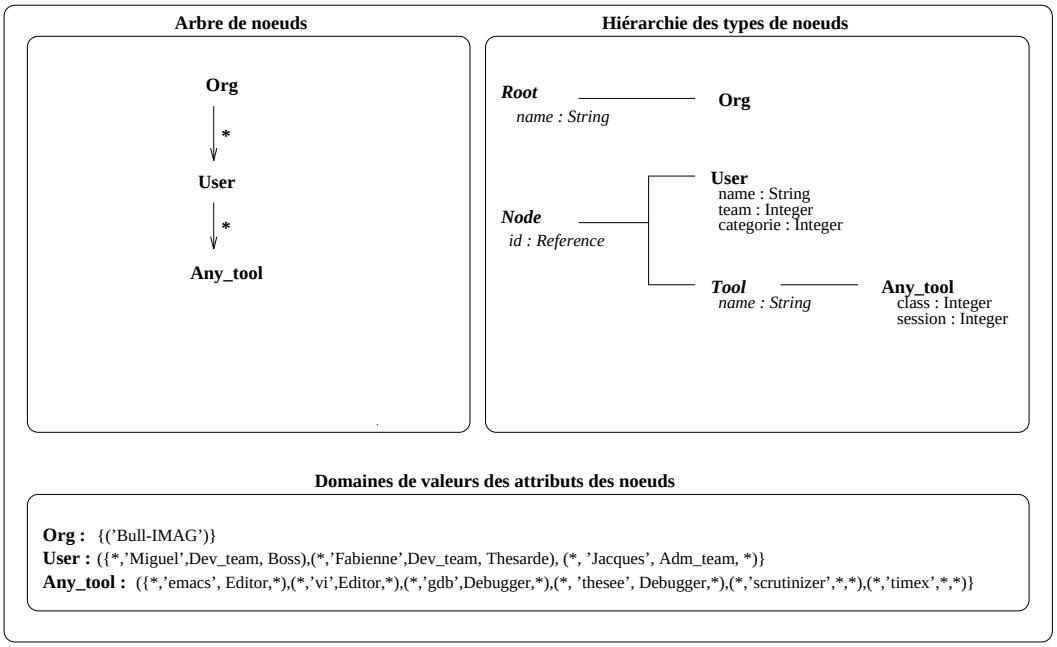


Fig. 3.12 : Deuxième exemple de schéma d'organisation des outils

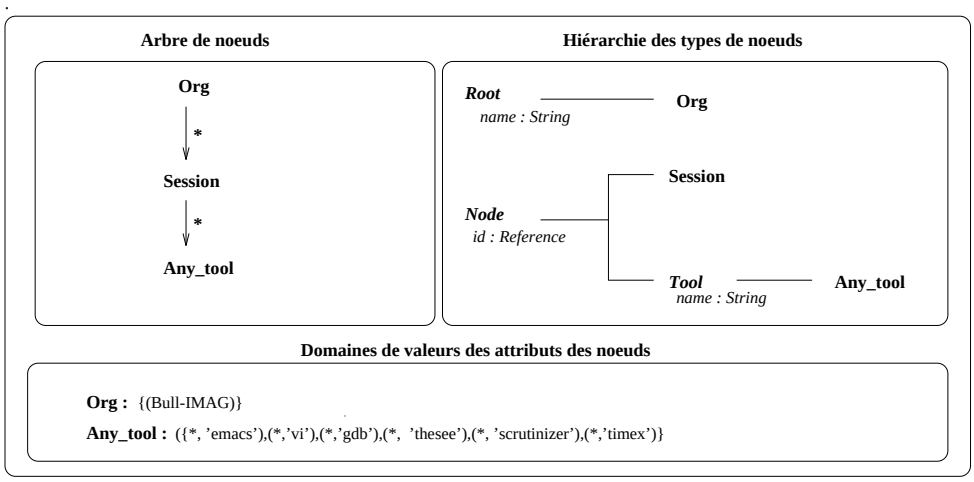


Fig. 3.13 : Troisième exemple de schéma d'organisation des outils

III.3.3 Désignation des outils

Cette section présente le principe de désignation des outils, indépendamment de l'utilisation des relations. Ces dernières sont abordées dans la section III.3.6.

Nous appelons **Arbre Schématisé (AS)** l'arbre définissant le schéma de l'organisation des outils. L'AS donne lieu à un arbre instancié dès qu'un premier outil est activé dans l'organisation qu'il définit. Cet arbre instancié est soumis à évolution : des nœuds peuvent être à tout moment créés et détruits, au gré des activations et terminaisons d'outils. L'arbre instancié est détruit dès que le dernier outil actif est désactivé. Le terme **Arbre Instancié Effectif (AIE)** dénote dans la suite l'état de l'arbre instancié à l'instant courant.

D'un autre coté, l'AS définit un **Arbre Instancié Potentiel (AIP)**, calculé par extension (Fig. 3.14). L'AIP comporte un nombre de branches infini. Notons que l'AIE est toujours une portion de l'AIP. Nous notons dans la suite $\overline{\textbf{AIE}}$ l'ensemble des nœuds de l'AIP n'existant pas dans l'AIE (et n'y ayant jamais existé jusqu'à l'instant présent).

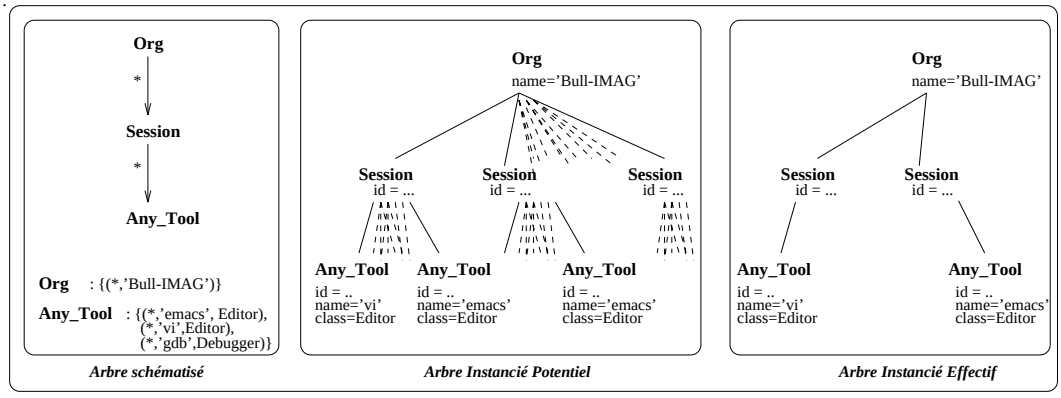


Fig. 3.14 : Arbres utilisés pour désigner les outils

Pour désigner un ou plusieurs outils, le principe est de sélectionner un ou plusieurs sous−arbres dans l'AIE ou dans l'AIP ou encore dans l'$\overline{\text{AIE}}$ selon que l'on veut désigner des outils déjà actifs ou non. Dans tous les cas, les outils désignés sont les feuilles des sous−arbres sélectionnés.

Pour sélectionner un ou plusieurs sous−arbres dans un arbre instancié, il faut spécifier les chemins d'accès à ces sous−arbres par un adressage relatif(8) ou absolu qui peut utiliser les types de nœuds se trouvant sur les chemins et la valeur de leurs attributs.

L'expression `Org.name='Bull−IMAG'/Dev_Team/Dev.categorie=Boss` sélectionne ainsi les sous−arbres se trouvant sous des nœuds de type `Dev` et ayant `boss` pour valeur d'attribut `catégorie`, eux−mêmes se trouvant sous des nœuds de type `Dev_Team`

(8) "A la Unix", le symbole '..' permet de remonter au nœud prédécesseur dans l'arbre instancié.

qui se trouvent à leur tour sous une racine de type `Org` ayant `Bull−IMAG` pour valeur d'attribut `name` (Fig. 3.15).

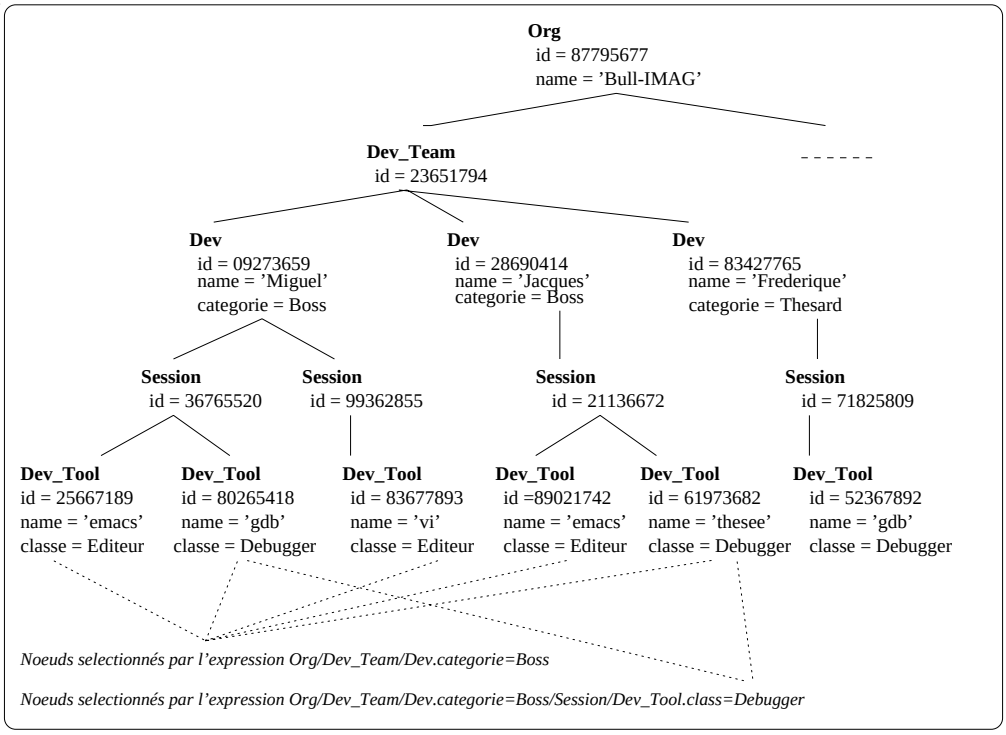


Fig. 3.15 : Exemples de désignation d'outils dans un arbre instancié

Expliquons maintenant comment sélectionner des sous−arbres dans l'AIE ou dans l'AIP ou encore dans l'$\overline{\text{AIE}}$. Un outil qui émet une requête doit pouvoir spécifier qu'elle doit être servie par un outil déjà actif, ou bien par un outil à activer ou encore par un outil à activer si aucun outil actif parmi ceux désignés ne veut bien servir la requête.

Un outil qui spécifie qu'une requête doit être servie par un outil à activer peut vouloir placer cet outil dans une nouvelle *Session*, ou encore dans un nouveau nœud *Dev*.

Pour répondre à ces besoins, l'expression de la désignation d'outils comprend trois parties, chacune optionnelle :

```
<AI_designation> ::= [<AIE_designation> EX]
                     [<AIP_designation>]
                     [NEW <AIE_designation>]
```

Ex : `Org.name='Bull−IMAG' / Dev_Team` **EX**(9) `/ Dev.name='Miguel' /` **NEW** `Session / Dev_Tool.class=Editor`

(9) EX signifie EXistant (EXisting) dans l'AIE.

L'AIE_designation spécifie des chemins qui sont recherchés dans l'arbre instancié effectif. Cette recherche part de la racine de l'arbre instancié effectif (si l'adressage est absolu) ou du nœud courant (si l'adressage est relatif). Elle donne pour résultat un ensemble de nœuds {n1,..,nk}, nœuds terminant ces chemins et appartenant à l'AIE. Notons que l'expression de la désignation peut ne comprendre que l'AIE_designation. Dans ce cas les outils désignés sont les feuilles des sous−arbres de racines {n1,..,nk}. Ces outils sont forcément des outils déjà actifs.

> Par exemple, l'expression `Org.name='Bull−IMAG' / Dev_Team / Dev.name='Miguel' EX` sélectionne l'ensemble des nœuds de type `Dev` ayant `Miguel` pour valeur d'attribut `name`, placés sous un nœud existant (dans l'AIE) de type `Dev_Team`, lui−même placé sous un nœud existant racine de type `Org` et de nom `'Bull−IMAG'`.

L'AIP_designation spécifie des chemins qui sont recherchés en premier lieu dans l'AIE, à partir des nœuds résultats de l'expression précédente ({n1,..,nk}). Si seuls des chemins partiels sont trouvés dans l'AIE, alors on essaie de compléter ces chemins partiels par une recherche dans l'AIP, à partir des nœuds terminant ces chemins partiels dans l'AIE. La recherche des chemins sélectionnés par l'AIP_designation donne pour résultat un ensemble de nœuds {n1',..,nj'}, appartenant à l'AIP. Si l'expression de la désignation ne comprend que l'AIP_designation, on désigne des outils indépendamment du fait qu'ils soient déjà actifs ou non.

> Par exemple, l'expression `Org.name = 'Bull−IMAG' / Dev_Team EX / Dev.name = 'Miguel' / Session` sélectionne les nœuds de type `session` existants ou non, placés sous un nœud de type `Dev` existant ou non et ayant `Miguel` pour valeur d'attribut `name`, ce nœud `Dev` étant placé sous un nœud de type `Dev_Team` déjà existant, etc.

Enfin, l'$\overline{\text{AIE}}$_designation spécifie des chemins qui sont recherchés dans l'AIP et qui doivent être formés de nœuds appartenant à l'$\overline{\text{AIE}}$. Ceux−ci sont recherchés à partir des nœuds résultat de l'expression précédente : {n1',..,nj'}. Cette recherche donne pour résultat un ensemble de nœuds {n1'', .., nl''}, appartenant à l'AIP et n'appartenant pas à l'AIE.

> Par exemple, l'expression `Org.name='Bull−IMAG' EX/ Dev_Team / Dev.name='Miguel'/ NEW Session / Dev_Tool.name='emacs'` désigne un outil de nom `emacs`, non encore actif et qui doit être activé dans une nouvelle `session` appartenant à un développeur de nom `Miguel` déjà existant ou non. Les nœud `Dev_Team` et `Dev` sont automatiquement crées si ils n'existent pas déjà dans l'AIE. Le nœuds `Org` doit être déjà existant dans l'AIE.

III.3.4 Instanciation du schéma d'une organisation

L'activation d'un *premier* outil au sein d'une organisation définie par un AS donné crée un AIE, qui possède un unique chemin allant du nœud racine au nœud représentant l'outil activé.

Le choix des nœuds qui composent ce chemin est dicté par l'expression de la désignation de l'outil à activer. Celle−ci se trouve dans la commande de l'utilisateur donnée au langage de commande de l'environnement.

Le fonctionnement est conforme à celui décrit précédemment (cf. III.3.3) : l'utilisateur peut spécifier que l'outil à activer doit être placé dans une `Session` existante ou non, appartenant à un nœud de type `Dev` existant ou non, etc. Notons que l'utilisateur n'a pas besoin de savoir si l'AIE existe déjà ou non, c'est à dire si un outil a déjà été activé par un autre utilisateur ou non.

Ainsi, un type de commande courant peut être `"REQ edit TO /Org.name='Bull−IMAG' / Dev_Team / NEW Dev.name='Miguel' / Session / Dev_tool.name = 'emacs' "`(10), entraînant l'activation d'un outil de nom `emacs` au sein d'une nouvelle `Session` placée sous un nouveau nœud de type `Dev`, etc.

III.3.5 Destruction d'un schéma d'organisation instancié

Un AIE est détruit lorsqu'il ne comprend plus de nœuds. La destruction d'un nœud est régie par la règle suivante : tout nœud de l'AIE qui n'est pas représenté par une feuille dans l'AS est automatiquement détruit lorsqu'il devient une feuille dans l'AIE. En outre, la terminaison d'un outil entraîne la destruction automatique de la feuille qui représentait cet outil dans l'arbre.

Un AIE est donc détruit lorsqu'il n'y a plus d'outils actifs s'exécutant dans l'organisation représentée par cet arbre.

III.3.6 Utilisation des relations

Une relation est une association nommée qui peut être dynamiquement placée ou enlevée entre deux feuilles de l'AIE. Toute relation est caractérisée par son nom et par l'identité des deux feuilles qu'elle lie. L'objectif des relations est défini ci−après.

1. Permettre de différencier deux outils identiques, se trouvant dans le même état de coordination et étant placés sous un même nœud père dans l'organisation.

   Par exemple, si deux éditeurs (E1 et E2) et deux compilateurs (C1 et C2) sont actifs dans une session donnée, l'éditeur Ei étant coordonné au compilateur Ci, alors l'utilisation de relations entre les couples (Ei, Ci) permet à Ei et Ci de se désigner respectivement sans possibilité d'erreur.
2. Gérer les désactivation d'outils intervenants dans une relation. Par exemple, si l'éditeur E1 est désactivé et qu'une requête est émise par C1 vers "*un éditeur relié à moi−même*", alors une activation automatique d'un nouvel éditeur, relié avec C1, a lieu.
3. Fournir des critères de désignation dynamiques, dans la mesure où les créations et destructions de relations peuvent prendre place à tout moment. Par exemple, un

(10) Certaines parties de cette expression peuvent bien sûr être automatiquement calculées par l'environnement de développement.

éditeur devient dynamiquement "relié" à un compilateur, lorsque l'utilisateur sélectionne la fonction *compiler* dans le menu de l'éditeur.

De manière générale, les relations permettent de modéliser le fait que deux outils peuvent, durant un certain temps, être coordonnés de manière forte : chacun joue un rôle spécifique vis−à−vis de l'autre.

III.3.6.1 Création et destruction de relations

Un outil qui désigne d'autres outils peut spécifier que les outils désignés doivent être *déjà* reliés à lui par la relation exprimée, ou bien ne doivent pas être déjà reliés, ou encore peuvent être ou non déjà reliés.

La désignation d'outils comprend donc la spécification de leur position et des valeurs de leurs attributs et / ou la spécification de leurs relations avec le désignant :

```
<designation>  ::=  [<AI_designation>]    [<REL_designation>]
<REL_designation> ::=   NEW_REL <rel_name> |
                        EX_REL <rel_name>  |
                        REL   <rel_name>
```

L'expression `EX_REL <rel_name>` désigne l'ensemble des outils reliés au désignant par une relation de nom `<rel_name>`. L'expression `NEW_REL <rel_name>` désigne l'ensemble des outils qui ne sont pas reliés au désignant par une relation de nom `<rel_name>` et qui acceptent de l'être en participant à la coordination dans laquelle apparaît cette expression. Enfin, l'expression `REL <rel_name>` désigne l'ensemble des outils soit déjà reliés au désignant par une relation de nom `<rel_name>`, soit non encore reliés mais acceptant de le devenir en participant à la coordination dans laquelle apparaît cette expression.

Par exemple, l'expression ci−après placée dans un état de coordination signifie que l'outil associé veut bien servir la requête si celle−ci provient d'un outil non déjà relié à lui par une relation de nom `collaboration`, mais demandant à le devenir au travers de l'acceptation de la requête.

```
RECV   REQ ...
FROM   NEW_REL  'collaboration'
```

Inversement, l'expression suivante signifie que la requête émise doit être servie par un outil de classe `editor` non encore lié à l'émetteur par une relation de nom `collaboration`, mais acceptant de le devenir au travers de la réalisation de la requête.

```
REQ  ...
TO   ./Dev_Tool.class  =  Editor      NEW_REL  'collaboration'
```

L'expression qui suit signifie que l'outil associé accepte de recevoir la notification si elle provient d'un outil déjà relié à lui par une relation de nom `collaboration`.

```
RECV    NOTIF ...
FROM    EX_REL   'collaboration'
```

Enfin, l'expression suivante signifie que la requête doit être servie par un outil déjà relié à l'émetteur par une relation de nom `collaboration` si un tel outil existe, et sinon par un outil acceptant de le devenir au travers de la réalisation de la requête.

```
REQ   ...
TO    REL  'collaboration'
```

La création d'une relation est associée à une coordination de type requête, la relation créée l'étant entre l'appelant et l'appelé. La destruction d'une relation peut être provoquée par deux causes :

- par la terminaison d'un outil : celle−ci entraîne la destruction de l'ensemble des relations dont l'outil constitue une extrémité,
- par une commande explicite (`RM_REL`) spécifiant le nom de la relation. Par exemple :

```
ON RECV ...
FROM    EX_REL  'collaboration'
DO      ...
        RM_REL  'collaboration'
```

III.3.6.2 Transmission de relations

Dans certains cas, il est utile de pouvoir établir plusieurs relations lors de la réalisation d'une requête. Notamment, on veut pouvoir établir des relations par transitivité, comme illustré dans la figure Fig. 3.16 pour la relation entre O_1 et O_3.

Pour fournir ce moyen, un outil O_1 déjà relié avec un ensemble d'outils $\{O_{11}, .., O_{1k}\}$ par une relation de nom `<rel_name>` peut faire "hériter" O_2 de relations avec $\{O_{11},..,O_{1k}\}$ comme suit :

```
REQ ...
TO  ...
    INHERITS_REL   <rel_name>
```

Inversement, un outil récepteur peut également filtrer les messages qu'il accepte de recevoir en fonction des relations dont il "hérite" comme illustré dans l'exemple ci−après :

```
RECV ...
FROM ...
 WHITH_INHERITED_REL  'collaboration'  ../Tool.class=Debugger
```

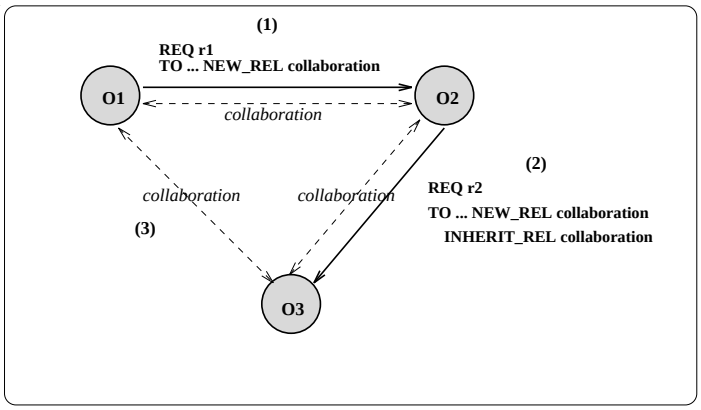


Fig. 3.16 : Transmission de relations

III.4 Conclusion

Nous résumons ci−après les trois caractéristiques majeures de notre proposition, et les contre−parties engendrées. Une évaluation plus approfondie est donnée dans le chapitre VI.

Premièrement, nous avons choisi le modèle mixte qui est le plus apte à satisfaire l'ensemble des critères qui ont formé nos objectifs (cf. II.3.3.3).

L'implication de ce choix est la symétrie impliquée par le modèle : une coordination est définie de manière symétrique au niveau de l'outil émetteur et au niveau de l'outil récepteur. Pour l'administrateur, cette symétrie peut être ressentie comme une duplication d'information, dont il doit vérifier la cohérence.

Deuxièmement, les coordinations dans lesquelles un outil est susceptible d'intervenir sont décrites sous la forme d'états de coordination. Les avantages procurés sont la simplicité d'expression et de compréhension du comportement d'un outil vis−à−vis de ses coordinations. L'implication majeure de ce choix est que l'exécution des coordinations par un outil actif donné doit être atomique. Cette atomicité peut introduire des situations d'interblocage dont la gestion est complexe. Par ailleurs, elle implique que les instructions autorisées par le langage Indra soient placées dans un certain ordre.

Enfin, le troisième point clé concerne les fonctions de désignation des outils, adaptées à un environnement dans lequel les outils évoluent fortement. Cette désignation se fonde sur un schéma d'organisation des outils sous forme d'un arbre de nœuds attribués dans lequel la notion d'outil actif/non actif est prise en compte. Les outils peuvent être désignés par leurs attributs, par leur position dans l'organisation et par le fait qu'ils sont déjà actifs ou non. Les impacts de nos choix à l'égard du service de désignation portent essentiellement sur la mise en œuvre de ce service (cf. VI).

Chapitre IV

Le langage Indra : une illustration par l'exemple

Ce chapitre illustre l'utilisation du langage Indra, au travers d'un scénario qui s'inspire d'une expérimentation réalisée avec l'environnement de développement Cybèle.

Après avoir présenté l'état initial de l'environnement (IV.1), les étapes principales mises en jeu dans le scénario sont décrites (IV.2).

IV.1 Etat initial de l'environnement

Le scénario prend place dans un environnement défini par :

- les descriptions Indra des outils,
- la définition de l'Arbre Schématisé (AS) (IV.1.1),
- l'état de l'Arbre Instancié Effectif (AIE) à l'instant où commence le scénario (IV.1.3),
- l'état des outils actifs au sens de leur état de coordination courant, à l'instant où commence le scénario (IV.1.3).

Les descriptions des états de coordination des outils ne sont pas données, car nous avons préféré présenter celles−ci par portions, tout au long du scénario(1). Seules quelques explications informelles sur le comportement des outils sont données en IV.1.2 .

IV.1.1 Définition de l'Arbre Schématisé

La définition de l'AS est donnée par la figure ci−après. Dans le scénario, on s'intéresse uniquement aux outils de développement. Ceux−ci sont regroupés en *sessions*, elles−mêmes regroupées en *développeurs*. Les développeurs sont à leur tour regroupés en *équipes*. Deux catégories d'outils de développement sont distinguées : ceux de niveau *session* et ceux de niveau *équipe*.

(1) En Annexe C, le lecteur intéressé pourra trouver les descriptions Indra de certains outils impliqués dans le scénario.

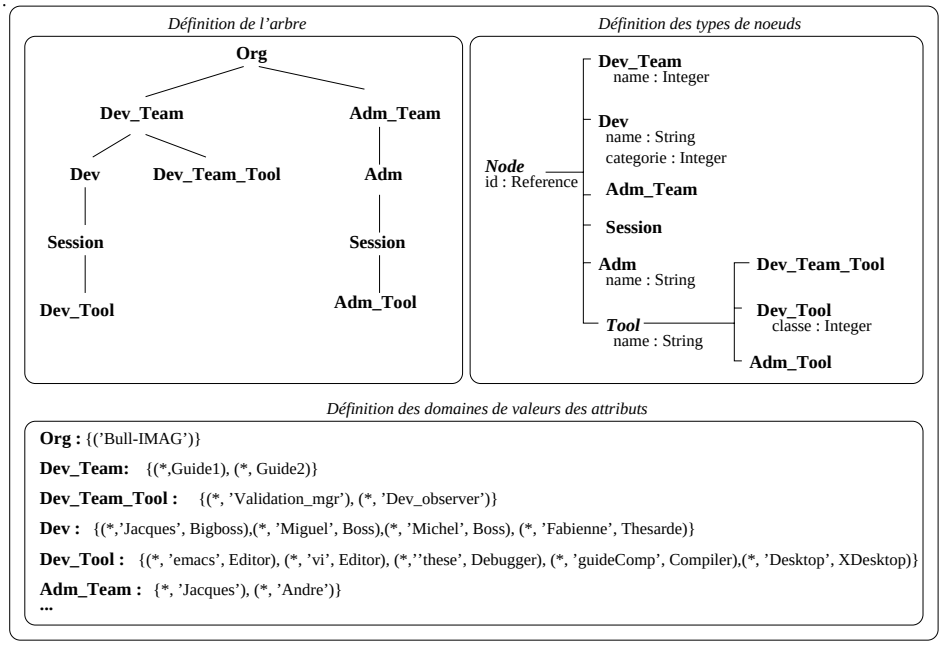


Fig. 4.1 : Arbre schématisé du scénario

IV.1.2 Description des outils

Six outils dont nous précisons brièvement le rôle et le comportement sont mis en jeu(2).

Desktop

Le rôle de l'outil nommé "Desktop" est de présenter graphiquement les données et les outils qui sont à la disposition des développeurs et de permettre à ceux−ci d'appliquer ces outils sur ces données.

Le Desktop a en outre un rôle de gestionnaire de session. Son activation entraîne la création d'une session. Tout outil qu'il active est placé dans la même session que lui. Enfin, sa terminaison provoque la terminaison des outils appartenant à la même session que lui.

Les coordinations dans lesquelles le Desktop accepte d'intervenir ne varient pas durant son exécution. Un seul état de coordination (outre son état d'activation) est donc suffisant pour décrire ses coordinations. Dans cet état, il est spécifié que :

- lorsqu'un développeur demande au Desktop d'éditer, de compiler ou de mettre au point un composant donné, une requête est émise, spécifiant que l'outil recevant la requête doit être un nouvel outil, placé dans la même session que le Desktop,
- lorsqu'un développeur demande au Desktop d'enregistrer la validation d'un composant qu'il vient de développer (ou de visualiser la liste des composants déjà

(2) Le lecteur intéressé trouvera plus d'information sur ces outils dans [27] .

validés), une requête est émise vers l'outil de nom Validation_Mgr (décrit plus loin) qui gère la liste des composants validés pour l'équipe de développement,

- lorsqu'un développeur ferme un Desktop, une notification est émise vers l'ensemble des outils appartenant à l'équipe de développement.

GuideComp

L'outil guideComp est un compilateur qui a pour rôle de produire, à partir d'un code source écrit en langage Guide, un code exécutable. Lorsque cette production n'est pas possible en raison d'erreurs dans le code source, guideComp affiche graphiquement les erreurs et leurs causes(3). Pour permettre aux développeurs de cibler rapidement ces erreurs et de les corriger sur le champ, il se coordonne avec un éditeur. Toute sélection d'une erreur affichée par le compilateur entraîne la visualisation automatique de la ligne comportant cette erreur dans le code source affiché par l'éditeur. Inversement, la demande de recompilation du code source effectuée depuis cet éditeur est automatiquement traitée par le compilateur avec qui il est coordonné si celui−ci est toujours actif.

guideComp est activé pour servir une requête de compilation donnée. Une fois cette requête servie, il reste actif pour servir de nouvelles requêtes de compilation. Néanmoins, il refuse de traiter plusieurs requêtes en parallèle, et lorsqu'une requête émise n'est servie par aucun compilateur déjà actif, une nouvelle activation du compilateur a lieu.

Pour respecter ce fonctionnement, guideComp possède deux états de coordination, l'un représentant l'état "occupé", l'autre l'état "prêt à servir". En outre, pour que la coordination avec un éditeur fonctionne correctement, il établit une relation entre lui−même et l'éditeur avec qui il est coordonné. Ainsi, d'autres éditeurs actifs et coordonnés avec d'autres compilateurs ne risquent pas de recevoir des requêtes qui ne leur sont pas adressées.

Emacs

L'outil de nom emacs est un éditeur (bien connu), qui est utilisé :

- par les développeurs pour éditer un composant,
- par le compilateur pour visualiser les erreurs de compilation,
- par le metteur au point pour visualiser le pas à pas de l'exécution.

Ces diverses utilisations impliquent de contrôler les coordinations de l'éditeur pour que celui−ci ait un comportement cohérent.

Nous utilisons deux états de coordination. Dans le premier, l'éditeur accepte de servir les requêtes d'édition en provenance de tous les outils de la session (par exemple en provenance d'un browser de types et de classes). Dans le deuxième, l'éditeur n'accepte que les requêtes en provenance du compilateur et / ou du metteur au point avec qui il est coordonné. Pour bien distinguer ce compilateur et ce metteur au point des autres compilateurs et metteurs au point, une relation est établie entre l'éditeur et le compilateur, de même qu'entre l'éditeur et le metteur au point.

---

(3) En réalité, guideComp est interfacé par un outil indépendant (appelé "Browser" d'erreurs), qui se charge de l'appel à guideComp et qui présente graphiquement les résultats de compilation.

Thésée

L'outil nommé thésée [65] est un metteur au point pour les applications écrites en langage Guide. Sur la demande du développeur, le pas à pas d'exécution suivi par ce metteur au point peut être visualisé par un éditeur. Cet éditeur, coordonné avec le metteur au point, donne la possibilité d'ajouter et de retirer des points d'arrêts dans le code source.

A la différence de l'outil guideComp, Thésée veut distinguer l'état dans lequel il est coordonné avec un éditeur de l'état dans lequel il ne l'est pas, afin de n'envoyer des requêtes à l'éditeur que si nécessaire.

Deux états de coordination sont donc utilisés (avec / sans éditeur). Si la requête de mise au point reçue par Thésée provient d'un éditeur, alors celui−ci est directement coordonné avec Thésée. Une relation est établie entre les deux outils.

En revanche, si la requête de mise au point provient du Desktop, alors Thésée se trouve dans l'état "sans éditeur". A la demande du développeur, Thésée peut ensuite émettre une requête d'édition, qui le fera changer d'état. Une fois dans ce nouvel état, la fermeture de l'éditeur par le développeur le fait retourner dans l'état précédent.

Dev_observer

L'outil nommé Dev_observer conserve l'historique des outils activés et désactivés par les développeurs. Un Dev_observer est affecté à chaque équipe de développeurs. Tout outil de développement, lorsqu'il est activé (resp. désactivé), émet une requête d'enregistrement de son activation (resp. désactivation) vers cet outil.

Notons qu'un outil n'a pas à se préoccuper de savoir si le Dev_observer est déjà actif ou non. La requête qu'il émet l'est dans un mode entraînant l'activation du Dev_observer si nécessaire.

Le Dev_observer est donc automatiquement activé dès qu'un premier outil est activé par l'un des développeurs d'une équipe de développeur. Il reste ensuite actif jusqu'à la désactivation du dernier outil actif appartenant à un développeur de l'équipe qu'il gère.

Validation_mgr

L'outil nommé Validation_mgr permet aux développeurs de connaître les composants que chacun a validés. Un développeur peut enregistrer la validation d'un composant qu'il vient de développer et peut connaître la liste des composants validés à un instant donné. Dans le scénario, un Validation_mgr est affecté à chaque équipe de développeurs.

IV.1.3 État de l'Arbre Instancié Effectif

L'état de l'AIE à l'instant où commence le scénario est illustré ci−après. Cet état montre qu'un outil de nom Desktop est actif et appartient à une session appartenant elle−même à un développeur de nom Michel. Un outil de nom Dev_observer est également actif et appartient à l'équipe de développeurs de nom 'Guide2'.

L’outil Desktop et l’outil Dev_observer sont tous deux dans leur unique état de coordination.

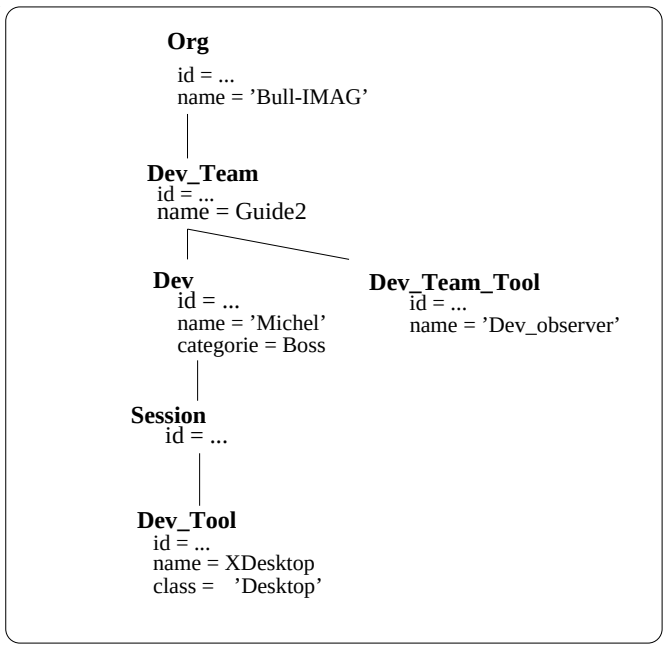


Fig. 4.2 : AIE_0

IV.2 Le scénario

Le scénario est composé des 11 actions présentées ci−après.

(Action a)

Le développeur Jacques lance l’outil `Desktop` depuis le langage de commande de l’environnement. Il donne la commande suivante :

```
REQ desktop
TO /Org.name = ’Bull−IMAG’/Dev_Team.name = Guide2/ Dev.name
   =’Jacques’/Session/NEW   Dev_Tool.name=’Desktop’
```

Dans l’état d’activation du `Desktop`, on trouve la déclaration suivante :

```
ON REQ desktop
FROM *
DO COMMAND "/usr/bin/Desktop −rf Desktop.P"
   CHANGE_CS CS_Desktop
```

L’AIE_0 devient l’AIE_1.

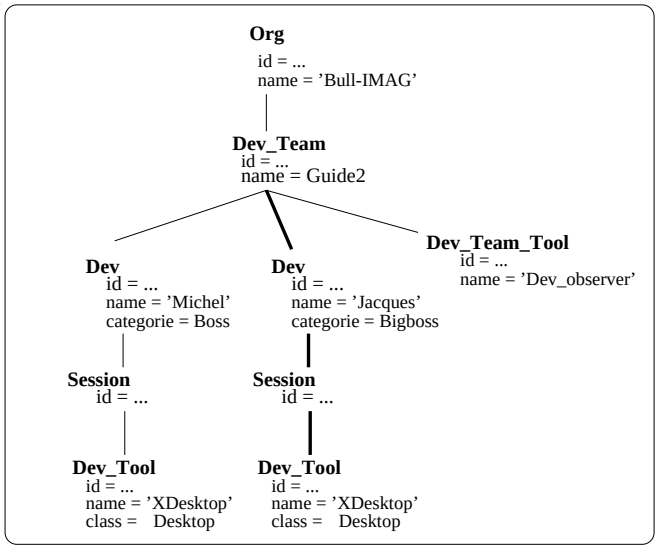


Fig. 4.3 : AIE_1

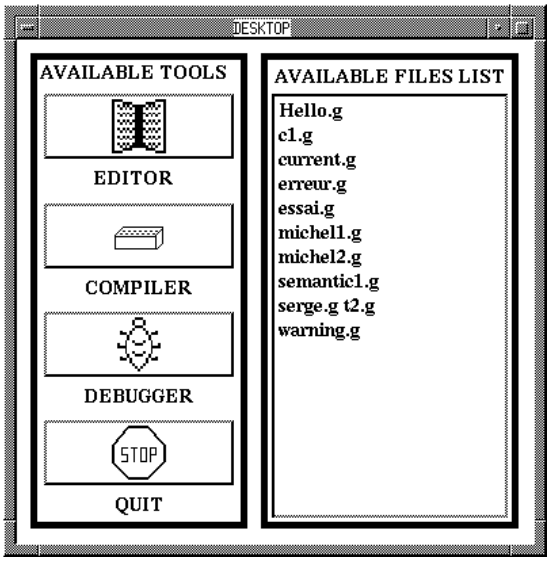


Fig. 4.4 : Une vue du Desktop par copie d'écran

(Action b)

Dès qu'il commence à s'exécuter, tout outil émet un signal qui entraîne l'émission d'une requête vers un outil de nom Dev_observer. Si aucun outil actif correspondant à la désignation donnée ne veut servir la requête, un outil Dev_observer est automatiquement activé. Dans notre cas, il existe un Dev_observer actif acceptant de servir la requête. L'AIE_1 ne change donc pas.

Dans l'état de coordination courant du Desktop, on trouve :

```
ON SIGN start
 DO  REQ_ASYNC_AR    registrate_activation  ('Desktop')
```

```
        TO  ../../../Dev_Team_Tool.name='Dev_observer'
```

Dans l'état de coordination courant du Dev_observer, on trouve :

```
ON REQ   registrate_activation (IN tool_name : String)
   FROM  ../Dev/Session/Dev_Tool
DO nb_active_tool := nb_active_tool + 1;
CALL  reg_activation  (tool_name);
```

(Action c)

Le développeur Jacques demande à son Desktop d'éditer le composant `essai`. Le Desktop signale cette demande. Dans son état de coordination, ce signal engendre l'émission d'une requête d'édition, qui entraîne l'activation d'un éditeur. L'AIE_1 devient l'AIE_2.

Dans l'état de coordination courant du Desktop, on trouve :

```
ON SIGN user_ask_edit (IN c : T_comp_name)
DO  REQ_ASYNC_AR  edit  (c)
    TO  ../NEW  Dev_Tool.name='emacs'
```

Dans l'état d'activation de l'éditeur emacs, on trouve :

```
ON REQ edit (IN c : T_comp_name)
DO COMMAND   "lemacs –edit <c> "
   CHANGE_CS     CS_edition
```

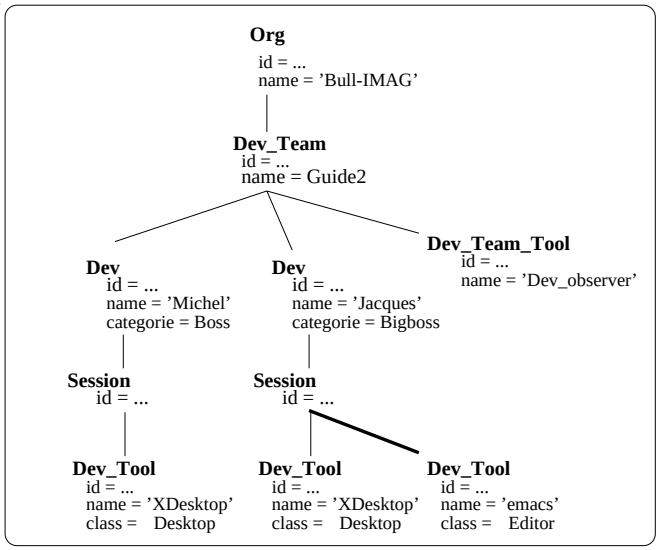


Fig. 4.5 : AIE_2

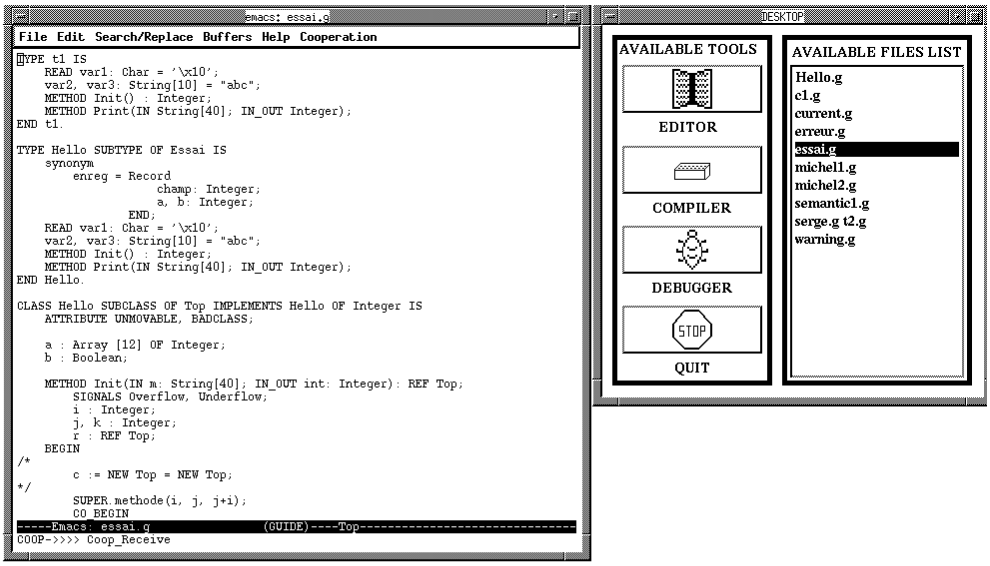


Fig. 4.6 : Le développeur demande l'édition de essai.g au Desktop

(Action d)

Le développeur Michel demande à son Desktop de fermer la session. Le Desktop signale cette demande avant de se fermer. Dans son état de coordination, ce signal engendre l'émission d'une notification vers l'ensemble des outils de son équipe. Tous les outils travaillant dans la même session que ce Desktop reçoivent cette notification et se ferment à leur tour.

Dans l'état de coordination courant du Desktop, on trouve :

```
ON SIGN close
DO    NOTIF   close
      TO   ../../../*
```

Dans l'état de coordination courant de tout outil de type Dev_Tool, on trouve :

```
ON NOTIF close
FROM  ../Dev_Tool.class=Desktop
DO     CALL   close
```

L'AIE_2 devient l'AIE_3.

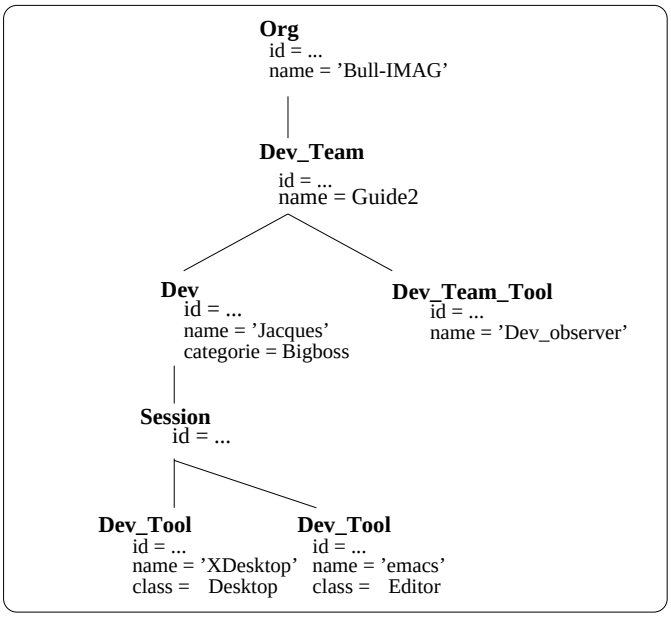


Fig. 4.7 : AIE_3

(Action e)

Le développeur Jacques veut compiler le composant édité `essai`. Il demande la compilation à l'éditeur, qui signale cette demande. Ce signal engendre l'émission d'une requête de compilation et un changement d'état de coordination pour l'éditeur. Ainsi ce dernier ne recevra plus que les requêtes d'édition provenant du compilateur. Il passe de l'état `cs_edition` à l'état `cs_edition_busy`.

Pour ne pas recevoir les coordinations provenant d'un autre compilateur, une relation de nom `collaboration` est établie entre l'éditeur et le compilateur. La mise à jour de la variable `compiler` de l'état `cs_edition_busy` permet à l'éditeur de savoir avec qui il est lié par la relation `collaboration` (compilateur et / ou metteur au point dans notre scénario) (voir action (h)).

Aucun compilateur correspondant à la désignation donnée par l'éditeur n'étant actif, un compilateur est automatiquement activé. L'AIE_3 devient l'AIE_4.

Dans l'état de coordination courant de l'éditeur, on trouve :

```
ON SIGN  user_ask_compile (IN c : T_comp_name)
DO REQ_ASYNC_AR   compile (c)
   TO ../Dev_Tool.class=Compiler  NEW_REL   'collaboration'
   cs_edition_busy.compiler := 1;
   CHANGE_CS   CS_edition_busy
```

Dans l'état d'activation de l'outil compilateur, on trouve :

```
ON REQ   compile (IN comp_ref : Integer)
```

```
    FROM  /Org/Dev_Team/Dev/Session/Dev_Tool.class=Editor
          NEW_REL  'collaboration'
DO COMMAND  "guideCompile -p5  <c>"
   CHANGE_CS  CS_compilation
```

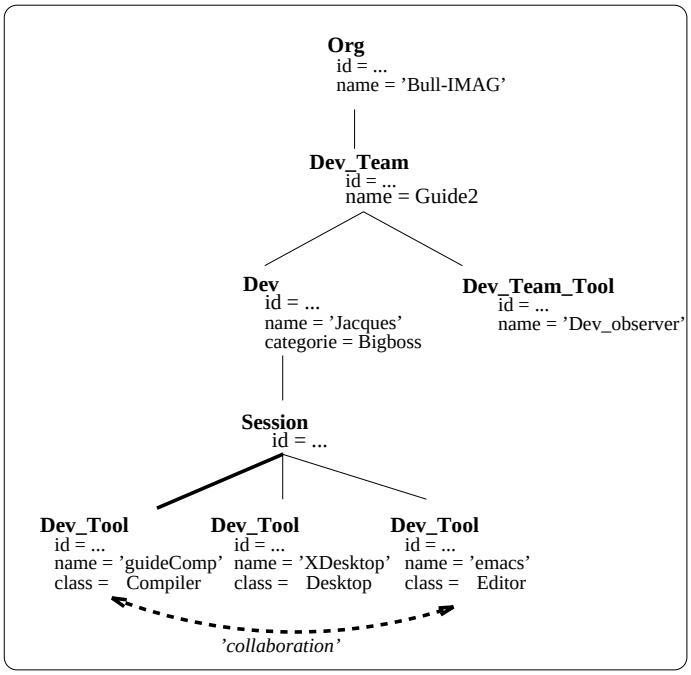


Fig. 4.8 : AIE_4

(Action f)

Le compilateur, après avoir compilé `essai`, affiche les résultats dans une fenêtre graphique, listant les erreurs et leurs causes. Lorsque le développeur sélectionne une erreur dans cette fenêtre, la ligne comportant l'erreur est automatiquement imprimée en inverse−vidéo dans l'éditeur. Pour ce faire, chaque sélection entraîne l'émission d'une requête demandant l'affichage en inverse vidéo de la ligne d'erreur à l'éditeur. Si, entre−temps, l'éditeur a été fermé par un développeur, le compilateur n'a pas à s'en préoccuper car un nouvel éditeur, à nouveau lié à lui par un lien de nom `collaboration`, est automatiquement activé. Dans le scénario, l'éditeur n'a pas été fermé, et l'AIE reste donc identique.

Dans l'état de coordination courant du compilateur, on trouve :

```
ON SIGN user_select_err (IN c:T_comp_name, IN line: Integer)
DO REQ_ASYNC_AR show_line (c, line, RED)
   TO  ../Dev_Tool.class=Editor   REL  'collaboration'
```

Dans l'état de coordination courant de l'éditeur, on trouve :

```
ON REQ show_line (IN c : T_comp_name, IN line : Integer,
                  IN color : Integer)
```

```
   FROM  ../Dev_Tool.class=Compiler  EX_REL  'collaboration'
DO CALL  show_line(c,  i,  color)
```

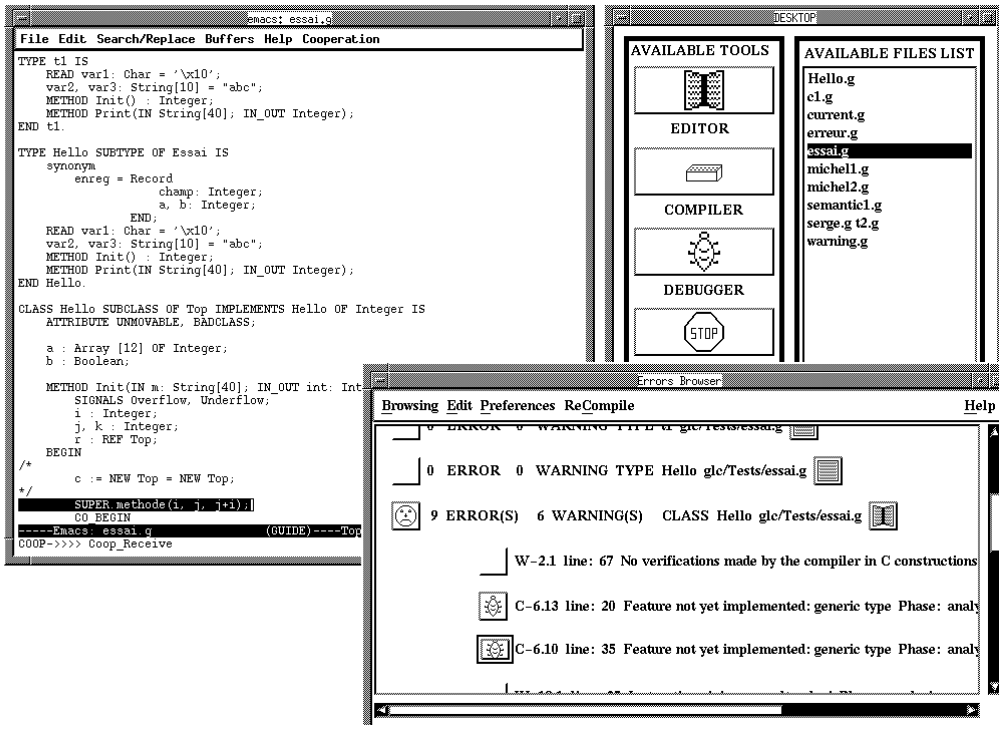


Fig. 4.9 : La selection de l'erreur C6.10 dans le compilateur provoque l'affichage en inverse vidéo de cette ligne dans l'éditeur

(Action g)

Le développeur Jacques demande la mise au point du composant `essai` à l'éditeur. Le signal émis par l'éditeur engendre l'émission d'une requête de mise au point.

Un metteur au point est automatiquement activé pour servir la requête. L'AIE_4 devient l'AIE_5.

Le développeur peut ensuite poser ou retirer des points d'arrêts depuis l'éditeur. En outre, durant l'exécution, l'éditeur visualise en inverse−vidéo la ligne courante d'exécution.

Dans l'état de coordination courant de l'éditeur, on trouve :

```
ON SIGN user_ask_debug (IN c : T_comp_name)
DO REQ_ASYNC_AR debug (c)
   TO ../Dev_Tool.class=Debugger  REL   'collaboration'
   cs_edition_busy.debugger  :=  1;

ON   REQ show_line (IN c: T_comp_name, IN line:Integer,
         IN color : Integer)
    FROM ../Dev_Tool.class=Debugger  EX_REL   'collaboration'
DO  CALL  show_line (c,  line,  color);
```

Dans l'état de coordination du metteur au point activé, on trouve :

```
ON SIGN  new_step (IN c: T_comp_name, IN line : Integer)
DO REQ_ASYNC_AR   show_line (c, line, BLACK)
   TO ../Dev_Tool.class=Editor  EX    EX_REL  'collaboration'
```

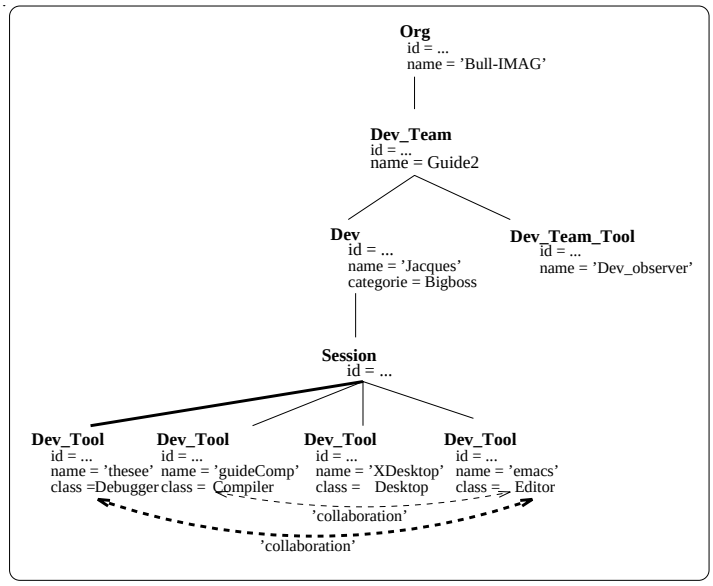


Fig. 4.10 : AIE_5

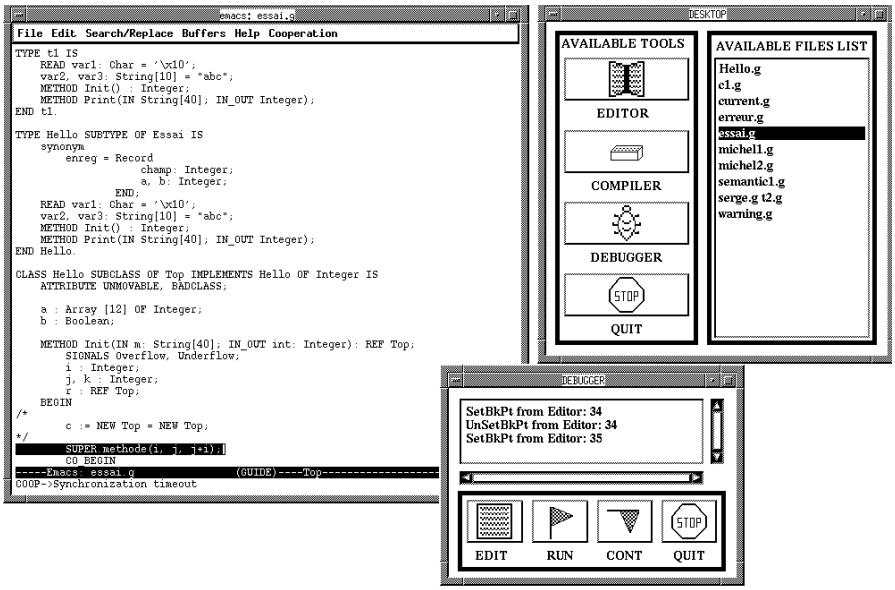


Fig. 4.11 : Le développeur peut déposer et retirer des points d'arrêts depuis l'éditeur

(Action h)

Le développeur Jacques ferme le metteur au point et le compilateur. Avant de se fermer, ces derniers signalent leur terminaison. Chaque signal engendre l'émission d'une notification. L'éditeur recevant ces notifications repasse dans un état de coordination dans lequel il accepte de servir les requêtes d'édition en provenance de tous les outils de la session. L'AIE_5 devient l'AIE_6.

Dans l'état de coordination courant du compilateur, on trouve :

```
ON SIGN close
DO NOTIF  close
   TO   ../../../*
```

Dans l'état de coordination courant de l'éditeur, on trouve :

```
ON NOTIF close
   FROM   ../Dev_Tool.class=Compiler
          EX_REL  'collaboration'
DO   compiler := compiler − 1;
     IF  (compiler = 0) AND (debugger = 0)
     THEN  CHANGE_CS  CS_edition
     END
```

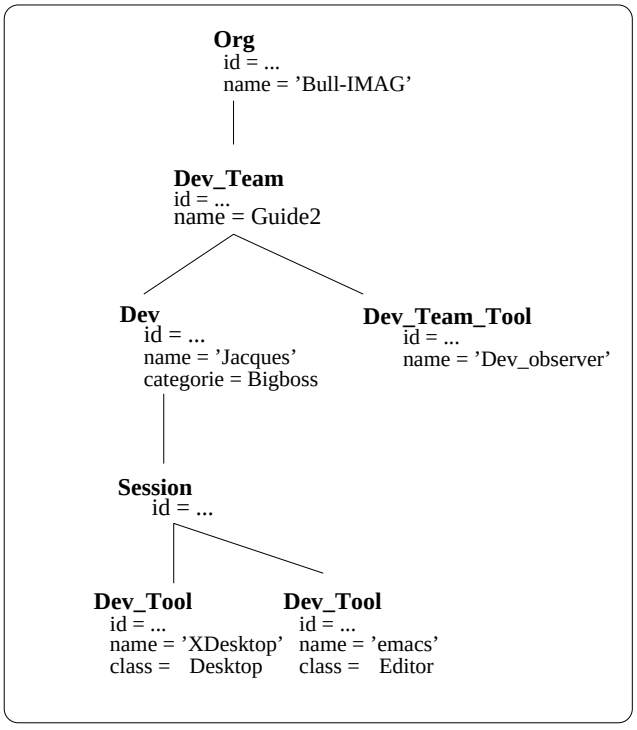


Fig. 4.12 : AIE_6

(Action i)

Le développeur Jacques demande, au travers du Desktop, d'enregistrer la validation du composant c. Une requête est émise vers le Validation_mgr et ce dernier n'étant pas déjà actif est automatiquement activé. L'AIE_6 devient l'AIE_7.

Dans l'état de coordination du Desktop, on trouve :

```
ON SIGN user_ask_for_validation (IN c : T_comp_name)
DO REQ_ASYNC_AR registrate_validation (c)
    TO   ../../../Dev_Team_Tool.name='Validation_mgr'
```

Dans l'état d'activation du Validation_mgr, on trouve :

```
ON   REQ registrate_validation (IN c : T_comp_name)
     FROM  ../*
DO    COMMAND   'Validation_mgr'
     CHANGE_CS  CS_validation
     CALL   validate  (c)
```

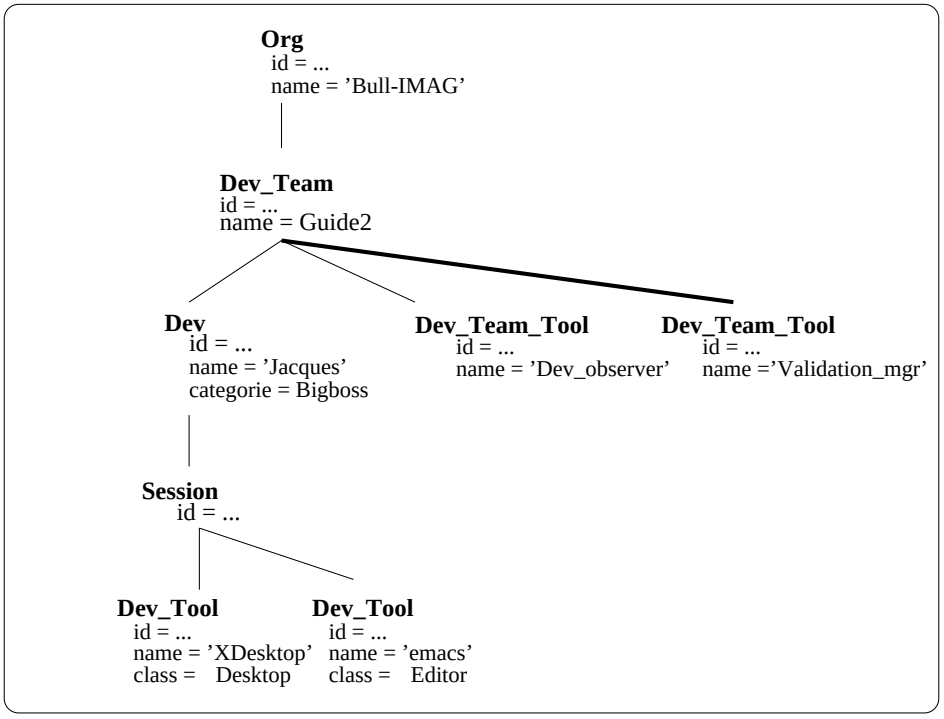


Fig. 4.13 : AIE_7

(Action j)

Le développeur Jacques veut obtenir, au travers du Desktop, la liste de l'ensemble des composants validés par les développeurs de son équipe. Une requête est à nouveau émise vers le Validation_mgr.

Dans l'état de coordination courant du Desktop, on trouve :

```
ON SIGN user_ask_validation (OUT c_list : T_comp_name_list)
```

```
DO  REQ_SYNC   get_validation_list  (c_list)
    TO   ../../../Dev_Team_Tool.name='Validation_mgr'
```

Dans l'état de coordination courant du Validation_mgr, on trouve :

```
ON  REQ   get_validation_list(OUT  c_list:T_comp_name_list)
   FROM      ../*
DO CALL  get_validation  (c_list)
```

(Action k)

Le développeur Jacques a terminé son travail. Il ferme le Desktop, ce qui provoque l'émission d'une notification aux outils placés dans l'équipe de développeurs. L'ensemble des outils placés dans la session du Desktop se ferment à leur tour. L'outil Dev_observer, qui reçoit les notifications de fermeture des outils, sait qu'aucun outil de type Dev_tool n'est encore actif. Il envoie une notification nommée `no_more_active_tool` au Validation_-mgr qui se ferme à son tour. L'AIE se désinstancie, car l'ensemble des nœuds qui n'ont plus de fils se détruisent.

Dans l'état de coordination du Dev_observer, on trouve :

```
ON NOTIF close
   FROM  ../Dev/Session/Dev_tool
DO nb_active_tool := nb_active_tool − 1;
   IF  nb_active_tool = 0
   THEN   CALL close
   END


ON SIGN close
DO  NOTIF   no_more_active_tool
   TO   ../Dev_Team_Tool.name='Validation_mgr'
```

Dans l'état de coordination du Validation_mgr, on trouve :

```
ON  NOTIF   no_more_active_tool
   FROM   ../Dev_Team_Tool.name='Dev_observer'
DO CALL  close
```

IV.3 Conclusion

Le scénario présenté dans cette section a eu pour but de concrétiser la présentation du langage Indra donnée dans le chapitre précédent. Les aspects suivants ont été montrés.

1. L'utilisation de l'espace de coordinations global fourni par le modèle de coordination choisi (cfII.3.3.3). Prenons par exemple l'action (f).

   Le programme exécutable de l'éditeur signale l'événement `user_ask_compile` lorsque le développeur sélectionne, dans un menu donné par l'éditeur, le bouton `Compilation`.

Le programme exécutable du compilateur comprend une méthode exécutable (méthode réaction) de nom `guideCompile`.

Le lien entre l'événement émis par l'éditeur et la méthode réaction du compilateur est fait au niveau de l'expression de la coordination transmise : dans l'état de coordination de l'éditeur, le signal émis engendre l'émission d'une requête de nom `compile`, et dans l'état de coordination du compilateur, la réception de la requête de nom `compile` entraîne l'appel à la méthode réaction `guideCompile`.

2. L'utilisation et le fonctionnement de la notion d'état de coordination pour gérer l'évolution des coordinations d'un outil.

   Les coordinations de l'éditeur, du compilateur et du metteur au point sont exprimées en utilisant ce concept.

3. L'utilisation et le fonctionnement du service de désignation des outils.

   Différents types de désignation sont utilisés dans le scénario : utilisation de l'organisation arborescente au travers d'adressages absolus et relatifs, utilisation des attributs, utilisation des relations (entre l'éditeur, le compilateur et le metteur au point).

Chapitre V

Un environnement d'exécution pour le langage Indra

Ce chapitre présente la mise en œuvre de l'environnement d'exécution pour le langage Indra. Après avoir introduit l'architecture logicielle générale de cet environnement, le fonctionnement des principaux composants est décrit.

V.1 Architecture logicielle générale

V.1.1 Introduction

De prime abord, on peut distinguer trois parties intervenant dans l'environnement d'exécution (Fig. 5.1).

1. La partie *Bibliothèque de fonctions de coordination*, qui permet à un outil de faire appel aux fonctions fournies par le mécanisme de coordination.
2. La partie *Système support*, qui correspond au système d'exploitation sur lequel est implémenté le mécanisme de coordination.
3. La partie *Mécanisme de coordination*, qui implémente les fonctions de coordinations pour des outils qui s'exécutent sur des sites éventuellement distants. Ces fonctions sont paramétrées par les *définitions Indra* (définitions des états de coordination, des états d'activation et de l'organisation) des outils à coordonner.

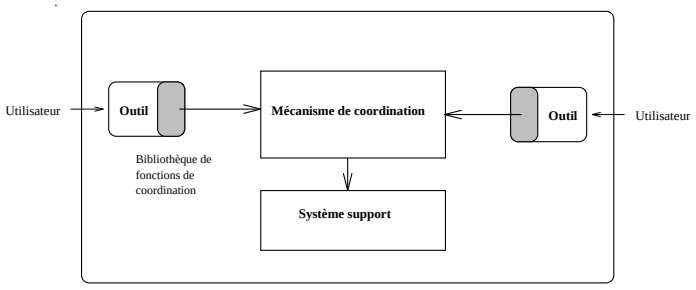


Fig. 5.1 : Architecture logicielle générale de l'environnement d'exécution d'Indra

V.1.2 Hypothèses

La première hypothèse provient de limitations (énoncées en II) effectuées au niveau de notre travail : nous nous plaçons au niveau d'un environnement de développement de moyenne envergure (niveau *groupe*), et le mécanisme de coordination est prévu pour fonctionner avec des outils de gros grain (applications). Cette hypothèse permet d'établir :

1. la fréquence moyenne de coordinations (cf. II.2),
2. des délais de communication bornés entre les machines supportant l'environnement de développement, connectées par un réseau local.

La deuxième hypothèse provient d'une observation du comportement des outils : les coordinations entre outils sont d'autant plus fréquentes que ceux−ci sont proches dans l'organisation. Par exemple, deux outils travaillant dans une même session se coordonnent plus fréquemment que deux outils travaillant dans des sessions différentes, pour des développeurs distincts.

Ces deux hypothèses ont été particulièrement exploitées pour la mise en œuvre du service de désignation des outils (cf. V.5.2).

V.1.3 Objectifs et principes

Notre premier objectif a été de concevoir un mécanisme de coordination ayant des performances d'exécution satisfaisantes au regard des besoins des développeurs : ces derniers ne doivent pas avoir l'impression que les coordinations entre les outils engendrent des temps d'attente notables.

Les deux principes suivants ont été adoptés à cet égard :

- les définitions Indra des outils sont compilées et non interprétées,
- l'environnement d'exécution doit être décentralisé pour autoriser une exécution parallèle des coordinations.

Notre deuxième objectif a été de supporter l'évolution des définitions Indra (évolution des états de coordination, des états d'activation et de l'organisation arborescente des outils) en cours de développement.

Cet objectif et le précédent étant quelque peu antagonistes, nous avons opté pour un compromis qui consiste à utiliser un mécanisme de *liaison tardive* entre les définitions Indra compilées et le système sous−jacent.

Notre troisième objectif a été de fournir un environnement d'exécution permettant de coordonner des outils écrits dans des langages de programmation divers. La partie Bibliothèque de fonctions de coordination doit donc être multi−langages. La gestion des

différences de représentation des données entre les machines supports de l'environnement doit également être assurée[1].

V.1.4 Architecture logicielle générale

L'architecture logicielle générale est illustrée ci−après (Fig. 5.2). La partie *Mécanisme de coordination* (cf. fig Fig. 5.1) se décompose en deux sous−parties :

1. la partie nommée *Partie compilée*, qui comprend les traductions compilées des définitions Indra,
2. la partie *Machine d'exécution*, qui implémente les fonctions du mécanisme de coordination paramétrées par la *Partie Compilée*.

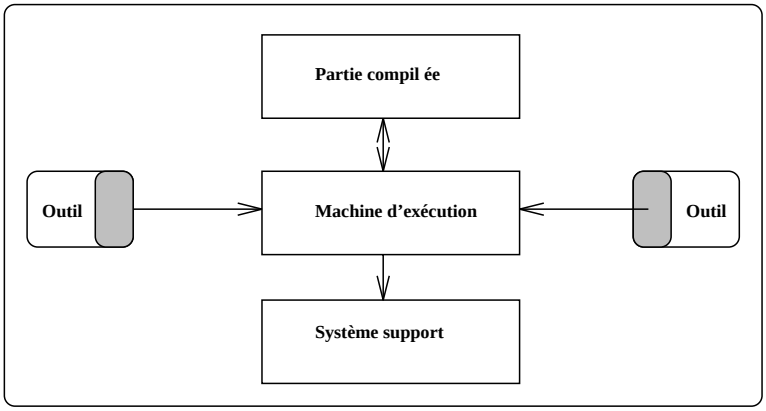


Fig. 5.2 : Architecture logicielle générale de l'environnement d'exécution d'Indra

V.2 Système support

Le système support choisi est le système réparti et orienté objet Guide car ce dernier fournit des fonctions très adaptées aux besoins de l'environnement d'exécution. Seules les caractéristiques principales de Guide sont ici évoquées. Celles−ci concernent d'une part son modèle d'objet, et d'autre part son modèle d'exécution. Une description complète de ces caractéristiques se trouve dans [6].

(1) Cet aspect repose sur la gestion offerte par le système support de l'environnement d'exécution. Le système choisi (Guide, cf. V.2) doit, à terme, gérer les hétérogénéités des machines d'exécution.

V.2.1 Le modèle d'objets de Guide

Le modèle d'objet de Guide permet de structurer une application à base d'objets. Un *objet* encapsule des données et des opérations de manipulation de celles−ci. Un *type* décrit un comportement partagé par tous les objets de ce type. Ce comportement est défini par la signature des méthodes. Une *classe* définit une implémentation spécifique d'un type d'objets : elle contient une description interne de la représentation des données ainsi que du code des méthodes.

La notion de *sous−typage* permet d'enrichir ou modifier un type par la définition de nouvelles méthodes ou par surcharge de celles existantes. De manière identique, il existe une hiérarchie de sous−classes limitées actuellement à un héritage simple.

Les objets sont *persistants* et sont désignés par des identificateurs appelés références. Une référence est indépendante de la localisation de l'objet qu'il désigne. Un objet peut contenir des références à d'autres objets, permettant de construire des objets composés.

Ce modèle d'objet est accessible aux utilisateurs par l'intermédiaire d'un langage (également nommé Guide) qui permet de manipuler tous les concepts définis par ce modèle.

V.2.2 Le modèle d'exécution de Guide

Le modèle d'exécution fourni par Guide permet de contrôler l'exécution concurrente et répartie des activités d'une application, tout en préservant la transparence de la localisation.

Un *domaine* définit une machine virtuelle distribuée qui regroupe dans un espace d'adressage commun un ensemble d'objets et un ensemble d'entités de contrôle (*activités*) qui agissent sur ces objets. L'exécution d'une activité dans un domaine consiste en une succession d'appels de méthode sur ces objets. Les objets Guide sont donc *passifs*, les flots d'exécution étant externes aux objets et représentés par le concept d'activité.

Pour exprimer la concurrence, une activité peut créer à tout moment un ensemble d'activités Guide qui sont exécutées en parallèle au sein du même domaine. L'activité mère est suspendue jusqu'à ce qu'une condition de terminaison (e.g. terminaison de la première activité fille ou de toutes les activités filles, etc.) soit vérifiée.

Le partage d'objets entre activités du même domaine ou de domaines différents constitue le seul moyen de communication et de synchronisation. Des contraintes de synchronisation peuvent être associées à chaque méthode d'un objet partagé.

V.3 Fonctions de coordination fournies aux outils

Les fonctions fournies à un outil par la partie *Bibliothèque de fonctions de coordination* (cf. Fig. 5.2) sont essentiellement les suivantes.

1. Initialiser la coordination

   Étant donné le nom de l'outil, cette fonction décide des états de coordination qui lui seront associés et le place dans son état de coordination initial. Cette décision est

donnée par les liaisons (nom d'outil − liste de ses états de coordination) exprimées dans le langage Indra.

2. Signaler un événement

   Cette fonction exécute les instructions de la partie opérative associée à l'événement signalé dans l'état de coordination courant de l'outil.

3. Recevoir une coordination

   Cette fonction demande à recevoir une coordination (requête ou notification) qui est acceptée dans l'état de coordination courant de l'outil. Si une telle coordination peut être reçue, les instructions de la partie opérative associée à cette coordination dans l'état de coordination courant de l'outil sont exécutées.

4. Terminer la coordination

   Cette fonction supprime l'outil de l'environnement de coordination : celui−ci ne peut plus être coordonné avec les autres outils.

La dernière instruction d'une partie opérative pouvant spécifier un appel à une méthode *réaction* de l'outil (cf. III.2.2.2), les fonctions 2 et 3 retournent comme résultat la description de cette méthode (nom, paramètres effectifs).

Les fonctions décrites ci−dessus sont des fonctions de base. Pour faciliter leur utilisation, des adaptations peuvent être faites en fonction des caractéristiques des différents langages de programmation et de leurs applications. Par exemple, le langage C permet de manipuler des méthodes par leurs adresses. L'appel aux méthodes réactions peut alors être automatiquement géré *dans* les fonctions de coordination, si les liaisons (nom de méthode *réaction*, adresse de méthode *réaction*) ont été préalablement spécifiée par l'outil.

Pour l'instant, seules des fonctions pour les langages C (C++) et Guide sont fournies.

V.4 Partie compilée

Les éléments paramétrables du mécanisme de coordination sont les définitions Indra des états d'activation et de coordination des outils et la définition du schéma d'organisation de ceux−ci (appelé Arbre Schématisé).

Ces descriptions sont compilées en classes Guide, dont les superclasses et les types sont prédéfinis par la machine d'exécution.

Toute description d'un état de coordination est compilée en une classe Guide de type *Indra_CoordState* et de nom *Indra_CoordState_<StateName>*. Lors de l'exécution d'un outil, chacun des états de coordination par lesquels il est susceptible de passer est représenté par une instance de la classe générée pour cet état (Fig. 5.3). L'interface fournie par ces instances comprend essentiellement :

- une méthode *Mge_signal* qui étant donné un événement émis par l'outil, exécute les instructions de la partie opérative associée à cet événement dans l'état de coordination,
- une méthode *Mge_coord* qui étant donné une coordination (requête ou notification), exécute les instructions de la partie opérative associée à cette coordination dans l'état de coordination,
- une méthode *Init* qui spécifie les coordinations acceptées dans cet état de coordination.

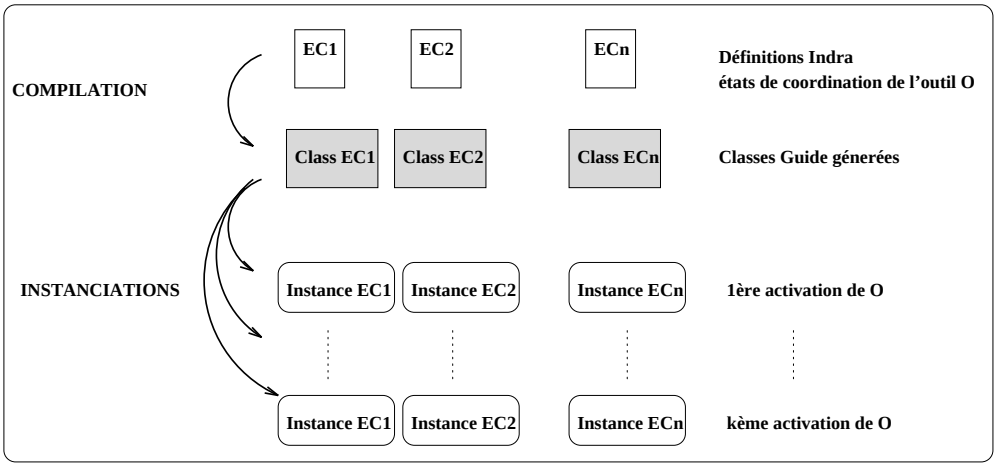


Fig. 5.3 : Compilation des états de coordination

La compilation des descriptions des états d'activation des outils produit deux sorties. Premièrement, la description d'*un* état d'activation produit une classe de type *Indra_CoordState* et de nom *Indra_CoordState_<StateName>* (comme précédemment). La méthode *Mge_coord*, étant donné une coordination qui a entraîné l'activation de l'outil, exécute les instructions qui ont été spécifiées dans la partie opérative associée à cette coordination(2).

Deuxièmement, à partir de l'*ensemble* des descriptions des états d'activation est générée une classe de type *Indra_Activation* et de nom identique. Cette classe comprend une méthode qui, étant donné une coordination, spécifie si un outil peut être activé pour servir la coordination et si tel est le cas, l'active par la commande d'activation spécifiée dans l'état d'activation de cet outil. Lors de l'exécution, seule *une* instance de cette classe est créée.

Enfin, la description du schéma d'organisation des outils est compilée en une classe Guide, de type *Indra_Organisation* et de nom identique. A la différence des descriptions précédentes, celle−ci est entièrement déclarative. La compilation se réduit donc à la définition de données constantes qui représentent, sous une forme spécifique, le schéma arborescent de l'organisation des outils (cf. III.3). Les méthodes de la classe

(2) A l'exception de la commande d'activation de l'outil, qui aura déjà été exécutée pour activer l'outil.

Indra_Organisation sont entièrement héritées d'une superclasse définie par la machine d'exécution. Elles permettent de récupérer des informations sur la définition de l'organisation des outils. Comme précédemment, lors de l'exécution, seule une instance de cette classe est créée.

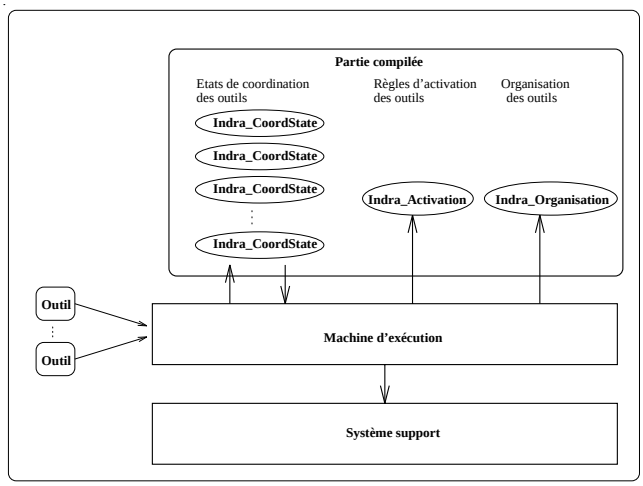


Fig. 5.4 : Classes Guide générées pour la partie compilée et instances créées.

V.5 Machine d'exécution

La machine d'exécution implémente les fonctions de coordination fournies aux outils. Deux rôles lui sont attribués. Premièrement, elle doit gérer la liaison entre un outil et ses états de coordination. Deuxièmement, elle doit implémenter les fonctions de transmission des coordinations (émission, réception, acceptation).

L'architecture de la machine d'exécution reflète ce double rôle, en étant divisée en deux parties (Fig. 5.5): la partie *Frontal de coordination* et la partie *Service de communication*.

La partie Frontal de coordination, présentée en V.5.1, est locale à un outil actif. Un frontal joue le rôle d'intermédiaire entre un outil, ses états de coordination et le service de communication. Il crée et utilise les instances d'objets de la partie compilée qui représentent les états de coordination par lesquels l'outil qu'il gère est susceptible de passer.

La partie Service de communication, présentée en V.5.2, est globale, répartie et partagée dans l'environnement d'exécution. Elle implémente les fonctions d'émission, de réception et d'acceptation de coordinations. C'est cette partie qui active les outils lorsqu'une requête doit être, ou ne peut être que, servie par un outil à activer. Pour ce faire, elle crée et utilise les instances d'objets de la partie compilée qui définissent l'organisation et les règles d'activation des outils.

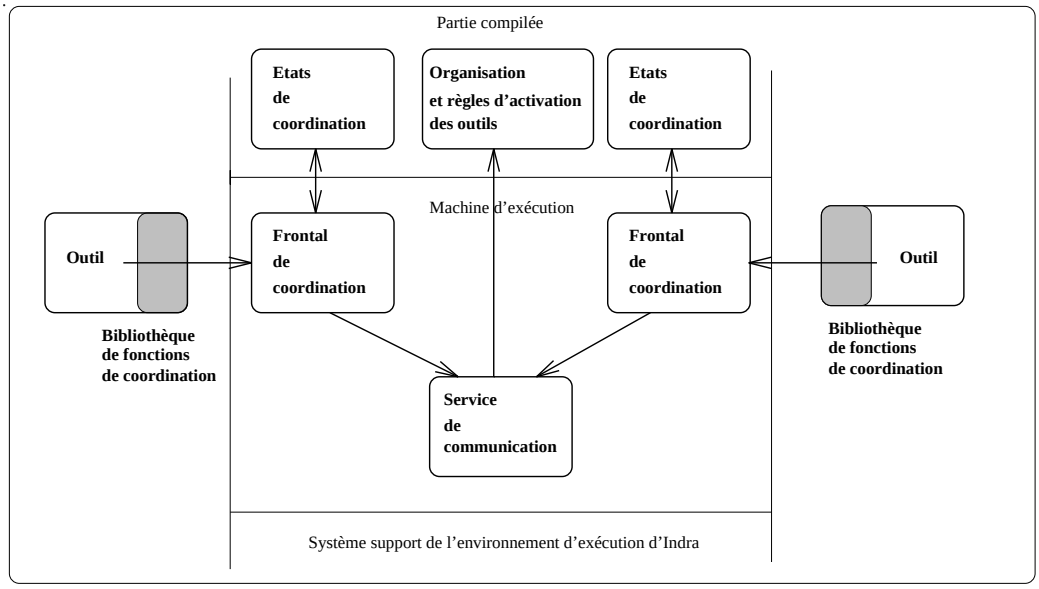


Fig. 5.5 : Architecture logicielle générale de l'environnement d'exécution d'Indra

V.5.1 Frontal de coordination

V.5.1.1 Fonctions offertes

Un objet Frontal fournit deux types d'interfaces (Fig. 5.5).

L'interface fournie à l'outil auquel il est associé est le reflet des fonctions fournies par la bibliothèque de fonctions de coordinations (cf. V.3).

L'interface fournie aux états de coordination comporte essentiellement :

- une méthode *CS_accept_coord*, qui permet de spécifier qu'une requête ou notification donnée est acceptée dans un état de coordination donné,
- une méthode *CS_send_coord*, qui permet d'émettre une requête ou une notification,
- une méthode *CS_change_state*, qui permet de changer d'état de coordination courant.

Les objets Guide étant passifs, le Frontal ne fonctionne pas comme un serveur. Ce sont des flots d'exécution "externes" qui exécutent, éventuellement en parallèle(3), les fonctions offertes par son interface. Ainsi, le fait qu'il fournisse des interfaces à deux clients différents ne pose aucun problème.

(3) La cohérence des données du Frontal est assurée par l'utilisation des mécanismes de synchronisation offerts par Guide.

V.5.1.2 Fonctionnement général

Pour mieux comprendre la dynamique générale du système, nous donnons ci−après sous forme d'algorithme le déroulement des méthodes Tool_init, Tool_signal et Tool_receive.

La méthode Tool_init, étant donné le nom de l'outil, crée l'objet qui va représenter son état de coordination initial et demande ensuite à cet objet de spécifier les coordinations qu'il accepte de recevoir. Ces acceptations sont enregistrées auprès du service de communication. La figure Fig. 5.6 illustre ce fonctionnement.

```
METHOD Tool_init ( IN Tool_name : String)
Begin
   initial_CS  :=  Indra_CoordState_<ToolActivationStateName>.New;
   initial_CS.Init;
End
```

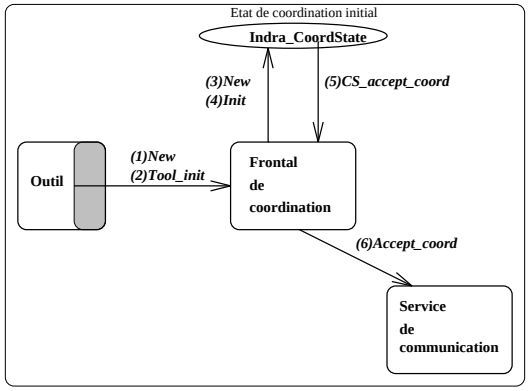


Fig. 5.6 : Fonctionnement de la méthode Tool_init

La méthode Tool_signal demande à l'état de coordination courant de l'outil d'exécuter la partie opérative associée à l'événement signalé. L'exécution de cette partie opérative peut entraîner des appels au Frontal lui demandant par exemple d'émettre une coordination ou de changer d'état de coordination. Le changement d'état entraîne des appels au service de communication, demandant l'invalidation des acceptations de coordinations émanant de l'état précédent et validant les acceptations émanant du nouvel état (Fig. 5.7).

```
METHOD Tool_signal (
                IN evt_name : String, IN evt_args : REF T_List OF T_Args,
                 OUT is_react : Boolean,
                 OUT react_name: String, OUT react_args: REF  T_List OF T_Args)
 Begin
   current_CS.Mge_signal  (evt_name,  evt_args,  is_react,  react_name,  react_args);
End
```

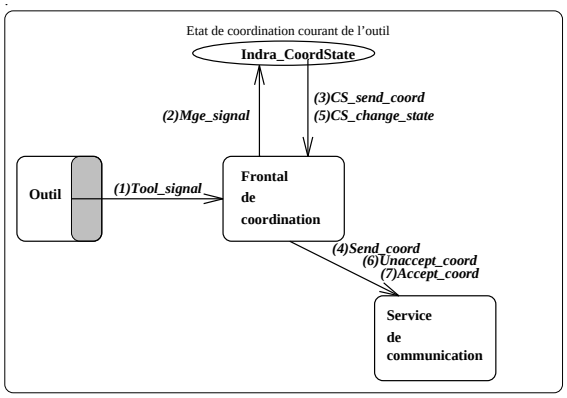


Fig. 5.7 : Illustration du fonctionnement de la méthode Tool_signal

Enfin, la méthode Tool_receive demande à recevoir une coordination au service de communication. Aucune attente active n'est engendrée car les mécanismes de synchronisation fournis par Guide permettent de bloquer un flot d'exécution, jusqu'à ce qu'une condition soit remplie.

Après réception de la coordination, l'état de coordination courant de l'outil doit exécuter la partie opérative associée à celle−ci. L'exécution de cette partie opérative peut entraîner des appels au Frontal lui demandant d'émettre une coordination (non représenté dans l'exemple donné par la figure ci−après) ou de changer d'état de coordination courant.

```
METHOD Tool_receive (
            IN max_delay : Integer, OUT is_react : Boolean,
            OUT react_name : String, OUT react_args : REF  T_List OF T_Args)
 Begin
    comm_service.Recv_coord  (max_delay,  coord);
    current_CS.Mge_coord (coord, is_react, react_name, react_args)
End
```

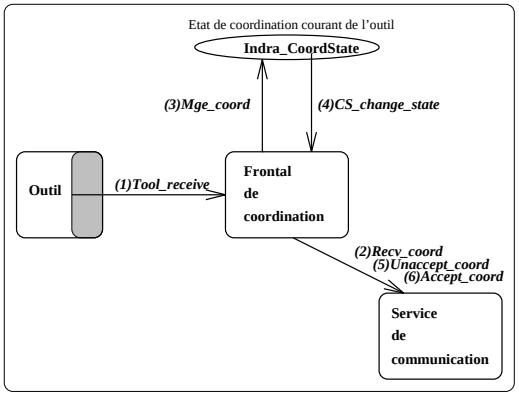


Fig. 5.8 : Fonctionnement de la méthode Tool_recv

V.5.2 Service de communication

Le service de communication transmet des coordinations entre un ensemble de clients(4).

V.5.2.1 Fonctions offertes

Les fonctions offertes par le service de communication sont essentiellement :

- la méthode *Accept_coord* (resp. *Unaccept_coord*) qui permet à un client de spécifier les coordinations qu'il accepte (resp. n'accepte plus) de recevoir ; une coordination est ici essentiellement caractérisée par son type (requête / notification), son nom et par la désignation de ses émetteurs potentiels,
- la méthode *Recv_coord* qui permet à un client de recevoir une coordination qui lui a été adressée et qui fait partie des coordinations qu'il accepte,
- la méthode *Send_coord* qui permet à un client d'émettre une coordination. Celle−ci est essentiellement caractérisée par son type (requête/notification), son mode (synchrone, asynchrone avec accusé, asynchrone), son nom, ses arguments et par la désignation de ses récepteurs potentiels.

V.5.2.2 Propriétés assurées

Le service de communication :

- respecte l'ordonnancement causal quant à la transmission des coordinations.
- détecte les pannes de sites ou de clients, mais ne les gère pas (5).

V.5.2.3 Architecture générale à objets

L'architecture générale du service de communication est représentée dans la figure Fig. 5.9.

Les éléments paramétrables de ce service sont représentés par deux instances de classes Guide au niveau de la partie compilée (nommées Indra_organization _<OrgName> et Indra_Activation) qui ont été introduites en V.4.

Ces deux instances n'étant accédées qu'en lecture par les différents flots d'exécution qui peuvent prendre place au niveau du service de communication, les accès parallèles à celles−ci n'entraînent pas de goulot d'étranglement.

---

(4) Dans l'environnement d'exécution, les clients sont les objets Frontal.

(5) Le problème des pannes repose essentiellement sur le support fourni par le système Guide. Celui−ci détecte les pannes de sites et renvoie le traitement de ces pannes au programmeur au travers d'un mécanisme d'exception [76]. Un mécanisme support pour la reprise après panne basé sur une réplication des objets est en cours d'étude [30]. Ces mécanismes supplémentaires permettront d'améliorer la fiabilité du service de communication.

A la différence des objets appartenant à la partie compilée, les objets du service de communication qui font partie de la machine d'exécution (cf. Fig. 5.5) ont besoin de se synchroniser. Pour éviter un trop fort goulot d'étranglement, nous avons décentralisé ce service en s'appuyant sur une architecture à base d'objets qui reflète l'arbre instancié effectif (AIE)(6) (Fig. 5.9). Tout nœud de l'AIE est représenté par un objet Guide. Ces objets sont des types suivantes.

- Les objets de type *Client_mgt* représentent les nœuds feuilles de l'AIE. Ils définissent les points d'entrée du service de communication : un Client_Mgt est associé à chaque client, et fournit à ce dernier les fonctions citées en V.5.1.1.

  Lorsqu'un *Client_mgt* est crée pour servir une requête donnée, il fait appel à l'objet de classe *Indra_activation* appartenant à la partie compilée pour activer l'outil adéquat.

- Un objet de type *Node_mgt* représente un nœud non feuille de l'AIE. Il fournit une méthode *Send_coord* qui, étant donnée une coordination à transmettre dont les récepteurs potentiels sont désignés par un adressage relatif partant de ce nœud, envoie cette coordination aux bons clients.

  Lorsqu'un *Node_Mgt* doit créer un nouvel objet fils de type *Node_Mgt*, il utilise l'objet de classe *Indra_organization* _<*OrgName*> appartenant à la partie compilée pour que la création soit conforme à la définition de l'organisation.

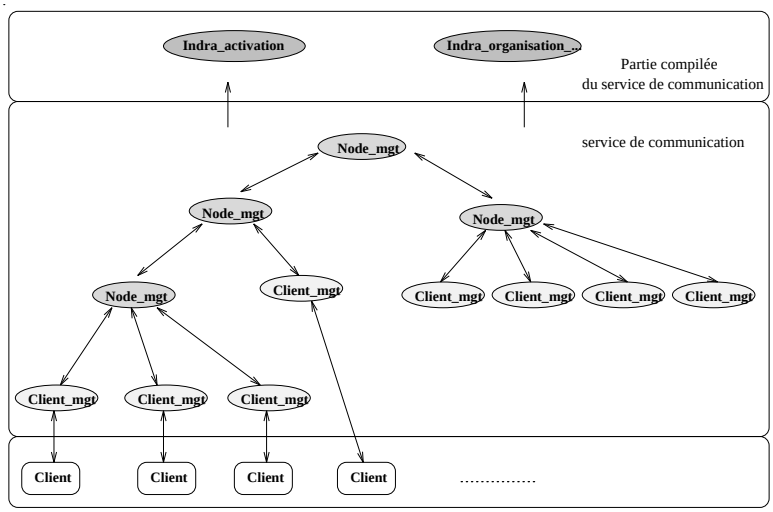


Fig. 5.9 : Objets composant le service de communication

Cette architecture, qui pourrait être contestée dans un environnement composé d'objets *actifs*, n'entraîne pas de surcharge de calcul dans un environnement composé d'objets *passifs* tel que Guide.

(6) Le concept d'AIE a été présenté en III.3.3.

V.5.2.4 Communications entre les objets et gestion de la dynamicité des clients vis−à−vis de leurs acceptations

Entre les objets qui représentent les nœuds de l'AIE, les communications sont régies par les règles suivantes.

- Toute coordination à transmettre passe par les nœuds qui sont sur les chemins d'accès aux récepteurs de cette coordination.
- Tout nœud fils est créé par son nœud père. Une opération de création prend place dynamiquement lorsqu'un père doit transmettre une coordination à un fils qui n'existe pas encore dans l'AIE.
- Toute destruction d'un nœud fils est préalablement notifiée à son nœud père. Tout nœud non feuille qui n'a plus de fils s'auto−détruit.

Les communications entre nœuds de type *Node_mgt* correspondent à des appels de méthodes directs depuis un nœud fils vers un nœud père, ou inversement.

Les communications entre un nœud de type *Node_mgt* et un nœud de type *Client_mgt* sont plus complexes. Lorsqu'un *Node_mgt* doit transmettre une coordination à un *Client_-mgt*, deux flots d'exécution sont mis en jeu : celui provenant de l'objet *Node_mgt* chargé de transmettre une coordination au client et celui provenant du *Client_mgt* demandant à recevoir une coordination. La réception d'une coordination par un client est en effet une opération asynchrone, provoquée par l'appel de la méthode *Recv_coord* par ce client. Un asynchronisme entre les deux flots d'exécution peut être requis : celui qui transmet la coordination ne veut pas forcément attendre que celle−ci soit consommée par le client.

Pour gérer cet asynchronisme d'une part, et pour gérer la dynamicité des clients vis à vis des coordinations qu'ils acceptent de recevoir d'autre part, un objet de type *Msg_bus* est utilisé comme intermédiaire entre un objet *Node_mgt* et ses objets *Client_mgt* fils. Cet objet fonctionne globalement comme un bus de message classique : il ne connaît pas la sémantique des messages et se contente de router ceux−ci entre ses clients au travers d'une technique de *pattern−matching* (cf. I.3.3).

Les demandes de réception et d'acceptation de coordination émises par un *Client_mgt* sont reçues et traitées par cet objet *Msg_bus*.

En revanche, la demande de transmission d'une coordination émise par un *Client_mgt* est reçue par l'objet *Node_mgt*. Si la coordination à transmettre reste locale (ne peut être transmise qu'à des *Client_mgt* qui sont ses fils), cet objet dépose la coordination dans son objet *Msg_bus*. Si la coordination à transmettre est globale, il renvoie la demande de transmission à son objet *Node_mgt* père.

Note Les objets de type *Msg_bus* utilisés ont toutefois une particularité qui les différencie des bus de messages usuels et qui découle du modèle d'exécution du mécanisme de

coordination proposé (cf. III.2.4). Celui−ci implique que le dépôt d'un message (coordination dans notre cas) chez un client ne soit fait qu'au moment où le client demande à recevoir un message (et non pas par anticipation). Ceci permet de garantir qu'un message *m* arrivant à un moment où il n'est pas accepté par un client *C,* pourra quand même être reçu par *C* si *C* change ses acceptations avant de demander à recevoir un message.

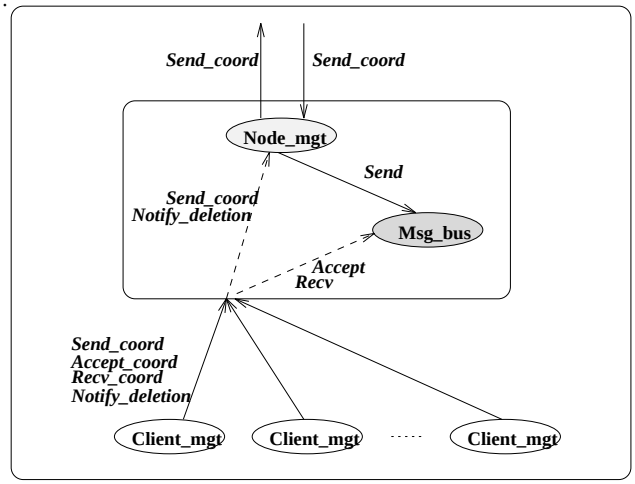


Fig. 5.10 : Communications entre un Node_mgt et un Client_mgt

V.5.2.5 Transmission d'une coordination

La majorité des actions effectuées au niveau du service de communication sont provoquées par la transmission d'une coordination.

La première transmission de coordination prend place lors de la première commande d'activation d'un outil au sein d'une organisation de nom fixé. Cette commande effectue les actions suivantes :

- vérifier si l'objet *Node_mgt* représentant la racine de l'AIE correspondant à l'organisation fixée existe. Si cet objet n'existe pas, le créer,
- appeler la méthode *Send_coord* sur cet objet. La coordination donnée en paramètre de cette méthode reflète la commande d'activation de l'outil : elle est de type requête, demande l'activation d'un outil, et spécifie sa position et ses attributs dans l'AIE.

Nous nous retrouvons ensuite dans le cas général de l'appel de la méthode *Send_coord* sur un nœud de type *Node_mgt* de l'AIE. Soit C la coordination à transmettre et soit C.designation la désignation relative des récepteurs potentiels de cette coordination partant du nœud courant.

C.désignation peut être composée de trois parties : l'AIE_designation, l'AIP_designation, et l'$\overline{\text{AIE}}$_designation (cf. III.3.3). Le principe de fonctionnement consiste à distinguer les cas suivants.

- Celui dans lequel C.designation ne comprend pas d'AIP_designation. Nous disons que C.designation est *non ambiguë*. Les nœuds placés sur les chemins menant aux récepteurs potentiels sont soit existant (de type *Ex*), soit à créer (de type *New*).
- Celui dans lequel C.designation comprend une AIP_designation. Nous disons que C.designation est *ambiguë.*

METHODE Send_coord (IN C : T_coord)

Soit C la coordination à transmettre,

C.type donne le type de la coordination (requête ou notification)

C.mode donne le mode de communication de la requête

C.designation donne la désignation des récepteurs potentiels de la coordination (au travers de leurs positions et attributs)

Soit N l'objet Node_mgt sur lequel la méthode Send_coord est appelée,

N.designation donne la désignation (position et attributs) de N

N.pere donne la référence de l'objet Node_mgt père

N.fils donne la liste des références des objets Node_mgt fils

N.mbus donne la liste des références des objets Client_mgt fils

Les actions effectuées par N sont :

1. Si N.designation n'est pas en accord avec C.désignation [i], i désignant le niveau de N dans l'AIE, retourner le code UNACCEPTED.
2. Sinon Si C.designation [i+1] = '..' alors (N.pere).Send_coord (C)

   Sinon // il faut descendre dans l'AIE :

   a) Si C.type = notification

      Si C.designation [i+1] est de catégorie feuille alors (N.mbus).Send (C)

      Sinon //appeler Send_coord sur tous les nœuds fils :

      Pour tout fils dans N.fils (N.fils).Send_coord (C)

   b) Sinon (C.type = requete)

      Si C.designation est ambiguë alors

      Lever_ambiguïté (C.designation, unamb_designation_list)(7).

      // Exécuter la liste d'instructions ci−après pour chacune des désignations non
      // ambiguës retournées jusqu'à ce que la requête ait été acceptée. Si elle n'est
      // pas acceptée retourner un code d'erreur.

      Si C.designation non ambiguë alors

      Si C.designation [i + 1]est de type *New* et de catégorie *feuille*

      // Verifier que la création d'un client pour servir C est possible
      // sinon retourner le code UNACCEPTED :

(7) Le fonctionnement de la méthode Lever_ambiguïté est rapporté juste après.

```
                Indra_organisation.Is_creation_possible (C, i + 1)
                // Créer un Client_mgt qui activera un client pour servir C :
                Client_mgt.New(C, i + 1)
                // Envoyer la cordination au bus de messages. Celle−ci sera transmise
                // au Client_mgt crée qui fera appel à l'instance de classe Indra_activation
                // pour activer le bon client.
                retourner ((N.mbus).Send (C))
            Si C.designation [i + 1] est de type New et de catégorie non feuille
                //Vérifier que la création d'un nœud en accord avec
                // C.designation [i+1] est possible. Sinon retourner le code UNACCEPTED.
                Indra_organisation.Is_creation_possible (C, i + 1)
                // Créer le nœud et lui renvoyer la demande de transmission de C :
                Node_mgt.New (C, i + 1)
                retourner (Node_Mgt.Send_coord (C))
            Si C.designation [i + 1] est de type Ex et de catégorie feuille
                // Il suffit de demander au bus de messages de l'envoyer à un client
                retourner (N.mbus.Send (C))
            Si C.designation [i + 1] est de type Ex et de catégorie non feuille
                // Pour chacun des nœuds Node_mgt donnés par N.fils, appeler
                // Send_coord (C) jusqu'à ce que la coordination soit acceptée. Si elle
                // n'est acceptée par personne, retourner le code UNACCEPTED
                Pour tout fils dans N.fils
                    Si fils.Send_coord (C) = ACCEPTED alors retourner ACCEPTED
Fin
```

La méthode Lever_ambiguïté (Fig. 5.11) transforme une désignation ambiguë en une liste de désignations non ambiguës, ordonnées de celle ayant la longueur maximale d'extension de la partie AIE_designation à celle ayant la longueur minimale (cf. III.3.3).

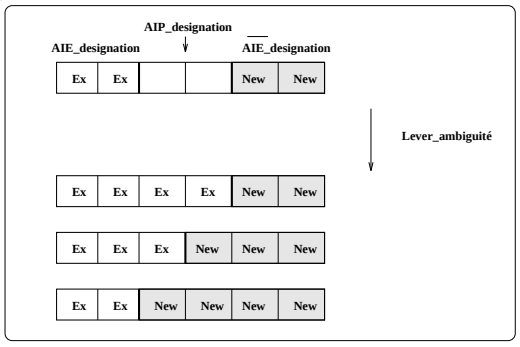


Fig. 5.11 : Gestion des désignations ambiguës

V.5.2.6 Gestion des relations

Les relations (définies en III.3.6) sont gérées par les objets de type Client_mgt. Tout Client_mgt conserve les identifications des autres Client_mgt qui lui somt reliés par une relation de nom donnée.

Les mises à jour de ces informations, prenant place lors de la création et de la destruction de relations, fonctionnent comme suit.

1. Lorsqu'une relation est créée au travers de la transmission d'une requête, les Client_mgt représentant les extrémités de cette relation profitent de cette communication pour effectuer la mise à jour.
2. Les mises à jour requises lors de la création d'une relation par transitivité (cf. III.3.6.2) ou lors de la destruction d'une relation sont régies par un appel de méthode direct entre les Client_mgt concernés par la relation.

V.5.2.7 Ordonnancement assuré

L'ordonnancement assuré par le service de communication respecte l'ordonnancement causal. Autrement dit :

1. si un client Cl émet une coordination c_1 et émet ensuite (au sens du chronologique) une coordination c_2, alors aucun client du service de communication ne recevra c_2 avant c_1,
2. si un client Cl_1 émet une coordination c_1 et qu'un client Cl_2, après avoir reçu c_1, émet une coordination c_2, alors aucun client du service de communication ne recevra c_2 avant c_1.

La première propriété est assurée par le fait que la méthode de transmission d'une coordination, exécutée par le client demandant la transmission, ne se termine que lorsque la coordination a été *déposée* chez tous les clients susceptibles de la recevoir.

Le respect de la deuxième propriété repose en plus sur le fait que, pour tout objet de type Node_mgt de même que tout objet de type Msg_bus, la méthode de transmission d'une coordination (Send_coord) est exclusive.

Supposons trois clients Cl_1, Cl_2, et Cl_3. Cl_1 émet une coordination c_1 vers Cl_2 et Cl_3. Après réception de c_1, Cl_2 émet une coordination vers Cl_3. Montrons que Cl_3 ne peut recevoir c_2 avant c_1.

Notons n_{ij} le nœud de plus bas niveau dans l'arbre, se trouvant sur les chemins reliant les clients Cli et Clj à la racine. Pour montrer que Cl_3 ne peut recevoir c_2 avant c_1, montrons qu'il existe un nœud de l'arbre sur lequel Cl_1 et Cl_2 vont appeler la méthode Send_coord pour envoyer la coordination à Cl_3 (et vont donc être synchronisés au niveau de ce nœud). Montrons en outre que l'appel de Cl_2 arrive forcément après celui de Cl_1. Cl_2 devant alors attendre que Cl_1 ait terminé son appel, la coordination émise par Cl_1 aura été déposée chez Cl_3 et celle émise par Cl_2 ne pourra être déposée qu'après.

Quatre cas sont à distinguer (les plus complexes sont illustrés Fig. 5.12)[8]:

a) n12 = n23 = n13

b) n12 = n23

c) n23 = n13

d) n12 = n13

Dans les quatre cas, il existe un nœud commun utilisé par Cl1 et par Cl2 pour émettre leur coordination vers Cl3. Ce nœud est respectivement n12 (égal à n23 et à n13) dans le cas (a), n23 dans les cas (b) et (c), et n13 dans le cas (d).

Dans les quatre cas, le nœud commun est aussi le nœud par lequel Cl2 reçoit la coordination émise par Cl1. L'appel de Cl2 à ce nœud ne pourra donc commencer qu'après que celui de Cl1 soit terminé.

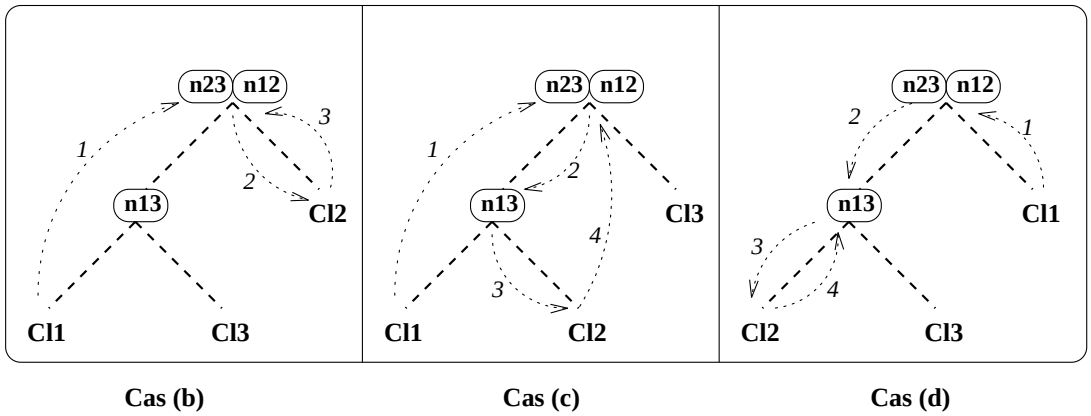


Fig. 5.12 : Respect de l'ordonnancement causal par le service de communication

Les propriétés qui permettent de respecter un ordre causal réduisent donc le parallélisme autorisé au niveau des transmissions des coordinations. Celles−ci ne sont soumises à aucun point de synchronisation que si elles passent par des nœuds distincts de l'arbre.

V.6 Conclusion

Outre le fait de permettre de coordonner des outils écrits dans des langages de programmation divers et s'exécutant éventuellement sur des machines hétérogènes, les objectifs essentiels qui ont régi la conception de l'environnement d'exécution ont été :

1. fournir des performances d'exécution satisfaisantes au regard des besoins des développeurs.
2. permettre l'évolution des définitions Indra en cours d'utilisation de l'environnement d'exécution,

(8) Le cas (n12 <> n23 <> n13) ne peut exister.

La mise en œuvre proposée a deux particularités du point de vue des performances d'exécution.

1. Une architecture décentralisée qui permet aux clients du mécanisme de coordination d'exécuter les fonctions fournies par celui−ci de manière parallèle. Le parallélisme des transmissions de coordination est toutefois limité par des points de synchronisation qui permettent d'assurer un ordonnancement causal au niveau des réceptions de coordinations.
2. L'environnement d'exécution, étant composé d'objets passifs, n'entraîne pas de surcharge de calcul inutile pour les machines support.

Ces deux particularités auraient été difficiles à obtenir avec un système support plus classique que Guide (Unix par exemple). Une évaluation des performances d'exécution est donnée dans le chapitre VI.

L'évolution des définitions Indra en cours d'utilisation de l'environnement, bien que non réalisée, a été considérée. La conception de l'environnement d'exécution devrait permettre, par l'ajout de mécanismes additionnels, de supporter les cas d'évolution les plus courants (ajout / retrait d'un outil, modification de la coordination d'un outil) en cours de développement.

Chapitre VI

Evaluation

Après avoir présenté l'état des travaux (VI.1), ce chapitre fournit une évaluation du langage Indra et de son modèle (VI.2) ainsi que de son environnement d'exécution (VI.3).

VI.1 Etat des travaux

Un prototype de l'environnement d'exécution du langage Indra a été réalisé, avec les restrictions suivantes.

Le *Service de désignation* de la partie *Machine d'exécution* autorise une organisation des outils limitée à deux niveaux hiérarchiques.

Le compilateur du langage Indra, chargé de traduire les descriptions Indra des outils en une représentation manipulable par la machine d'exécution, n'a pas été réalisé.

Ce prototype comporte environ 14000 lignes de code, dont 11500 lignes écrites en langage Guide (39 types et 35 classes) et 2500 lignes écrites en langage C.

Il a donné lieu à une petite expérimentation (mettant en jeu un petit nombre d'utilisateurs et six outils) qui permet de discuter sur les apports et les insuffisances du langage Indra et de son environnement d'exécution (cf. VI.2).

VI.2 Le langage Indra et son modèle

VI.2.1 Satisfaction des objectifs fixés

Nous pensons que les principes que nous avons adoptés (énoncés en III.1.2), ont permis d'atteindre les objectifs sur lesquels nous nous sommes concentrés (énoncés en II.5), au dépens des contre−parties rappelées ci−après.

1. **Objectif** : séparer l'expression des coordination du code des outils et modulariser cette expression.

   Principe : le choix du modèle de coordination mixte (cf. II.3.3.3).

Contre−partie : l'administrateur doit exprimer un grand nombre d'informations pour coordonner des outils. La vérification de la cohérence des informations exprimées n'est pas toujours facile.

2. **Objectif** : séparer l'expression de l'évolution dynamique des coordinations du code des outils et la rendre explicite.

   Principe : on exprime les coordinations d'un outil sous la forme d'états et de transitions entre états.

   Contre−partie 1 : le modèle ne prévient pas les interblocages.

   Contre−partie 2 : Les instructions autorisées par le langage Indra doivent être placées dans un certain ordre.

3. **Objectif** : fournir des moyens de désignation des outils adaptés leur évolution dynamique.

   Principe : les outils sont organisés dans un arbre attribué. Pour désigner des outils, on peut utiliser leur position dans l'arbre, leurs attributs, leurs relations et le fait qu'ils sont actifs ou non.

   Contre−partie : l'impact sur les performances d'exécution (voir VI.3.2).

VI.2.2 Expérimentation

L'expérimentation réalisée s'est déroulée en deux étapes. Premièrement, quatre outils ont été coordonnés : un "Desktop", un "Browser" d'erreurs, un éditeur (lemacs) et un metteur au point. Dans un deuxième temps, nous avons ajouté un "Browser" de types et de classes Guide et un "Builder".

Cette expérimentation permet de discuter sur :

1. la pertinence des objectifs fixés,
2. l'importance des contre−parties des choix effectués pour atteindre ces objectifs.

VI.2.2.1 Expression explicite et modulaire des coordinations

Le fait que l'expression des coordinations soit explicite et modulaire nous a paru avantageux, notamment lorsque nous avons du faire évoluer les coordinations pour ajouter de nouveaux outils à l'environnement. La compréhension du comportement des outils déjà intégrés a été rapide, facilitant la définition des états de coordination des nouveaux outils.

En revanche, l'expérimentation a confirmé que la contre−partie du choix effectué pour atteindre ces objectif est très ressentie. La quantité *importante* de noms à manipuler (noms d'événements, de réactions et de coordinations) est incontestablement source d'erreurs. Cet inconvénient peut toutefois être allégé en offrant des outils (par exemple, un outil graphique

pour lier les événements aux réactions, ainsi qu'un outil de vérification des concordances des noms).

VI.2.2.2 Expression explicite des coordinations et de leur évolution dynamique

Trois aspects ont été montrés par l'expérimentation réalisée concernant l'évolution dynamique des coordinations.

1. Il s'est avéré nécessaire d'avoir, à un instant donné de l'exécution, deux outils identiques (par exemple deux éditeurs) ne fonctionnant pas de manière identique vis−à−vis des coordinations car ne jouant pas le même rôle dans l'environnement. L'association d'états de coordination différents à chacune des deux instances d'outil a répondu à ce besoin.

   Par exemple, un éditeur activé par un metteur au point pour visualiser la trace de l'exécution d'un programme agit différemment d'un éditeur activé par le développeur pour des besoins d'édition.

2. La possibilité de faire évoluer dynamiquement les coordinations d'un outil s'est également avérée nécessaire, même si cette possibilité n'a été utilisée que pour certains outils. Pouvoir exprimer des transitions entre états de coordination a donc été requis.

   Par exemple, le metteur au point "bascule" entre deux états, selon qu'il est coordonné avec un éditeur ou non. Ainsi, c'est seulement lorsqu'il est coordonné avec un éditeur que chaque pas d'exécution engendre l'émission d'une requête vers cet éditeur.

3. Lorsque nous avons dû rajouter des outils à l'environnement, la notion d'état de coordination nous a paru avantageuse pour comprendre rapidement comment se comportent les outils déjà intégrés face aux coordinations.

   Par exemple, pour l'ajout du Browser de types et de classes Guide, nous avons voulu établir des coordinations permettant de visualiser sous forme textuelle ces types et classes depuis le browser. Pour ce faire, nous avons regardé quels étaient les états de l'éditeur dans lesquels ce dernier était susceptible de répondre à ce type de requête.

Enfin, il faut noter que la contre−partie engendrée par l'utilisation du concept d'état de coordination (interblocages) n'a pas été ressentie lors de notre expérimentation : les coordinations que nous avons exprimées n'étaient pas réentrantes.

VI.2.2.3 Désignation des outils

L'organisation arborescente des outils nous a semblé pratique pour désigner ceux−ci. Etant limités à deux niveaux hiérarchiques par le prototype, nous avons exprimé la désignation "sur papier" avec plus de niveaux, et avons ensuite "traduit" cette désignation en utilisant des attributs pour représenter les différents niveaux hiérarchiques.

Outre cet aspect, la gestion du caractère actif ou non actif des outils a été utile, notamment en ce qui concerne :

- la possibilité, pour un émetteur, de ne pas préciser si le récepteur d'une coordination doit être un outil déjà actif ou non, sachant qu'une activation automatique est engendrée si aucun outil déjà actif ne peut être récepteur,
- la possibilité de spécifier que seul un outil non encore actif peut être récepteur d'une coordination.

Enfin, la possibilité d'établir dynamiquement des relations entre outils a également été utile pour différencier, par exemple, deux outils identiques travaillant dans une même session et se trouvant placés dans le même état de coordination.

L'expérimentation a toutefois montré que les relations sont souvent utilisées pour lier des outils travaillant sur une même donnée. Autrement dit, elles permettent de sélectionner émetteurs et récepteurs par rapport aux données sur lesquelles ils travaillent. En conséquence, nous jugeons nécessaire d'approfondir la notion de relation, et d'étudier si celle−ci ne doit pas être remplacée par une autre notion, axée sur les données.

VI.3 L'environnement d'exécution du langage Indra

VI.3.1 Satisfaction des objectifs fixés

Objectifs majeurs et principes adoptés, énoncés en V.1.3, sont brièvement rappelés ci−après. Une discussion sur la satisfaction de l'objectif 1 (performances d'exécution) que nous considérons comme le plus critique est ensuite donnée.

1. **Objectif** : respecter des performances satisfaisantes au regard des besoins des développeurs.

   Principe : choix du système réparti et orienté objet Guide(1), qui permet de décentraliser l'environnement d'exécution pour autoriser des exécutions parallèles des coordinations.

2. **Objectif** : permettre de coordonner des outils écrits dans des langages de programmation divers et s'exécutant sur des machines éventuellement hétérogènes.

   Principe : s'appuyer sur le fait que Guide rend ses fonctions accessibles à de tels outils.

3. **Objectif** : supporter l'évolution des outils, de leur coordination, et de leur organisation durant le développement.

   Principe : exploiter le mécanisme de liaison dynamique entre objets adopté par Guide.

(1) Les apports essentiels du système Guide à l'égard de la mise en œuvre de l'environnement d'exécution, et en particulier des performances d'exécution obtenues sont expliqués en Annexe D.

VI.3.2 Performances d'exécution

Deux aspects sont à considérer au niveau des performances d'exécution : les coordinations que nous qualifions d'*intra−session* et celles que nous qualifions d'*inter−session*. A la différence des secondes, les premières prennent place entre des outils "proches" (le chemin reliant ces outils dans l'organisation arborescente ne comporte qu'un nœud, voir exemple Fig. 6.1) qui s'exécutent généralement sur une même machine.

VI.3.2.1 Coordinations intra−session

L'expérimentation réalisée a montré que les performances d'exécution étaient satisfaisantes pour les coordinations intra−session, tant que les ressources (calcul, mémoire) requises par les outils actifs ne sont trop importantes.

Par exemple, avec une session comprenant un éditeur (lemacs), un compilateur, un browser d'erreur et un browser de types et de classes s'exécutant sur une station de travail DEC 5000, le développeur n'a pas l'impression que les coordinations engendrent des temps d'attente.

En revanche, une augmentation importante du nombre d'outils entraîne un fort ralentissement des performances d'exécution, essentiellement causé par l'accroissement de la taille de mémoire centrale utilisée. Les mesures effectuées pour les coordinations intra−session, dans un environnement "hors−charge" (composé d'une dizaine d'outils actifs ne consommant qu'un minimum de ressources) ont en effet montré que les temps d'exécution n'engendraient pas, à eux seuls, une attente visible pour les développeurs (cf. table ci−après).

Nous pensons que le problème soulevé peut être fortement allégé en utilisant un mécanisme qui permette de partager du code exécutable en mémoire centrale, car les outils de développement utilisent souvent des bibliothèques communes (X/Motif par exemple).

Enfin, il faut noter que le fait que deux coordinations émises par un même outil ne soient pas traitées en parallèle n'a pas eu d'impact, car dans l'expérimentation réalisée les coordinations intra−session étaient en majorité provoquées par les actions du développeur, ces dernières étant forcément séquentielles.

Types de coordination	Notification vers 1 outil	Notification vers 10 outils	Requête avec accusé	Requête avec accusé et activation d'outil
Temps (ms) sur DEC 5000	8.70	13.95	11.48	74.24

Temps des coordinations intra−session, dans un contexte hors−charge ne comprenant que des outils locaux.

Note : le temps donné pour la requête avec activation d'outil comprend le temps de connexion de l'outil activé au bus de messages, et l'enregistrement de 20 messages acceptés. Ce temps ne comprend pas le temps de lancement de l'outil (différent selon les outils).

VI.3.2.2 Coordinations inter−session

L'expérimentation réalisée n'a pas permis d'évaluer les coordinations inter−session, car seule une organisation des outils limitée à deux niveaux hiérarchiques est pour l'instant permise par le prototype [(2)]. L'ordre de grandeur des temps requis par ces coordinations a toutefois été évalué. Nous nous sommes fondés sur le fait que le surcoût engendré par rapport aux coordinations intra−session provient essentiellement du passage par *n* nœuds de l'organisation arborescente des outils, représentés par des objets éventuellement *distants* les uns des autres.

L'étude effectuée se base sur une organisation que nous jugeons représentative d'un environnement de moyenne envergure, illustrée par la figure Fig. 6.1.

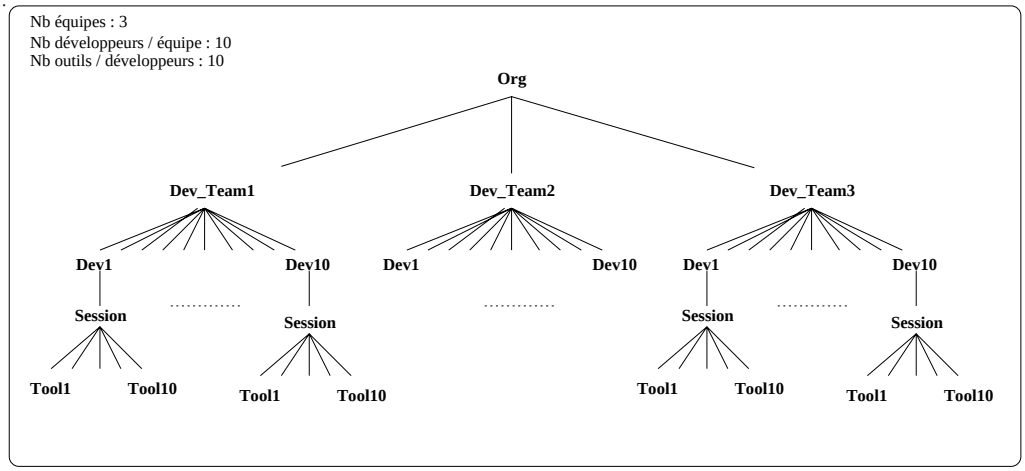


Fig. 6.1 : Situation choisie pour calculer les coûts des coordinations inter−session

Nous donnons (cf. table ci−après) le nombre de nœuds traversés pour différents types de coordinations, sachant que la traversée d'un nœud correspond à un appel de méthode sur un objet Guide. Nous avons supposé que les nœuds Dev, Dev_Team et Org sont représentés par des objets distants les uns des autres.

Les cas choisis sont :

1. Une émission de notification ayant la forme suivante :
   ```
   NOTIF notif_name( )
   TO  ../../../../*
   ```
2. Une émission de requête ayant une désignation non ambiguë des récepteurs, sous la forme suivante :
   ```
   REQ−ASYNC−AR req_name( )
   TO  ../../../../* EX
   ```

(2) Pour expérimenter le mécanisme de coordination proposé dans un contexte multi−utilisateur, nous avons utilisé les attributs pour représenter le concept de développeur.

3. Une émission de requête ayant une désignation non ambiguë des récepteurs et entraînant la création de nœuds Dev_Team, Dev, Session et Tool dans l'arbre, sous la forme :

   ```
   REQ_ASYNC−AR req_name( )
   TO /Org EX NEW /Dev_Team/Dev/Session/Tool.name = emacs
   ```

4. Une émission de requête ayant une désignation ambiguë des récepteurs, sous la forme suivante

   ```
   REQ−ASYNC−AR req_name( )
   TO ../../../../Org/Dev_Team/Dev.name=n/Session/Tool.name=emacs
   ```

Type de coordination inter−session	(1)	(2)		(3)	(4)	
		Cas fav.	Cas défav.		Cas fav.	Cas défav.
Nb appels de méthode distants	35	5	36	5	5	109
Nb appels de méthode locaux	32	3	32	3	3	67
Nb créations objets distants	0	0	0	2	0	2
Nb créations objets locaux	0	0	0	1	0	1
Temps calculé (ms) sur DEC 5000	~ 850	~ 65	~ 750	~ 155	~ 65	~ 1800

Nb appels de méthodes et créations d'objets additionnels par rapport aux coordinations intra−session, et temps approximatif calculé.

Les temps donnés montrent que des temps d'attente visibles pour le développeur peuvent être engendrés par certaines coordinations de grande portée (surtout dans les cas défavorables). Ces temps risquent d'être encore plus marqués par les contraintes de synchronisation qui s'appliquent sur les transmissions de coordination (cf. V.5.2.7).

Fournir un moyen qui permette à un outil de continuer à s'exécuter en parallèle avec l'exécution de la coordination qu'il émet peut réduire ce temps d'attente. Néanmoins, deux coordinations émises par un même outil étant traitées séquentiellement, si le développeur attend un résultat visible d'une coordination C2 émise après une coordination C1 de grande portée, alors ce dernier aura l'impression d'attendre.

L'inconvénient présenté peut être pondéré par le fait les coordinations de grande portée ne sont pas fréquentes(3). Par ailleurs, une deuxième version de Guide (Guide 2) qui fournit des performances beaucoup plus attractives (4 ms contre 12 ms pour les appels de méthode distants) est en cours de consolidation et permet de compter sur des coûts de coordinations inter−session plus satisfaisants pour les développeurs.

Nous pensons que cet inconvénient remet toutefois en cause l'adéquation de nos choix de conception et de réalisation du service de désignation pour coordonner des outils dans des environnements de grande taille. La dégradation des performances est en effet d'autant plus grande que le nombre de nœuds de l'arbre représentant l'organisation des outils est important.

(3) Cette affirmation, qui a été une hypothèse effectuée pour la conception du mécanisme de coordination proposé, a été corroborée par l'expérimentation que nous avons réalisée.

Conclusion

.1 Rappel du cadre et des objectifs du travail

Le travail présenté dans ce document relève du domaine des environnements de développement intégrés, et plus particulièrement d'un axe d'intégration appelé *Intégration du contrôle*.

L'objectif des environnements de développement intégrés est d'accroître la productivité des développeurs et d'améliorer la qualité du logiciel développé en intégrant les composants de l'environnement de développement.

Dans ce cadre, l'intégration du contrôle vise à permettre aux outils de développement d'échanger des informations pour agir de manière cohérente et homogène. Ces échanges, que nous qualifions de *coordinations*, sont principalement de deux types:

- les notifications qui permettent à un outil de transmettre une information à d'autres outils,
- les requêtes qui permettent à un outil d'appeler un service fourni par un autre outil.

Un mécanisme d'intégration du contrôle met en œuvre un modèle de coordination au travers d'un environnement d'exécution.

Un modèle de coordination doit permettre de définir des coordinations sous la forme d'associations entre des évenements de type requête ou notification émis par des outils et des réactions exécutables par d'autres outils.

Nous nous sommes intéressés aux mécanismes d'intégration du contrôle qui permettent de coordonner des outils de gros grain. Les principaux critères de qualité qui doivent être remplis par ces mécanismes sont rappelés ci−après. Ceux−ci ont formé nos objectifs majeurs.

Du point de vue du modèle

1. L'expression des interfaces de coordination des outils, en termes des événements qu'ils peuvent émettre et des réactions qu'il peuvent exécuter, doit être explicite.
2. L'expression des coordinations doit être modulaire et séparée du code des outils.

3. L'expression de l'évolution dynamique des coordinations doit être explicite et séparée du code des outils.
4. Les moyens de désignation des outils doivent être adaptés au fait que l'ensemble des outils qui sont actifs à un instant donné évolue fortement.
5. Les outils doivent pouvoir utiliser un vocabulaire différent pour les coordinations (noms d'événements et de réactions différents bien que sémantiquement identiques).
6. Les outils doivent pouvoir échanger des données au travers des coordinations même si leur modèle de donnée respectif est différent.

Du point de vue de l'environnement d'exécution

a) Respecter des performances satisfaisantes au regard des besoins des développeurs : ceux−ci ne doivent pas avoir l'impression que les coordinations engendrent des temps d'attente notables.
b) Permettre de coordonner des outils écrits dans des langages de programmation divers et s'exécutant sur des machines éventuellement hétérogènes.
c) Supporter l'évolution des outils, de leur coordination, et de leur organisation en cours de développement.

L'analyse des environnements existants fait apparaître que les critères 2), 3), 4) et 6) sont encore mal remplis.

Nous nous sommes essentiellement concentrés sur les critères 2), 3) et 4). Notre démarche a été de proposer des solutions qui satisfassent ces critères et de valider ces solutions par la réalisation d'un prototype.

.2 Travail réalisé

Notre travail a consisté en :

1. la proposition d'un modèle de description de l'intégration du contrôle,
2. la conception d'un langage nommé Indra mettant en œuvre ce modèle,
3. la conception et la réalisation d'un environnement d'exécution pour le langage Indra,
4. la mise en place d'une expérimentation de nos travaux, mettant en jeu un petit nombre d'utilisateurs et six outils.

La section suivante rappelle les choix essentiels effectués et les résultats obtenus pour les points 1 et 3.

.3 Rappel des principes adoptés et des résultats obtenus

.3.1 Le modèle mis en œuvre par le langage Indra

Les principes qui ont permis d'atteindre les objectifs 2), 3) et 4) énoncés en .1 sont rappelés ci−après.

1. Le modèle de coordination choisi est un modèle mixte, basé sur l'utilisation d'un mécanisme événementiel de type bus de message, et sur l'utilisation de l'appel de procédure. Il fournit un espace global et explicite de coordinations, auxquelles les événements et les réactions des outils peuvent être dynamiquement liés ou déliés.

   Ces liaisons expriment les coordinations de manière modulaire, à un grain de niveau outil. Elles sont définies en dehors des programmes exécutables des outils.

2. L'expression de l'évolution dynamique des coordinations d'un outil s'inspire du concept d'automate d'états fini. Chaque état décrit d'une part les coordinations dans lesquelles l'outil est susceptible d'intervenir lorsque durant son exécution il se trouve dans cet état et d'autre part les événements qui font passer l'outil à un autre état.

3. Les outils sont organisés dans un arbre attribué. La définition de cet arbre est paramétrable. Alors que la position et les attributs d'un outil ne varient pas dynamiquement, des relations peuvent être dynamiquement créées ou détruites entre les outils. Pour désigner des outils, on peut utiliser leur position dans l'arbre, leurs attributs, leurs relations et le fait qu'ils sont actifs ou non.

Ces principes ont permis d'atteindre les objectifs fixés, aux dépens des contre−parties suivantes.

1. L'administrateur doit exprimer un grand nombre d'informations pour coordonner des outils (interfaces de coordination et coordinations). La vérification de la cohérence des informations exprimées n'est pas toujours facile.

2. Le concept d'état de coordination implique, pour fournir un modèle simple d'utilisation, que les exécutions des coordinations associées à un outil actif soient atomiques. Ceci permet en effet d'offrir un modèle déterministe : la même séquence d'événements (au sens des événements de coordination entrants et sortants pour un outil) reproduit le même comportement.

   Cette atomicité implique que les instructions autorisées par le langage Indra soient placées dans un certain ordre, ce qui peut être fastidieux pour l'administrateur.

   Cette atomicité peut par ailleurs produire des interblocages dont la résolution n'est pas toujours réalisable. Il faut alors fournir un moyen pour gérer ces interblocages *au niveau du langage* Indra. Or les événements de coordination sont asynchrones

(peuvent être déclenchés à tout instant) et la présence d'interblocages ne peut être considérée comme une erreur d'expression des coordinations.

3. La puissance du service de désignation des outils nécessite une mise en œuvre assez lourde, qui a un impact sur les performances d'exécution.

   Par ailleurs la manipulation des relations est complexe. Cette notion doit être approfondie et clarifiée pour la rendre plus simple d'utilisation.

.3.2 L'environnement d'exécution du langage Indra

L'élément majeur a été de choisir le système réparti et orienté objet Guide comme support de l'environnement d'exécution. Ce choix a permis :

- de décentraliser fortement l'environnement d'exécution, pour autoriser des exécutions parallèles des coordinations et améliorer les performances,
- de s'appuyer sur les fonctions fournies par Guide pour le support d'outils multi−langages et la gestion de l'hétérogénéité des machines : Guide offre ses fonctions à des applications écrites dans des langages de programmation divers (Guide, C, C++) ; le support de l'hétérogénéité est en cours d'étude,
- de supporter l'évolution des composants paramétrés, grâce au mécanisme de liaison tardive adopté par Guide. Cet aspect a toutefois été réduit au stade d'une étude qui fait apparaître que les types d'évolution les plus courants (ajout / retrait d'outils, modification de la coordination d'un outil) ne requièrent pas d'arrêter l'environnement d'exécution. En revanche, la modification de l'organisation des outils (notamment l'ajout ou le retrait d'un niveau hiérarchique dans l'arbre) nécessite cet arrêt.

Les résultats obtenus font apparaître que les performances d'exécution sont satisfaisantes pour les environnements de petite et moyenne taille, impliquant moins d'une trentaine de développeurs. En revanche elles sont moyennement satisfaisantes pour les environnements de grande taille, dans lesquels les coordinations peuvent passer par de multiples niveaux hiérarchiques de l'arbre attribué pour arriver à leurs récepteurs.

.4 Les solutions existantes

Les points originaux qui distinguent notre proposition des solutions existantes sont les suivants.

- Le concept d'états de coordination est exploité dans des mécanismes existants [40], mais seuls les états sont décrits de manière explicite en dehors des programmes exécutables des outils. Les transitions entre ces états sont exprimées dans les programmes exécutables.

- Notre proposition homogénéise la gestion de l'activation des outils, en considérant que tout outil non actif est placé dans un état d'activation dans lequel les coordinations spécifiées entraînent l'activation de l'outil et son passage dans un état de coordination donné.

- Le concept d'arbre attribué pour désigner des agents est utilisé par la norme X500 [88]. Néanmoins, cette norme n'étant pas spécialement dédiée à la gestion d'outils de développement, la notion d'outil actif ou non actif n'est pas gérée.

  Par ailleurs, dans le Koala Bus [11], un mécanisme de groupe est fourni pour désigner les outils. En utilisant un groupe par attribut, on obtient des possibilités de désignation similaires aux nôtres, à la différence qu'aucun moyen ne permet d'établir une organisation arborescente des outils ni d'utiliser leur placement dans cette organisation pour les désigner.

.5 Perspectives

Les tâches suivantes constituent des perspectives à court terme.

1. Approfondir la notion de relation utilisée dans le service de désignation des outils. En fonction des objectifs de celle−ci, étudier si elle ne doit pas être remplacée par une autre notion, axée sur les données sur lesquelles travaillent les outils à un instant donné.
2. Permettre d'activer des outils à la suite de notifications.
3. Améliorer la gestion des interblocages (détection et résolution).
4. Travailler la syntaxe du langage Indra pour la rendre plus concise et plus simple.
5. Fournir des moyens qui permettent aux développeurs de configurer dynamiquement le choix des outils qu'ils désirent utiliser, lorsque plusieurs outils de même fonction leur sont fournis.
6. Concevoir et réaliser des outils graphiques de visualisation, de mise au point et de simulation des coordinations pour rendre plus facile leur expression.
7. Etudier comment les fonctions de coordination fournies peuvent être utilisées par un mécanisme d'intégration du processus de développement.

Les perspectives qui nous paraissent intéressantes à plus long terme sont les suivantes.

1. Augmenter la puissance du langage Indra en vue de son utilisation dans un contexte autre que l'intégration du contrôle.

   L'administration de systèmes, la conception assistée par ordinateur, les applications bureautiques et de manière générale les applications qui mettent en jeu la coopération de plusieurs agents peuvent nécessiter un mécanisme de coordination.

Certains concepts additionnels peuvent être requis par ces applications. De prime abord, nous jugeons intéressants les quatre suivants.

Notion de temps

Il s'agirait d'une part de permettre de différer la réception d'une coordination et d'autre part de pouvoir conditionner les émissions et les réceptions de coordinations par rapport au temps physique ou logique.

Notion d'état global

Cette notion permettrait de conditionner les émissions et les réceptions de coordinations par rapport à l'état global de l'environnement, défini par l'état de l'ensemble des agents actifs à un instant donné, ainsi que par leur position et par leurs attributs dans l'organisation arborescente. La prise en compte de l'état global peut être motivée pour gérer l'allocation de ressources.

Notion d'historique

En tant qu'extension directe de la proposition précédente, il s'agirait de pouvoir conditionner les émissions et les réceptions de coordinations par rapport à la suite des états globaux qui se sont succédés depuis le lancement de l'environnement jusqu'à l'instant présent.

Partage de données

Certaines catégories d'applications ne partagent pas une base de données commune. Le mécanisme de coordination doit alors non seulement gérer les divergences portant sur des données de "structures simples", mais également celles portant sur des données de structures complexes.

2. Adapter le modèle et le langage Indra, ainsi que son environnement d'exécution pour les environnements de développement de grande taille.

   Nous pensons qu'il devient nécessaire de dissocier un niveau intra−environnement (correspondant à un environnement de taille moyenne) de celui inter−environnement, pour pouvoir différencier les critères requis (ordonnancement des coordinations, performances d'exécution, etc) et adapter les fonctions fournies en conséquence.

3. Etudier les besoins spécifiques aux applications écrites dans un langage orienté objet.

   En général, les applications écrites en langage orienté objet ne connaissent que la notion de méthode, qui ajoute la notion d'identification d'objet à celle de procédure.

Or l'interface fournie par le langage Indra se fonde sur la notion de procédure. Faut−il fournir un autre type d'interface ?

Le langage Indra étant conçu pour exprimer des coordinations inter−applications, il faut tout d'abord définir le concept d'application dans le langage orienté objet visé. Si une application est une unité globale qui ne rend pas visibles les objets qui la composent et qui fournit une interface externe au travers d'un objet particulier, alors l'interface fournie par Indra peut être adéquate.

Pour la majeure partie des langages orientés objet, la notion d'application reste toutefois mal définie. Nous jugeons donc nécessaire d'étudier plus en profondeur les problèmes posés.

4. Adapter le langage Indra et son environnement d'exécution pour les coordinations intra−applications.

   Fonder une méthode de développement d'une application sur la notion de coordination commence à être envisagé [31][54].

   Le grain des agents à coordonner étant ici fin, le mécanisme de coordination doit fournir des performances d'exécution adéquates. Il doit en outre permettre d'associer une interface de coordination à chaque agent de l'application.

   Comment alors intégrer cette interface de coordination avec l'interface classique d'un agent, définie en termes des méthodes ou procédures appelables ? Dans le cas d'un langage orienté objet, comment concilier cette interface de coordination avec les mécanismes d'héritage et de réutilisation ?

   Quels sont par ailleurs les besoins au niveau de la désignation des agents ? ceux−ci n'évoluent pas dynamiquement comme dans le cas des coordinations inter−applications.

   Enfin, comment fournir un langage qui permette de bien dissocier l'expression des coordinations intra−applications de celles inter−applications ?

Ces perspectives visent à concevoir un mécanisme de coordination générique, répondant aux besoins d'applications différentes, tant au niveau de leur domaine qu'à celui de leur taille. Elles expriment notre avis quant aux axes de recherche futurs pour les mécanismes de coordination.

ANNEXES

Annexe A

Grammaire du langage Indra

A.1 Description de l'interface de coordination d'un outil

```
<tool_coord_itf> ::= TOOL_COORD_ITF    OF    <tool_name>
                     <reaction_desc>*
                      <event_desc>*
                      END

<reaction_desc>   ::=  <reaction_completion_type>
                     <reaction_name>
                     <reaction_form_args>
<reaction_completion_type>  ::=  ASYNC_REACT  |  SYNC_REACT

<event_desc>    ::=    <event_type>
                     <event_name>
                     <event_form_args>
<event_type>  ::=  NOTIF_EVT  |  REQ_EVT
```

A.2 Description d'un état de coordination d'un outil

```
<toool_coord_state>::=  TOOL_COORD_STATE  <tool_coord_state_name>
                        OF <tool_name>*
                       [INHERITS   <tool_coord_state_name>]
                       [RM_VAR   <var_name>*]
                       [RM_COORD   <coord_item>*]
                         <var_decl>*
                         <coord_desc>*
                        END

<var_decl>  ::=          <var_name>  ':'  <var_type>  ';'

<coord_desc>  ::=       ON  <declarative_part>  DO  <operative_part>

<declarative_part>  ::= <evt_signal>  |  <coord_receiving>
<evt_signal>::          SIGN   <evt_name>
                        <evt_form_args>
                       [WHERE <cond_on_args>]
<coord_receiving>::=    <coord_type>
                        <coord_name>
```

```
                        <coord_form_args>
                        FROM  <tools_naming>
                        [WHERE  <cond_on_args>]
                        [WHITH_INH_REL   <rel_name><nodes_naming>]
<coord_type> ::=        REQ | NOTIF
<cond_on_args> ::=      <arg_name> '=' <arg_val>          |
                        <cond_on_args> AND cond_on_args>  |
                        <cond_on_args> OR <cond_on_args>

<operative_part> ::=    [<coord_sending>]
                        <var_affect>*
                        [RM_REL  <rel_name>+  <nodes_naming>*]
                        [CHANGE_STATE   <coord_state_name>]
                        [CALL   <reaction_nsme><reaction_eff_args]
<coord_sending> ::=     <coord_type>
                        <coord_name>
                        <coord_eff_args>
                        TO  <tools_naming>
                        [INH_REL  <rel_name>+]
<coord_type>   ::=      REQ_ASYNC_AR | REQ_SYNC | NOTIF
<var_affect>   ::=      [<coord_state_name>'.']
                        <var_name>':='<var_val>   ';'

<tools_naming> ::=      [AI_naming] [REL_naming]
<AI_naming>    ::=      [<nodes_naming> EX]
                        [<nodes_naming>]
                        [NEW  <nodes_naming>]
<nodes_naming> ::=      '//' <nodes_naming>           |
                        '..//'   <nodes_naming>       |
                        <node_type> [cond_on_attrs]   |
                        <nodes_naming>  '//'  <nodes_naming>
<cond_on_attrs> ::=     '.' <cond_on_attr>                    |
                        '.' <cond_on_attr> AND <cond_on_attrs> |
                        '.' <cond_on_attr> OR <cond_on_attrs>
<cond_on_attr> ::=      <node_attr> '=' <node_attr_val>
<REL_naming>   ::=      NEW_REL  <rel_name>  |
                        EX_REL  <rel_name>   |
                        REL  <rel_name>
```

A.3 Description de l'état d'activation d'un outil

```
<tool_act_state> ::=   TOOL_ACT_STATE   OF <tool__name>
                       <var_decl>
                       <coord_desc>
                      END

<coord_desc>     ::=  ON  <declarative_part>  DO <operative_part>

<declarative_part>  ::= REQ  <coord_name>  <coord_form_args>
                        FROM   <tools_naming>
                        [WHERE   <cond_on_args>]
                        [WHITH_INH_REL    <rel_name><nodes_naming>]

<operative_part>  ::=   COMMAND <Unix_command>
                        [<coord_sending>]
                        <var_affect>*
                        CHANGE_STATE   <coord_state_name>
                        [CALL    <reaction_nsme><reaction_eff_args]
```

A.4 Description de l'organisation des outils

```
<nodes_type_def>  ::= NODES_TYPE_DEF   <node_type_def>*
<node_type_def>   ::= <node_type_name>
                      SUBTYPE  OF  <node_type_name>
                      [ATTRIBUTES    <attr_decl>*]
                      [ATTRIBUTES_VALUES    '{'<attrs_value>+'}']
<attr_decl>       ::= <attr_name>  ':'  <attr_type>  ';'
<attrs_value>     ::= '('  <value>  ')'
<value>           ::= <attr_val>  |  <attr_val>  ','  <value>

<AS_def>          ::=  AS_DEF  <nodes_def>
<nodes_def>       ::=  '('  <valuation>
                       <node_type_name>
                       <nodes_def>*   ')'
```

Annexe B

Gestion des cas d'erreurs et des retours d'information dans le langage Indra

La gestion des cas d'erreurs et des retours d'information est un aspect non encore exploré dans le langage Indra. Seuls des mécanismes primitifs sont fournis à cet égard.

B.1 Cas d'erreurs

Parmi les cas d'erreurs pouvant se produire au niveau des coordinations, il faut distinguer les erreurs *système* des erreurs *applicatives*. Les premières proviennent du mécanisme de coordination. Elles peuvent être provoquées par une panne de site, ou par un dysfonctionnement du système de communication sous−jacent. Les secondes proviennent d'une mauvaise spécification des coordinations.

Les erreurs systèmes qui peuvent survenir dépendent de la fiabilité du mécanisme de coordination que l'on veut assurer. Nous avons choisi de nous baser sur la fiabilité assurée par le système support de l'environnement d'exécution du langage Indra. Ce système est le système réparti et orienté−objet Guide. Il détecte les cas d'erreurs (pannes de sites ou incohérence d'un objet par exemple), sans les gérer. Nous nous sommes donc limité à supposer que des erreurs système pouvaient se produire, et qu'il fallait donner à l'administrateur un moyen minimal pour gérer ces erreurs (1). Deux types d'erreurs systèmes sont distinguées : les erreurs fatales (incohérence d'un objet par exemple), et celles non fatales (problème de communication avec un site donné par exemple). A l'inverse des erreurs non fatales, une erreur fatale stoppe définitivement la coordination d'un outil.

Parmi les erreurs applicatives, il faut distinguer celles pouvant être détectées statiquement de celles qui ne peuvent être détectées que dynamiquement. Un grand nombre d'erreurs peuvent être détectées statiquement : par exemple, un nom de requête qui n'est accepté par personne, un passage à un état de coordination inexistant, etc. Dynamiquement, deux types d'erreurs peuvent se produire et concernent les coordinations de type requête :

- une requête n'est recevable par aucun outil (2),
- une requête ne peut pas être servie pour cause d'inter−blocage. Ce cas d'erreur est provoqué par l'atteinte d'un *temps limite*.

A court terme nous voulons, comme pour les erreurs systèmes, fournir à l'administrateur un moyen minimal lui permettant de gérer ces erreurs.

Pour ce faire, toute émission de coordination peut être suivie par un test. Ce test peut lui−même être suivi par une nouvelle partie opérative, permettant à l'administrateur, selon le cas, de renvoyer une requête ou une notification, de manipuler les variables des états de coordination, de changer d'état de coordination et enfin, comme dernière instruction, d'appeler une méthode *réaction* de l'outil.

```
REQ / NOTIF ...
TO  ...
CASE
  NO_ERR  :    <operative_part>
  SYST_ERR :  <operative_part>
  SYST_FATAL_ERR  :  <operative_part>
  NO_RECEIVERS_ERR  :  <operative_part>
  TIME_OUT_ERR  :  <operative_part>
END
```

Note 1 : De ce fait, le mécanisme de coordination n'assure pas l'atomicité de transmission des coordinations : parmi les outils devant recevoir une notification, celle−ci pourra être reçue par certains et pas par d'autres.

Note 2 : Notons que ce type d'erreur peut être prévu par l'administrateur, lorsque par exemple il sait que la requête émise peut n'être servie par aucun outil.

B.2 Retours d'information

Le retour d'information par le mécanisme de coordination ne se limite pas à un retour du résultat sur la bonne ou mauvaise exécution d'une coordination. D'autres informations sont susceptibles d'intéresser un outil qui émet une coordination. Dans un premier temps, nous avons jugé deux informations de ce type : le nombre de récepteurs de la coordination (variable NB_RECEIVERS), et leur identification (variable ID_RECEIVER). Connaître le nombre de récepteurs peut permettre à un outil de vérifier qu'une notification est reçue par au moins un outil. Connaître l'identification du ou des récepteurs peut permettre à un outil de diriger ensuite certaines coordinations vers ce ou ces outils récepteurs.

Exemple :

```
ON SIGN AskForVisualization (IN file_name : String)
DO REQ AskForVisualization  (file_name : String)
   CASE
    NO_ERR: cs_with_visualization.id_visualizer:= ID_RECEVERS;
            CHANGE_CS   cs_with_visualization
    ...
  END_CASE
END_DO
```

Annexe C

Exemples de description Indra d'outils de développement

Ci−après sont donnés des exemples de description d'états de coordination et d'activation pour différents outils que nous avons intégré dans l'environnement de développement Cybèle.

C.1 Description Indra de l'outil "XDesktop"

```
TOOL_ACT_STATE   OF   XDesktop
COORD
   ON REQ  desktop ( )
   DO   COMMAND "XDesktop −rf XDesktop.P"    // Commande d'activation
        CHANGE−CS   CS_XDesktop
END

TOOL_COORD_STATE    CS_XDesktop   OF   XDestop
COORD
   //Le développeur veut activer un nouvel outil
   ON SIGN  user_ask_tool (IN tool_fct : String, IN c : T_comp_name)
   DO   REQ−ASYNC−AR  (tool_fct, c)
        TO   ../NEW Dev_tool.class = tool_fct
   END

  //Le développeur veut lancer une nouvelle session
  ON  SIGN   user_ask_desktop
  DO REQ_ASYNC_AR  desktop
      TO   ../../NEW   Session/Dev_Tool.name='XDesktop'
   END

   //Le XDesktop se ferme sur la demande du développeur
   ON SIGN  close ( )
   DO NOTIF  close
        TO   ../../../*
   END
END
```

C.2 Description Indra de l'outil "ErrBrowser"

```
TOOL_ACT_STATE OF ErrBrowser
VAR   err_list : T_err_list;
COORD
  ON REQ err_browse (IN c : T_comp_name)
  DO  COMMAND  "/usr/bin/ErrBrowse"
        REQ_SYNC compile (c, err_list)        //Demande la compilation de c
        CHANGE_CS   CS_Err_Browser
        CALL  browse (c, err_list)            //Affiche la liste des erreurs de compilation
  END
END

TOOL_COORD_STATE   CS_ErrBrowser
 COORD
   //Le développeur veut visualiser la ligne d'erreur
   ON SIGN user_select_err (IN c:T_comp_name, IN line: Integer)
   DO REQ_ASYNC_AR show_line (c, line, RED)
        TO  ../Dev_Tool.class=Editor   REL  'collaboration'
   END

 //Le "browser" d'erreurs se ferme
 ON SIGN close
 DO  NOTIF close
       TO   ../*
 END

   //Nouvelle requête de compilation demandée par l'éditeur
   ON REQ err_browse (IN c : T_comp_name)
       FROM  ../*      EX_REL  'collaboration'
   DO  REQ_SYNC compile (c, err_list)
       CALL  browse (c, err_list)
   END

 ON NOTIF close
       FROM   ../Dev_Tool.class = Desktop
 DO   CALL  close
 END

 //Si l'éditeur est fermé, le "browser" d'erreurs se ferme aussi
 ON NOTIF close
      FROM   ../Dev_Tool.class = Editor EX_REL 'collaboration'
```

```
  DO  CALL  close
  END
END
```

C.3 Description Indra de l'outil guideComp

```
TOOL_ACT_STATE    OF   guideComp
COORD
 ON REQ   compile (IN c : T_comp_name, OUT err_list : T_err_list)
      FROM  /Org/Dev_Team/Dev/Session/Dev_Tool
 DO  COMMAND "/usr/bin/glc −p5  <c> <err_list>"
      CHANGE_CS  CS_guideComp_busy
 END
END.

TOOL_COORD_STATE   CS_guideComp_busy
 //Le compilateur est occupé à traiter une requête.
COORD
    //Fin de traitement de la requête
   ON SIGN   end_compilation (IN c : T_comp_name)
   DO  CHANGE−CS   CS_guideComp_free
   END

   // Le compilateur se ferme
   ON  SIGN  close ( )
   DO  NOTIF  close
        TO   ../*
   END

 // Lorsque le XDesktop se ferme, les  autres outils de la session se ferment aussi
 ON  NOTIF  close
      FROM   ../Dev_Tool.class = Desktop
  DO CALL  close
  END
END

TOOL_COORD_STATE   CS_guideComp_free  INHERITS  CS_guideComp_busy
COORD
  RM_COORD   1
```

```
 // OK pour servir une nouvelle requête de compilation
 ON  REQ   compile (IN c : T_comp_name, OUT err_list : T_err_list)
 DO   CHANGE_CS  CS_guideComp_busy
       CALL  compile (c, err_list)
 END
END
```

C.4 Description Indra de l'outil Thésée

```
TOOL_ACT_STATE  OF  Thesee
 COORD
  ON REQ debug (IN c : T_comp_name)
      FROM /Org.name=Bull−IMAG/Dev_Team/Dev/Session/Dev_Tool.class = Editor
      NEW_REL  'collaboration'
  DO  COMMAND  "/usr/bin/Thesee"
       CHANGE_CS   CS_Thesee_busy_with_ed
       CALL   debug  (c)
  END

  ON REQ debug (IN c : T_comp_name)
  DO  COMMAND  "/usr/bin/Thesee"
       CHANGE_CS   CS_Thesee_busy
       CALL   debug  (c)
  END
END

TOOL_COORD_STATE   CS_Thesee_busy
COORD
  //La mise au point de c est terminée
  ON  SIGN  end_debug
  DO NOTIF end_debug TO ../*
      CHANGE_CS  CS_thesee_free
  END

  //Le metteur au point se ferme
  ON SIGN close
  DO NOTIF close TO ../*
  END

  //Le développeur veut visualiser le composant c
  ON SIGN visualize (IN c : T_comp_name)
  DO  REQ_ASYNC_AR  edit  (c)
```

```
      TO ../Dev_Tool.class = Editor   NEW_REL 'collaboration'
      CHANGE_CS   CS_Thesee_busy_with_ed
 END

 //Le développeur veut visualiser les points d'arrêts posés dans c
 ON SIGN bkpts_list (IN c : T_comp_name, IN lines : T_line_list)
 DO  REQ_ASYNC_AR  show_lines  (c, lines, RED)
      TO ../Dev_Tool.class = Editor   NEW_REL 'collaboration'
      CHANGE_CS   CS_Thesee_busy_with_ed
 END

  ON NOTIF close
       FROM   ../Dev_Tool.class = Desktop
  DO   CALL  close
  END

END
```

C.5 Description Indra de l'outil cmacs

```
TOOL_ACT_STATE  OF  emacs
COORD
   ON REQ edit (IN c : T_comp_name)
      DO   COMMAND "lemacs −edit <c>"
           CHANGE−CS   CS_edition
    END

   ON REQ edit (IN c : T_comp_name)
      FROM /Org.name='Bull−IMAG'/Dev_Team/Dev/Session/Dev_Tool.class = Debugger
            NEW_REL   'collaboration'
      DO   COMMAND "lemacs −visualize <c>"
           CS_edition_busy.debugger  =  1;
           CHANGE−CS   CS_edition_busy
    END

   ON REQ   show_line (IN c: T_comp_name, IN no_line : Integer, IN color : Integer)
       FROM /Org.name = Bull−IMAG/Dev_Team/Dev/Session/Dev_Tool.class = ErrBrowser
             NEW_REL   'collaboration'
       DO    COMMAND "lemacs −edit <c>"
             CS_edition_busy.browserErr  =  1;
             CHANGE−CS  CS_edition_busy
```

```
            CALL  show_line (c, no_line, color)
        END

        ON REQ   show_lines (IN c: T_comp_name, IN lines: T_line_list, IN color : Integer)
            FROM /Org.name = Bull−IMAG/Dev_Team/Dev/Session/Dev_Tool.class = Debugger
                 REL   'collaboration'
            DO    COMMAND "lemacs −visualize <c>"
                  CS_edition_busy.debugger  =  1;
                  CHANGE−CS  CS_edition_busy
                  CALL  show_lines (c, lines, color)
        END
        END
```

TOOL_COORD_STATE   CS_edition

```
  COORD
  //Le développeur demande la compilation de c
  ON SIGN  user_ask_compile (IN c : T_comp_name)
  DO REQ_ASYNC_AR   errBrowse (c)
       TO ../Dev_Tool.class=ErrBrowser  NEW_REL   'collaboration'
       cs_edition_busy.browserErr := 1;
       CHANGE−CS   CS_edition_busy
  END

  //Le développeur demande la mise au point de c
  ON SIGN user_ask_debug (IN c : T_comp_name)
  DO REQ_ASYNC_AR debug (c)
       TO ../Dev_Tool.class=Debugger  NEW_REL    'collaboration'
       cs_edition_busy.debugger  :=  1;
       CHANGE−CS   CS_edition_busy
  END

  //L'éditeur se ferme
  ON SIGN close
  DO  NOTIF close
        TO   ../*
  END

  //On accepte les requêtes d'édition qui viennent des autres outils de la session
  ON REQ edit (in c : T_comp_name)
       FROM  ../Dev_Tool
```

```
  DO CALL edite (c)
   END

 ON NOTIF close
   FROM ../Dev_Tool.class = Desktop
  DO  CALL close
  END
END
```

TOOL_COORD_STATE CS_edition_busy INHERITS CS_edition

```
VAR
  browserErr : Integer = 0;
  debugger : Integer = 0;
COORD
   RM_COORD 1,2,4

 ON SIGN user_ask_compile (IN c : T_comp_name)
 DO REQ_ASYNC_AR errBrowse (c)
   TO ../Dev_Tool.class=ErrBrowser REL 'collaboration'
   cs_edition_busy.browserErr := 1;
    END

 ON SIGN user_ask_debug (IN c : T_comp_name)
 DO REQ_ASYNC_AR debug (c)
   TO ../Dev_Tool.class=Debugger REL 'collaboration'
        cs_edition_busy.debugger := 1;
 END

ON SIGN set_bkpt (IN c: T_comp_name, IN line : Integer)
DO REQ_ASYNC_AR set_bkpt(c, line)
  TO ../Dev_Tool.class = Debugger EX_REL 'collaboration'
END

ON REQ show_line (IN c: T_comp_name,IN line: Integer,
            IN color : Integer)
  FROM ../Dev_Tool EX−REL 'collaboration'
DO CALL show_line(c, line, color)
END

ON REQ show_lines (IN c: T_comp_name,IN lines: T_line_list, IN color : Integer)
```

```
    FROM ../Dev_Tool  EX−REL 'collaboration'
  DO CALL show_lines (c, lines, color)
   END


  ON NOTIF close
     FROM   ../Dev_Tool.class=ErrBrowser
          EX_REL  'collaboration'
     DO  errBrowser := 0;
          RM_REL 'collaboration' ../Dev_Tool.class = ErrBrowser
          IF  debugger  =  0
          THEN  CHANGE_CS  CS_edition
          END
    END

  ON NOTIF close
     FROM   ../Dev_Tool.class=Debugger
          EX_REL  'collaboration'
     DO   debugger  :=  0;
          RM_REL 'collaboration' ../Dev_Tool.class = Debugger
          IF  errBrowser  =  0
          THEN  CHANGE_CS  CS_edition
          END
     END

  ON  NOTIF  end_debug
     FROM   ../Dev_Tool.class=Debugger
          EX_REL  'collaboration'
     DO   debugger  :=  0;
          RM_REL 'collaboration' ../Dev_Tool.class = Debugger
          IF  errBrowser  =  0
          THEN  CHANGE_CS  CS_edition
          END
     END
  END
```

Annexe D

Apports de Guide comme système support

Les éléments du système réparti et orienté objet Guide qui ont joué un rôle important pour la mise en œuvre de l'environnement d'exécution du langage Indra sont décrits ci−après. Sur certains points, afin de mieux cerner cet apport, la mise en œuvre actuelle est comparée à une mise en œuvre au dessus d'un système plus classique tel qu'Unix.

Partage de code et de données entre agents répartis

La machine d'exécution est constituée d'un ensemble de données qui varient dynamiquement, et de procédures de manipulation de ces données qui peuvent être exécutées par les clients du mécanisme de coordination. Ces derniers sont répartis sur un ensemble de sites.

Conceptuellement, la machine d'exécution est donc une entité *passive* et *partagée* entre des clients répartis.

Avec le système Guide, cette vision conceptuelle est directement représentable en un ensemble d'objets Guide : d'une part les objets Guide sont constitués d'un ensemble de données et de méthodes de manipulation de celles−ci ; d'autre part, ces objets sont passifs et partagés : ils sont rendus accessibles à des activités réparties.

Avec le système Unix, nous devons faire de la machine d'exécution une entité active, c'est dire composée d'un ensemble de processus Unix. Le concept de processus est en effet le seul moyen de partager du code et des données de manière répartie au dessus du système Unix standard. Décentraliser la gestion de la machine d'exécution de manière aussi poussée qu'il est possible de le faire avec Guide est alors très difficile : placer un processus par nœud de l'arbre instancié effectif (voir architecture proposée en <Impl>) rend le nombre de processus trop important.

Par ailleurs, avec Guide, les communications entre les nœuds de l'arbre instancié effectif correspondent à des appels de méthodes. Seuls les appels effectuées entre deux objets distants entraînent des communications inter−processus. Dans la deuxième version de Guide, la rapidité d'exécution d'un appel local n'est que six fois supérieure à celle d'un appel de procédure intra−processus dans Unix (0.0038 contre 0.00059 millisecondes). Cette amélioration est très intéressante car la majorité des coordinations ne font intervenir que des objets locaux.

Avec Unix, toute communication entre les nœuds de l'arbre instancié effectif entraîne des communications inter−processus, diminuant les performances d'exécution et requérant la possibilité d'utiliser simultanément un très grand nombre de sockets de communication. Optimiser les appels locaux en utilisant un mécanisme de mémoire partagée n'est en effet pas du tout évident, impliquant de jongler entre deux types de communications inter−processus.

Persistance des données

Une coordination n'est autre qu'un ensemble de données, créées par un client et transmises à un ou plusieurs autres clients. La réception d'une coordination peut prendre place après que son créateur ait terminé son exécution.

Avec le système Unix, les données transmises doivent être copiées pour assurer que celles−ci existent encore lors de leur réception par différents clients.

Avec le système Guide, les données qui constituent une coordination sont représentées par un objet. La transmission d'une coordination se limite à la transmission d'une référence d'objet, au travers d'appels de méthodes. Le dernier appel dépose cette référence dans des buffers associés aux clients récepteurs. Outre la simplicité de mise œuvre, les données ne sont pas copiées : elles sont placées dans de la mémoire partagée. L'accès à ces données par une activité distante entraîne une "extension" d'activité.

Par ailleurs, le système Guide fournit un ramasse−miettes qui se charge de détruire les objets qui ne peuvent plus être accédés (c.a.d dont la référence n'est présente dans aucun autre objet qui ne doit pas être détruit).

Avec le système Unix, il faut gérer la destruction des données qui composent une coordination lorsque celle−ci a été reçue par tous les clients qui devaient la recevoir.

Serveur de noms

Lorsqu'un utilisateur active un outil, il faut savoir si l'organisation dans laquelle l'outil doit être placé est déjà "existante" (représentée par un ensemble d'objets Guide déjà crées) ou non. Les objets qui composent cette organisation sont persistents : leur durée de vie est supérieure à celle de l'outil qui a entraîné leur création.

Le système Guide répond directement à ce besoin au travers d'un serveur de noms qui est disponible sur toutes les machines connectées, même en cas de panne de l'une des machines (1).

Le système Unix implique en revanche de stocker dans un fichier l'information spécifiant que l'organisation existe. Comment dès lors assurer un accès réparti à ce fichier ? Placer celui−ci sur une machine donnée pose un problème en cas de panne de cette machine. Le dupliquer sur plusieurs machines implique de gérer la cohérence des informations qu'il conserve.

Notions de types et de classes

Guide distingue la signature des méthodes exportées par un objet (type) de son implémentation (classe). Ceci a été utile pour réaliser le lien entre la machine d'exécution et la partie compilée de l'environnement d'exécution du langage Indra.

Tout environnement d'exécution qui comprend une partie compilée et une machine d'exécution implique que l'interface fournie par la partie compilée à la machine d'exécution soit clairement et explicitement définie. Cette interface peut être composée de définition de données et de méthodes. Elle doit être identique pour tous les composants compilés : dans notre cas, toutes les descriptions d'états de coordination sont compilées de manière à produire des objets d'interfaces identiques. Les notions fournies par Guide répondent directement à ce besoins : ces objets sont de même type et de classes différentes.

Conformité

La relation de conformité entre types Guide a permi de réaliser un objet ayant la fonction de Bus de messages qui se contente de "router" des messages dont il ne connaît pas la sémantique. Le bus conçu est ainsi générique car chaque client spécifie sa propre fonction de sélection d'un message au travers d'un objet Guide de type conforme à un type prédéfini par le Bus de message. De même, les clients sont très libres pour structurer les messages qu'ils émettent : un message doit simplement être d'un type conforme a un type prédéfini par le Bus de message. Ceci permet de typer les champs d'un message. En conséquence, la technique classique de sélection, basée sur le "pattern matching", n'est qu'un cas particulier de l'ensemble des techniques qui peuvent être utilisées.

Synchronisation

Les mécanismes de synchronisation fournis par Guide ont été très utiles pour gérer des situations d'asynchronisme dans lesquelles deux flots d'exécution doivent se "rencontrer". En permettant de bloquer un clôt jusqu'à ce qu'une condition soit remplie, aucune attente active n'a été engendrée. La possibilité d'associer un délai maximum de blocage a également été utile, notamment pour détecter des situations d'interblocage.

Liaison tardive

Le mécanisme de liaison tardive entre objets Guide a été indispensable pour assurer une indépendance entre la partie compilée et la machine d'exécution : la partie compilée peut ainsi évoluer sans qu'il faille recompiler la machine d'exécution.

Ceci permet notamment d'ajouter, de retirer ou de remplacer un outil par un autre sans avoir à stopper l'environnement d'exécution.

Bibliographie

[1] Accetta M.J., Baron R., Bolosky W., Golub D., Rashid R., Tevanian A., Young M., ''Mach : a new kernel foundation for Unix development'', *Proceedings of the USENIX 1986 Summer Conference*, pp. 93−112, 86.

[2] Ambriola V., Ciancarini P., Montangero C., ''Software Process Enactment in Oikos'', *Proceedings of ACM SIGSOFT Conference on Software Development Environment, ACM SIGSOFT Software Engineering Notes*, pp. 183−192, 90.

[3] Ambriola V., Montangero C, ''Oikos at the age of Three'', *Lecture Notes in Computer Sciences 635*, pp. 84−93, Sept 92.

[4] Arnold J., Memmi G., ''Control Integration and its Role in Software Integration'', *Toulouse 92 Software Engineering Conference*, pp. 1−15, Dec 92.

[5] Arnold J., Memmi G., Kaplan S.M., ''Software Brokers and the Evolution of Software Integration'', *Submitted to 15th International Conference on Software Engineering*, Baltimore, Maryland pp. 1−18, May 93.

[6] Balter R., Bernadat J., Decouchant D., Duda A., Freyssinet A., Krakowiak S., Meysembourg M., Le Dot P., Nguyen Van H., Paire E., Riveill M., Roisin C., Rousset de Pina X., Scioville R., Vandôme G., ''Architecture and Implementation of Guide, an Object−oriented Distributed System'', *Computing Systems*, 4(1), pp. 31−67, 91.

[7] Bancilhon F., Barbedette G., Benzaken V., Delobel C. et al., ''The Design and Implementation of O2, an Object−Oriented Database System'', *LNCS 334, Springer−Verlag*, Sept 88.

[8] Bandinelli S., Fugetta A., Ghezzi C., Grigolli S, ''Process Enactment in SPADE'', *Lecture Notes in Computer Sciences 635*, pp. 67−83, Sept 92.

[9] Barghouti N. S, ''Supporting Cooperation in the MARVEL Process−Centered SDE'', *5th ACM SIGSOFT Symposium on Software Development Environment*, pp. 21−31, Washington DC Dec 92.

[10] Barghouti N.S., Kaiser G.E., ''Modeling Concurrency in Rule−Based Development Environments'', *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pp. 15−27, Dec 90.

[11] Beust C., Nahaboo C. , ''The Koala Bus Group Communication Software'', *Programmer's manual for version 1.28, Bull Document* , pp. 1−35, Fev 93.

[12] Black E., ''ATIS, CIS, PCTE and Software Backplane'', *Toulouse 91 : Software Engineering & its applications*, pp. 601, EC2, 269−287 rue de la Garenne 92024 Nanterre Cedex France, Dec 91.

[13] Boudier, Gallo, Minot, Thomas, ''An Overview of PCTE & PCTE+'', *ACM Software Engineering Notes*, 13(15), pp. 248−257, Nov 88.

[14] Brown A., ''Control Integration Through Message Passing in a Software Development Environment'', *Software Engineering Institute Technical Report CMU/SEI−92−TR−??*, pp. 1−29, Sept 92.

[15] Brown A., Feiler P., Wallnau K., ''Understanding Integration in a Software Development Environment'', *Software Engineering Institute Technical Report CMU/SEI−91−TR−31*, pp. 1−26, Jan 92.

[16] Brown A., Feiler P., Wallnau K., ''Past and Future Models of CASE Integration'', *CASE 92, Canada*, pp. 36−45, July 92.

[17] Brown A., Mc Dermid J., ''Learning from IPSE's Mistakes'', *IEEE Software*, pp. 23−28, March 92.

[18] Brown A., Penedo M., ''An Annotated Bibliography on Integration in Software Engineering Environments'', *ACM Software Engineering Notes*, 17(3) pp. 47−55, July 92.

[19] Brown A.W, Feiler P.H.,, ''An Analysis Technique for Examining Integration in a Project Support Environment'', *5th ACM SIGSOFT Symposium on Software Development Environments*, pp. 139−148, Washington DC Dec 92.

[20] Bull, ''OMG Object Request Broker Submission'', Feb 91.

[21] Military Standard Common APSE Interface Set (CAIS), ''Os DoD'', Jan 85.

[22] Proposed Military Standard − Common Ada Programming Support Environnement (APSE) Interface Set (Revision A), *Vol I,II,III, Softech*, Jan 88.

[23] Catill V., Balter R., Harris N.R., Rousset de Pina X., ''The COMANDOS Distributed Application Platform'', *Research Report ESPRIT, Project 2071 Springer−Verlag*, Vol 1 pp. 1−312, 93.

[24] Cap Gemini Innovation, ''PROCESS WEAVER Concepts and tools overview'',

[25] Cagan M., ''The HP SoftBench Environment: An Architecture for a New Generation of Software Tools'', *Hewlett−Packard Journal*, Vol 41(3), pp. 36−47, June 90.

[26] Carey M.J, DeWitt D.J, ''An Overview of the EXODUS Project'', *IEEE Database Engineering*, 10(2), pp. 47−54, juin 87.

[27] Chabert A., ''Manuel d'utilisation des outils de l'environnement de développement Cybele'', Sept 93.

[28] Chen M., Norman R., ''A Framework for Integrated CASE'', *IEEE Software*, pp. 18−22, March 92.

[29] Chen P.P, ‘‘The Entity−Relationship model : Toward a unified view of Data’’, *ACM TODS*, 1(1), pp. 9−36, mars 76.

[30] Chevalier P.Y„ ‘‘A Replicated Object Server for Distributed Object−Oriented System’’, *11th Symposium on Reliable Distributed Systems*, Oct 92.

[31] Clement D., ‘‘A Distributed Architecture for Programming Environment’’, *4th ACM SIGSOFT Symposium on Software Development Environments*, pp. 11−21, Dec 90.

[32] Clemm G., Osterweil L., ‘‘A Mechanism for Environment Integration’’, *ACM Transactions on Programming Languages & Systems*, Vol. 12(1), pp. 1−25, Jan 90.

[33] Clemm G., Osterweil L., ‘‘The Toolpack/IST Approach to Extensibility in Software Environment’’, *Lecture Notes in Computer Sciences Ada Software Tool Interfaces*, pp. 133−163, 83.

[34] Clow G., Ploedereder E., ‘‘Issues in Designing Object Management Systems’’, *Lecture Notes in Computer Sciences 467*, pp. 204−209, 90.

[35] Cutkosky M.R., Engelmore R.S., Fikes R.E., Genesereth M.R., Gruber T.R., Mark W.S., Tenenbaum J.M., Weber J.C, ‘‘PACT: An Experiment in Integrating Concurrent Engineering Systems’’, *IEEE Software*, pp. 28−37, Jan 93.

[36] Date C.J, ‘‘An Introduction to Database Systems’’, *Vol I of Addison−Wesley Systems Programming Series*, Reading, MA: Addison−Wesley Pub. Co., Inc.pp. 144−154, 86.

[37] Digital Equipment Corporation, Hewlett Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc. and SunSoft, Inc., ‘‘The Common Object Request Broker: Architecture and Specification’’, OMG Document Number 92.12.1, Dec 91.

[38] Deiters W. Gruhn V., ‘‘Managing Software Process in the environment MELMAC’’, 4th ACM SIGSOFT Symposium on Software Development Environments, pp 193−205, dec 90.

[39] Dewan P., Rield J.„ ‘‘Toward Computer Supported Concurrent Software Engineering’’, *IEEE Computer*, pp. 17−27, Jan 93.

[40] Digital Equipment Corporation, ‘‘DEC FUSE EnCASE Manual’’, *Order Number AA−PKRXA−TE*, pp. 1−118, Dec 91.

[41] Digital Equipment Corp., SiliconGraphics Computer Systems, SunSoft Inc., ‘‘The CASE Interoperability Message Sets : Release 1.0’’, pp. 1−179, Oct 92.

[42] Dowson M., ‘‘Integrated Project Support with ISTAR’’, *IEEE Software*, pp. 6−15, Nov 87.

[43] Dowson M.,, ‘‘Fundamental Software Process Concepts’’, *Proceedings of the 1st*

European Workshop on Software Process Modeling, Milano, May 91.

[44] ECMA (European Manufacturers Association), ''Reference Model for Frameworks of Software Engineering Environments'', *NIST Special Publication 500−20 & TR ECMA TR/55, 2nd Edition*, Dec 91.

[45] Emmerich W., Schafer W., Welsh J, ''Suitable Database for Process−centred Environments Do not yet Exist'', *Lecture Notes in Computer Sciences 635*, pp. 94−98, Sept 92.

[46] Fernström C., Narfelt K.H., Ohlsson L., ''Software Factory Principles, Architecture and Experiments'', *IEEE Software*, 9(2), pp. 36−44, March 92.

[47] Fernstrom C., Ohlsson L., ''Integration Needs in Process Enacted Environments'', *Proceedings of the 1st International Conference on the Software Process*, pp. 142−158, Oct 91.

[48] Ferrans J., Hurst D., Sennet M., Covnot B., Ji W., Kajka P., Ouyang W., ''HyperWeb: A Framework for Hypermedia−Based Environments'', *5th ACM SIGSOFT Symposium on Software Development Environments, Washington D.C.*, pp. 1−10, Dec 92.

[49] Fleming R., Wybolt N.„ ''CASE Tools Frameworks'', *Unix Review*, 8(12), pp. 24−32, Dec 90.

[50] Freyssinet A. Krakowiak S., Lacourte S., ''A generic object−oriented virtual machine'', *Proc. of the 2nd International Workshop on Object Orientation in Operating Systems*, Palo Alto, Oct 91.

[51] Garlan D., ''A Structural Approach to the Maintenance of Structure Oriented Environments'', *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environment*, Palo Alto pp. 160−170, Dec 86.

[52] Garlan D., ''Views for Tools in Integrated Environments'', *Lecture Notes in Computer Sciences*, Vol. 244 pp. 314−343, June 86.

[53] Garlan D., Ilias E., ''Low−cost, Adaptable Tool Policies for Integrated Environments'', *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pp. 1−10, Dec 90.

[54] Garlan D., Kaiser G.E., Notkin D., ''Using Tool Abstraction to Compose Systems'', *IEEE Computer*, 25(6), pp. 30−38, June 92.

[55] GIE Emeraude, ''Environment Guide'', *Sales Department, 153 Bureaux de la Colline, 92213 St Cloud Cedex*, 92.

[56] Griffiths P., Oldfield D., Legait A., Menes M., Oquendo F.„ ''ALF: Its Process Model and its Implementation on PCTE'', *Software Engineering Environments − Research and Practice*, Ellis Horwood 89.

[57] Habermann A.N., Notkin D., ‘‘Gandalf: a Software Development Environments’’, *IEEE Transactions on Software Engineering* , pp. 1117−1127, Dec 86.

[58] Harrison W., ‘‘RPDE: A FrameWork for Integrated Tool Fragments’’, *IEEE Software*, Vol. 4 (6), pp. 46−56, Nov 87.

[59] Harrison W., Ossher H., Kavianpour M., ‘‘OOTIS : extending PCTE with fine−grained tool composition’’, *PCTE no 11*, pp. 11−19, Dec 92.

[60] Harrison W., Kavianpour M., Ossher H., ‘‘Integrating Coarse−Grained and Fine−Grained Tool Integration’’, *CASE 92 Proceedings*, pp. 1−34, July 92.

[61] Hazzah A., ‘‘Making Ends Meet: Repository Manager’’, *Software Magazine*, 9(15), pp. 59−71, Dec 89.

[62] Heimbigner D.,, ‘‘The Process Wall: A Process State Server Approach to Process Programming’’, *5th ACM SIGSOFT Symposium on Software Development Environment*, Washington DC pp. 159−169, Dec 92.

[63] Hoffman C., Kramer B., Dinter B., ‘‘Multi Paradigm Description of System Development Process’’, *Lecture Notes in Computer Sciences 635*, pp. 123−137, Sept 92.

[64] Ison R., ‘‘An Experimental ADA Support Environment in the HP CASEdge Integration Framework’’, *Software Engineering Environments, number 467 in Lecture Notes in Computer Science*, pp. 179−193, Springer−Verlag 90.

[65] Jamrozik H., ‘‘Aide à la mise au point des applications parallèles et réparties à base d’objets persistents’’, *Thèse de doctorat, Univ. Joseph Fourier*, pp. 1−123, mai 93.

[66] Jordan M., ‘‘An Extensible Programming Environment for Modula−3’’, *SIGSOFT 90*, pp. 66−76, Dec 90.

[67] Kaiser G. Kaplan S.M., Micallef J., ‘‘Multi−User, Distributed Language Based Environments’’, *IEEE Software* , pp. 58−67, Nov 87.

[68] Kaiser G., Barghouti N.S., ‘‘Modeling Concurrency in Rule Based Development Environment’’, *IEEE Expert*, Dec 90.

[69] Perry D.E, Kaiser G.E, ‘‘Models of Software Development Environments’’, *IEEE Transactions on Software Engineering*, Vol. 17(3), pp. 283−295, March 91.

[70] Kaiser G., Feiler P., Popovich S., ‘‘Intelligent Assistance for Software Development and Maintenance’’, *IEEE Software*, pp. 40−49, May 88.

[71] Kadia R., ‘‘Issues Encountered in Building a Flexible Software Development Environment, Lessons from the ARCADIA Project’’, *5th ACM SIGSOFT Symposium on Software Development Environment*, pp. 169−180, Washington DC Dec 92.

[72] Kaplan S.M, Johnson R.E, Campbell R.H., Kamin S.N. Purtilo J.M., Harandi M.T, Liu

J.W., ‘‘An Architecture for Tool Integration’’, *Lectures Notes in Computer Sciences*, Vol. 244 pp. 112−125, June 86.

[73] Kaplan S. M.,, ‘‘Operating Systems Support for Collaborative Work’’, *IWOOOS*, 92.

[74] Kaplan S.M., Tolone W. J., Carroll A.M., Bogia D.P., Bignolli C., ‘‘Supporting Collaborative Software Development With Conversation Builder’’, *5th ACM SIGSOFT Symposium on Software Development Environment*, Washington DCpp. 11−20, Dec 92.

[75] Krakowiak S., Meysembourg M., Nguyen Van H., Riveill M., Roisin C., ‘‘Design and Implementation of an Object−oriented, Strongly Typed Language for Distributed Applications’’, *Journal of Object−oriented Programming*, 3(3), pp. 11−22, sept−oct 90.

[76] Lacourte S., ‘‘Exception in Guide, an object−oriented language for distributed applications’’, *Proc. ECOOP’91, Lectures Notes in Computer Sciences (512), Springer−Verlag*, pp. 268−287, Geneva, July 91.

[77] Legras I., ‘‘ODE (OSF Development Environment)’’, *User’s Guide*, pp. 1−140, juin 92.

[78] Lewis G. R., ‘‘CASE Integration Frameworks’’, *SunTech Journal*, 3(5), pp. 50−51, Nov 90.

[79] Lonchamp J., Godart C., Derniame J.C, ‘‘Les environnements integrés de production de logiciel’’, *Technique et science informatiques*, 11(1), pp. 31−95, 92.

[80] Maier D., Stein J., ‘‘Development and Implementation of an Object−Oriented DBMS’’, *In Schriver B. and Wegner P. "Research Directions in Object−Oriented Programming*, MIT Press pp. 355−392, 97.

[81] Martin D., ‘‘La norme de référentiel IRDS : objectifs et fonctions’’, *Génie Logiciel & Systèmes Experts*, (22), pp. 16−23, Mars 91.

[82] Meyers S., ‘‘Difficulties in Integrating Multiviews Development Systems’’, *IEEE Software*, pp. 49−57, Jan 91.

[83] Meysembourg M.L, ‘‘Modèle et langage à objets pour la programmation d’applications réparties’’, *Thèse de 3ème cycle, Institut National Polytechnique de Grenoble*, Juil 89.

[84] Mi P., Scacchi W., ‘‘Process Integration in CASE Environments’’, *IEEE Software*, 8(2), pp. 45−53, March 92.

[85] Nelson B., ‘‘Remote Procedure Call’’, *PhD Thesis, Carnegie Mellon University, Technical Report CMU−CD−81−119*, 81.

[86] Notkin D., ‘‘The RelationShip Between Software Environments and the Software

Process'', *ACM 88*, pp. 107−109, 88.

[87] Oliver H., ''Adding Control Integration to PCTE'', *Software Development Environments and CASE Technology, number 509 in Lecture Notes in Computer Sciences*, pp. 69−80, Springer−Verlag 91.

[88] Open Software Foundation, *OSF DEC 1.0 Administration Guide*, Vol(1), August 92.

[89] Oquendo F., Boudier G., Gallo F., Minot R., Thomas I., ''The PCTE+'s OMS, A Software Engineering Distributed Database System for Supporting Large−Scale Software Development Environments'', *Proceedings of the 2nd International Symposium on Database Systems for Advanced Applications*, Tokyo Japan pp. 1−12, Apr 91.

[90] Osher H. , ''Object Request Brokers'', *Byte*, pp. 172−174, Jan 92.

[91] Osterweil L., ''A Process Object Centered View of Software Environment Architecture'', *Lecture Notes in Computer Sciences*, Vol 244 86.

[92] Osterweil L.,, ''Software Processes are Software Too'', *Proceedings of the 9th International Conference on Software Engineering*, pp. 2−13, March 87.

[93] Paseman W.,, ''Tools on a New Level'', *Unix Review*, 7(6), pp. 69−77, June 89.

[94] Penedo M. H., Ploedereder E., Thomas I., ''Object Management Issues for Software Engineering Environments'', *SIGPLAN NOTICE*, Vol. 24(2), pp. 226−234, Feb 89.

[95] Perry D.A., Kaiser G., ''Infuse: A Tool or Automatically Managing & Coordinationg Source Changes in Large Systems'', *ACM fifteenh Annual Computer Science Conference*, pp. 292−299, Feb 87.

[96] Quester R., ''obTIOS: A CAx−Framework Service for Building Concurrent Engineering Environments'', *5th ACM SIGSOFT Symposium on Software Development Environments*, pp. 32−40, Washington DC Dec 92.

[97] Raynal M., ''Order Notions and Atomic Multicast in distributed Systems : A short survey'', *IRISA Publication Interne no 524*, pp. 1−14, Mars 90.

[98] Reiss S., ''Interacting with the FIELD Environment'', *Brown University Department of Computer Science, Technical Report No CS−89−51*, pp. 1−25, May 89.

[99] Reiss S., ''Connecting Tools Using Message Passing in the FIELD Environment'', *IEEE Software*, pp. 57−66, July 90.

[100] Reiss S., ''PECAN: Programming Development Systems that Support Multiple Views'', *IEEE Transactions on Software Engineering*, pp. 276−285, March 85.

[101] Roncancio C., ''Interopérabilité entre SGBD : systèmes fédérés et systèmes

multi−bases'', *A paraître dans Techniques et Sciences Informatiques*, pp. 1−37.

[102] Rosenblatt W.R., Wileden J.C., Wolf A.L., ''OROS: Toward a Type Model for Software Development Environments'', *OOPSLA Conference Proceedings*, pp. 297−304, Oct 89.

[103] Satyanarayanan M., ''A Survey of Distributed File Systems'', *Rapport Technique no CMU−CS−89−116, Pittsburgh PA (USA), Dept of Computer Sciences, Carnegie Mellon University*, Fev 89.

[104] Schafer W., Weber H., ''The ESF−Profile'', *Handbook of CASE*, 89.

[105] Sheth A., ''Building Federated Database Systems'', *Distributed Computing Technical Comitee Newsletter (Quarterly)*, 10(2), Nov 88.

[106] Snodgrass R., ''The Interface Description Language: Definition and Use'', *Rockville, MD: Computer Science Press*, 89.

[107] Snodgrass R., Shannon K., ''Supporting Flexible and Efficient Tool Integration'', *Lecture Notes in Computer Sciences*, Vol. 244 pp. 290−311, June 86.

[108] Snodgrass R., Shannon K., ''Fine Grained Data Management to Achieve Evolution Resilience in a Software Development Environment'', *4th ACM SIGSOFT Symposium on Software Development Environments*, pp. 144−155, Dec 90.

[109] Stonebraker M.E, Rowe L.A, ''The Design of POSTGRES'', *Proceedigns of ACM SIGMOD Conference on Management of Data*, Washington pp. 340−355, mai 86.

[110] Sullivan K.J, Notkin D., ''Reconciling Environment Integration and Component Independance'', *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pp. 22−33, July 90.

[111] Sullivan K.J, Notkin D., ''Reconciling Environment Integration and Software Evolution'', *ACM Transactions on Software Engineering and Methodology*, Vol 1(3), pp. 229−268, July 92.

[112] SunSoft White Paper, ''Introduction to the ToolTalk Service'', pp. 1−15, June 91.

[113] Sutton S.M., Heimbigner D., Osterweil L., ''Process Constructs for Managing Change in Process Centered Environment'', *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments*, pp. 206−217, Dec 90.

[114] Taylor R. et al, ''Foundations for the ARCADIA Environment Architecture'', *Proceedings of the Symposium on Practical Software Development Environments*, pp. 1−13, Nov 88.

[115] Thomas I., , ''PCTE Interfaces : Supporting Tools in Software Development Environments'', *IEEE Software*, 6(6), pp. 15−22, Nov 89.

[116] Thomas I., , ''Tool Integration in the PACT Environment'', *Proceedings of the 11th International IEEE Conference on Software Engineering*, pp. 13−22, May 89.

[117] Thomas I., Nejmah B., ‘‘Definitions of Tool Integration for Environments’’, *IEEE Software*, pp. 29−35, March 92.

[118] Thomas I., ‘‘Goals and Requirements for Integration Frameworks’’, *ACM SIGSOFT*, 15(6), pp. 201−203, Dec 90.

[119] Wasserman A., ‘‘Visible Connections’’, *Unix Review*, 4(10), Oct 86.

[120] Wasserman A.,, ‘‘Tool Integration in Software Development Environments’’, *F. Long Editor, Software Engineering Environment, number 467 in Lectures Notes in Computer Science*, pp. 138−150, Sept 89.

[121] Wileden J.C., Wolf A., ‘‘Environment Object Management Technology: Experiences, Opportunities and Risks’’, *Lecture Notes in Computer Sciences 467*, pp. 251−259, 90.

[122] Wileden J.C., Wolf A.L., Rosenblatt W.R., Tan P.L, ‘‘Specification Level Interoperability’’, *Communications of the ACM* , Vol34(5), pp. 73−87, May 91.

[123] Wolf A.L., Clarke L. A., Wileden J.C. , ‘‘The AdaPIC Tool Set: Supporting Interface Control and Analysis througout the Software Development Process’’, *IEEE Transaction on Software Engineering* , 15(3) pp. 250−263, March 89.

[124] Wolfgang E., Schafer W., Welsh J., ‘‘Suitable DataBases for Process Centred Environments Do Not Yet Exist’’, *Lecture Notes in Computer Sciences 635*, Sept 92.

[125] Wybolt N., ‘‘Perspectives on CASE Tool Integration’’, *ACM Software Engineering Notes*, 16(3), pp. 56−60, July 91.

Introduction

Chapitre I

Environnements de développement intégrés

Chapitre II

Intégration du contrôle : position du problème

Chapitre III

Proposition d'un modèle et d'un langage de description de l'intégration du contrôle

Chapitre IV

Le langage Indra : une illustration par l'exemple

Chapitre V

Un environnement d'exécution pour le langage Indra

Chapitre VI

Evaluation

Conclusion