Ingénierie Concurrente en Génie Logiciel: Céline

Sergio Garcia Camargo

UNIVERSITE JOSEPH FOURIER – GRENOBLE I

THESE

pour obtenir le grade de

DOCTEUR de l'Université Joseph Fourier de Grenoble

(Arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

Discipline : Informatique

présentée et soutenue publiquement par

Sergio Garcia-Camargo

le 20 décembre 2006

Ingénierie Concurrente en Génie Logiciel : Céline

Directeur de thèse :

Jacky ESTUBLIER

JURY :

	Président de jury
Claude Godart, Université Henri Poincaré, Nancy	Rapporteur
Jean-Marc Andréoli. Xerox Research Center. Grenoble	Rapporteur
	Examinateur
	Examinateur
Jacky Estublier, Directeur de recherche au CNRS, Grenoble	Directeur de thèse

Thèse préparée au sein du Laboratoire LSR – Équipe ADELE

Table de matières

UNIVERSITE JOSEPH FOURIER – GRENOBLE I 1
THESE 1
Table de matières 2
Chapitre I. Introduction 5
1. Contexte 5
2. Portée du travail 7
3. Problématique 7
3.1. Le problème de la concurrence 7
3.2. Hétérogénéité 8
4. Objectifs 8
5. Plan de la thèse 9
5.1. Etat de l'art 9
5.2. Support à l'ingénierie concurrente 9
5.3. Politiques 9
5.4. Augmentation de l'information contextuelle 10
5.5. Validation 10
6. Cadre de l'expérimentation 10
7. Résumé des principaux résultats 11
Chapitre II. Etat de l'art 12
1. Les procédés 12
2. Les procédés logiciel 13
2.1. Modèles de procédé 13
2.2. Langages de modélisation de procédés (PMLs) exécutables. 16
2.3. Classification des PMLs 17
2.4. PMLs basées sur des règles 17
2.4.1. Marvel 18
2.4.2. Merlin 18
2.4.3. EPOS 19
2.5. PMLs basées sur des réseaux de pétri 19
2.5.1. SPADE 19
2.5.2. FunSoft 20
2.6. PMLs procéduraux 20
2.6.1. APPL/A 20
2.7. PMLs se focalisant sur les types abstrait de donnés 21
2.7.1. ADELE 21
2.8. Discussion 21
2.9. Contrôle de la concurrence dans les bases de données 22
2.10. Modèles de transactions avancées 25
2.11. Transactions imbriquées 25
2.12. Split Transactions 26
2.13. Hiérarchie des transactions coopératives 27
2.14. Structure 27
2.15. Copies multiples 28
2.16. Définition des critères de correction 28
2.17. Exemple 29
2.18. COO 30
2.19. Discussion 31

3. Systèmes de gestion de la configuration 32
3.1. Traçabilité des modifications 33
3.2. Modèle de produit et sélection de configurations 35
3.3. Contrôle des espaces de travail 35
3.4. Sélection de configurations 35
3.5. Contrôle des changements 36
3.6. Gestion des utilisateurs multiples 37
4. Fusion 38
4.1. Fusion textuelle 39
4.2. Fusion syntaxique 40
4.3. Fusion sémantique 40
4.4. Fusion structurelle 40
5. Travail Coopératif Assisté par Ordinateur 40
5.1. Edition coopérative 41
5.2. Observations 42
6. Conscience du contexte 43
6.1. Conscience du contexte dans les éditeurs collaboratifs 44
6.2. Systèmes de conscience du contexte avec des copies multiples 45
6.3. Palantir 46
6.4. State Treemap 46
6.5. Discussion 47
7. Conclusions 48
Chapitre III. Support à l'ingénierie concurrente 52
1. Hypothèses de base 53
1.1. Génie logiciel coopératif 53
1.2. Espaces de travail 53
1.3. Entités logiques 55
1.4. Opérations basiques 56
1.5. Communication entre espaces de travail 56
1.6. Copies de référence et groupes 57
Chapitre IV. Langage de définition de politiques 59
1. Introduction 59
2. Contexte de la politique 59
2.1. Politiques : déterminisme et non déterminisme 60
2.2. Rôles 61
2.3. Contrôle d'accès 62
2.4. Contrôle de la topologie 62
2.5. Contrôle de la concurrence 65
3. Modèle d'exécution du procédé 69
4. Application de une politique d'ingénierie concurrente 73
4.1. Introduction 73
L'application d'une politique correspond à la procédure suivante : 73
4.2. Validation d'une politique : les transferts 74
4.3. Validation d'une politique : La concurrence 75
Chapitre V. Augmentation de l'information contextuelle 79
1. Introduction 79
2. L'approche « naïve »: la notification de tous les événements 82
3. Objectifs 84
4. Groupes et passage à l'échelle 84
5. Pertinence 85
5.1. Filtrage de l'information contextuelle 86

5.2. Distances 87
6. Exemple : Topologie en étoile simple 89
7. Un système de règles pour la paramétrisation d'un SAIC 92
7.1. INCOMING 93
7.2. OUTGOING 95
8. Opportunité 98
9. Résumé et conclusions 99
Chapitre VI. Validation 101
1. Objectifs et méthodologie 102
2. Cadre d'expérience 103
3. Implémentation 107
3.1. Gestionnaire des espaces de travail 108
3.2. Gestionnaire de l'état du procédé 109
3.3. Gestionnaire de politiques 111
3.4. SAIC 111
4. Évaluation 112
4.1. Introduction 112
4.2. Niveau d'adoption de CELINE dans les équipes pilote 113
4.3. Analyse 114
4.4. Concurrence 116
4.5. Conclusions 117
Conclusion 119
Bibliographie 121

Chapitre I. Introduction

La croissance en taille et en complexité des logiciels actuels, ainsi que les contraintes du marché, rendent de plus en plus nécessaires les techniques d'ingénierie concurrente. Ces techniques servent à réduire les temps de développement en permettant à plusieurs développeurs de travailler simultanément sur les mêmes objets. Malheureusement, la concurrence pose de nombreux problèmes mal gérés par les systèmes de support à la collaboration qui existent aujourd'hui.

Le contrôle de la concurrence est donc un défi pour les concepteurs d'environnements de génie logiciel. Cette thèse s'intéresse au support informatique des procédés collaboratifs de génie logiciel et, en particulier, au contrôle de la concurrence.

1. Contexte

Depuis longtemps, les procédés qui impliquent des données en format électronique sont supportés par des systèmes logiciels spécialisés. Ces systèmes aident considérablement les organisations en fournissant des services tels que la formalisation des procédés, la surveillance, le guidage, l'automatisation de certaines tâches répétitives, le transfert des données, et d'autres encore. En particulier, la formalisation joue un rôle fondamental dans la compréhension des procédés et dans la capacité des organisations à les améliorer.

Cependant, les procédés par lesquels les systèmes logiciels sont créés et maintenus présentent des caractéristiques particulières qui rendent les technologies des procédés traditionnelles peu adaptées. Ceci est dû, entre autres, à une instabilité qui ne se trouve pas dans les ingénieries traditionnelles. En effet, les projets de développement de systèmes logiciels sont sujets à plusieurs sources d'instabilité: de fréquents changements dans les techniques, méthodologies et outils rendent obsolètes les techniques précédentes avant même que ces dernières arrivent à maturité. A côté de ces changements généraux dans la discipline, les développeurs d'un projet en particulier doivent faire face souvent à des changements imprévus dans les besoins, au cours du développement.

Par exemple, le développement d'une nouvelle fonctionnalité peut devoir être interrompu pour régler un « bug » urgent trouvé au cours d'un nouveau développement. Cette nouvelle activité ne pouvait pas être prévue d'avance. Ce type d'instabilité est l'une des sources de la

difficulté à maîtriser les procédés collaboratifs de génie logiciel. Une deuxième source est le haut niveau de concurrence existant dans des projets de développement aujourd'hui.

Nous pouvons résumer de la façon suivante les caractéristiques propres aux procédés de développement collaboratifs des systèmes logiciels:

- **Non-determinisme**: La notion classique de procédé est celle d'un ensemble d'activités qui s'exécutent dans un ordre précis. Dans cette notion, les activités concrètes et l'ordre effectif d'exécution sont connus à l'avance. En génie logiciel, en général, l'ensemble précis des activités de développement et l'ordre dans lequel ils s'exécutent sont extrêmement difficiles à planifier, car ils changent au fur et à mesure que des nouveaux besoins arrivent. [Coc 01]
- **Itérativité**: Le génie logiciel privilégie les cycles de développement courts et la publication fréquente des résultats partiels. L'itérativité réduit les temps de communication entre les développeurs et ses clients, ce qui facilite l'adaptation des procédés aux nouvelles circonstances. Dans les procédés traditionnels, un procédé définit la façon de finaliser un produit. Dans un procédé itératif, le concept de produit finalisé est remplacé par une suite de résultats intermédiaires.
- **Concurrence**. Les données qui constituent un système logiciel sont difficiles à découper en modules indépendants. Ainsi, la modification d'une partie d'un système a une haute probabilité d'affecter une grosse partie du reste. Ceci signifie qu'il est en pratique très difficile d'exécuter simultanément des activités complètement indépendantes. Les contraintes du marché font de la concurrence un "mal nécessaire", car il devient de plus en plus impératif de réduire les temps de développement.

Ces propriétés expliquent en partie la transition graduelle du génie logiciel depuis un modèle de développement "en cascade", adapté à des procédés séquentiels et déterministes, vers des techniques "en spirale" et, plus récemment, vers la "programmation extrême" et les méthodologies dites "agiles" [Coc 01]. Ces dernières méthodologies reconnaissent le caractère itératif et non-déterministe du développement logiciel : ils privilégient une vision des procédés, basée sur les acteurs et ses interactions, qui contraste avec la vision des procédés basés sur la détermination à l'avance des étapes de développement et de son ordre, typique du modèle en cascade.

La réduction des temps des cycles de développement et la croissance du nombre d'ingénieurs nécessaires pour le développement d'un seul logiciel rendent les procédés de génie logiciel de

plus en plus difficiles à maîtriser, et des techniques de contrôle de l'ingénierie concurrente de plus en plus nécessaires.

2. Portée du travail

Nous avons volontairement restreint la portée de notre travail avec les hypothèses suivantes sur les projets collaboratifs:

- **Résultat unique :** La collaboration de plusieurs participants a pour objectif de produire un **résultat unique**. Par la suite, le terme “collaboration” sera utilisé dans ce sens.
- **Copies multiples** : Dans ce travail nous avons adopté l'hypothèse de l'existence de **copies multiples** du logiciel dans des espaces de travail différents. Cette hypothèse correspond au besoin de chaque développeur de travailler en isolation à fin d’éviter d'être perturbé par des modifications faites par d’autres.

Ces deux hypothèses excluent des techniques tels que l'édition coopérative synchrone, où la collaboration dont les résultats n'ont pas la vocation d'être réconciliés dans une version collective. Cependant, nos hypothèses couvrent un grand nombre de scénarios réels de développement logiciel.

Ces hypothèses nous permettent de préciser les problèmes propres de la collaboration dans les procédés de génie logiciel.

3. Problématique

Les procédés collaboratifs de développement de logiciels sont mal compris et mal assistés par les environnements de génie logiciel d’aujourd’hui. Ce manque de maîtrise des procédés explique en partie le retard du génie logiciel, comparé à d'autres génies, dans la prédictibilité des temps et des coûts de développement. Les problèmes spécifiques qui empêchent la maîtrise de ce type de procédés sont la concurrence et l’hétérogénéité.

3.1. Le problème de la concurrence

Le problème qui nous intéresse le plus, dans la problématique générale du support de la collaboration, est celui du contrôle de la concurrence. La question “comment contrôler la concurrence?” est pertinente pour deux raisons:

1. La concurrence est un problème actuel: les responsables des projets de développement collaboratif de logiciel avec de nombreux développeurs ont du mal à gérer les risques associés à la modification simultanée des données. La croissance de taille, de la complexité, et du nombre des développeurs qui collaborent dans un même projet ne peuvent qu'aggraver ce problème
2. Aujourd'hui, aucun environnement ne supporte le contrôle explicite de la concurrence. Les systèmes de gestion des procédés ignorent ou laissent le contrôle de la concurrence aux utilisateurs. Les systèmes de gestion de configuration permettent soit de bloquer la concurrence, soit de la permettre sans aucune restriction.

Les risques de la concurrence sont liés à la difficulté de réconcilier deux copies qui ont été modifiées simultanément, comme nous l'expliquerons dans la première section de la thèse. Or la réconciliation est inévitable à cause de nos hypothèses de copies multiples et de résultat unique.

3.2. Hétérogénéité

Le génie logiciel n'implique pas uniquement l'ingénierie concurrente. Il nécessite aussi des procédés de type déterministe. Par exemple, lorsqu'une nouvelle version du logiciel va être livrée au client, l'ordre des étapes pour le livrer doit souvent être strictement respecté.

La difficulté que représente la maîtrise de cette hétérogénéité (déterminisme – non déterminisme, séquentielle – concurrence) se traduit dans un manque de compréhension globale des procédés de génie logiciel.

4. Objectifs

L'objectif de cette thèse est de proposer des mécanismes de support informatique à la collaboration, spécifiquement dans le cadre du génie logiciel. Ce support doit couvrir aussi bien les besoins associés à l'ingénierie concurrente que ceux liés aux aspects déterministes de la collaboration.

Nos objectifs spécifiques sont:

- Expliquer les principes généraux de l'ingénierie concurrente, basée sur des copies multiples et résultat unique, ainsi que les problèmes liés à sa gestion. Expliquer les limitations des technologies existantes par rapport à ces principes et problèmes.

- Proposer un langage permettant la modélisation des procédés qui soit adaptée tant à l'ingénierie concurrente qu'aux aspects déterministes des procédés de génie logiciel.
- Proposer des mécanismes de support à l'ingénierie concurrente, basés sur ce langage.
- Proposer des stratégies d'implémentation de ces mécanismes.
- Expérimenter ces mécanismes dans un cadre réel et analyser les résultats.

5. Plan de la thèse

5.1. Etat de l'art

Dans la première section de cette thèse nous étudierons les différents systèmes de support des procédés qui existent aujourd'hui, ainsi que les différents systèmes utilisés dans les environnements de génie logiciel, tels que les systèmes de gestion de configuration. Nous présenterons aussi la façon d'aborder la concurrence d'autres outils tels que les éditeurs coopératifs, le CSCW (*Computer Supported Cooperative Work*), et les outils de fusion. Nous discuterons des limitations des systèmes présentés par rapport aux besoins du génie logiciel.

5.2. Support à l'ingénierie concurrente

Dans cette section nous introduirons les concepts de base de notre proposition. Nous proposerons deux types de support basés sur les modèles de procédé: les « *politiques* » et la "*augmentation de l'information contextuelle*".

5.3. Politiques

Le premier besoin pour maîtriser les procédés de génie logiciel est d'avoir un langage approprié à sa description. Notre approche inclut la définition d'un tel langage. Ce langage permet de définir des *modèles de procédés* de génie logiciel. Un modèle de procédé est la description explicite d'un procédé du monde réel.

Un modèle sert à communiquer sur le procédé à un niveau de détail adéquat, en permettant d'ignorer les détails inutiles. Les modèles de procédé facilitent ainsi la discussion nécessaire pour comprendre, modifier, améliorer et faire des prédictions sur un procédé réel. Pour être réellement un facilitateur de la communication, le langage doit être simple pour les utilisateurs et ne doit décrire que les aspects des procédés qui les intéressent.

Pour pouvoir implémenter des outils spécialisés dans le support des procédés, les modèles doivent être définis dans un langage à la sémantique précise. Les outils de support des procédés sont des logiciels capables d'interpréter des modèles ainsi formalisés, et de fournir des services adaptés.

Notre langage est donc conçu pour remplir les deux caractéristiques suivantes:

- Une sémantique précise, pour pouvoir construire des logiciels de support basés sur le langage. Notre langage sera donc expliqué en détail.
- Un haut niveau d'abstraction : Le langage doit se focaliser sur les principaux concepts du domaine et les préoccupations des utilisateurs. L'identification et l'explication en profondeur de ces concepts et préoccupations sont présentées dans le chapitre 3.

Pour illustrer les concepts, nous fournissons plusieurs exemples de scénarios de développement et de politiques appropriées aux différents besoins.

Dans cette section nous définirons l'automatisation comme l'intervention active et autoritaire de l'environnement de développement pour garantir le respect des procédés modélisés avec notre langage. Les difficultés et les stratégies d'implémentation seront également présentées.

5.4. Augmentation de l'information contextuelle

La deuxième stratégie de support consiste à fournir aux utilisateurs l'information nécessaire pour qu'ils puissent prendre des décisions pertinentes sur le quand et le comment suivre les règles du procédé, lorsque c'est possible. La relation entre le modèle de procédé et la transparence sera expliquée.

5.5. Validation

CELINE est notre implémentation d'un environnement de support des procédés de génie logiciel collaboratifs et concurrents. Nous avons réalisé un outil de qualité industrielle implémentant un sous-ensemble de nos idées, utilisé en exploitation réelle depuis 2005. Dans cette section nous présenterons les conclusions de cette utilisation industrielle.

6. Cadre de l'expérimentation

Cette thèse a été développée dans le cadre d'une collaboration entre la société STMicroelectronics et le laboratoire LSR. STMicroelectronics est une société dédiée à la conception et fabrication de produits microélectroniques. Le sujet de l'ingénierie concurrente

est une préoccupation dans les équipes de conception de cette société. Nous nous sommes focalisés sur les activités des équipes LVT (“Layout Verification Team”) et PCELL qui s'occupent du développement et de la maintenance des données utilisées dans le cycle de développement microélectronique, qui peuvent être vus comme des fichiers de code source. Si le nombre de développeurs dans les équipes est loin de ceux des grands projets de génie logiciel, le cadre d'expérimentation est intéressant dans la mesure où l'équipe est répartie géographiquement (Crolle et Tunisie). Ce qui rend la gestion de la concurrence plus difficile est aussi le besoin croissant d'augmenter le niveau de concurrence dans le développement.

L'équipe Adèle du laboratoire LSR est une équipe dédiée à l’étude des problèmes liés à l'évolution des logiciels de grande taille. L'équipe a une longue expérience dans le domaine de la gestion de configurations ainsi que dans le domaine des procédés.

7. Résumé des principaux résultats

Nous considérons que les principaux résultats de cette thèse sont :

- Avoir illustré les principes du contrôle de la collaboration dans les procédés de génie logiciel.
- Avoir montré que les procédés de génie logiciel collaboratifs peuvent être modélisés avec un niveau de détail assez fin pour contrôler l'ingénierie concurrente tout en restant très simple à utiliser.
- Avoir proposé un langage dans lequel la description d'une large gamme de procédés, du complètement déterministe jusqu'au hautement concurrentiel et non déterministe peut être fait dans un formalisme unique.
- Avoir proposé un formalisme de haut niveau où les détails qui ne font pas partie des préoccupations principales ne sont pas exposés à l'utilisateur.
- Avoir réalisé un outil capable d'imposer un procédé ainsi que de fournir de la transparence aux utilisateurs.
- Avoir utilisé industriellement cet outil et avoir obtenu une évaluation réaliste de nos idées.

Chapitre II. Etat de l’art

Notre approche au support du génie logiciel collaboratif s’appuie sur la formalisation des procédés. Il est donc pertinent d’explorer les différentes recherches qui ont été menées dans ce domaine. Les *procédés* et les *workflows* sont des technologies qui ont permis la formalisation et le support des procédés dans de nombreux domaines, mais dont leur application au génie logiciel reste limitée pour plusieurs raisons. Nous expliquerons dans cette section les principes de ce type de technologies, en exposant ses limitations par rapport aux besoins du génie logiciel. Nous présenterons aussi les différentes tendances qui existent pour rendre ces types de technologies plus adaptés à des procédés concurrents et non déterministes.

La concurrence est l’une des sources de difficultés pour la gestion de la collaboration dans les procédés de développement. Plusieurs technologies ont abordé le problème de la modification concurrente des données dans des procédés de conception. Nous présenterons les modèles des transactions avancées, les outils de travail coopératif « synchrones » tels que les CSCW et les éditeurs coopératifs.

Finalement, nous présenterons les outils de gestion de la configuration. Ces outils sont utilisés dans pratiquement tout scénario de développement logiciel aujourd’hui, et constituent le principal point de coordination entre les développeurs. Cependant, ces outils se focalisent sur les données et ils ne résolvent pas le problème de la gestion de la concurrence ni de la formalisation des procédés.

Dans tous les cas, nous nous concentrerons sur les aspects de ces technologies qui permettent d’illustrer ses principes fondamentaux ainsi que ses limitations par rapport à notre problématique. Nous ne prétendons pas couvrir intégralement tous les aspects des technologies étudiées.

1. Les procédés

Un procédé est une approche *systématique* pour la réalisation d’un objectif [Ost 87]. Le terme *systématique* implique la séparation entre l’exécution *réelle* des taches et *l’idée abstraite* sur la façon dont les taches doivent être ou sont exécutées. Des exemples de procédés sont la préparation d’un plat d’après une recette, l’assemblage d’une chaise par un opérateur d’usine ou la procédure d’inscription d’un étudiant à l’université. Dans tous ces cas, les acteurs suivent une idée pré-établie des étapes de la procédure, et de comment ces étapes doivent

s’enchaîner. Le concept de procédé est très général.

De son coté, le génie logiciel est l’approche systématique et disciplinée pour la construction de systèmes logiciels. Le lien entre génie logiciel et procédés est donc naturel: pour pouvoir aborder la construction des logiciels de façon systématique, une idée précise ou même formelle de la procédure à suivre est nécessaire. Malheureusement, la description des procédés de génie logiciel collaboratifs reste encore très précaire actuellement. Dans cette section nous expliquerons les systèmes de support des procédés qui existent aujourd’hui ainsi que ses limitations par rapport au génie logiciel collaboratif.

2. Les procédés logiciel

Dans les années 80, la recherche et la pratique du génie logiciel ont été fortement influencées par la compréhension du fait suivant:

> ***La qualité des systèmes logiciels dépend de la qualité des procédés par lesquels ils sont créés.***

Étant donné que les procédés de développement de logiciel sont collaboratifs et complexes, leur support doit tenir compte de l’intégralité du *procédé* et ne pas se limiter à la programmation et à ses outils (les langages de programmation, les compilateurs, les éditeurs, etc.). Le domaine de recherche qui s’est intéressé aux procédés de création des systèmes logiciels fut nommé « software process » [Fug 00] (*procédé logiciel*).

Dans cette section nous expliquerons les principes généraux des procédés logiciels et les concepts communs entre plusieurs travaux qui ont été menés dans cette direction.

2.1. Modèles de procédé

Très souvent, dans un même domaine d’application ou dans une même organisation, les procédés suivent un patron commun. Il est donc naturel, lorsqu’on s’intéresse à une famille de procédés, de vouloir exprimer ses aspects communs. Ces aspects communs sont exprimés dans un « modèle de procédé ».

Concrètement, un modèle de procédé est la description explicite d’une famille de procédés. Pour [Ost 87], la différence entre un *procédé* et un modèle *de procédé* est semblable à celle qui existe dans les langages de programmation entre une classe ou un type, et une instance de la classe. Le modèle de procédé définit donc certains traits généraux du procédé, mais pas les

détails spécifiques. Par exemple, un modèle de procédé peut décréter qu'une activité doit être exécutée par un développeur, mais ne pas spécifier lequel.

La modélisation des procédés est l'une des préoccupations fondamentales des procédés logiciels. Les modèles de procédé servent un ou plusieurs objectifs [Gru 02] [ABG 92] [CKO 92]. Nous pouvons lister les suivants :

- **Communication**: Le modèle de procédé sert d'abord à faciliter la communication entre les participants d'un procédé :
    - Il est l'instrument avec lequel plusieurs acteurs peuvent discuter sur les procédés réels.
    - Il est le point d'entrée au procédé pour les nouveaux arrivants.
    - Il est le moyen pour expliquer le procédé à d'autres branches de la même organisation, etc.
- **Compréhension, analyse et amélioration**: Pour [Ost 87], l'un des bénéfices de l'utilisation des modèles de procédés est qu'il est en général plus facile de réfléchir sur le plan d'un procédé (une vision statique) que sur l'exécution du procédé lui-même (la vision dynamique). Ainsi, un modèle facilite la compréhension des procédés qu'il décrit. Également, avec un modèle de procédé, plusieurs techniques d'analyse tels que la simulation, l'évaluation statiques des traces de la simulation ou l'extraction des propriétés générales applicables à toutes les instances du modèle, deviennent possibles. La compréhension et l'analyse sont importantes pour l'évolution des procédés, car elles aident à mesurer l'impact qu'une modification du modèle peut avoir sur l'exécution des instances du procédé [Hum 89].
- **Support automatisé:** Un modèle de procédé exécutable peut être interprété par des outils de support spécialisés [Per 89], capables de fournir des services adaptés aux particularités de chaque modèle. Parmi ces services nous trouvons la traçabilité, le monitorage (obtention en temps réel des informations sur l'état d'un procédé) et l'automatisation des tâches répétitives (allocation de ressources, envoi de messages à des systèmes associés, transfert de données, notification à des personnes concernées…).

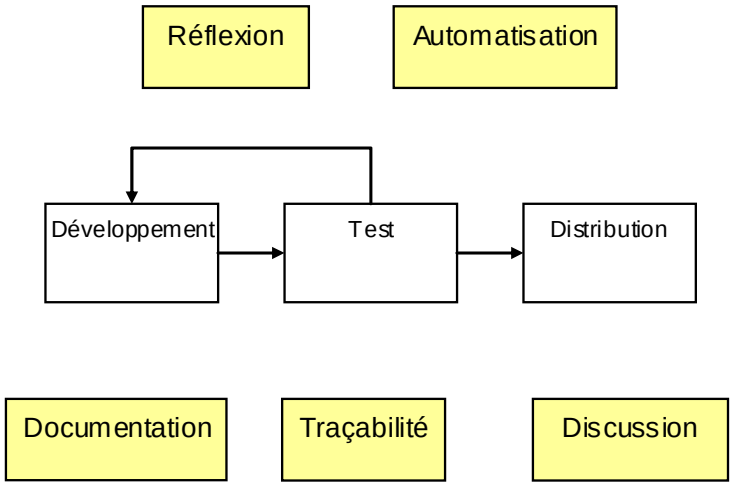


Figure 1 : Procédés

Selon les objectifs pour lesquels les modèles sont conçus, ils peuvent décrire les procédés à différents niveaux de détail. [Mon 99] classifie les modèles de procédés dans les catégories suivantes :

- **Cycle de vie :** Le cycle de vie est une représentation informelle et très générale d'un procédé. Le cycle de vie est surtout intéressant pour la discussion sur des problèmes de méthodologie globale. Des exemples de cycles de vie connus sont le modèle en *Cascade* et le modèle en *Spirale*. Ces types de modèles sont souvent exprimés de façon narrative dans des langages naturels.
- **Modèle de Procédé Générique :** Le modèle de procédé générique est une représentation abstraite d'un procédé, qui peut être réutilisée dans plusieurs contextes similaires. Un exemple de procédé générique peut être la représentation formelle du cycle de vie.
- **Modèle de Procédé Spécialisé :** Le modèle de procédé spécialisé est l'adaptation d'un modèle de procédé générique à un contexte spécifique. Par exemple, l'adaptation d'un standard pour son utilisation dans une organisation particulière.
- **Modèle de Procédé Exécutable :** Le modèle de procédé exécutable est le modèle le plus détaillé qui peut être interprété par des outils spécialisés dans le support, l'automatisation, la simulation, le guidage, etc.
- **Modèle de Procédé en exécution :** Le modèle de procédé « en exécution » est la représentation d'un procédé concret qui est en train de se dérouler. Le modèle contient donc une représentation de l'état de développement du procédé.

Notre démarche s'inscrit dans les modèles *de procédés exécutables*, nous allons donc explorer

les différents systèmes existants basés sur les modèles exécutables.

2.2. Langages de modélisation de procédés (PMLs) exécutables.

Nous parlerons dans cette section des langages de modélisation des procédés qui fournissent une syntaxe et sémantique précises, et qui peuvent de ce fait supporter l'analyse, la simulation et l'exécution par des moyens informatiques. L'architecture générale d'un système basé sur des modèles de procédés exécutables est illustrée dans la figure suivante [Mon 99] :

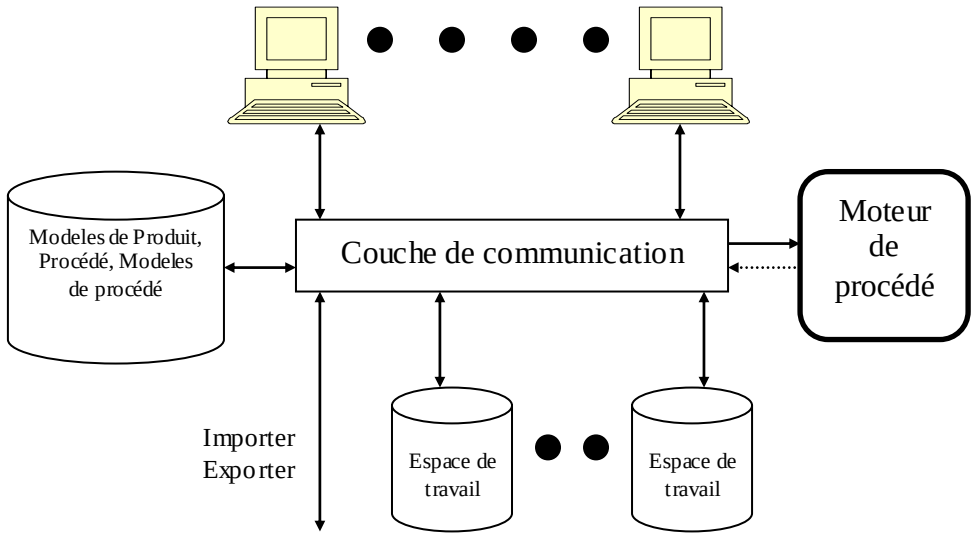


Figure 2 : Architecture des environnements de génie logiciel basé sur des modèles de procédés.

L'architecture comporte trois entités principales: un **dépôt** où les modèles de procédés sont stockés, des **outils de création** des modèles de procédés, et des **outils de support** capables de rendre des services basés sur l'interprétation des modèles.

Les concepts centraux trouvés dans les PMLs sont [Mon 99]:

- **Activité** : Une étape dans un procédé qui opère sur des objets logiciels. Les activités peuvent être décomposées en sous tâches, ce qui permet de définir un modèle à plusieurs niveaux de détail. Une activité peut se dérouler en parallèle ou en concurrence avec d'autres activités, et elle peut être déterministe ou non déterministe. Typiquement un procédé de développement logiciel contient des tâches de spécification, de développement, d'intégration et de test. Les activités sont souvent associées à des ressources nécessaires pour les accomplir (humains et outils).

- **Produit** : Un produit est un artefact logiciel, simple ou composite, qui est à la fois créé, transformé et consommé par des activités. Exemples: des systèmes logiciels, des bibliothèques de logiciels, des documents de conception, la documentation d'un « bug », etc.
- **Rôle** : Un rôle regroupe un ensemble de droits et de responsabilités. Cet ensemble peut être assumé, dans son intégralité, par un humain.
- **Humain**: Une personne qui participe au procédé. Un humain peut faire partie d'un ou de plusieurs groupes, et il peut jouer un ou plusieurs rôles.

2.3. Classification des PMLs

Nous utiliserons une catégorisation des PMLs inspirée par [ABG 92][EB 92], qui organisent les langages selon ses différents « paradigmes ». Évidemment, cette catégorisation ne prétend pas diviser les différents PMLs dans des camps complètement disjoints, mais elle est un outil pour comprendre les traits communs et les différences entre les langages.

2.4. PMLs basées sur des règles

Ces systèmes modélisent les procédés de développement logiciel en termes de règles, qui encapsulent l'information concernant les activités. Les règles sont typiquement décrites par trois éléments : 1) une action (souvent associée à un outil), 2) les conditions requises pour que l'action soit effectuée (souvent décrites en termes des expressions sur les attributs des objets dans une base de données) et 3) les effets de l'action sur le système (typiquement, des modifications dans la base de données).

Par exemple, la condition qui définit une activité d'édition peut être définie de la façon suivante, dans le PML MSL (*Marvel Strategy Language*) [BK 91] :

```
edit[?c:CFILE]:
:
(?c.reservation_status = CheckedOut)
{ EDITOR edit ?c.contents }
(and (?c.compile_status = NotCompiled (?c.time_stamp = CurrentTime));
```

Dans cet exemple, les trois éléments d'une règle sont bien illustrés : le fichier doit être dans l'état CheckedOut pour pouvoir être édité (la condition), l'action est l'édition du fichier, et l'action déclenche une transition d'état des attributs du fichier (son état et sa date d'édition).

2.4.1. Marvel

Dans le langage *Marvel Strategy Language* (MSL) [KFP 88][Bar 92] [ABG 92], le procédé est décrit en terme des éléments suivants [KBF 88] :

- Les objets logiciels, stockés dans une base de données, et ses relations (par exemple, un programme écrit dans le langage C est constitué par des modules, qui sont eux-mêmes constitués par des fonctions et des variables).
- Les règles, qui définissent à la fois:
    - les pré-conditions requises pour l'application d'un outil (autrement dit, une activité),
    - les conséquences possibles de l'application d'un outil.

Par exemple, pour éditer un fichier, celui-ci doit être affecté à la personne qui l'édite, et le résultat de l'édition est le passage du fichier à l'état « pas compilé ». Par la suite, une activité de compilation sur le fichier peut être lancée.

Lorsque un utilisateur exécute une opération (une activité) avec un outil, ses résultats peuvent déclencher d'autres activités, selon les règles, ou permettre l'exécution d'autres opérations par d'autres utilisateurs.

Un des points forts des systèmes basés sur des règles du style Marvel est que les règles peuvent aussi être appliquées « backwards » : lorsqu'un utilisateur déclare vouloir exécuter une opération pour laquelle les pré-conditions ne sont pas satisfaites, le système peut proposer une chaîne d'opérations à effectuer pour satisfaire les pré-conditions manquantes.

2.4.2. Merlin

Merlin est un autre PSEE (*Process Centered Software Engineering Environment*) basé sur la notion des règles. Le langage de modélisation (ESCAPE) se décompose en trois types de modèles différents [RJ 99][JPS 94] : le premier (*Extended Entity Relationship*) décrit les données (documents), le deuxième (organisation) concerne les humains et les rôles, et le troisième (coordination) définit les activités en terme des règles : pré-conditions, post-conditions et changements d'état.

Les modèles définis dans le langage ESCAPE sont traduits par des règles écrites dans un langage de style prolog, qui peut être directement interprété par le PSEE. L'exécution simultanée d'activités n'est pas supportée dans Merlin [ABG 92] [JPS 94].

2.4.3. EPOS

EPOS est un environnement de support des procédés qui fournit un PML nommé SPELL. Le système EPOS combine plusieurs approches [ABG 92][JC 93][JR99], ce qui rend son analyse difficile. Dans SPELL, la description d'une activité est constituée par trois types d'éléments :

- les pré-conditions/post-conditions,
- les entrées/sorties,
- les relations de composition des activités (division fonctionnelle).

2.5. PMLs basées sur des réseaux de pétri

Les réseaux de pétri sont des représentations mathématiques des systèmes distribués. L'utilisation des réseaux de pétri pour la formalisation des procédés permet d'utiliser les nombreux travaux théoriques qui existent sur le sujet pour extraire des propriétés d'un procédé à partir de l'analyse statique du modèle.

Un réseau de pétri est constitué de « lieux », de « transitions », de « jetons» et d'arcs dirigés qui relient les lieux et les transitions.

2.5.1. SPADE

SPADE est un système de support de procédés intégré au gestionnaire de bases de données orienté objets O2. SPADE fournit le PML SLANG, basé sur un formalisme de haut niveau des réseaux de pétri nommé ER [ABG 92]. Un procédé est modélisé dans SPADE en traduisant les concepts des réseaux de pétri de la façon suivante:

- les jetons représentent les documents (les objets logiciels),
- les « lieux » représentent des dépôts de documents,
- les transitions représentent des évènements qui se produisent dans une période de temps négligeable.

[BFGR 93][BBF 94]

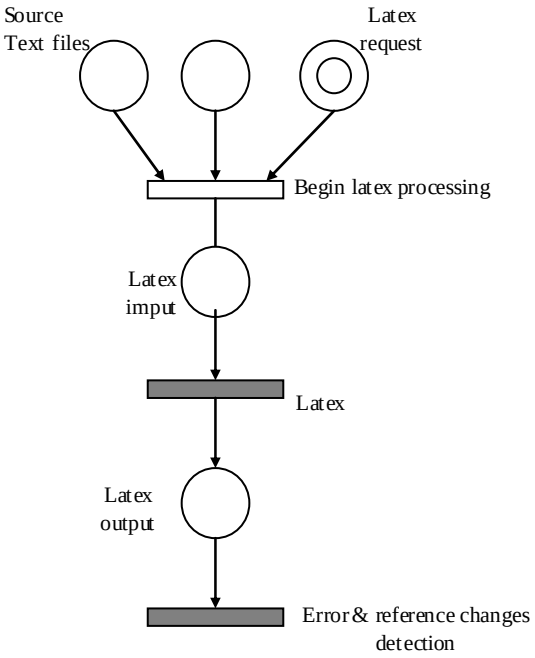


Figure 3 : Réseau de pétri

2.5.2. FunSoft

FunSoft est un autre système basé sur les réseaux de pétri. Particulièrement, il permet d'associer à chaque « lieu » une politique et à chaque transition une valeur estimée de la durée de l'activité. Ceci permet la simulation avancée des modèles définis avec ce langage. [ABG 92] [EG 91]

2.6. PMLs procéduraux

Les PMLs procéduraux correspondent à la vision de [Ost 97] selon laquelle « les procédés logiciels sont aussi des logiciels » (« *software processes are software too* »). Ils utilisent donc des langages de style procédural (ou des adaptations de ces langages) directement empruntés au domaine des langages de programmation traditionnelle.

2.6.1. APPL/A

APPL/A est une extension du langage de programmation ADA, développé dans le contexte du projet ARCADIA. En tant qu'extension du langage ADA, il s'agit d'un langage essentiellement procédural. Cependant, certaines parties (tels que les conditions de cohérence) sont décrites de façon déclarative. [ABG 92][RJ 99]

Example: [RJ 99]

```
Procedure SingleUserModel is...
Begin
      while not Done loop
            case CurrentApproach is
                  when TopDown =>
                        if NeedRool then CreateRoot;
                       elsif NeedChildren then
                             SelectNodeToElaborate(currentNode);
CreateChildren( CurrentNode);
                  else SelectNodeToEdit(CurrentNode);EditNode(CurrentNode);
                  end if;
```

2.7. PMLs se focalisant sur les types abstrait de donnés

Il s'agit des PML dont l'objectif est la modélisation des objets logiciels.

2.7.1. ADELE

Adèle [EC 94] est un environnement construit autour d'une base de données versionnée. Cette base de données est fondée sur le modèle entité relation, étendue par des relations inspirées du modèle orienté objet, ainsi que par des activités et des transactions longues. L'accent dans le développement d'Adèle a été mis sur un modèle de produit avancé. Le concept d'activité est supporté par la notion de « trigger ». Un trigger est une règle définie par un triplet événement/condition/actions-dans-la-base-de-données.

2.8. Discussion

Un grand nombre d'acteurs coopèrent dans le but de construire des systèmes logiciels. La coopération [God 99] fournit plusieurs bénéfices, tels que la capacité à produire des systèmes plus complexes que ceux qu'il est possible de créer individuellement, ainsi que la réduction des temps de développement.

Quand l'objectif de la coopération est de réduire les temps de développement, le procédé doit minimiser le temps qu'un participant reste sans travailler, en permettant l'exécution d'activités simultanées. La plupart des PSEE supportent le parallélisme. En particulier, les PMLs basés sur les réseaux de pétri le décrivent avec précision. Cependant, aucun des PML que nous avons étudié ne fournit des moyens pour exprimer la façon dont les données qui se

trouvent sous l'influence des activités en parallèle doivent être (ou pas être) modifiées simultanément. Or c'est de cela que dépend l'intégrité des données manipulées.

Idéalement, un processus pourrait être conçu de manière à ce que chaque activité soit en charge d'une partie indépendante des données, éliminant ainsi les interférences entre les activités. Mais les données qui constituent un système logiciel sont hautement couplées et difficilement divisibles en parties complètement indépendantes. Dans la pratique, la gestion de la coopération dans des projets de génie logiciel n'est donc complète que lorsqu'elle inclut des moyens de contrôle de la concurrence.

Pour [God 99], il existe trois façons d'aborder ce problème dans les PMLs :

1. La responsabilité de l'intégrité des données revient aux utilisateurs. Le système ignore les problèmes liés à la concurrence (Il n'y a donc pas de vrai support dans le langage).

2. Le système ne supporte pas directement des concepts liés à la concurrence, mais il est (en théorie) possible de décrire dans le PML toutes les interactions possibles entre les différents acteurs (par exemple, dans ADELE, des triggers peuvent décrire toutes les interactions possibles entre les espaces de travail des participants).

3. Des politiques indépendantes du modèle de procédé sont définies et peuvent être appliquées à plusieurs modèles.

La première solution est insuffisante lorsque le nombre des développeurs travaillant simultanément est important [Est 96]. La deuxième conduit rapidement à des modèles trop complexes lorsque la collaboration va au-delà des cas basiques, ce qui détruit la fonction principale des modèles (la compréhension du procédé).

Plusieurs systèmes de gestion des procédés utilisent une base de données unique pour stocker les données manipulées par le procédé, et ils délèguent la gestion de l'accès concurrent au système de gestion de la base de données. Nous expliquerons dans la prochaine section les mécanismes de gestion de la concurrence dans les bases de données traditionnelles, leurs limitations et les modèles alternatifs qui ont été conçus pour faire face à ces limitations.

2.9. Contrôle de la concurrence dans les bases de données

Les gestionnaires des bases de données fournissent un mécanisme uniforme pour stocker et manipuler de larges ensembles de données. L'un des principaux objectifs des systèmes de

gestion des bases de données est de supporter l'interaction de plusieurs utilisateurs avec la base de données [Elm 92], en garantissant sa cohérence.

Dans une base de données, les utilisateurs interagissent avec le système par l'intermédiaire des *transactions*, c'est à dire des séries d'opérations de lecture et écriture qui satisfont les propriétés ACID:

- Atomicité : Une transaction doit être exécutée dans sa totalité (si ceci n'est pas possible, elle ne doit pas être exécutée du tout).
- Cohérence : Chaque transaction doit être individuellement correcte: lorsqu'elle est exécutée seule, la base de données doit finir dans un état cohérent (par rapport au contraintes de la base de données). En général, les gestionnaires des bases de données assument que chaque transaction est correcte, en isolation.
- Isolation : Chaque transaction doit observer une base de données cohérente (elle ne doit pas lire des résultats intermédiaires produits par d'autres transactions).
- Durabilité : Les résultats d'une transaction doivent persister, une fois que la transaction est finie. Cette persistance doit résister aux pannes du gestionnaire de la base de données.

Pour illustrer les problèmes liés à l'exécution concurrente des transactions dans une base de données, nous fournissons l'exemple suivant :

Dans ce scénario, deux transactions sont exécutées simultanément. La première transaction encapsule l'activité de louer un film dans un magasin, et la deuxième encapsule l'activité de recharger la carte d'abonné. Les opérations qui constituent les deux transactions s'exécutent ainsi :

	2.9.1.1 Transaction 1 : Louer un film	**Transaction 2 : Recharger la carte**
1	**read(X,solde)**	
2		**read(X,solde)**
3	X = X – prix du film	
4	**write(X,solde)**	
5		X = X + somme rechargee

6		X -> write(X,solde)

La transaction 2 recharge la carte d'une somme égale à celle dépensé dans la transaction 1. Tenant compte de la sémantique des deux transactions, le solde final devrait être égal à celui initial, car l'utilisateur a rechargé sa carte de la même somme qu'il a dépensé. Cependant, dans l'ordre des opérations décrit, le solde final est égal au solde précédent plus la somme rechargée, car la transaction de rechargement ne prend pas en compte le nouveau solde, modifié dans l'action numéro 4, mais celui qu'elle avait lu dans l'action numéro 2. Ceci est ce qui est appelé le problème de *lecture incohérente.* [Elm 92]

Nous pouvons constater que l'exécution des deux transactions aurait été correcte si elles avaient été exécutées l'une *entièrement* avant l'autre, indépendamment de leur ordre d'exécution. Ceci illustre la notion principale de correction dans les transactions traditionnelles : la *sérialisation*. Un ensemble de transactions exécutées simultanément est donc correct s'il est n'est pas distinguable d'une exécution quelconque en série des transactions impliquées.

De multiples protocoles existent et servent à assurer une exécution sérialisable des transactions dans une base de données [Elm 92] : 2 phase-locking, timestamp ordering, serialization graph testing, etc. Nous n'allons pas explorer ces protocoles en détail, mais il suffit de dire que nous pouvons séparer les différentes stratégies en deux camps : le premier camp garantit la sérialisation en bloquant certaines opérations en utilisant des *verrous*, le deuxième camp permet l'exécution des transactions sans les bloquer et fait une vérification à la fin.

Lorsque les bases de données ont commencé à être utilisées dans des domaines tels que le CAD/CAM, les environnements de publication et le génie logiciel, la sérialisation comme critère de correction a montré ses limites. En se focalisant sur le génie logiciel, les activités qui manipulent les objets logiciels pressentent des propriétés qui rendent la sérialisation peu adaptée. En effet, les différentes données qui composent un logiciel sont souvent liées par des dépendances sémantiques, ce qui rend très difficile de réaliser une tâche sans lire une vaste partie des données. Certaines activités, par exemple la compilation du système, nécessitent la totalité des données qui composent le logiciel. Cette propriété des données qui composent les systèmes logiciels a comme conséquence que, dans la pratique, deux « transactions » simultanées sur un système logiciel ne sont presque jamais sérialisables.

Pour aggraver les choses, et en contraste avec les transactions traditionnelles qui ne durent généralement que quelques fractions de secondes, dans le génie logiciel une transaction peut durer plusieurs heures ou plusieurs jours. En conséquence, il est inacceptable d'avorter une transaction, car ceci entraînerait la perte du travail.

2.10. Modèles de transactions avancées

Plusieurs modèles de transactions avancées ont été proposés, afin de résoudre ces problèmes. [WS 92] propose la classification suivante des objectifs cherchés par les modèles de transactions avancées:

1. Supporter les activités de longue durée.
2. Relâcher les contraintes ACID.
3. Supporter la coopération entre des activités de conception (logiciel/CAD/autres).
4. Supporter des modèles de données « orientés objet ».
5. Rendre explicite de nouveaux aspects de la sémantique des opérations sur les bases de données.
6. Augmenter le parallélisme entre des transactions en utilisant des connaissances sur la sémantique des opérations.
7. Supporter la modélisation du parallélisme à l'intérieur des transactions.
8. Supporter la définition de points de sauvegarde intermédiaires (« partial rollbacks ») par l'utilisateur.
9. Supporter les transactions « conversationnelles ».
10. Supporter l'hétérogénéité des systèmes autonomes fédérés autour d'une base de données distribuée.

Nous nous intéressons aux points 1 à 7. Dans les pages suivantes nous expliquons brièvement quatre alternatives aux transactions traditionnelles qui s'occupent de ces limitations.

2.11. Transactions imbriquées

Les transactions imbriquées ont été conçues pour : 1) la modularité et le facteur d'échelle, 2) le traitement en cas de panne et 3) la parallélisation à l'intérieur d'une transaction [WS 92]. Pour faciliter la modularité, une transaction imbriquée est séparée en sous-transactions, de

façon successive, en formant un arbre de transactions. Les transactions « racines » (celles qui n'ont aucune sous transaction) sont traitées avec les contraintes ACID traditionnelles. Cependant, une transaction n'est pas obligée d'avorter lorsqu'une de ses sous transactions échoue : elle peut soit 1) ignorer le résultat de la sous-transaction 2) re-essayer la sous-transaction, ou 3) initier une sous-transaction de compensation, permettant de garder la cohérence de l'ensemble et basée sur la sémantique de la sous-transaction qui échoue. Ces caractéristiques permettent un traitement arbitrairement fin des pannes, par rapport aux transactions non imbriquées. Finalement, étant donné que les sous-transactions peuvent être exécutées simultanément, les transactions imbriquées augmentent le niveau de parallélisme à l'intérieur d'une transaction.

La notion de hiérarchie de transactions est fondamentale dans une grande partie des modèles des transactions avancées.

2.12. Split Transactions

Les « Split Transactions » [PKH 88] ont été conçues pour supporter des activités dont la durée et les données qui vont être utilisées sont difficiles à connaître à l'avance. Un « split » divise une transaction en deux parties sérialisables, ce qui permet de « publier » les données modifiées par une nouvelle transaction, sans publier la totalité des modifications de la transaction entière.

Exemple :
Remy commence une transaction pour ajouter une nouvelle fonctionnalité à une application :

- Il lit le fichier **Plugin.c** et le modifie.
- Il lit le fichier **Framework.c**, dont le fichier **Plugin.c** dépends, pour faire tourner un test.
- Il se rends alors compte que le fichier **Framework.c** contient une erreur, qu'il corrige.
- En suite il modifie le fichier **Plugin.c** pour l'adapter au nouvel état de **Framework.c**.
- Ensuite il lit le fichier **PluginExtra.c**, qui dépends de **Plugin.c**, et continue à travailler.

A ce moment, Fred se rend compte qu'il a besoin des modifications de Remy sur le fichier **Framework.c.** Malheureusement, une lecture de ce fichier serait risquée, car si la transaction de Remy est avortée, un roll-back affecterait aussi le travail de Fred. Remy fait donc un split

de sa transaction originale (T) dans deux nouvelles transactions A et B, de la façon suivante :

Operation	Transaction
Read Plugin.c	Transaction B
Write Plugin.c	Transaction B
Read Framework.c	Transaction A
Write Framework.c	Transaction A
Read PluginExtra.c	Transaction B

La transaction A “précède” la transaction B, même si les deux premières opérations de la transaction globale T sont affectées à la transaction B. Lorsque la transaction A fait «commit», un avortement de la transaction B ne signifie plus la perte des modifications sur Framework.c.

Le SPLIT des transactions permet ainsi de communiquer des résultats partiels d’une transaction à d’autres, en diminuant le risque de perte du travail dû aux pannes.

2.13. Hiérarchie des transactions coopératives

Une « hiérarchie des transactions coopératives » cherche à fournir aux gérants des procédés de CAD ou de génie logiciel des mécanismes pour définir des critères de correction alternative à la sérialisation [Elm 92]. Dans le contexte des transactions, un critère de correction définit les séquences d’opérations invoquées par les participants qui sont valides.

Dans ce modèle, les transactions sont imbriquées et des critères de correction peuvent être définis pour chaque groupe de sous-transactions *sœurs* (qui font partie de la même transaction immédiatement supérieure), appelé groupe de transactions. Ceci correspond à l’idée que, dans le domaine des applications coopératives, il n’y a pas un unique critère de correction, mais qu’ils varient selon les cas.

2.14. Structure

Les transactions et sous-transactions forment une structure composée des éléments suivants :

- Groupe des transactions :
    - Il correspond à une tâche qui est effectuée par les membres du groupe.
    - Il est associé à son propre critère de correction.

- Il opère sur ses propres copies des objets

- Transaction coopérative :
  - Correspond à une séquence d'opérations effectuées par un membre d'un groupe.

- Opération :
  - Un tuple <M, o, O> ou **M** est le membre qui effectue l'opération, **o** est le type d'opération et **O** est l'objet sur lequel l'opération est effectuée.

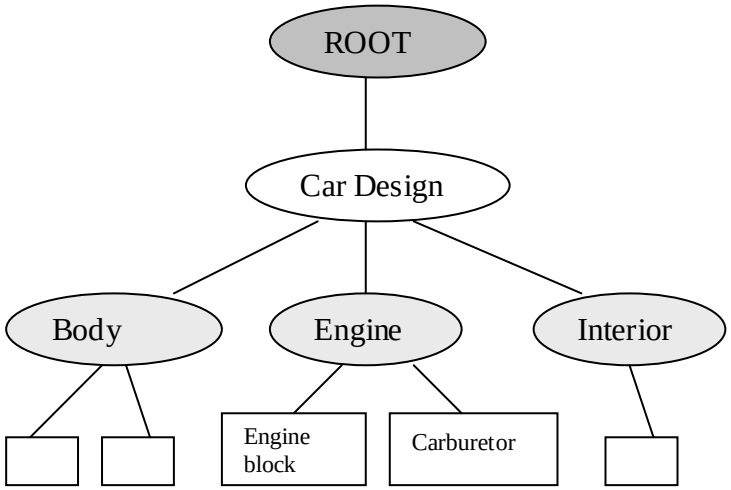


Figure 4 : Hierachie de transactions

2.15. Copies multiples

L'un des objectifs fondamentaux des hiérarchies des transactions coopératives est de cacher les détails de la gestion des sous transactions. Avoir une seule copie des données pour toutes les transactions signifierait vérifier chaque opération contre *tous* les critères de correction des différents groupes de transactions, ce qui serait une violation de cet objectif. En conséquence, chaque groupe de transactions doit avoir une copie privée de la base de données. Lorsqu'un des membres du groupe finit une tâche, ses opérations sont traduites dans une série d'opérations dans la base de données supérieure. Cette traduction dépend du type de l'objet.

2.16. Définition des critères de correction

Le critère de correction dans un groupe des transactions est définit en terme de *patrons* (des séquences d'opérations qui doivent avoir lieu) et de *conflits* (des séquences d'opérations qui ne doivent pas avoir lieu).

Un exemple informel de patron qui correspond à un groupe de transactions dans le cadre de la

conception d'une voiture: [NRZ 92] : « Si le plan du carburateur change, il doit être vérifié que le moteur marche selon le règlement d'utilisation d'essence ». Il est important de remarquer que ce type de critère implique des opérations issues de multiples transactions, en contraste avec le sérialisation, qui bannit l'interférence intra transactionnelle.

Pour sa part, un conflit peut être déclaré de façon informelle ainsi : « Si vous n'avez pas lu le dernier plan du carburateur, vous ne pouvez pas le modifier ».

L'exécution d'un groupe de transactions est donc correcte si l'histoire des opérations satisfait tous les patrons et ne correspond à aucun conflit.

Les patrons et les conflits sont définis avec une grammaire LR(0), représentée avec des expressions écrites ainsi :

```
Etat <- [<membre, operation, object> nouveau etat] *
```

Chaque règle exprime un changement d'état dans une machine d'état déterministe. L'état final est l'état « empty ».

2.17. Exemple

Nous allons montrer un exemple très simple pour illustrer le langage

Patron :

```
S <- <any, write, carb> A | empty
A <- <any, write, carb> A | <verify, read, carb> B
B <- <verify, w, spec> S
```

Conflit:

```
S <- <any, write, carb> A
A <- <any, write, carb> A | <verify, read, carb> B
B <- <verify, w, spec> S | <any, w, carb> empty
```

Dans le patron, un groupe de transactions doit suivre un cycle où plusieurs transactions peuvent modifier le plan du carburateur (<any, write, carb>), et la transaction *verify* doit lire le plan du carburateur, et mettre à jour la spécification pour enregistrer sa vérification.

Dans le conflit, la troisième règle indique que la modification du carburateur est interdite après que la transaction *verify* l'ait lu.

2.18. COO

COO [God 99][GPS 99][CGM 98] est un projet qui s'intéresse au support de la coopération, en particulier au problème de la cohérence entre plusieurs artefacts logiciel manipulés dans un procédé coopératif.

L'opérateur COO est un élément de modélisation qui sert à représenter l'interaction entre deux activités simultanées, qui peuvent échanger plusieurs versions d'un document pendant son déroulement. Cet opérateur est adapté à la modélisation des scénarios de coopération suivants [GPS 99]:

- Producteur/Consommateur : Dans ce scénario, une des activités (le producteur) est la seule à modifier un certain objet. Une deuxième activité (le consommateur) lit régulièrement les résultats de la première pour les incorporer dans son propre travail.
- Développeur/Inspecteur : Dans ce cas, une activité développe un objet et envoie des résultats partiels à une deuxième. La deuxième activité est en charge de lire ces objets, et de retourner une réponse (un nouvel objet), qui va influencer la travail de la première.
- Écriture collaborative : Deux activités participent activement à l'évolution d'un objet et peuvent échanger des résultats intermédiaires pendant qu'ils convergent vers une version commune.

L'opérateur COO déclare deux activités comme étant liées par une contrainte de *COO-Sérialisation*, qui, dit de façon très informelle, signifie que les activités peuvent partager des résultats intermédiaires, mais qu'ils « sont d'accord » sur la valeur finale des objets qu'ils ont partagés.

La COO-Sérialisation est garantie par l'utilisation suivante du protocole [GPS 99]:

1. Un résultat produit avant la fin d'une activité est appelé un résultat intermédiaire.
2. Les résultats finaux sont produits de façon atomique, avec l'opération *terminate*.
3. Dans la phase de terminaison d'activité, un résultat final est produit pour chaque objet pour lequel un produit intermédiaire a été produit dans une activité.
4. Lorsqu'une activité lit une valeur intermédiaire en provenance d'une autre activité, cette activité *dépend* de la valeur finale de l'objet dans l'activité de provenance. La

dépendance est finie lorsque la première activité lit la valeur finale provenant de la deuxième activité.

5. Une activité ne peut pas terminer lorsqu'elle dépend des valeurs finales d'autres activités.

6. Les activités qui forment des cycles de dépendances constituent un *groupe*.

7. Lorsqu'une activité dans un groupe essaie de terminer, le protocole produit des tentatives de valeurs finales, et met l'activité dans l'état *prêt-a-terminer*.

8. Lorsqu'une activité dans un groupe essaie de terminer, si elle est la seule qui reste active, tout le groupe termine, et les tentatives de valeurs finales deviennent les valeurs finales.

9. Lorsqu'une activité produit un nouveau résultat intermédiaire, la tentative de terminaison est avortée, et toutes les activités re-deviennent actives.

Il faut noter qu'un résultat intermédiaire d'une activité peut rendre « invalide » le travail d'une deuxième activité, dans le sens des transactions traditionnelles, celle la peut être « valide » à nouveau en lisant la nouvelle valeur de l'objet. Ceci se base sur l'hypothèse que la lecture de la nouvelle valeur d'un objet déclenchera la mise à jour des objets qui en dépendent. Ceci est raisonnable car les activités dont COO s'occupe sont censé être des activités interactives avec un responsable humain.

2.19. Discussion

Nous avons analysé quatre modèles qui essaient d'adapter la notion de transaction à des environnements de conception coopérative, en relaxant les contraintes propres des transactions traditionnelles.

Comme nous l'avons expliqué dans la section 2.9, les transactions traditionnelles sont incompatibles avec plusieurs des aspects des procédés de conception coopérative, qui doivent être assistés par un outillage adapté. Nous pouvons résumer la façon dont ces aspects sont abordés par les modèles présentés ci-dessus :

- Facteur d'échelle : Dans des scénarios de conception coopérative, il est souvent nécessaire de pouvoir accommoder un grand nombre de participants sans affecter la capacité à coordonner le travail collectif. Ce facteur d'échelle est atteint par la division des participants en groupes et sous-groupes, qui peuvent être gérés en gardant toujours un nombre maîtrisable de participants par unité de gestion. Cette notion de hiérarchie se

trouve dans les transactions imbriquées et les groupes de transactions coopératives, qui divisent les transactions à plusieurs niveaux, permettant de gérer chaque niveau de manière indépendante.

- Support des transactions non sérialisables : Sous le critère de la sérialisation, lorsqu'une transaction modifie un objet, le travail qui a été fait en utilisant une ancienne lecture de cet objet n'est plus valable. Ceci, dans les transactions traditionnelles, est empêché par l'utilisation de protocoles tels que le 2-phase locking. Autrement, la modification d'un objet peut entraîner l'invalidation des transactions parallèles. Ce type de contrainte doit donc être modifié, pour pouvoir supporter des activités multiples et simultanées, qui travaillent souvent sur les mêmes objets. Les transactions coopératives cherchent à proposer des mécanismes de définitions des critères de correction adaptés à chaque cas. L'imbrication des transactions permet de traiter l'échec d'une transaction de façon plus fine, car l'échec peut être traité à différents niveaux, sans avoir besoin chaque fois de remonter jusqu'à la racine. La COO-Sériabilité permet de partager des résultats intermédiaires, mais elle garantit que le résultat final de chaque activité est basé uniquement sur des résultats finaux.

- Longue durée et non-déterminisme des activités : La longue durée des activités de conception et développement rend encore plus grave les limitations de la sérialisation comme critère de correction. Imposer la sérialisation signifierait soit le blocage du travail des autres pendant des périodes longues, soit la perte de longues périodes de travail au moment d'avorter une transaction. Les Split-Transactions essaient de résoudre ce problème, en gardant la notion de sérialisation. Ils permettent aux utilisateurs de diviser une transaction en deux transactions sérialisables, qui peuvent passer à l'etat « *commited* » indépendamment, pour pouvoir partager une partie du travail avec les autres. Cette division des transactions « a posteriori » correspond à la difficulté à prévoir les opérations qui vont faire partie d'une transaction, et comment les activités vont s'enchaîner et communiquer leurs résultats intermédiaires.

3. Systèmes de gestion de la configuration

La gestion de la configuration des systèmes logiciels (ou Software Configuration Management, SCM) est la discipline qui s'occupe du **contrôle de l'évolution des systèmes logiciels** [ELC 92] [Est 00].

Le SCM est l'une des disciplines du génie logiciel qui a connue le plus de succès au niveau

industriel [ELC 02]. Le SCM est reconnu comme un des éléments clé dans la gestion des projets par le Capability Maturity Model (CMM). Le CMM [CMM 93] est un effort du Software Engineering Institute (SEI) pour guider l'amélioration des procédés logiciels en proposant une classification des différents niveaux de maturité, basés sur les stratégies et pratiques utilisées par les procédés. Aujourd'hui, pratiquement tout processus de création de logiciel avec un minimum de complexité utilise un système de gestion de configuration.

Dès son acceptation dans les milieux industriels, l'évolution des systèmes SCM a été étroitement guidée par les besoins et les pratiques courantes dans les projets réels de développement. Ceci rend l'analyse de ces systèmes fort intéressante.

Les systèmes SCM fournissent des services organisés autour de quelques concepts de base, que nous exposerons dans cette section.

3.1. Traçabilité des modifications

L'une des motivations principales des SCM est le besoin de garder l'historique des transformations effectuées sur un système logiciel. Les systèmes de SCM gardent un registre des différentes transformations, ainsi que de l'information associée sur la date de création de l'entrée, l'utilisateur qui a effectué la modification, des commentaires, et d'autres données associées. Le stockage des différentes modifications d'un système logiciel sert à plusieurs fonctions, telles que l'étude de l'évolution, la reconstruction des anciennes configurations du logiciel, la création des nouvelles configurations basées sur différentes modifications correspondant à un critère particulier, etc.

Pour garder la trace de l'évolution d'un système il existe plusieurs stratégies, ou modèles *de versionnement* [CW 98]. Les systèmes SCM traditionnels utilisant un modèle basé sur les états du logiciel pendant son cycle de vie. Ce modèle est le suivant : 1) Chaque nouvel état créé par une modification est stocké avec un identificateur unique. 2) Les relations de prédécesseur-successeur entre les différents états sont stockées. Ces états et les relations entre eux forment un graphe; plus précisément, un graphe dirigé et acyclique (car un état peut avoir plusieurs chaînes de prédécesseurs et de successeurs, mais nul état ne peut être un prédécesseur ou un successeur de lui-même).

La relation de successeur peut être de plusieurs types, selon l'intention des modifications qui relient un ou plusieurs états vers un nouvel état :

1. Historique : Un état (ou version) est appelé une révision d'un ancien état si le premier est créé comme une amélioration directe du deuxième. Par exemple, la correction d'une

erreur entraîne la création d'une révision.

2. Variation : Deux états partagent une partie commune, mais ils ont des parties différentes (correspondant à différentes fonctionnalités ou implémentations) qui peuvent exister et évoluer en parallèle. Ces versions sont appelées « variantes ». Lorsque plusieurs versions d'un objet co-existent pour supporter la coopération entre plusieurs participantes, ils sont appelés *copies coopératives*.

3. « Fusion » : Une relation de *fusion* est établie entre une version et deux (ou plusieurs) prédécesseurs, qui sont des variantes entre eux. La relation signifie que le nouvel état est la combinaison de ces prédécesseurs.

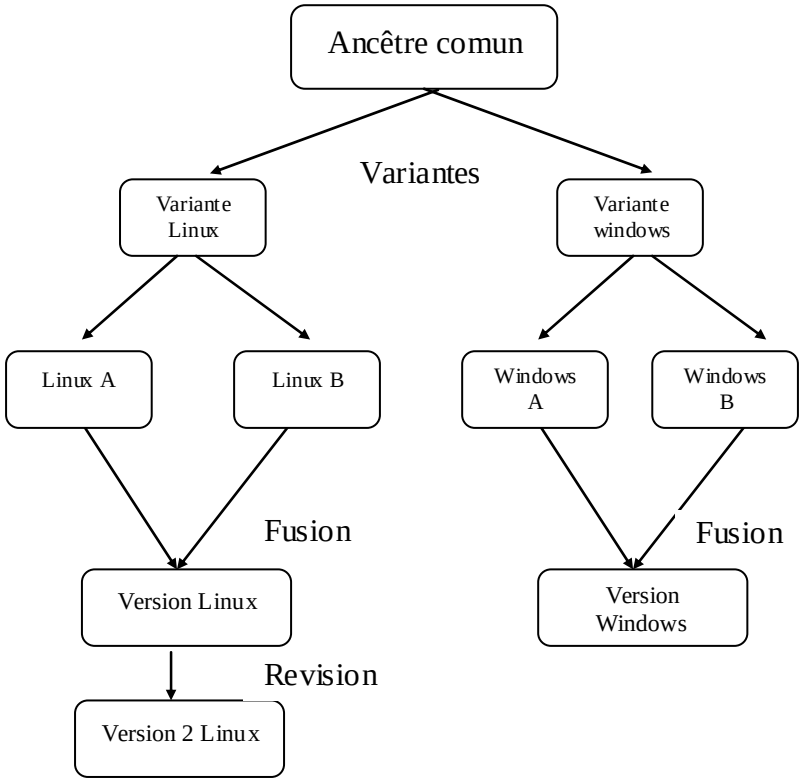


Figure 5 : Revision, Variantes, Fusions

Le modèle antérieur se focalise sur le stockage des différents états des objets logiciels. Un deuxième modèle de versionnement (dit orienté « change-set ») met les changements eux-mêmes comme étant les objets centraux à manipuler par le système SCM. Dans ce cas, les états du logiciel sont traités comme des ensembles de changements appliqués à un état d'origine. Ceci facilite la construction des états du logiciel qui n'ont jamais existé, en combinant des modifications qui ont été créées indépendamment les unes des autres.

3.2. Modèle de produit et sélection de configurations

Les systèmes logiciels sont des systèmes complexes qui doivent être gérés par les systèmes SCM comme des ensembles d'objets reliés par des relations de plusieurs types. Plusieurs langages existent pour faciliter la modélisation des systèmes logiciels : Module Interconnection Languages (MILs), Architecture Definition Languages (ADLs), des langages de conception orienté objets tels que UML, etc. Les systèmes SCM se sont focalisés d'abord sur le contrôle des fichiers et ainsi les systèmes logiciels étaient traités par les SCM comme des hiérarchies des fichiers, ce qui empêche des fonctionnalités avancées basées sur des modèles de produit plus complets (tels que la construction des espaces de travail basée sur des critères sophistiqués).

Certains systèmes ont implémenté des modèles de produit spécialisés pour la gestion de la configuration. Ceci est particulièrement le cas du système ADELE [EC 94] qui fournit un langage de modélisation orienté objets versionnés permettant la définition des relations telles que la composition, la dérivation et la dépendance entre les différentes parties d'un système. Cependant, le système de fichiers (parfois enrichi avec des attributs) reste encore le modèle de produit standard pour la plupart des systèmes SCM commerciaux.

3.3. Contrôle des espaces de travail

Un espace de travail est une partie d'un système de fichiers ou les fichiers d'intérêt pour développer une tâche sont stockés. L'espace de travail sert plusieurs fonctionnalités :

1. Exposer les objets logiciels au développeur dans un format manipulable par les outils de développement (éditeurs, compilateurs, logiciels de test, etc.).
2. Protéger le déroulement de la tâche des interférences extérieures en n'acceptant des modifications qu'à l'initiative de son responsable.
3. Protéger des objets publics des modifications partielles.

Un système de gestion de configuration est chargé de créer des espaces de travail avec les fichiers nécessaires et de sauvegarder les modifications lorsque le travail est fini.

3.4. Sélection de configurations

La *sélection de configurations* s'adresse au besoin de construire des espaces de travail appropriés à des activités particulières, sans avoir à choisir à la main chaque version de chaque objet qui constitue le produit logiciel. Des systèmes tels que CVS ou Subversion

peuvent manipuler des configurations en fournissant un nom symbolique associé à un ensemble de versions précises de fichiers, ou une date précise (sélection par *extension*). Autrement, le système construit toujours les espaces de travail basé sur les dernières versions de la variante principale du système. N'importe quelle autre configuration doit être construite à la main par l'utilisateur.

Il existe plusieurs stratégies qui permettent de sélectionner des configurations par *intention* :

1. Des requêtes générales basées sur le modèle de produit versionné. Par exemple « sélectionner la dernière configuration approuvée par Pierre, avant la première configuration approuvée pour le système d'exploitation Windows ».

2. Dans des systèmes orientés « change-sets », des configurations peuvent être sélectionnées « par intention » avec des requêtes du style « configuration initiale + bug fix 3 + fonctionnalité de recherche + adaptation du plugin 2 ». Il existe cependant plusieurs problèmes qui rendent difficile l'implémentation de ce type d'approche, tels que les possibles conflits entre des modifications incompatibles.

3. L'utilisation des hiérarchies d'espaces de travail, dont la relation espace de travail parent – espace de travail fils correspond à une division des procédés en tâches et sous-tâches. Ainsi, un espace de travail utilise par défaut les objets utilisés par son espace de travail père, mais il peut utiliser aussi des versions locales de certains objets. La spécification des objets utilisés par un espace de travail est donc réutilisé, minimisant les choix manuels à réaliser.

4. Des systèmes de règles ordonnées. C'est le système utilisé par des systèmes de SCM tels que *ClearCase*. Ce type de système utilise plusieurs règles qui définissent les versions à sélectionner. Le système applique la première règle utilisable, de la sélection la plus appropriée à la plus générale permettant à l'utilisateur de définir plusieurs critères de sélection [Leb 94].

3.5. Contrôle des changements

Une approche disciplinée à la modification du logiciel nécessite la compréhension des raisons pour lesquelles le logiciel va être modifié. Ceci est acquis en mettant en place un procédé par lequel les demandes de modification et les rapports de problèmes sont capturés et traités. Ce procédé, initialement supporté « manuellement » avec de la documentation papier, fut incorporé dans les systèmes SCM pour être automatisé, en stockant les demandes et les rapports, ainsi que ses relations avec les modifications réelles du logiciel.

3.6. Gestion des utilisateurs multiples

Il est habituel dans de grands projets, que plusieurs espaces de travail hébergent des copies concurrentes des mêmes fichiers. Les tâches fondamentales des gestionnaires de configuration à ce sujet sont 1) contrôler le travail concurrent (qui peut modifier quoi, quand) et 2) supporter la synchronisation des espaces de travail, lorsque des modifications concurrentes sont effectuées.

Dès le début de la discipline du SCM [Tic 85] le protocole « check-in check-out » avec verrouillage des fichiers fut le moyen primordial de contrôle de la concurrence, et il continue à être à la base de multiples systèmes actuels. Ce protocole marche ainsi:

Lorsqu'un utilisateur veut travailler sur un fichier, il exécute une opération de *check-out,* qui crée une copie du fichier dans son espace de travail, et met un verrou sur le fichier. Le verrou permet aux autres utilisateurs de lire le fichier, mais pas de le modifier. Lorsque l'utilisateur a finit son travail, il exécute un *check-in* du fichier, une opération qui consiste à copier le fichier à partir de l'espace de travail vers le dépôt. Quand le fichier a été copié, le verrou est enlevé.

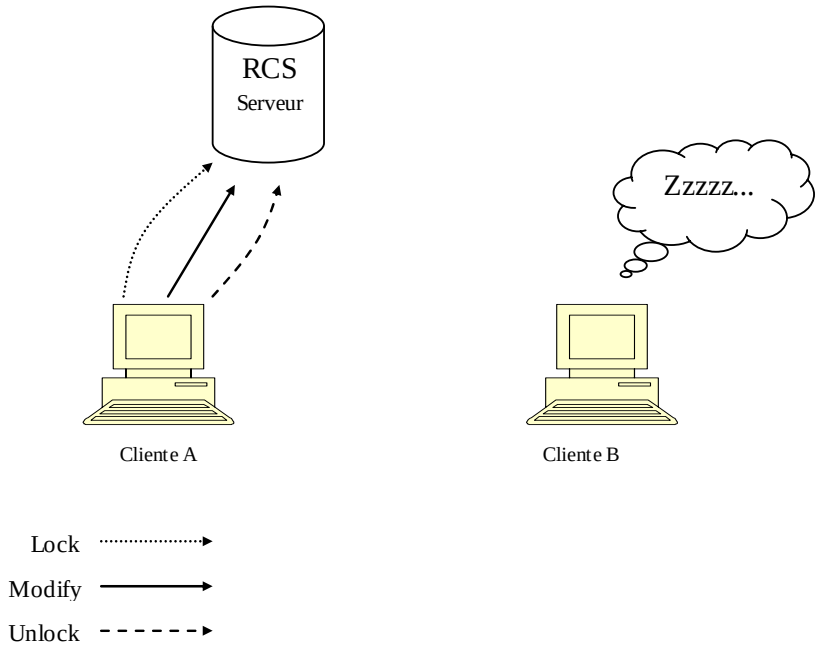


Figure 6 : Modèle *check-in check-out*

Lorsque la concurrence est permise parmi les participants, les systèmes de gestion de la configuration doivent supporter la re-synchronisation des espaces de travail concurrents. La re-synchronisation est couramment limitée à la « fusion » des répertoires et des fichiers source, traités comme des listes ordonnées de lignes de code. Des mécanismes de

synchronisation plus avancés existent, mais ils sont malheureusement moins utilisés ; ce qui est peut-être dû à l'applicabilité plus générale de fusion de texte par rapport aux outils spécialisés dans un langage spécifique.

4. Fusion

Deux besoins s'opposent dans l'édition simultanée des systèmes logiciels. Le premier est le besoin d'isolement, nécessaire pour les activités de développement, et le deuxième est le besoin d'avoir un résultat commun à ceux qui travaillent en concurrence. L'isolation est fournie par les espaces de travail, qui garantissent que les objets qui y sont ne changent pas sans le consentement du propriétaire de l'espace de travail. Le besoin d'un résultat commun implique qu'il doit exister un moyen de combiner les différentes modifications dans un unique résultat, partagé par tous les participants. Ce moyen est la réconciliation de deux versions différentes d'un même objet dans une version unique, qui est censée remplacer les deux versions initiales. Cette opération est couramment appelée une « fusion ».

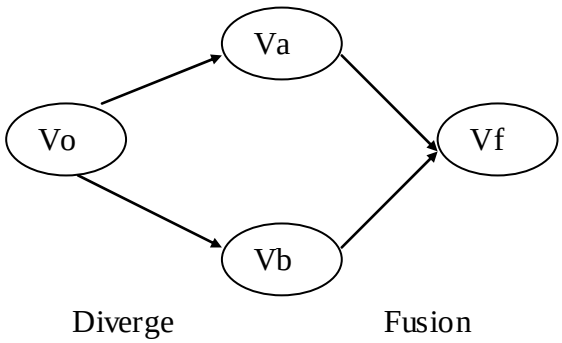


Figure 7 : Divergence et fusion

Selon le type de donnée modifié, il peut exister une opération de fusion garantissant que le résultat de la fusion est correct. Par exemple, si l'objet modifié est un ensemble d'objets, manipulé par des flots d'opérations d'ajouts et de suppression d'objets, une opération de fusion peut être définie en utilisant les sémantiques associées aux opérations, de la manière suivante:

Ensemble Final = (Ensemble Original ∪ Éléments ajoutés par A ∪ éléments ajoutés par B) - (Éléments supprimés par A ∪ Éléments supprimés par B)

La détection de tous les conflits sémantiques entre deux modifications simultanées d'un logiciel est un problème indécidable [Men 02], ce qui veut dire qu'il n'existe pas de critère de correction formel permettant de savoir si une fusion est correcte ou non. Cela se traduit par le besoin d'intervention humaine pour toute fusion effectuée sur des versions concurrentes. La

complexité des systèmes logiciels rend l'activité de fusion complexe, longue, et risquée pour la cohérence des données.

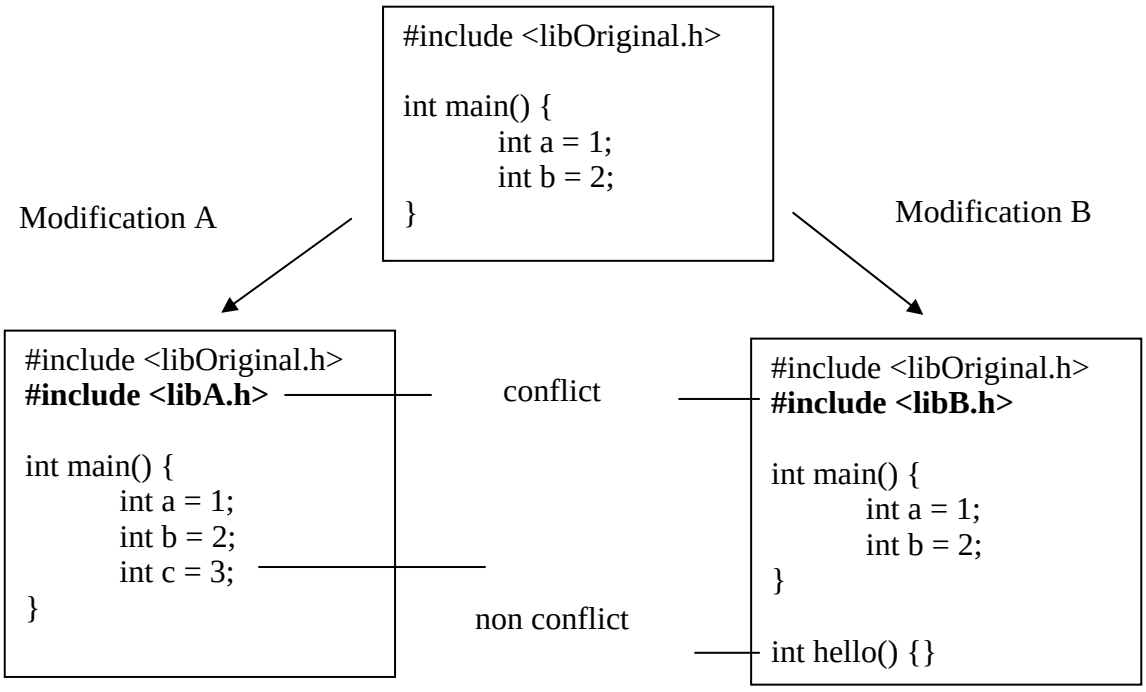
Néanmoins il existe plusieurs algorithmes qui guident le processus de fusion, principalement ils essaient d'identifier les conflits possibles. Les algorithmes et les outils qui les implémentent peuvent être classifiés en plusieurs types [Men 02] : textuel, syntactique, sémantique.

4.1. Fusion textuelle

Cette famille d'outils de fusion traite les objets logiciels comme étant soit des fichiers de texte, soit des fichiers binaires. Il n'est pas possible de guider la fusion des objets binaires, car ils sont interprétés comme des objets atomiques. Les fichiers de texte sont assimilés à des listes de *lignes de code*, qui sont eux, des objets atomiques ordonnés.

Le seul conflit détecté par la fusion textuelle est la modification concurrente d'un même segment de la liste des lignes de code.

Exemple:



Selon des études statistiques menées dans des environnements industriels, 90% des fusions ont correctement marché avec des outils de fusion textuelle. Dans 10% des cas, la fusion a eu besoin d'intervention humaine. Cela explique peut-être la persistance des fusions textuelles

par rapport aux outils plus avancés: aucun d'entre eux est 100% fiable, mais la fusion textuelle est « assez bonne » et a l'avantage de ne pas être liée à un domaine ou langage spécifique.

Il faut noter que les statistiques sur le nombre de conflits sont difficiles à interpréter, car sûrement la précision de l'outil de fusion a une incidence sur la façon dont les gens travaillent en concurrence (par exemple, en évitant les conflits eux-mêmes).

4.2. Fusion syntaxique

Les fusions prennent en compte la syntaxe du langage de programmation dans lequel les entités logicielles sont représentées [Buf 95].

Les outils de fusion syntaxiques peuvent être classifiés selon la structure de représentation de code qu'ils utilisent : des arbres syntaxiques [Ask 94] ou des graphes.

4.3. Fusion sémantique

Les fusions sémantiques font typiquement des analyses statiques du code source pour trouver des conflits fréquents.

Par exemple, lorsque la déclaration d'une variable est effectuée dans un espace de travail et une nouvelle utilisation de la même variable est ajouté dans un autres espace de travail, une fusion basée sur les arbres syntaxiques ne serait pas capable de trouver le conflit, car la relation entre l'utilisation de la variable et sa définition n'est pas représentée dans l'arbre.

4.4. Fusion structurelle

Les outils de fusion dits structurels se spécialisent dans la réconciliation des changements qui ne changent pas le comportement d'un programme mais sa structure. Ce type de modification (appelé *refactoring*) est typiquement associé au besoin de rendre le code plus facile à maintenir.

5. Travail Coopératif Assisté par Ordinateur

Le travail coopératif assisté par ordinateur (CSCW) est un terme utilisé pour regrouper plusieurs efforts destinés à proposer des solutions informatiques aux problèmes propres au travail en groupe. [God 94] définit le travail coopératif assisté par ordinateur comme la « collaboration à l'aide de l'ordinateur en vue d'augmenter la productivité et/ou la

fonctionnalité des processus personnes à personnes ». Pour [BS 89], le terme *travail coopératif* implique la production d'un produit ou d'un service. La littérature qui est considérée habituellement comme faisant partie du domaine CSCW est très hétérogène ; elle incorpore des éléments de psychologie sociale, théorie organisationnelle, anthropologie, pédagogie et, en général, des disciplines qui s'intéressent aux activités de groupe, ainsi que des domaines purement « techniques » comme les technologies de synchronisation de données, des interfaces homme-machine et autres. Le domaine du CSCW est donc multidisciplinaire. [Grud 94] [BS 89]

[Grud 94] propose une classification des outils de CSCW basée sur deux axes : la position géographique des membres du groupe (le même endroit, des endroits différentes mais prévisibles, des endroits différents mais pas prévisibles) et les temps de participation (simultanés, différents mais prévisibles, différents pas prévisibles). Le tableau suivant montre plusieurs applications représentatives du CSCW classifiées selon ce schéma :

Temps Lieu	Simultanés	Temps différents, mais prévisibles	Temps différents, non prévisibles
Même endroit	Facilitateurs des réunions		
Sites différents, mais prévisibles	Teleconference, videoconference	Email	Editeurs collaboratives
Sites différentes, non prévisibles	Séminaires interactives	Web « forums »	WorkFlow

5.1. Edition coopérative

L'édition coopérative (EC) est un des sous-domaines importants du travail coopératif. Le EC cherche à offrir un système permettant à des participants géographiquement distribués, de modifier simultanément des documents partagés. [LCL 94] définit l'édition coopérative comme "un processus itératif et social qui implique un groupe concentré sur une destination commune, et qui communique, négocie et se coordonne pendent la création d'un document commun"

Il existe plusieurs types d'éditions coopératives, qui suivent plusieurs décisions de conception :

- Synchrone vs Asynchrone. Avec l'édition dite synchrone, tous les participants travaillent simultanément sur une même copie du document. Les modifications du document sont propagées automatiquement par l'outil. L'édition synchrone correspond au paradigme WYSIWYG ("Ce que vous voyez c'est ce que vous avez"). En contraste, dans le style asynchrone chaque utilisateur réalise ses modifications sans l'interférence des autres et les modifications deviennent publiques seulement à l'initiative de son auteur.

- Document structuré / non structuré: Les éditeurs coopératifs peuvent utiliser des documents structurés, c'est à dire qu'ils peuvent traiter les documents comme des ensembles d'élément *typés* reliés entre eux par plusieurs types de relation [DQS 96]. Un modèle de document est une description des types possibles d'éléments et les ensembles d'éléments et des relations entre eux qui constituent un document "légal". Par exemple, un modèle typique utilisé par les éditeurs coopératifs est celui qui organise les documents en "chapitres", "sections" et "paragraphes" (avec des relations de type "une section contient des paragraphes"). L'éditeur Alliance est un exemple d'éditeur de documents structurés [DQS 95]. Le langage XML Schema [XML][XSD] permet de décrire des modèles de documents dont les éléments sont organisés hiérarchiquement.

  Un type particulier de document structuré est le document hypertexte, qui définit un document comme un ensemble d'éléments "media" (texte, images, vidéo, etc.) avec des "liens" entre eux, mais dont la structure des "liens" ne suit pas forcément un format prédéfini. Parmi les éditeurs qui travaillent sur ce type de document, nous trouvons Sepia [SHH 92], Quilt [FKL 88], et les "Wiki" [Wiki]. Le protocole WebDav [WebDav] est un effort pour standardiser le protocole par lequel des documents hypertexte sont modifiés.

  Dans l'abstrait, tout document est "structuré", cependant, lorsque la structure du document ne fournit pas d'information significative sur la nature du document (par exemple, le document est considéré comme un simple ensemble de caractères), nous parlons d'un document non structuré. [Gobby] est un exemple d'éditeur coopératif de documents non structurés.

5.2. Observations

De façon générale, les éditeurs coopératifs se limitent à fournir les mécanismes d'édition multi utilisateurs et ils ne fournissent aucune manière de diriger le travail, ni de contrôler la concurrence (lorsqu'ils l'acceptent).

Dans une étude menée par [NR 04] sur l'utilisation des éditeurs coopératifs, la plupart des

utilisateurs ont signalé la difficulté à intégrer les différentes modifications ainsi que la difficulté à affecter les différentes responsabilités comme étant les principaux problèmes de l'édition coopérative. Ceci suggère que le support de procédé est indispensable pour gérer un procédé d'une complexité significative.

6. Conscience du contexte

Un des moyens d'assister un processus collaboratif est d'enrichir l'information dont chaque utilisateur dispose, avec de l'information relative au groupe et aux autres. L'hypothèse sous-jacente est qu'une bonne information contextuelle peut permettre à chaque individu de prendre des décisions plus aptes au travail commun. C'est l'approche nommée *conscience du contexte* (en anglais "awareness"). [DB 92] définit *awareness* comme l'information sur les activités des autres qui fournit un contexte pour l'activité d'un individu.

[DB 92] propose une classification des systèmes de conscience du contexte pour les systèmes CSCW, selon la façon dont l'information du groupe est collectée. Dans la première catégorie, appelée *informationnelle*, l'information est fournie par les utilisateurs avec des opérations explicites (c'est-à-dire, avec un effort additionnel de la part des utilisateurs). Dans cette catégorie, on peut classer le système de commentaires dans le système de versionnement RCS, où les utilisateurs doivent fournir explicitement de l'information sur la nature de ces modifications. Dans la deuxième catégorie, appelée 'restrictive' par [Dou 95], l'information est récupérée par le système (avec peu ou pas d'intervention des utilisateurs) en tenant compte de l'information existante sur la nature des activités développées par chaque utilisateur (par exemple, selon ses rôles).

Dans notre approche, de façon plus générale, nous pouvons dire qu'un système d'*awareness* peut prendre l'avantage d'un modèle de procédé explicite pour fournir une information contextuelle plus riche et pertinente sans avoir à payer le prix d'un effort additionnel de la part des utilisateurs.

L'information fournie par les systèmes de conscience du contexte est considérée comme secondaire par rapport à l'information sur laquelle les utilisateurs travaillent directement. De ce fait, une des préoccupations principales des systèmes d'*awareness* est de trouver un moyen de faire parvenir l'information du groupe aux utilisateurs sans interrompre ses activités principales.

Nous illustrerons par la suite des systèmes de conscience du contexte avec quelques exemples.

6.1. Conscience du contexte dans les éditeurs collaboratifs

Les éditeurs collaboratifs ne disposent pas, dans la plupart des cas, de mécanismes de guidage de procédés pour supporter la collaboration entre les participants. Cette absence rend intéressant le support de la conscience du contexte dans ce type d'environnements collaboratifs, car avec une bonne connaissance des activités des autres, les utilisateurs peuvent eux-mêmes prendre en main la coordination de ses propres activités d'édition.

Les éditeurs collaboratifs Quilt [FKL 88] et Perp comptent plusieurs mécanismes pour la collecte et la distribution de l'information relative aux activités des participants. Ils utilisent un système de rôles qui permet de réduire le non déterminisme sur la nature des activités développées par quelqu'un. De même, les deux outils permettent l'écriture de « notes » : des commentaires que chaque utilisateur peut écrire sur les documents pendant qu'ils sont édités. Les commentaires ne font pas partie du document édité, ils servent uniquement à supporter la communication et coordination entre les participants. Quilt intègre dans son environnement des outils de courrier électronique, ainsi que de vidéo-conférence, pour faciliter la communication entre les participants sur ses différentes activités.

ShrEdit [MO 92] est un éditeur collaboratif du style synchrone où tous les participants partagent une même copie du document, et la vue de chacun sur le document est mise à jour « en temps réel ». ShrEdit fournit aussi des espaces d'édition isolés où les participants peuvent écrire plusieurs lignes pour finalement faire un «copier/coller» sur le document partagé. ShrEdit ne fournit aucun moyen de contrôler le travail des participants: chacun peut éditer n'importe quelle partie du document à tout moment. La granularité des modifications est le « caractère ». Même si ShrEdit ne fournit pas de mécanismes de guidage des procédés, il fournit plusieurs facilités pour s'informer sur le travail des autres: chaque utilisateur peut demander une vue représentant exactement ce que quelqu'un d'autre est en train de faire, et ainsi avoir une image fidèle de ses activités en temps réel. ShrEdit, comme la plupart des éditeurs collaboratifs synchrones, ne supporte pas les concepts de procédé ni de rôle, la philosophie est de montrer en temps réel le travail de tout le monde pour que chacun puisse adapter ses activités dynamiquement au fur et à mesure que le document change.

Les problèmes liés au passage à grande échelle sont souvent négligés dans ce type d'approche, propre aux éditeurs collaboratifs, car ils n'y a pas de moyen de distinguer ce qu'il est important de contrôler lorsque un grand nombre d'utilisateurs modifient le document

simultanément.

6.2. Systèmes de conscience du contexte avec des copies multiples

Nous présenterons ici deux systèmes qui cherchent à assister la conscience du contexte dans des scénarios de collaboration basés sur des copies multiples. Dans ce type de collaboration, l'objectif principal d'un système de conscience du contexte est de permettre aux utilisateurs d'anticiper la complexité des intégrations à venir, et ainsi réagir en conséquence [MSB 01].

Dans un système de collaboration avec des copies multiples, la concurrence peut être contrôlée directement par les utilisateurs avec les actions suivantes:

1. Réduire la concurrence, en évitant de modifier les mêmes fichiers que les autres (lorsque c'est possible).
2. Réaliser des réconciliations anticipées, pour réduire une divergence avant que l'écart soit trop important.
3. Donner la responsabilité de la réconciliation à la personne la plus qualifiée pour le faire.

Pour cela, les utilisateurs doivent être au courrant de ce que les autres sont en train de faire, d'où l'importance d'un système de support de la conscience du contexte.

Les deux systèmes étudiés ici font l'hypothèse que les développeurs travaillent sur des copies privées dans des espaces de travail personnels, et qu'ils synchronisent leur travail avec un procédé très simple (nommé CMM, pour « Copy/Modify/Merge »):

1. Il existe une copie commune qui ne peut pas être modifiée directement.
2. Chaque utilisateur récupère une copie des objets à éditer, en copiant dans un espace de travail local la copie commune.
3. Chaque utilisateur peut ensuite modifier sa copie privée.

Ils synchronisent ses copies avec la copie commune. Si la copie commune a changé depuis qu'un utilisateur l'a récupérée dans son espace de travail, il est obligé de faire une réconciliation des deux nouvelles versions, dans son espace de travail local.

6.3. Palantir

Palantir [SNV 03] est un système qui cherche à fournir la conscience du contexte dans des scénarios de collaboration autour d'un outil de gestion de versions tel que RCS, CVS ou Subversion. Palantir traite l'état d'un objet comme étant la relation entre la valeur d'une copie dans un espace de travail et la copie dans le dépôt commun. De ce point de vue, Palantir considère les états suivants: *populated, unpopulated, synchronized, changesInProgress, changesReverted, changesCommited, added, removed, renamed, moved, severityChanged*.

Chaque utilisateur peut être notifié des changements d'état dans les espaces de travails des autres. Pour cela, Palantir utilise le système de notification d'événements Siena, qui permet à chaque utilisateur de faire un filtrage des messages selon l'espace de travail de provenance et le type de changement d'état.

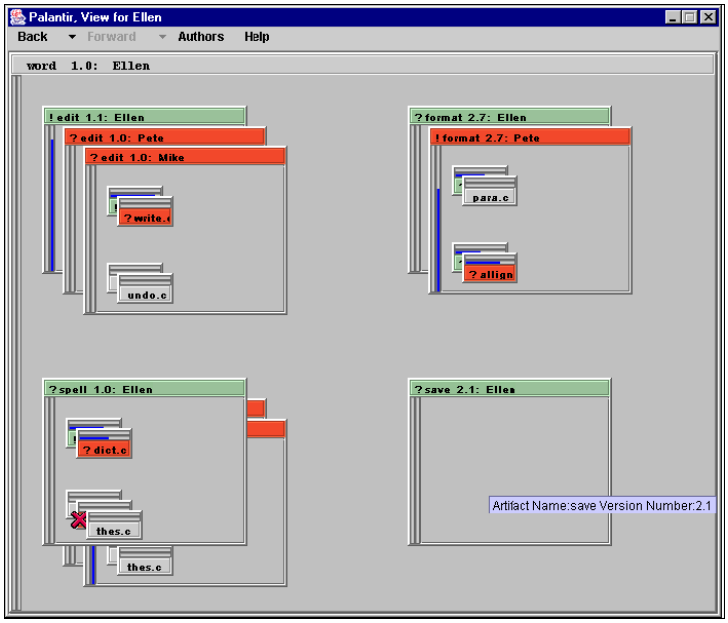


Figure 8 : Palantir

6.4. State Treemap

"State Treemap" [MSB 01] est une vue graphique qui permet aux propriétaires des espaces de travail de visualiser rapidement l'état des hiérarchies d'objets contenues dans leurs espaces de travail. En comparaison avec Palantir, où les états représentent uniquement le rapport entre un espace de travail particulier et la copie commune, State Treemap synthétise également dans l'état de l'objet le rapport entre la copie dans un espace de travail, et les copies dans les autres espaces de travail.

L'état d'un objet correspond à une de ces valeurs:

1. mis à jour (la copie locale est identique à la copie commune),
2. modifié localement,
3. modifié par quelqu'un d'autre,
4. obsolète (la copie commune a été modifiée après avoir été copiée par l'utilisateur),
5. conflit potentiel (modifié localement et modifié par quelqu'un d'autre),
6. conflit (modifié localement et obsolète).

State Treemap utilise une représentation de la hiérarchie qui synthétise l'état général d'une hiérarchie dans un espace graphique limité, de façon à ce que l'appréhension de l'état des objets puisse se faire d'un seul « coup d'œil », limitant ainsi l'effort fait par les utilisateurs.

Figure 9 : State treemap

6.5. Discussion

Palantir et State Treemap fournissent la conscience du contexte pour les procédés qui 1) utilisent des copies des produits, 2) suivent un procédé du style « copy/modify/merge ».

State Treemap fait un effort pour supporter de grands nombres d'objets, en représentant l'état de l'espace de travail dans une vue qui synthétise la relation entre les valeurs des objets dans une copie locale, les valeurs dans la copie commune et les espaces de travail des autres, tandis que Palantir cherche à réduire le nombre des messages avec des mécanismes de filtrage qui doivent être calibrés par chaque utilisateur manuellement.

Nous considérons qu'aucune de ces approches n'est apte au support des scénarios avec de grands nombres d'utilisateurs, où l'information contextuelle cesse d'être utile à cause de l'effort nécessaire pour l'interpréter. Également, nous croyons que le fait de considérer un unique procédé (simpliste) rend ces approches très limitées dans la vie réelle.

Nous exposerons une approche plus générale qui tire partie des connaissances sur le modèle de procédé pour faire face au problème de passage à l'échelle.

7. Conclusions

La modification concurrente des objets électroniques dans l'ingénierie collaborative pose des problèmes complexes qui ont inspiré plusieurs domaines de recherche.

Ces problèmes sont particulièrement importants en génie logiciel, car il est en pratique impossible d'avoir de grands nombres de développeurs travaillant simultanément sans permettre qu'un même objet soit modifié par plusieurs personnes en même temps. Malheureusement, le génie logiciel est encore loin d'avoir une vraie maîtrise des procédés permettant de contrôler de façon fine la manière dont les développeurs travaillent et interagissent; et plus précisément, la manière dont ils effectuent des modifications simultanées. Cela entraîne des nombreux risques pour la cohérence des données.

La première approche au problème de la concurrence que nous avons explorée dans cette section et qui est dérivée de la recherche sur les gestionnaires de bases de données. Cette approche est une évolution naturelle car ce domaine a étudié depuis longtemps les problèmes liés à la manipulation concurrente de données, mais pas dans des contextes d'ingénierie coopérative.

Dans l'approche classique aux bases de données, la sérialisation a été adoptée comme le critère qui garantit la cohérence des données lorsque plusieurs flots d'opérations s'exécutent parallèlement. En ayant la sérialisation comme critère, plusieurs protocoles ont été perfectionnés pour garantir ou vérifier la sérialisation de deux transactions. Malheureusement, les tentatives pour utiliser les bases de données dans l'ingénierie collaborative ont révélés plusieurs difficultés, qui ont inspirés des travaux cherchant à assouplir le critère de sérialisation, trop strict pour les réalités dans ce nouveau cadre d'utilisation.

Parmi les difficultés identifiées dans ce genre de travaux, on peut mentionner 1) la durée des activités, qui aggrave le problème de l'interférence, entre les activités qui sont actives simultanément pendant plus de temps que les transactions traditionnelles ; ce qui rend inacceptable l'avortement d'une série opérations, 2) le non déterminisme des activités qui rend difficile de prévoir les ressources qui vont être utilisées pendent le déroulement de l'activité (ce qui a incité des travaux tels que les split-transactions), 3) la nature des données manipulées par le génie logiciel, qui sont hautement couplées. Cela veut dire que toute activité a souvent besoin d'une grande partie des données, rendant ainsi les différentes transactions non sérialisables. 4) le besoin de pouvoir structurer les différents flots de données (activités) pour répondre à la division du travail en tâches et sous-tâches, et ainsi pouvoir

répondre aux problèmes de passage à l'échelle lorsque de nombreux intervenants font partie d'un même projet.

Parmi les conclusions qui ont été tirées de ces travaux, nous retenons qu'il est nécessaire de **permettre aux utilisateurs de définir leurs propres critères de correction**, car il est impossible d'établir un critère adapté à tous les types de situation (types de données, types d'activité, niveaux de responsabilité des gens impliqués, etc.). Certaines approches, dans les modèles avancés de transactions, ont essayé de fournir cette flexibilité en proposant un langage pour la définition d'historiques valides des opérations.

L'idée d'adapter la notion de correction à chaque cas particulier s'approche des stratégies des technologies des procédés. Ces technologies se basent sur la description explicite de la manière dont un objectif doit être réalisé (i.e. comment créer un produit). Les outils de gestion de procédé utilisent ce type de descriptions pour piloter une infrastructure informatique de support qui inclut le guidage du procédé réel, l'automatisation des tâches répétables, la traçabilité, etc. Mis à part le support informatique, l'existence même des descriptions explicites des procédés est une aide considérable à la compréhension, la gestion, la communication et l'amélioration de la façon dont les objectifs d'une organisation sont atteints. Nous inscrivons notre proposition dans ce genre de stratégie.

Cependant, la technologie des procédés, qui donne de bons résultats pour certains types de procédés comme la gestion documentaire et les procédés administratifs, n'a pas obtenu le même succès dans les domaines créatifs et de conception collaboratrice. Il est donc nécessaire d'identifier les particularités de ce genre de procédé pour trouver les besoins qui ne sont pas satisfaits par les systèmes de support de procédés actuels.

Il est intéressant d'étudier les systèmes de gestion de la configuration de logiciel (SCM) car, de facto, ce sont les systèmes autour desquels les participants des projets complexes de développement logiciel travaillent coopérativement aujourd'hui. Les systèmes de gestion de configuration fournissent des espaces de travail pour satisfaire le besoin d'isolement pendant le développement des tâches, et garantir un minimum de sécurité en permettant de récupérer des anciennes versions des objets modifiés. Ces systèmes ont identifié la fusion des modifications parallèles comme un des éléments centraux à supporter pour la collaboration asynchrone. Pour cela, des outils de fusion automatique sont mis à disposition des utilisateurs, qui doivent néanmoins vérifier la validité des fusions de manière manuelle.

Le fait est que, même avec des outils de fusion, la fusion de modifications concurrentes reste difficile et les données restent vulnérables à des erreurs dues à l'incompatibilité des

modifications fusionnées. C'est pourquoi les outils de gestion de la configuration permettent d'interdire totalement la concurrence, pour les scénarios où les utilisateurs trouvent la fusion trop risquée. Ceci est une mesure extrême, car la concurrence est nécessaire lorsqu'il est intolérable de bloquer le travail des développeurs. Il n'existe pas de gestionnaire de configuration permettant d'adapter le contrôle de la fusion aux particularités de chaque procédé, équipe, culture organisationnelle et vulnérabilité des données.

La popularité du développement asynchrone aidée par des outils de fusion de texte, ainsi que le besoin des utilisateurs de restreindre la concurrence pour limiter les risques, suggère que **la fusion doit être assistée et contrôlée explicitement**.

Finalement, nous avons étudié les outils CSCW. CSCW est un terrain de recherche très hétérogène qui recouvre les différents moyens d'améliorer le travail coopératif avec des ressources informatiques. Les éditeurs coopératifs sont un cas particulier des outils CSCW qui, dans le cas général, ne supportent pas la notion de procédé. Ils proposent d'autres mécanismes pour permettre un minimum de coordination entre les participants, tels que la conscience du contexte. La conscience du contexte permet aux utilisateurs de coordonner ses actions de manière dynamique, sans avoir à anticiper les activités précises et les échanges entre les activités. Ceci est intéressant, car une des limitations des systèmes de support des procédés traditionnels dans le contexte du génie logiciel est le fait que ces procédés soient souvent soumis à des changements inattendus dus à des modifications des besoins, des changements de priorité, de l'instabilité des modifications, qui sont difficilement prévisibles. Même si des « familles d'activités » peuvent être identifiées, les activités précises, le quand et comment des échanges entre ces activités, sont difficiles à anticiper, et l'application d'un plan rigide s'avère trop restrictif.

La plupart des systèmes de support des procédés font l'hypothèse implicite que:

- Un procédé est une séquence d'activités qui doivent être exécutées dans un ordre précis. Une étape ne peut pas commencer avant que l'étape précédente soit complètement finie.
- Le procédé s'achève avec la réalisation d'un produit complètement fini (et donc le versionnement n'est pas un problème).
- Il existe une copie unique des données, et donc la concurrence est bannie ou non-contrôlée.

Cette vision d'un procédé correspond au modèle de développement logiciel dit en "cascade".

Ce modèle de développement a été, et reste encore, populaire pour de nombreuses raisons [MJ 05] (ils sont faciles à expliquer et à retenir et ils donnent un sentiment d'ordre et mesurabilité.[1])

La critique la plus répandue au modèle en cascade est que celui-ci n'est adapté que lorsque les besoins et les technologies ne changent pas, ou changent très lentement, ce qui n'est pas souvent le cas dans le monde du logiciel.

Or, on observe que, en pratique, les procédés de développement logiciel sont :

- itératifs,
- concurrents,
- dynamiques.

Nous croyons que pour supporter l'ingénierie concurrente dans le génie logiciel, un système doit réunir les caractéristiques suivantes :

- Permettre aux utilisateurs de définir la façon dont ils veulent que le travail soit fait, avec un langage de définition de procédés.
- Ce langage doit tenir compte du caractère non déterministe des procédés de génie logiciel. C'est à dire, le langage ne doit pas imposer de définir toutes les instances d'activités ni l'ordre explicite des échanges qui seront réalisés.
- Le langage doit tenir compte du caractère itératif des procédés de génie logiciel.
- La fusion étant une des préoccupations fondamentales dans le génie logiciel concurrent, elle doit être traitée **explicitement** dans le langage de définition des procédés.
- Le passage à l'échelle doit être pris en compte.

[1] Le modèle en cascade a été parfois également renforcé par des régulations officielles des gouvernements [MJ 05]

Chapitre III. Support à l'ingénierie concurrente

Nous plaçons notre proposition dans le domaine de la modélisation et du support des procédés. Notre principale contribution à ce sujet est la proposition d'un langage qui permet la modélisation des procédés en tenant compte des aspects particuliers du génie logiciel, souvent négligés dans ce type d'approche, et en particulier, le contrôle de l'ingénierie concurrente.

Notre proposition sera divisée en deux parties. Dans la première partie nous définissons notre langage pour la modélisation des procédés de génie logiciel concurrents et nous expliquons les mécanismes qui permettent l'application dans la réalité des procédés ainsi définis. La deuxième partie propose un système dit d'augmentation de l'information contextuelle, qui prend avantage de l'existence des modèles de procédés, pour fournir aux utilisateurs une information pertinente pour leur travail.

Les deux approches ne sont pas opposées, elles se complètent [GHB 01][EG 06]. Cependant, chaque stratégie correspond à une vision différente sur le rôle des modèles de procédé dans un système de support. La première position peut être nommée **prescriptive**: la description du procédé est un plan qui **doit** être respecté, et la fonction du système de support est de garantir que le plan est suivi. La deuxième philosophie peut être qualifiée de **descriptive**, car elle prend la description du procédé simplement comme une source d'information sur la façon dont le procédé devrait se dérouler : la fonction du système de support est d'utiliser cette information pour enrichir les outils de collaboration dont les utilisateurs disposent, tels que le système d'augmentation de l'information contextuelle.

	Politique	Conscience du contexte
Philosophie	Prescriptive	Descriptive
Objectif	Application	Guidage
Avantages	Rigoureux, fiable	Indicatif, optionnel
Inconvénients	Rigide	Trop flexible
Responsabilité	Centralisée	Distribuée parmi les participantes

Malgré les différences, les deux philosophies sont complémentaires : l'application des procédés aide à élever la fiabilité du modèle de procédé comme une description, permettant à un système de conscience du contexte d'être plus fiable. Pour sa part, la conscience du contexte permet la définition des procédés flexibles, car le comportement des utilisateurs est censé être moins "risqué" grâce à l'information additionnelle dont ils disposent.

1. Hypothèses de base

Nous nous basons sur un certain nombre d'hypothèses sur la manière dont la collaboration est réalisée. Évidement, cela exclue plusieurs styles de collaboration dont la pertinence pour le génie logiciel peut être discutable. Cependant, nos hypothèses définissent un modèle de collaboration assez général où la plupart des scénarios de développement traditionnels peuvent s'inscrire.

1.1. Génie logiciel coopératif

La coopération implique un objectif partagé par les plusieurs participants. Dans le génie logiciel, cet objectif est de transformer un produit logiciel depuis un état initial vers un nouvel état. Ce nouvel état constitue un résultat unique et commun à tous les participants. Le rôle de l'ingénierie coopérative est de contrôler cette transformation.

1.2. Espaces de travail

La réalisation d'une activité de développement est souvent une tache réalisée de façon incrémentale. Une activité est souvent composée de petites modifications dont la signification ne peut être comprise que comme étant une partie d'une grosse modification qui, elle, peut être expliquée, documentée et partagée avec d'autres activités. Ceci fait qu'il soit nécessaire de donner à chaque activité un contexte isolé dans lequel ces modifications peuvent être réalisées sans interférer avec les activités des autres. C'est précisément le rôle des espaces de travail: héberger pendent des périodes arbitrairement longues des copies entières des produits, afin de permettre le déroulement simultanée et isolé des activités multiples.

Nos espaces de travail son divisés en deux zones, une **zone de travail** et une **zone de stockage historique**.

La **zone de travail** est un espace dans un système de fichiers ou les produits sont stockés dans des fichiers en utilisant la représentation standard utilisée par les outils de développement (éditeurs, compilateurs, etc). Ces fichiers sont modifiés uniquement à l'initiative du

responsable de l'espace de travail, fournissant ainsi l'isolement nécessaire pour le développement de ses activités. En effet, La zone de travail sert "juste" à stocker le produit pendent qu'il est transformé par des petites modifications, jusqu'au moment où la modification totale est considérée comme étant assez significative pour être traitée comme une modification finalisée. Les différents états traversés par un produit dans la zone de travail peuvent ne pas satisfaire les contraintes de cohérence sur le produit, car elles servent de support à la transition depuis un premier état "correct" vers un nouvel état "correct".

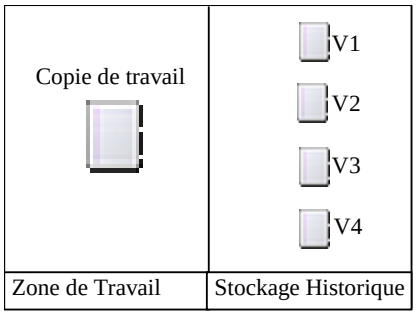


Figure 10 : Espace de travail

La **zone de stockage historique** garde tous les états successifs d'un produit que le responsable de l'espace de travail juge convenable d'enregistrer. Ces états peuvent toujours être récupérés à posteriori, par exemple pour revenir à un état antérieur à l'introduction d'une certaine modification ou pour étudier l'évolution du produit ou les modifications introduites pendant une période de temps quelconque.

Il est important de remarquer que l'espace ou les versions historiques sont stockées est propre à chaque espace de travail, ce qui rend possible d'avoir cette fonctionnalité installée directement sur le poste de travail du responsable de l'espace de travail. Cela permet d'avoir les fonctionnalités du versionnement historique sans établir une communication sur le réseau entre deux machines distantes. En ceci, cette approche contraste avec la plupart des systèmes de gestion de la configuration qui adoptent une architecture serveur/client, ou toutes les opérations relatives au versionnement requirent une communication sur le réseau. Notre approche requiert donc moins de communications sur le réseau, ce qui est un avantage de plus en plus important compte tenu de la taille et du nombre important des messages qui doivent être envoyés a cause de la taille des logiciels et du nombre de développeurs dans un projet. Parallèlement, les coûts moins élevés des moyens de stockage rendent moins significative la perte d'espace qu'implique la redondance des données nécessaire dans ce type d'approche.

Actuellement il existe plusieurs systèmes expérimentaux qui commencent à adopter ce style d'espace de travail [Monotone][Darcs][Arch].

1.3. Entités logiques

Pour la plupart des environnements de génie logiciel, un logiciel est un ensemble de fichiers. Cette interprétation du mot « logiciel » facilite leur implémentation, car la vraie structure du logiciel, souvent beaucoup plus complexe, est ignorée. Dans la réalité, un logiciel est caractérisé de manière plus proche de la réalité comme étant un ensemble d'unités logiques reliées par des liens sémantiques complexes. Les relations entre les différentes parties d'un logiciel vont largement au-delà des relations qui sont représentes par un système de fichiers.

La représentation simpliste du logiciel comme un ensemble de fichiers explique plusieurs limitations des environnements de génie logiciel actuels. Par exemple, les systèmes de SCM permettent de limiter la modification concurrente avec le verrouillage de chaque fichier avant la modification, mais ils ne fournissent pas des moyens de verrouiller des objets logiques tels qu'un « package », un « module » ou un « sous-système ». Dans ce cas, la modification des deux fichiers différents qui font partie d'une même entité est traitée comme une opération sans risque, même si la cohérence de l'objet composite est en danger.

Transformation entre entités logiques et fichiers

Nos politiques d'ingénierie concurrente expriment la façon dont les entités logiques doivent être modifiées et transmises par les utilisateurs. Étant donnée que les outils de développement opèrent sur des fichiers, une fonction pour traduire des entités logiques sur des ensembles des fichiers dans un espace de travail (et vice-versa) est nécessaire. Un système de support de l'ingénierie concurrente, lorsqu'il applique une politique définie dans notre langage, doit utiliser la fonction de projection pour appliquer la politique sur les fichiers réels (par exemple, en contrôlant l'accès concurrent à l'ensemble des fichiers qui constituent une entité logique protégée).

Les entités logiques ne se correspondent pas forcement à des ensembles disjoints: ils peuvent partager des fichiers. (Un « package » peut contenir les fichiers des deux modules, deux « composants » peuvent partager un même fichier de définition des interfaces fonctionnelles, etc.).

Dans notre langage nous faisons l'hypothèse qu'il existe un modèle de données définit par l'utilisateur ainsi que la fonction qui projette ces entités logiques sur des ensembles de fichiers. Cependant, pour faciliter l'utilisation de notre langage, nous permettons l'utilisation des expressions d'expansion des fichiers, qui permettent de définir des entités logiques par des expressions régulières. Cela est palliatif à un vrai modèle de produit, mais en combinaison

avec des règles de nommage adéquat il peut être une solution intermédiaire viable.

Exemples :

```
user-interface : src/ui/*/*
component-X : src/headers/X.h src/implementation/*-X.cpp
```

1.4. Opérations basiques

Le responsable d'un espace de travail dispose de deux opérations de base. La première est l'opération **sauvegarder**, qui crée une nouvelle entrée dans la zone de stockage historique avec exactement la même valeur que celle qui se trouve dans la zone de travail. L'opération **charger** change la copie dans la zone de travail pour qu'elle adopte la même valeur qu'une ancienne copie choisie par l'utilisateur.

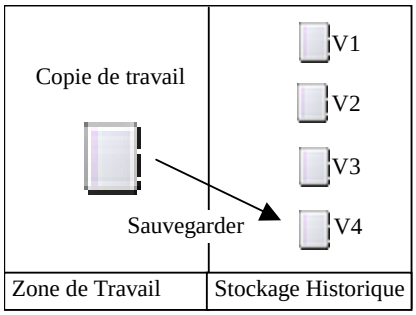


Figure 11 : Sauvegarde

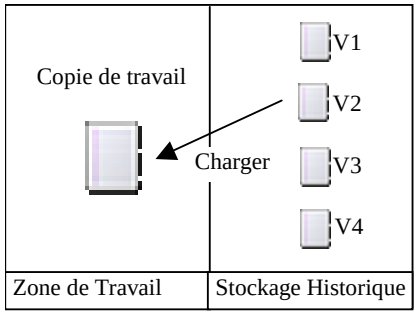


Figure 12 : Chargement

Avec ces deux opérations l'utilisateur a accès à son espace de versionnement historique.

1.5. Communication entre espaces de travail

Le travail développé par un participant doit pouvoir être communiqué aux autres pour établir une collaboration. Pour permettre le transfert de données d'un espace de travail vers un autre, une opération de **transfert** entre deux espaces de travail est mise en place. La synchronisation opère entre un espace de travail *source* et un espace de travail *destination*. La dernière version

dans la zone de stockage de l'espace de travail source est réconciliée avec la copie dans la zone de travail de l'espace de travail *destination*, et le résultat de la synchronisation remplace la valeur de la copie dans la zone de travail de l'espace de travail *destination* et deviens également la dernière valeur stockée dans la zone de stockage historique.

Un transfert ne peut avoir lieu que s'il existe des modifications dans l'espace de travail Source qui n'a pas encore été transmis à l'espace de travail destination. Si le transfert est possible, le transfert va entraîner soit:

- Une **copie**, si l'espace de travail Source connaît déjà tout ce qui est connu pour Destination.
- Une **fusion**, si l'espace de travail Destination et la Source sont tous le deux des modifications d'une même valeur d'origine.

1.6. Copies de référence et groupes

Pour pouvoir gérer un grand nombre de participants dans un projet, nous proposons une division des espaces de travail en unités appelés **groupes**. La division des espaces de travail en groupes permet de traiter un groupe, de l'extérieur, comme une unité de travail atomique, sans avoir à s'occuper des détails internes. Pour cela, chaque groupe a un **espace de travail de référence** qui représente le groupe pendent toute son existence et qui constitue le seul point de communication entre un groupe et l'extérieur

Comme nous l'avions déjà mentionné, nous supposons que le but de la coopération est de produire un résultat commun unique. Ce résultat commun unique n'est pas forcement un état complètement finalisé, car les procédés de génie logiciel sont de nature itérative: les produits sont transformés d'un état "intermédiaire" vers un nouvel état " intermédiaire " cycliquement, sans avoir à attendre la fin du procédé explicitement déclaré pour pouvoir livrer des résultats. Nous supposons que les copies qui se trouvent dans un espace de travail de référence représentent en permanence l'état courrant de la collaboration dans le groupe. En pratique, il est confortable pour les utilisateurs de savoir toujours quelle est la référence sur laquelle ils peuvent baser leur travail.

Dans les systèmes de gestion de versions, le "repository" rempli souvent le rôle de la référence. Dans notre cas, la référence n'est pas (forcement) passive, elle peut être un espace de travail tout à fait normal, avec un responsable.

Les groupes peuvent être structurés de façon hiérarchique, ce qui correspond de façon

naturelle à la division du travail en taches et sous-taches. Pour cela, un espace de travail peut être à la fois un espace de travail normal dans un groupe, et un espace de travail de référence dans un sous-groupe. Pour le groupe “parent”, le fait qu’un des ses espaces de travail soit ou pas la référence d’un autre groupe ne fait aucune différence.

Chapitre IV. Langage de définition de politiques

La définition de notre langage de modélisation de procédés de génie logiciel a été guide par les principes suivants:

- Le niveau d'abstraction doit rester assez élevé pour que le langage soit utilisable par des non-spécialistes. Pour cela, le langage doit utiliser, si possible, uniquement des concepts déjà familiers aux acteurs des projets de développement logiciel.
- Le travail concurrent doit pouvoir être géré par le langage. Pour contrôler la complexité de la gestion de la concurrence, nous privilégions un style déclaratif (contrôle de la concurrence) plutôt qu'un style impératif (programmation de la concurrence).
- Le langage doit supporter la définition d'un bon nombre des politiques connues, compte tenu de nos hypothèses de base, expliquées dans la section suivante.

1. Introduction

La description de chaque procédé de génie logiciel concurrent est définie dans ce que nous appelons une *politique d'ingénierie concurrente*. Dans cette section nous allons définir le langage avec lequel les politiques sont exprimées. Cette description sera effectuée initialement de façon informelle, où nous expliquerons la façon dont notre langage atteint les objectifs que nous nous sommes donnés. Ensuite, nous allons décrire précisément la sémantique du langage, permettant ainsi l'implémentation d'un système d'ingénierie concurrent basée sur ce langage.

2. Contexte de la politique

Une politique décrit un procédé à l'intérieur d'un groupe. Autrement dit, le groupe est le contexte dans lequel une politique est appliquée. Cela est en accord avec la notion du groupe comme une unité de gestion indépendante permettant le passage à l'échelle du développement collaboratif. En restreignant l'application d'une politique à intérieur d'un groupe on peut utiliser des politiques plus ou moins strictes selon les besoins de chaque groupe. Par exemple, le groupe "racine" qui est en charge de gérer les copies qui vont être directement livrés aux

clients, peut imposer des politiques très strictes sur la concurrence, tandis que les équipes chargées du développement de nouvelles fonctionnalités peuvent utiliser des politiques moins sécurisées mais permettant plus de souplesse.

A Dassault Systèmes, pour le développement du logiciel Catia, il existe 7 niveaux de hiérarchies [Est 96]. Les niveaux à la base de la pyramide sont responsable du prototypage et utilisent des politiques laissant beaucoup d'espace à la modification concurrente. Par contre, à la racine de la hiérarchie, ou réside la copie expose au publique (GA : General Availability), les politiques sont très strictes.

2.1. Politiques : déterminisme et non déterminisme

Sans une politique d'ingénierie concurrente, les modifications et les transferts de données peuvent se faire à tout moment et entre n'importe quel couple de participants. Cela veut dire que le travail concurrent, la communication et les réconciliations peuvent se produire de manière complètement chaotique, avec tous les risques associés.

Pour éviter le chaos, une politique définit la façon dont les transferts doivent être réalisés. Ce contrôle est incarné par des règles qui régissent les différents aspects concernés par un transfert. Ces règles sont regroupées dans trois niveaux

- Les **rôles** joués par les personnes dans la collaboration
- Le **contrôle d'accès**, qui contrôle quelles données peuvent être modifiées par quels rôles.
- Le **contrôle de la topologie**, qui prescrit quels espaces de travail, selon leur rôle, peuvent échanger des données
- Le **contrôle de la concurrence**, qui prescrit quelles données peuvent être modifiées *simultanément* par quels rôles, et qui peut les réconcilier.

Il est intéressant de remarquer que le contrôle d'accès ainsi que le contrôle de la topologie peuvent être appliqués en se basant uniquement sur une information statique : les rôles des espaces de travail, indépendamment de l'historique des opérations en exécution. Ces deux niveaux couvrent donc les aspects « déterministes » de la politique : ceux qui sont complètement définis à l'avance. Par contre, l'application du contrôle de la concurrence dépend de l'état courant d'un procédé qui, lui, dépend de l'historique des actions des participants.

Une politique est décrite dans notre langage avec la syntaxe suivante:

```
Policy name {
  // CORPS DE LA POLITIQUE
}
```

Les différentes partis du corps de la politique sont définies ci dessous

2.2. Rôles

Le rôle est un concept qui encapsule le type de responsabilités qui peuvent être attribuées à quelqu'un dans un projet, ainsi que les compétences qui peuvent être attendues de lui. Nous pouvons citer plusieurs exemples de rôles: "développeur senior", "développeur débutant", "responsable QA", "responsable documentation", "intégrateur", "testeur", etc. Dans notre langage, chaque espace de travail se voie attribué un rôle unique, qui caractérise les compétences et responsabilités de son propriétaire. En somme, le rôle d'un espace de travail définit le type d'activités que le propriétaire d'un espace de travail peut entreprendre.

Un rôle peut être "multiple", si plusieurs espaces de travail peuvent assumer simultanément le rôle, où il peut être « unique » s'il ne peut exister qu'un seul espace de travail à un moment donné avec ce rôle précis

La déclaration des rôles constitue la première partie de la politique:

```
Policy politique1 {
      Roles {
            Role1 [*] [REFERENCE][:Liste de produits]
            Role2 [*] [REFERENCE][:Liste de produits]
            …
      }
}
```

Chaque rôle est définit avec un nom et optionnellement suivi par une * indiquant que c'est un rôle multiple. Également optionnel, la mention `Reference` indique que l'espace de travail est la référence du groupe. Naturellement, il ne peut exister qu'un seul rôle, non multiple, correspondant à l'espace de travail de référence du groupe.

Après la déclaration d'un rôle, une liste des produits peut suivre. Cette liste définit les produits qui peuvent être présents dans un espace de travail jouant ce rôle. Par exemple, le rôle de « ingénieur de besoins » peut n'être intéressé que par le document de spécification, et

pas par le code source lui même, qui ne sera jamais amené dans un espace de travail jouant ce rôle. La liste de produits est obligatoire pour l'espace de travail de référence; pour les autres rôles, si la liste est omisse, la liste complète du rôle signalé comme « référence » est utilisé par défaut.

2.3. Contrôle d'accès

Avec des règles de contrôle accès, une politique exprime le besoin d'éviter que certaines données soient modifiés dans des espaces de travail peu appropriés (e.g. un développeur débutant qui touche le cœur du système).

Les règles de contrôle accès suivent la syntaxe suivante :

```
Read-Only {
      Role1 : [logical entities]
      Role2 : [logical entities]
      …
}
```

Par défaut, tous les espaces de travail peuvent modifier leurs données. Chaque ligne restreint les espaces de travail jouant *Rôle* à ne pas modifier les entités logiques définies dans la liste.

2.4. Contrôle de la topologie

Il est parfois nécessaire d'imposer un certain ordre dans les transferts des données parmi les utilisateurs. Par exemple, il peut être absolument nécessaire que les modifications passent par un control de qualité avant d'être intégrées à la version publique, qu'une documentation soit écrite avant de livrer une bibliothèque aux développeurs, ou qu'un manager puisse contrôler les échanges entre les développeurs.

Pour gérer l'ordre dans lequel les données sont acheminées, nous introduisons un niveau de contrôle du flot de données parmi les espaces de travail, selon le type d'activités qu'ils supportent, c'est-à-dire, selon leur rôles.

Le flot de données parmi les espaces de travail peut être représenté par un graphe dirigé (appelé le **graphe de communication**), ou les noeuds représentent les rôles, et les arcs représentent les canaux par lesquels un transfert de données est « légal ».

Contrairement aux graphes de plusieurs systèmes de workflow, dans notre graphe le *flot de*

contrôle n'est pas défini:

- Les noeuds sont des d'espaces de travail, pas des activités.
- Avant et après un transfert des données, l'espace de travail source et l'espace de travail destination travaillent simultanément. La direction d'un arc dans le graphe de communication n'indique pas une quelconque ordre dans exécution des activités hébergées par les espaces de travail.

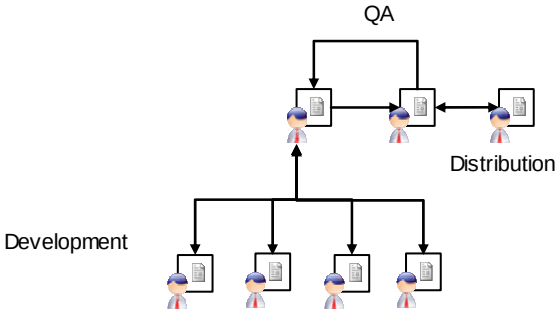


Figure 13 : Graphe de communication

Le patron le plus basique de graphe de communication est l'étoile. Dans l'étoile, les participants ne peuvent pas communiquer directement leur travail. Ils communiquent seulement à travers un espace de travail qui centralise tous les échanges. Cette topologie, qui est utilisée par un grand nombre de systèmes de génie logiciel, est intéressant car la copie du produit dans l'espace de travail de référence n'est jamais très éloigné de l'état d'avancement de chaque personne, représentant ainsi adéquatement le travail collectif.

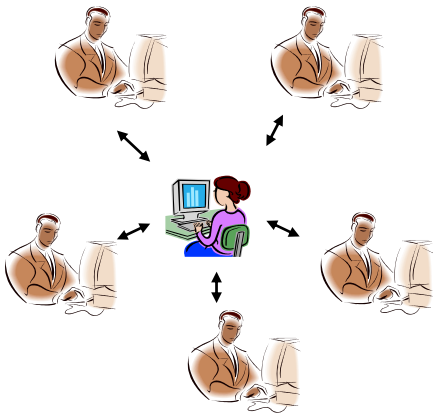


Figure 14 : Politique Etoile

Les règles de contrôle de la topologie suivent la syntaxe suivante :

```
graph {
```

```
      Role1[*] [-> | <->] [*] Role2 : [Entités logiques]
      …
}
```

Chaque règle définit un arc entre deux rôles, et la cardinalité de la relation entre les espaces de travail jouant ces rôles:

- **1 à 1**: Chaque instance du role1 est affecte à une instance du role2 dès le premier transfert entre les deux instances dans cette direction. Après ça, nul transfert ne peut être effectué entre la première instance et une instance du Role2, autre que l'instance affectée initialement.
- **1 à plusieurs**: Toute instance du rôle 1 peut être à l'origine de transferts vers n'importe quel espace de travail du rôle 2.

La liste des entités logiques définit pour quels produits les transferts entre les espaces de travail sont légaux. Si la liste est omisse, les transferts sont légaux pour toutes les entités logiques.

La topologie étoile est définit par une politique tel que la suivante :

```
Politique etoile {
      Roles {
            Database reference ;
            Contributeur *    ;
}
Graph {
      Database <-> * Contributeur   ;
}
}
```

Une politique possible de cycle:

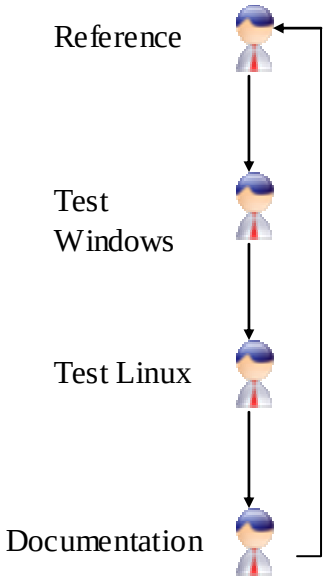


Figure 15 : Politique "loop"

```
Politique loop {
      Roles {
            Reference ;
            Test Windows ;
            Test Linux ;
            Documentation;
}
Graph {
      Reference -> Test Windows;
      Test Windows -> Test Linux;
      Test Linux -> Documentation ;
      Documentation -> Reference ;
}
}
```

2.5. Contrôle de la concurrence

Le contrôle accès définit quels rôles peuvent modifier quelles données, mais il laisse sans contrôle quelles données peuvent être modifiées simultanément et par qui. Pour sa part, le graphe de communication définit uniquement les chemins valides de données qui représentent le travail des participants selon leurs rôles, mais il ne contrôle pas les transferts lorsqu'ils signifient une fusion.

Une première approche pour contrôler la concurrence est de programmer les actions des

utilisateurs avec un langage de programmation traditionnel de style impératif. Cela peut se faire facilement avec un langage de programmation traditionnel. Par exemple, le comportement d'un espace de travail dans une politique du style CVS sans verrouillage peut être décrite ainsi (en pseudo code):

```
loop {
      Transfert from reference to contributor
      Edit
      if (reference_has_not_changed) {
            transfer from contributor to reference
      } else {
            do merge
      }
}
```

Pour un observateur, l'intention du concepteur de cette politique n'est pas tout à fait évidente. Sans une idée explicite de l'intention, il est difficile de raisonner sur la politique, en particulier lorsque les scénarios deviennent plus complexes. Dans le cas de notre exemple de politique CVS, l'intention est de permettre la concurrence parmi les développeurs, mais aussi de contraindre les fusions à ne jamais être faites directement dans la base de données centrale (l'espace de travail de référence).

Nous pouvons généraliser les préoccupations liées à la concurrence dans deux aspects principaux:

- Quels espaces de travail peuvent travailler simultanément
- Qui peut réconcilier des modifications concurrentes

Ainsi, la politique CVS peut être décrite avec une seule règle :

Le travail des développeurs peut être fusionné, mais uniquement dans l'espace de travail d'un développeur. Soit : merge { dev, dev : dev }

Notre syntaxe pour exprimer ce type de règles est la suivante:

```
Merges {
      [Role 1] [,Role 2] : Role 3 [: Liste d'entités logiques]
}
```

Chaque règle doit être interprétée ainsi :

1. Par défaut, toute fusion est interdite (et par conséquence, toute concurrence).
2. Ajouter une règle, permet la fusion des modifications (dans Rôle 3) des modifications provenant de deux types d'espace de travail (Rôle 1 et Rôle 2), permettant ainsi la concurrence entre deux espaces de travail de ce type précis.
3. Si le deuxième rôle (Rôle 2) n'est pas spécifié, un espace de travail jouant le rôle 3 peut faire la fusion des modifications réalisées dans des espaces de travail jouant le Rôle 1, avec des modifications réalisées dans n'importe quel autre espace de travail.
4. Si ni Rôle 1 ni Rôle 2 ne sont spécifiés, un espace de travail jouant le rôle 3 peut faire des fusions de modifications réalisées dans n'importe quel espace de travail.
5. Ces règles sont appliquées à la liste des entités logiques définies à la fin de la règle, ou à toutes les entités logiques si la liste n'est pas spécifiée.

Exemple : CVS pessimiste

La politique CVS dite optimiste (*copy/modify/merge*), peut maintenant être définie juste en déclarant que seul les développeurs peuvent effectuer des fusions. Si on ajoute la déclaration de la topologie (étoile) et le fait que l'espace de travail de synchronisation est une base de données qui n'est jamais modifié directement (read-only), la politique CVS optimiste peut être définie entièrement ainsi :

```
Politique CVS-Optimiste {
      Roles {
            Database REFERENCE: READ-ONLY  Project1, READ-ONLY Project2 ;
            Developpeur * ;
}
Graph {
      Database <-> * Developpeur ;
}
Merges {
      Developpeur ;
}
}
```

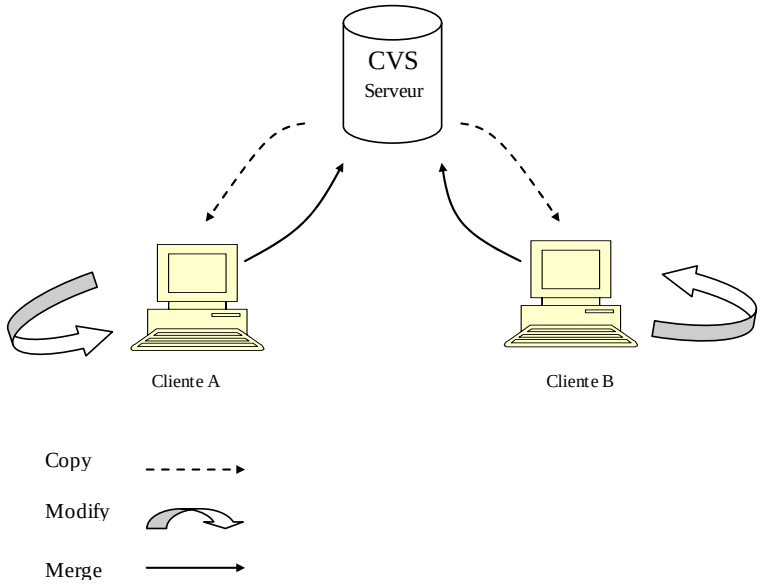


Figure 16 : Copy / Modify / Merge

Dans les systèmes tels que CVS, il est également possible de configurer le serveur pour qu'il applique une stratégie complètement pessimiste : nul fichier peut être modifié simultanément. Pour cela, le système exige le verrouillage d'un fichier avant sa modification. A nouveau, en pseudo code, on décrira cette politique de la façon suivante :

```
loop {
      Transfert from reference to contributor
      lock
      Edit
      transfer from contributor to reference
      unlock
}
```

Tandis que dans notre formalisme, il suffit d'éliminer toute référence aux fusions. C'est, en effet, l'objectif même de cette stratégie : éliminer les fusions!

```
Politique CVS-Pessimiste {
      Roles {
            Database, REFERENCE, READ-ONLY ;
            Developpeur * ;
}
Graph {
      Database <-> * Developpeur    ;
      }
}
```

On constante donc que dans notre formalisme l'intention est plus évidente que dans une stratégie de « programmation de la concurrence ». Il reste donc à définir plus strictement la sémantique du langage et expliquer la façon dont une politique peut être appliquée.

3. Modèle d'exécution du procédé

Dans cette section nous définirons les politiques de façon plus précise à l'aide d'un modèle qui formalise le déroulement d'un procédé. Cette définition est évidemment nécessaire pour qu'une politique puisse être exécutée par un logiciel.

Dans notre modèle, l'exécution d'un procédé est une suite **d'états**. L'état d'un procédé est défini par un ensemble ***d'espaces de travail (ET)*** et, pour chaque entité logique Ei, un ensemble de ***propagations en cours (PC(Ei))***. Les propagations en cours PC(Ei) représentent le travail qui a été effectué sur l'entité logique Ei et qui est entrain d'être communiqué parmi les différents espaces de travail. L'état contient donc un ensemble PC(Ei) pour chaque entité logique Ei, appartenant à l'ensemble des entités logiques contenu dans les espaces de travail du groupe.

Pour simplifier l'exposé, ce chapitre considèrera souvent une entité logique dans le groupe.

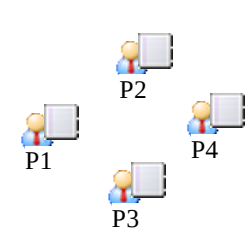
Les propagations en cours pour une entité logique Ei appelé PC(Ei) sont représentées par un ensemble de triplets d'espaces de travail (wsI, wsJ, wsK) dont la signification est que *wsI est à l'origine d'une modification de Ei qui a déjà atteint wsJ, mais qui est encore inconnu de wsK.*

Chaque triplet concerne donc trois espaces de travail

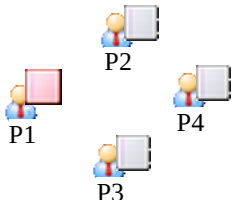
1. Un espace de travail qui a réalisé une modification
2. Un espace de travail qui a déjà reçu cette modification
3. Un espace de travail qui n'a pas encore reçu cette modification

Etat du procédé = Espaces de Travail(ET) + {Propagations en cours(PC(Ei))}

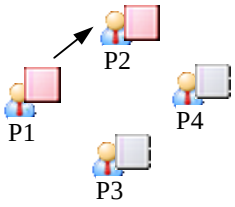
Exemple de succession possible d'états pour une entité logique donnée:



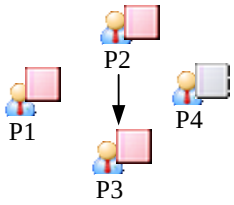
Propagations en cours = {}



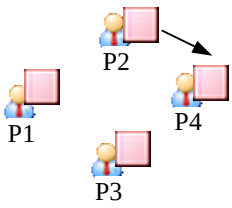
Propagations en cours = {
P1,P1,P2
P1,P1,P3
P1,P1,P4
}



Propagations en cours = {
P1,P1,P3
P1,P1,P4
P1,P2,P4
P1,P2,P3
}



Propagations en cours = {
P1,P1,P4
P1,P2,P4
P1,P3,P4
}



Propagations en cours = {}

Avec cette définition simple du concept d'état de l'exécution, nous allons spécifier toutes les opérations par l'ensemble des changements d'état quelles impliquent.

État initial

Nous supposons que, au début de tout procédé, tous les espaces de travail ont une copie identique du même produit. Pour toutes les entités logiques, les « propagations en cours » sont donc un ensemble vide, car il n'y pas d'unités de travail connue par les uns et pas connue par les autres.

```
Etat Initial =
      ET : {ws1,ws2,ws3…,wsN}
      PC(Ei) : {}
```

C'est également l'état atteint lorsque tous les participants se mettent d'accord sur une version commune. Nous appelons cet état **l'état de consensus**.

Edition

L'édition d'une EL Ej dans un espace de travail est modélisée par la transition depuis un état Ei vers un état Ei+1 ainsi:

Edition de l'EL Ai de l'espace de travail **wsX** :

```
Ei = ETi,PCi
Ei+1 = ETi, PCi+1.
PCi+1 = ∀κ≠j, PCi+1(Ek) = PCi(Ek),
        PCi+1(Ej) = PCi(Ej) ∪ { (wsX,wsX,xsY) : ∀ wsY ∈ ETi ∧ wsY ≠ wsX }
```

Le nouvel état est donc égal à l'état antérieur, sauf qu'une nouvelle modification réalisée par **wsX** existe, mais est inconnue de tous les autres espaces de travail.

Transferts

De façon informelle, un transfert est l'action qui met un espace de travail à jour par rapport à un autre espace de travail. Autrement dit, après un transfert, l'espace de travail destination « connaît » le travail hébergé par l'espace de travail source. L'état du procédé *après un transfert* doit donc respecter les deux conditions suivantes:

- *wsSource ne connaît plus rien qui ne soit également connu de wsDestination* :
  - *pre conditions* : (wsX,wsSource,wsDestination) ∈ PC
  - *post condition : ;*
- *si wsSource connaît une modification inconnue par un espace de travail wsY,après le transfert, wsDestination la connaît aussi*:
  - *pre condition :* (wsX,wsSource,wsY) ∈ PC ,
  - *post condition :* (wsX,wsSource,wsY) v (wsX,wsDestination,wsY) ∈ PC .

Le transfert peut être défini ainsi :

```
Ei = ETi,PCi
            // … transfer wsS -> wsD
Ei+1 = Eti, PCi+1
PCi+1 = (∀κ≠j, PCi+1(Ek) = PCi(Ek),
         PCi+1(Ek) = PCi(Ek) - {(wsI,wsS,wsD) : wsI ∈ Eti}
         ∪{(wsX,wsD,wsY): ∃ (wsX,wsS,wsY) : (wsX,wsS,wsY) ∈ PCi(Ek)}
```

Fusions

Une question se pose lorsque le transfert de données entraîne une fusion : la fusion est elle

aussi une « unité de travail » en soi? Une position consiste à assimiler la fusion comme étant uniquement le choix entre deux options concurrentes, et ainsi à considérer qu'elle ne constitue pas du travail original en soi. Cependant, il peut être argumenté qu'une fusion peut nécessiter également une modification active du logiciel, et que cette modification est du travail « indépendant ». Cette question reste un sujet de débat.

Nous adoptons pour notre travail la première position: nous ne considérons que la fusion se limite uniquement à réconcilier deux modifications en compétition et qu'elle ne doit pas ajouter une fonctionnalité indépendante.

Fusion

Le concept de « fusion » peut être maintenant défini précisément avec le concept d'état.

De façon informelle, dans un transfert depuis wsS vers wsD, le transfert **n'entraîne pas de fusion** si wsS connaît *tout le travail qui est connu* par wsD. Dans ce cas, la copie de wsS remplace simplement celle de wsD. Si par contre wsD connaît du travail pas encore connu par wsS ce travail doit être réconcilié avec le travail qui va être transféré. Dans ce cas, nous dirons qu'une **fusion** (manuelle ou pas) est nécessaire. Cette condition peut être représentée par l'expression suivante :

$$\exists\ wsX : (wsX, wsS, wsD) \in PC \wedge \exists\ wsY : (wsY, wsD, wsS) \in PC \wedge wsX \neq wsY$$

Création de nouveaux espaces de travail

Lorsque de nouveaux espaces de travail entrent dans un projet, ils commencent par se synchroniser avec un des espaces de travail avec lesquels ils peuvent légalement communiquer. Le nouvel état inclue le nouvel espace de travail, ainsi que de nouvelles propagations en cours qui correspondent exactement aux propagations en cours de l'état de travail de l'espace de travail avec lequel ils se synchronisent.

$$Ei = Eti, Pci$$

… Intégration d'un nouvel espace de travail wsX qui se synchronise avec l'espace de travail wsY

$$Ei+1 = Ei+1, PCi+1$$

$$Ei+1 = ETi \cup \{wsX\},$$

$$PCi+1(Ek) = PCi(Ek) \cup \{(wsY,wsX,wsK\} : (wsY,wsY,wsK) \in Pci\ (Ek)\}$$
$$\cup \{(wsI,wsX,wsK\} : (wsI,wsY,wsK) \in Pci\ (Ek)\}$$
$$\cup \{(wsI,wsK,wsX\} : (wsI,wsK,wsY) \in Pci(Ek)\}$$

Destruction d'un espace de travail

Un espace de travail peut être « détruit » lorsque, par exemple, son responsable abandonne définitivement le projet de développement. Naturellement, la destruction d'un espace de travail signifie son retrait de la liste des espaces de travail, ainsi que de tous les triplets dont lesquels il fait partie, sauf les triplets qui représentent du travail dont l'espace de travail a été à l'origine, car les modifications ne sont pas détruites, évidemment.

```
Ei = Eti,Pci
      … Destruction d'un espace de travail wsX
Ei+1 = Ei+1, PCi+1
      Ei+1 = ETi - {wsX},
      PCi+1(Ek) = PCi(Ek) – ({(wsI,wsX,wsJ) : (wsI,wsX,wsJ)∈ PCi(Ek)}
                              ∪{(wsI,wsJ,wsX) : (wsI,wsJ,wsX)∈ PCi(Ek)})
```

4. Application de une politique d'ingénierie concurrente

Dans cette section nous expliquerons comment une politique d'ingénierie concurrente est appliquée par notre système.

4.1. Introduction

Le rôle principal de notre système est d'appliquer une politique d'ingénierie concurrente, c'est à dire, de garantir que les actions des utilisateurs sont en accord avec les contraintes exprimées par la politique. Ceci est réalisé en contrôlant les actions des utilisateurs: le système ne doit permettre une modification ou un transfert des données que s'ils sont légaux d'après la politique et l'état courrant du système.

L'application d'une politique correspond à la procédure suivante :

1) L'utilisateur interagit avec son espace de travail en demandant au gestionnaire de l'espace de travail (un outil qui s'exécute sur son poste de travail) d'exécuter une opération (e.g. un transfert ou une modification).
2) Le gestionnaire de l'espace de travail (qui « connaît » la politique) analyse si

l'opération peut être réalisée, où s'il est nécessaire de demander l'autorisation du gestionnaire de politiques globale (une application qui peut être contactée sur le réseau).

3) Si la demande d'autorisation est nécessaire, le gestionnaire de l'espace de travail envoie la demande, et exécute l'opération si le gestionnaire de politiques donne un « avis » favorable (la demande et l'exécution de l'opération doivent être réalisés de façon atomique).

4) A la fin de l'opération, le gestionnaire de l'espace de travail informe le gestionnaire de politiques de la fin de l'opération, et ce dernier mets à jour l'état du procédé.

4.2. Validation d'une politique : les transferts

Dans l'application d'une politique, il y un premier niveau qui correspond tout simplement à la vérification directe du respect des règles de la politique. Ces règles ne définissent que des contraintes sur l'opération du transfert et doivent être vérifiées avant chaque transfert.

Validation du graphe de communication

La validation du transfert par rapport au graphe de communication est trivial: il suffit de vérifier que pour les entité logiques transmises, le transfert est valide entre les deux rôles concernés. Cette validation peut être faite directement par le gestionnaire de l'espace de travail, localement, car elle ne dépend que du graphe de communication et de l'état local des entités logiques.

Validation des règles de fusion

Pour valider qu'un transfert est légal par rapport aux règles du fusion, il est nécessaire de déterminer d'abord s'il va entraîner une fusion, selon la définition de la section 3, et si c'est le cas, la fusion doit être valide par rapport aux règles:

- quels sont les rôles des espaces de travail à l'origine des modifications à fusionner ?
- est-ce que l'espace de travail, destination du transfert, est habilité à réaliser la fusion?

On peut déterminer l'origine des modifications en conflit grâce aux propagations en cours. Pour cela, il suffit de prendre les triplets qui représentent le travail connu par l'espace de travail Destination et inconnues par l'espace de travail Source (wsX,wsDestination,wsSource) et vice-versa (wsY ,wsSource,wsDestination) et considérer les espaces de travail dont les modifications sont originaires, de chaque cote (WsX et WsY). Le transfert sera légal si et

seulement si il existe une règle : « merge {wsX, wsY : wsDestination} »

Supposons un scénario où existent trois rôles: développeur senior, développeur junior et un intégrateur ; et une instance de chacun de ces rôles désignés ici par le prénom de leur propriétaires : Christophe, Xavier, Philippe . Le scénario, pour l'entité logique « BankAccount », est le suivant :

1. Christophe, un « développeur senior », modifie la classe BankAccount, considéré comme une entité logique d'un logiciel bancaire. L'état du procédé est mis à jour avec des triplets qui représentent le fait qu'il est à l'origine de cette modification, inconnue encore de tous les autres. {(Christophe, Christophe, Xavier), (Christophe, Christophe, Philippe)}
2. Christophe envoie ses modifications à l'intégrateur, Philippe. L'état du procédé est mis à jour en éliminant le triplet (Christophe, Christophe, Philippe) et ajoutant le triplet (Christophe, Philippe, Xavier).
3. Xavier, un « développeur junior », modifie la même classe. Il essaie de se synchroniser avec Philippe. A ce moment là, le gestionnaire de politiques détecte la présence d'une fusion représenté par les triplets (Christophe, Philippe, Xavier) et (Xavier, Xavier, Philippe). La fusion concerne le travail réalisé dans un espace de travail « développeur senior » (Christophe) et le travail réalisé dans un espace de travail « Développeur junior » (Xavier) sur la classe BankAccount.

A ce moment là, le gestionnaire cherche s'il existe une règle permettant la réconciliation du travail provenant de ces deux types d'espace de travail dans l'espace de travail « developpeur junior » ; dans cet exemple cette règle doit être :

```
developpeur_senior, developpeur junior : developpeur_junior
```

4.3. Validation d'une politique : La concurrence

Nos règles expriment directement les conditions dans lesquelles les utilisateurs veulent que les fusions soient faites (ou pas faites du tout). Ceci correspond à notre conception selon laquelle la fusion est la préoccupation centrale dans l'ingénierie concurrente. La concurrence elle même (« qui peut modifier quoi simultanément avec qui ») n'est pas directement exprimée dans nos politiques ; ce n'est même pas un concept du formalisme.

Nous appellerons **blocage**, les cas où tous les transferts depuis un espace de travail donné implique une fusion interdite. Il s'agit d'un cas grave car le travail accumulé dans l'espace de

travail bloqué ne pourra jamais parvenir à l'espace de travail de référence et ne contribuera jamais à la version commune ; l'espace de travail bloqué est « mort » du point de vue du système et le travail qu'il a effectué est perdu. C'est donc une situation qu'il faut éviter à tout prix.

Or, avec l'application directe des règles de fusion, les risques d'une fusion sont sous contrôle, mais les situations de blocages ne sont pas évitées. Si la concurrence elle même n'est pas contrôlée par le système, certaines modifications vont mettre le système dans un état qui, inéluctablement va aboutir à un blocage, c'est-à-dire lorsque qu'il existe deux modifications concurrentes pour lesquelles aucune fusion n'est autorisé. Les éditions doivent donc être contrôlées pour éviter de tomber dans le cas des modifications simultanées impossibles à réconcilier.

Les **« modifications simultanées »**

Nous dirons que les espaces de travail wsX et wsY ont réalisés une modification simultanée d'une entité logique Ei, s'il existe simultanément pour Ei : (wsX,wsX,wsY) et (wsY,wsY,wsX), c'est a dire lorsque au même moment wsX a réalisé une modification encore inconnue par wsY, et vice-versa : wsY à réalisé une modification encore inconnue par wsX.

Étant donné notre hypothèse que toutes les modifications contribuent à un résultat unique commun, toute modification simultanée implique, tôt ou tard, une fusion. On appellera **état bloquant** un état tel qu'il est impossible de réconcilier une modification simultanée.

S'il n'existe aucune règle de fusion « merge {wsX, wsY : wsZ} » on peut en déduire que la modification simultané (wsX, wsY) aboutira à coup sur à un état bloquant. Mais même s'il existe des règles de fusion « merge {wsX, wsY : wsZ} », étant donné les transferts effectués, on peut se trouver dans un état bloquant.

Un état est dit bloquant s'il existe pour Ei une modification simultanée wsX et wsY et qu'il n'existe pas wsK tel que

- Il existe une règle « merge (wsX, wsY : wsK) »
- Il existe un chemin Cx entre wsX et wsK
- Il existe un chemin Cy entre wsY et wsK
- Cx et Cy ne partagent que wsK.

En d'autres termes, il n'existe pas d'espace de travail (wsK) qui peut fusionner la modification concurrente.

Nous étendons la définition d'un état bloquant: un état bloquant est celui depuis lequel il est impossible de faire parvenir les deux modifications à la référence. Ceci se produit lorsque une modification simultanée doit nécessairement être réconciliée dans un espace qui n'y est pas autorisé.

Toute modification ou tout transfert qui mène le procédé dans un état bloquant doit être interdite. Le système doit donc interdire tout changement d'état qui emmène le système dans un état bloquant, c'est-à-dire que le système doit soit :

- Empêcher l'édition locale de certaines entités logiques, soit
- Empêcher un transfert.

En d'autres termes, le système garanti que toute modification concurrente pourra être convenablement fusionnée par les espaces de travail habilités. Plus important, le système garanti que ne seront interdites que les modifications concurrentes pour lesquelles **aucune solution** n'est possible. Le système garanti donc la concurrence maximale, compte tenu des fusions autorisées.

L'algorithme utilisé est tel que le système peut, à tout moment, indiquer quelles sont les opérations permises, et lorsqu'une opération est interdite, indiquer la raison de l'interdiction et les opérations qui éviterons le blocage. Il est important de faire remarquer que ces situations de blocage peuvent être détectées très en amont, bien avant le blocage effectif, et l'expérience montre que, en dehors des graphes triviaux, il est impossible de détecter mentalement un état bloquant.

Prolifération de messages

Avec notre architecture, chaque action dans un espace de travail (édition ou transfert) implique une communication entre l'espace de travail et le gestionnaire de politiques. Cela est problématique pour deux raisons:

- Chaque espace de travail nécessite une connexion au réseaux chaque fois qu'une action y est exécutée.
- Le nombre de messages sur le réseau peut s'avérer trop important, pour de grands nombres d'espaces de travail.

Cependant, selon la politique, certaines vérifications peuvent être complètement superflues. Par exemple, dans le cas extrême d'une politique sans contrainte sur les fusions, aucune vérification sur les modifications ne serait nécessaire, ou les cas de modifications

concurrentes statiquement interdites.

Le système doit donc limiter les échanges entre les espaces de travail et la gestionnaire de politiques à ceux qui sont strictement nécessaires.

Chapitre V. Augmentation de l’information contextuelle

L'un des aspects les plus difficiles à contrôler pendent le développement d'un logiciel est la concurrence, c'est à dire la modification simultanée des données qui constituent un logiciel. Cet aspect est de grande importance car la concurrence est nécessaire pour diminuer les temps de développement, mais elle est également une source de risques. Ces risques proviennent du fait que la sérialisation, le seul critère connu pour vérifier la cohérence lorsque sont effectués deux modifications concurrentes, n'est pas utilisable pour le développement des systèmes logiciels.

La sérialisation n’est pas adéquate à cause des fortes liaisons entre les plusieurs parties qui constituent un logiciel ainsi que par la durée des activités de développement. Dans l'ingénierie concurrente la sérialisation est remplacée par la fusion, entendue comme la réconciliation (manuelle ou automatique) des modifications concurrentes. Cette opération peut entre assisté par des logiciels spécialisés, mais dans le cas général elle nécessite un humain qui se charge de réconcilier les deux modifications en vérifiant la cohérence du résultat. Malheureusement, la réconciliation de deux modifications est souvent difficile, et constitue un risque pour la cohérence du système. Dans ce chapitre nous présenterons notre approche à la stratégie dite d’augmentation de l’information contextuelle pour rendre l’ingénierie concurrente « sûre ».

1. Introduction

Dans le chapitre précédent nous avons présenté un langage pour la définition des politiques d’ingénierie concurrente. Ces politiques permettent de contrôler plusieurs aspects des procédés de développement coopératif, et en particulier elles permettent de contrôler la concurrence. Sur ce point, chaque politique d’ingénierie concurrente exprime un compromis entre 1) la concurrence totale (rapide, mais risquée) et 2) le respect de certaines pratiques qui réduisent la concurrence, mais qui donnent une meilleure sécurité pour les données. Ces pratiques qui augmentent la sécurité peuvent être classifiées en trois grandes familles:

- L'interdiction de la concurrence. Cette interdiction peut entre imposée sélectivement selon les données à protéger et les rôles des participants.

- L'affectation de la fusion à la personne la plus adapté à la réaliser. Ceci est contraignant, mais réduit le risque d’une mauvaise fusion due à un manque d’expertise.
- La synchronisation fréquente des espaces de travail. Cette pratique rend les conflits plus faciles à résoudre et moins enclins aux erreurs. Cependant, elle est en conflit avec le principe d'isolement car pour se synchroniser fréquemment, les utilisateurs peuvent se voir obligés de communiquer des états non cohérents du système.

Dans le chapitre précédent nous avons expliqué comment un mélange de ces pratiques peut être exprimé par une politique, pour ensuite être appliquée par l'environnement de génie logiciel. Le bénéfice de cette démarche est que les utilisateurs sont libérés de la responsabilité de veiller à l'application de la politique, rendant celle-la moins vulnérable à des erreurs d'origine humaine.

Malheureusement, même lorsqu'une politique est souple et qu’elle est adéquate la plupart des temps, elle peut se révéler trop stricte lorsque des situations non anticipées surgissent. Par exemple, pour un certain projet il peut être adéquat de toujours exiger que les fusions soient faites par une personne spécialisée dans cette tache, même s'il faut retarder par fois les intégrations. Si malheureusement les circonstances changent est que la résolution d'un « bug » doit être intégrée absolument a la version officielle sans aucun retard, et qu’elle demande une expertise spéciale, il serait nécessaire que la personne jugée la plus adapté parmi celles qui sont présentes prenne en charge cette responsabilité.

Ainsi, il parait judicieux de laisser parfois l'application des pratiques de « sécurité » à la responsabilité des participants, qui sont en effet ce qui connaissent le mieux les risques [PSV 98], sans imposer leur application stricte.

Malheureusement, sans une visibilité sur les activités des autres, les participants ne peuvent pas savoir quelles sont les modifications en cours, qui les ont réalisés, dans quelle intention, etc. En conséquence ils ne peuvent ni éviter une modification concurrente dangereuse, ni connaître l'existence d'un conflit pour avancer la fusion, ni éviter une fusion dans un espace de travail inapproprié. Le manque de visibilité est surtout significatif lorsque les participants travaillent de façon distribué dans plusieurs endroits géographiques, car ils ne bénéficient pas de la communication permanente qui peut exister dans un endroit commun.

Le rôle d'un système d’augmentation de l'information contextuelle (SAIC) est de réduire se manque de visibilité sur les activités des autres est ainsi permettre aux participants de rendre

l'ingénierie concurrente « sûre ».

Pour illustrer comment un SAIC peut aider à rendre l'activité du groupe plus sûre, nous pouvons imaginer les situations suivantes:

1. Un développeur se prépare à ajouter une nouvelle fonctionnalité dans un système logiciel, ce qui implique la modification des fichiers X, Y et Z. Il est informé par le SAIC qu'un deuxième développeur a commencé à modifier le fichier X. Il décide alors de commencer son travail par les fichiers Y et Z, en attendant la nouvelle version du fichier X, pour ainsi réduire la possibilité des conflits.
2. Le responsable d'un des composants d'un système logiciel a la responsabilité d'intégrer toutes les modifications effectuées sur le composant. Il est informé de l'existence de deux modifications concurrentes qui sont en train de se réaliser. En regardant de près les deux modifications, à l'aide du SAIC, il trouve que les deux modifications sont compatibles. Il permet en conséquence la poursuite des deux activités.
3. Un développeur est notifié que quelqu'un d'autre est en train de modifier un fichier sur lequel il se disposait à travailler. En regardant la modification de près, il trouve qu'elle est potentiellement incompatible avec son propre travail. Le SAIC lui permet de savoir qu'une fois le travail du deuxième développeur accompli; il pourra se synchroniser immédiatement. Il décide alors d'attendre la fin de la modification, pour se synchroniser avant de commencer à travailler.
4. Dans la même situation que l'exemple antérieur, le développeur sait grâce au SAIC que le travail doit encore parcourir une longue phase de révisions, avant qu'il puisse le récupérer. Il décide alors de travailler parallèlement, pour ne pas rester bloqué.
5. Un développeur a récemment effectué des modifications importantes qui modifient l'architecture du module sur lequel il travaille, et il a envoyé son travail au manager. Il se dispose à continuer son travail de re-structuration, mais il est averti par le SAIC du fait que plusieurs développeurs expérimentés n'ont pas encore intégré ses modifications. Il préfère attendre que tout le monde ait récupéré ses modifications avant de continuer le travail, pour avoir tous les commentaires possibles et éviter une perte du travail si des problèmes sur la nouvelle architecture sont détectés.

L'implémentation d'un SAIC présente de nombreux problèmes de conception qui sont liées surtout à l'optimisation de l'information présentée à chaque utilisateur. L'optimisation de l'information présentée est fondamentale, car l'effort dépensé dans l'appréhension de

l'information peut devenir trop élevé par rapport aux bénéfices qu'il rapporte. [SBC 00] explique ce problème ainsi:

« Du fait que la concentration humaine est une ressource limité qui doit être optimisé, l'information contextuelle (« awareness ») doit être synthétisée dans une forme utile, et livrée uniquement aux utilisateurs qui en ont besoin. Si les utilisateurs reçoivent trop peu d'information ou une information inapproprié, ils agiront de façon inappropriée et peu efficace. S'ils reçoivent trop d'information, la surcharge d'information augmentera leur travail et cachera l'information importante.

2. L'approche « naïve »: la notification de tous les événements

Une première approche possible, utilisée par exemple par des systèmes tel que Palantir [SNV 03], consiste tout simplement à notifier les utilisateurs chaque fois qu'une nouvelle action à eu lieu dans un espace de travail. Dans le système Palantir, l'utilisateur reçoit une information sur les actions des autres utilisateurs dans le style suivant:

[...]

[1] Check-Out – Pierre - Collector.cpp

[2] Edit – Me – Interface.h

[3] Edit – James – Collector.cpp

[4] Edit – Pierre – Interface.h

[5] Check-In – Chloe

[6] Update - James

[...]

A première vue, cette information ne permet pas aux utilisateurs d'identifier facilement quelles sont les conséquences de chacune de ces actions sur leur propre travail. Dans cet exemple précis, nous savons qu'il existe un procédé implicite, qui provient du couplage entre Palantir et le système CVS. Ce procédé, connu comme « copier, modifier, fusionner », correspond pour nous à une étoile simple, sans règles de concurrence (concurrence non contrôlé) ; la topologie étoile est imposée par le système. C'est également le « modèle » de procédé utilise par [MOS 02].

Connaissant cette information, nous savons que l'édition par James du fichier Collector.cpp (ligne 6) peut être dangereuse selon que quelqu'un d'autre l'a édité, sans avoir fait un « check-in » avant le dernier « check-out » de James. Notons que cette condition sur la « dangerosité » de l'action de James n'est pas évidente : elle dépend de l'information intuitive que nous possédons sur le procédé.

Dans cet exemple, la vérification de la situation conflictuelle implique une recherche dans la transcription des actions passées et une connaissance de la topologie. Ce travail de recherche est entièrement laissé à la responsabilité de l'utilisateur. Etant donné que le nombre des actions dépend du nombre de participants, plus le nombre d'utilisateurs est important et plus l'interprétation de l'information devient coûteuse pour les utilisateurs.

Ceci n'est pas un danger théorique: dans le travail que nous avons réalisé pour la société Dassault Systèmes lors du développement du gestionnaire de configuration ADELE, il a pu être mesuré [Est 01] qu'à tout moment il y a en moyenne 1000 espaces de travail actifs, chacun avec 2000 fichiers en moyenne. En moyenne toujours, une dizaine de fichiers est modifié quotidiennement dans chaque espace de travail. Dans ce contexte, le système génère (1000 * 10) = 10 000 événements, envoyés aux 1000 espaces de travails (soit 10 000 000 de messages). Le nombre de messages, et la faible information qu'ils renferment, font qu'il est virtuellement impossible de les interpréter. Un phénomène qui est qualifié de « surcharge cognitive » dans [SNV 03].

L'approche « naïve » se révèle inutilisable en dehors de petits projets menés par un nombre très restreint de participants. L'information doit être traitée avant d'être présentée aux utilisateurs. Dans Palantir, le seul moyen qui existe pour traiter cette information dans le but de réduire le coût d'interprétation est un système de filtrage que nous pourrions qualifier de 'syntaxique' car il ne dépend que de la forme des messages distribués par le système. Par exemple, les utilisateurs peuvent déclarer ne vouloir être informés que des événements qui commencent par le mot « check-in ».

Il faut noter que cette approche est la seule approche possible lorsque la notion de procédé ne fait pas partie de la conception du SAIC, car sans elle il n'existe pas de critère pour traiter l'action en tenant compte de sa sémantique. Notre hypothèse est qu'un système d'augmentation de l'information contextuelle ne doit pas se limiter à indiquer qui fait quoi, mais qu'elle doit tenir compte du procédé qui guide les actions des utilisateurs pour présenter l'information d'une façon qui soit vraiment utile.

3. Objectifs

L'objectif fondamental d'un SAIC est de donner aux utilisateurs une information sur le travail du groupe qui soit pertinente pour l'exécution de ses propres activités [EG 05][SNV 03]. Les points suivants sont déterminants pour l'accomplissement de cet objectif:

- **Passage à l'échelle**: La quantité d'information présentée doit rester dans des limites gérables par l'utilisateur, et cela indépendamment du nombre d'espaces de travail qui participent.
- **Pertinence**: La compréhension de l'information n'est pas sans coût. Ce coût est l'effort réalisé par l'utilisateur pour interpréter l'information de façon à trouver quelles sont les implications pour ses propres activités. Etant donné que le coût de compréhension de l'information dépend de la quantité d'information présentée, il est nécessaire de présenter uniquement l'information la plus pertinente à chaque utilisateur, de façon à maximiser les bénéfices apportés par l'information par rapport aux coûts de son interprétation.
- **Opportunité**: Les bénéfices de l'augmentation de l'information contextuelle sont plus significatifs lorsque l'information arrive aux utilisateurs au moment le plus opportun, car les risques peuvent être contrôlés avant qu'ils ne deviennent trop importants. Le système doit donc faire en sorte que l'information importante arrive à l'utilisateur le plus tôt possible.
- **Non intrusif**: L'information doit être transmisse le plus tôt possible aux utilisateurs, sans attendre une requête explicite de sa part, mais cette information ne doit pas être une source d'interférences négative pour l'utilisateur. Par exemple, le SAIC doit collecter l'information sur les activités sans requérir un effort additionnel de la part des utilisateurs. Dans tous les cas, le SAIC ne doit pas violer le principe d'isolation des espaces de travail.

4. Groupes et passage à l'échelle

Une question qui apparaît lors de la conception d'un SAIC est comment intégrer un nombre arbitraire de participants sans surcharger les utilisateurs avec des quantités trop élevés d'information.

Pour cela, nous avons décidé d'utiliser le concept de groupe comme étant le contexte dans lequel les utilisateurs seront informés [EG 05]. Cela veut dire que les utilisateurs ne seront informés que des activités qui ont lieu dans le groupe auquel ils appartiennent. Ainsi,

l'information contextuelle dépendra uniquement du nombre d'espaces de travail dans le groupe, et pas du nombre d'espaces de travail total. Étant donnée que de nouveaux groupes peuvent être ajoutés hiérarchiquement sans limite, le groupe comme contexte de l'augmentation de l'information contextuelle permet d'intégrer un nombre arbitraire d'espaces de travail sans surcharger l'information contextuelle de chaque groupe.

Le choix du groupe comme étant le contexte dans lequel l'information est collectée et transmise aux utilisateurs n'est pas arbitraire. La raison fondamentale est que la division en groupes et sous-groupes corresponds très souvent à une division du travail en taches et en sous taches, et/ou à des expertises partagées par les membres. Ceci a plusieurs conséquences importantes pour le SAIC :

1. Les modifications dans les espaces de travail d'un groupe sont souvent liées sémantiquement, et donc
2. les chances d'avoir plusieurs participants modifiant les mêmes données sont élevées, et
3. Ces modifications peuvent être analyses sur la lumière de la mission du groupe, qui peut être utilisé pour comprendre la logique et l'intention de la modification.

Dans notre travail avec Dassault Système, les 1000 espaces de travail sont structurés en groupes selon une arborescence à 7 niveaux de profondeur. Ainsi, le nombre de participants dans un groupe donné est le plus souvent inférieur à 10.

5. Pertinence

Une façon de présenter l'information sur le procédé à l'utilisateur est simplement de donner une vision sur l'état du procédé dans le groupe, selon notre définition d'état du chapitre 4. L'utilisateur aura ainsi une vue permanente sur une liste des propagations en cours.

Avec une visibilité sur l'état du procédé, il est possible d'identifier le travail en cours, et surtout les modifications concurrentes entre deux espaces de travail. Cette représentation de l'activité du groupe est un progrès important sur la stratégie consistant à montrer la totalité des actions exécutées car :

- L'information qui doit être interprétée pour vérifier s'il existe un conflit potentiel sur un fichier est limitée par le nombre d'espaces de travail ((N * (N^2 – N)) / 2). En comparaison, dans une transcription complète des actions des participants, il n'y pas de limitation sur l'information qui doit être interprétée.

- L’état contient toute l’information nécessaire pour vérifier l’existence d’une future fusion, et donc le fait de ne pas connaître la totalité les détails sur le procédé n’est pas une limitation, comme ce serait le cas dans une transcription totale des actions.

Cependant, sans un mécanisme pour réduire encore l’information contextuelle à l’intérieur du groupe, les utilisateurs peuvent très difficilement mesurer l’ampleur des conséquences que l'état général du procédé peut avoir sur leur propre travail. Elle est également incomplète: elle ne permet pas d'anticiper la proximité des fusions et ou ils vont se produire, ce qui réduit l'efficacité de l'information.

Pour réduire l'information en gardant la partie la plus riche, il est nécessaire de trouver des mécanismes pour la filtrer les « propagations en cours ». Pour cela, nous allons nous baser sur le modèle de procédé existant, comme nous l'expliquerons dans les sections suivantes.

5.1. Filtrage de l'information contextuelle

Comme les exemples que nous avons mentionnés dans l’introduction de ce chapitre l'illustrent, l'intérêt d’une information pour un participant particulier varie selon les taches qu'il réalise ainsi que sur sa responsabilité sur la fusion des modifications concurrentes. Ceci implique qu'il est possible de spécialiser l'information qui est présentée à chaque espace de travail selon leurs fonctions, en optimisant ainsi les mécanismes de filtrage. Pour [SBC 00], l'information contextuelle doit être « pertinente au rôle et à la situation de chaque participant spécifique ». Dans notre système, nous sélectionnerons les propagations en cours à présenter à un utilisateur selon son *rôle*, car le rôle représente les responsabilités des participants ainsi que son habilitation pour réaliser les fusions.

La spécialisation des filtres selon le rôle des participantes constitue une première stratégie pour éliminer une partie de l'information superflue sur l’état du procédé. Il est encore nécessaire d'avoir un critère pour filtrer les propagations en cours *pour chaque rôle particulier* selon leur pertinence, pour ainsi pouvoir présenter uniquement la partie la plus utile. Pour cela, nous expliquerons trois critères déterminants qui aident à juger la pertinence d'une propagation en cours pour un utilisateur particulier:

- Les rôles des espaces de travail à l'origine et en attente des modifications. L'hypothèse ici est qu'un participant est naturellement plus intéressé par certains modifications que par d'autres, selon l'origine de la modification et sa destination. Par exemple, le travail qui va arriver sous peu dans son espace de travail est plus intéressant pour un participant que celui qui ne va pas lui être transféré. Egalement, il est raisonnable de

penser qu'une modification ayant été intégrée à l'espace de travail de référence est plus importante qu'une modification dans un espace de travail quelconque.

- Le fait qu'une modification qu'il va réaliser soit ou non en concurrence avec une autre modification.
- L'imminence de l'arrivée d'une modification dans un espace de travail. Cette imminence dépend du concept de distance que nous expliquerons dans la section suivante.

5.2. Distances

Lors d'un transfert d'un espace de travail à un autre, le travail qui est transmis du premier espace au second a de fortes chances d'être modifié dans l'espace de travail destination, soit lors de la réconciliation, soit lors des modifications ultérieures effectuées dans l'espace destination.

Nous appellerons **distance** entre deux espaces le nombre de transfert minimal pour qu'une modification se trouvant dans le premier atteigne le second.

Ainsi deux espaces adjacents sont à une distance 1 ; dans une étoile, deux espaces autres que le centre sont à une distance 2, etc.

Lorsque la distance entre deux espaces est importante, les modifications présentes dans l'espace d'origine ont de grandes chances d'arriver significativement changées dans l'espace destination. De ce fait, une analyse trop anticipée d'une divergence entre deux espaces de travail risque d'être largement inutilisable lorsque la fusion est finalement réalisée.

Cette observation est à la base de notre critère pour mesurer la pertinence d'une propagation en cours:

Plus courte est la distance que doivent parcourir deux modifications pour être fusionnées, plus urgente est l'analyse des divergences entre ces modifications.

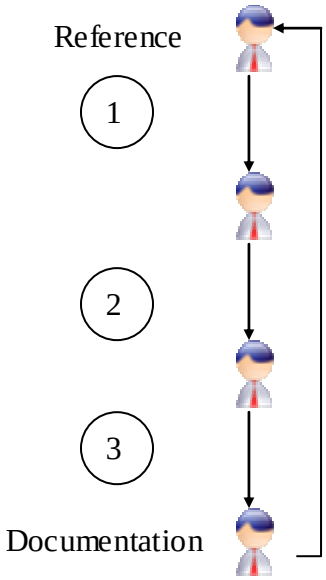


Figure 17 : Distance 3 entre la référence et la documentation

Une distance courte signifie que l'analyse de la divergence sera probablement encore valable au moment de la fusion, elle indique aussi l'imminence de la fusion, et donc l'urgence et l'intérêt d'effectuer l'analyse des divergences.

Un autre critère permettant de déterminer la pertinence de l'analyse de divergence est la connaissance de celui qui va être en charge de la fusion correspondante. Bien entendu, celui qui aura à effectuer la fusion est celui pour qui cette information a le plus d'intérêt.

L'information concernant la distance, et qui doit réaliser la fusion sont des informations qui peuvent être directement extraites du graphe de communication et des règles de concurrence. Grâce à ces informations, le système peut calculer, pour chaque participant, quelle est l'information pertinente.

Le concept de distance en l'absence de procédé

En l'absence de procédé (c'est-à-dire un procédé chaotique, sans topologie particulière) la distance entre deux modifications concurrentes est toujours 1, car il suffit d'un seul transfert pour les fusionner (la distance réelle est indéterminée). Dans ce scénario, toutes les modifications ont la même pertinence et la même urgence ; le développeur doit être informé de toutes les propagations en cours, ce qui conduit à la surcharge cognitive. Il y a trop d'information à considérer, sans aucune piste permettant de savoir quelle information est plus pertinente qu'une autre. L'augmentation de l'information contextuelle devient de fait inutilisable.

Calcul de la distance entre un espace de travail et une modification

La distance d'une modification donnée est calculée par rapport à l'espace de travail le plus proche contenant cette modification, et non pas par rapport à l'espace qui est à l'origine de la modification. Ceci nécessite de connaître l'état de système.

6. Exemple : Topologie en étoile simple

Chez STMicroelectronics, nous avons mis en production un environnement d'ingénierie concurrente qui comporte un SAIC comme une de ses fonctionnalités. La topologie en place dans l'équipe qui utilise CELINE est une étoile avec un espace de travail central passif (pas de modification ni de fusions). Le SAIC est paramétré pour associer les propagations en cours à des noms et symboles en utilisant la notion de distance, de la façon suivante.

- **Modifié**: L'utilisateur a réalisé une modification qui n'a pas encore été intégrée dans la copie de référence. La modification est donc à une distance 1 du dépôt central.
- **Modifié ailleurs**: Quelqu'un d'autre est entrain de modifier sa copie de travail. La modification n'a pas encore été publiée est à distance 2 de l'espace de travail du participant.
- **Obsolète**: L'espace de travail doit être mis à jour car la référence contient au moins une nouvelle modification. Il y a donc une modification à distance 1 depuis l'espace de travail de référence et l'espace de travail concerné.
- **Conflit**: Il existe deux modifications concurrentes qui nécessitent uniquement un transfert pour être fusionnées dans l'espace de travail du participant.

Notre système met à disposition de chaque utilisateur une interface d'utilisateur graphique où sont représentés :

La liste d'utilisateurs

Les copies de travail dans l'espace de travail de l'utilisateur (des hiérarchies de fichiers).

La liste des fichiers dont des propagations en cours sont d'intérêt pour l'utilisateur

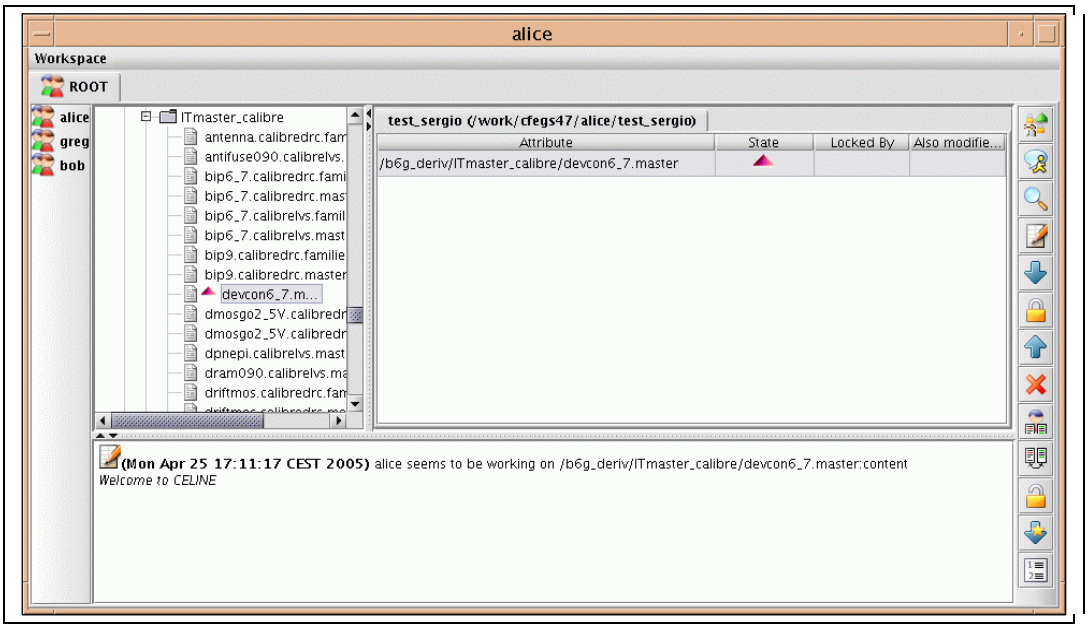
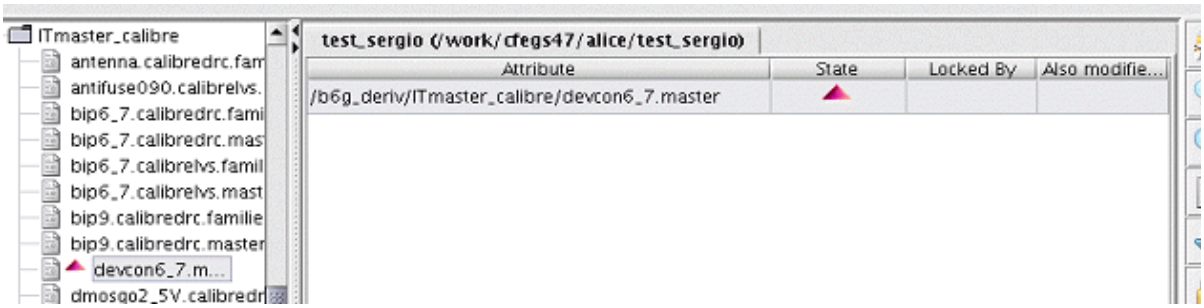


Figure 18 : CELINE

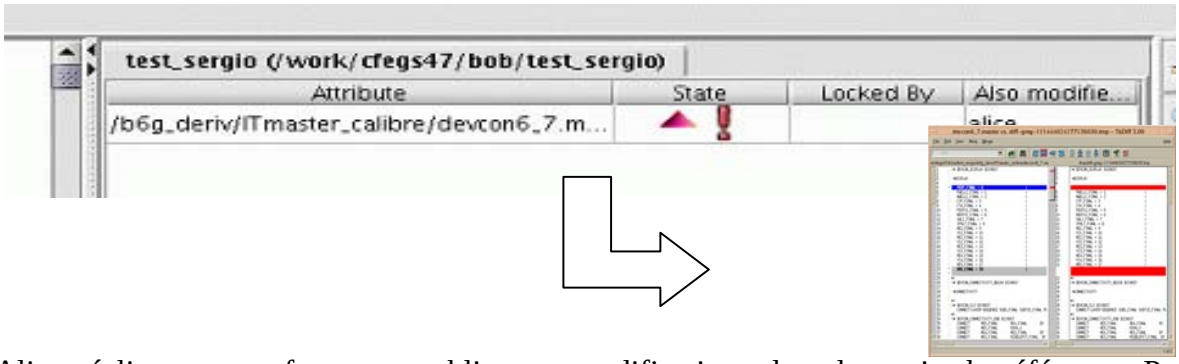
Nous allons par la suite illustrer le fonctionnement de notre SAIC avec l'exemple d'un possible déroulement d'activités. Dans cet exemple fictif, trois développeurs (Alice, Greg et Bob) travaillent simultanément :

Alice modifie le fichier devcon6_7.master. Dans son interface graphique, le SAIC affiche l'icône associée à la propagation en cours *modifié*.



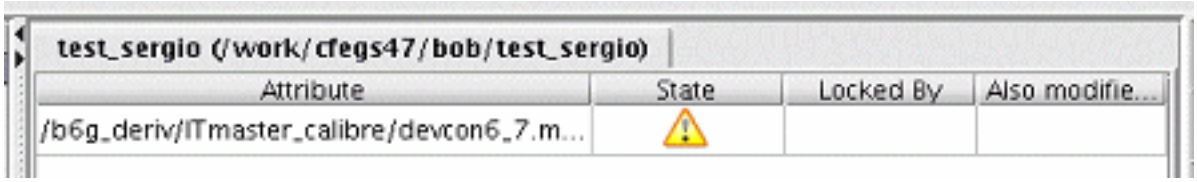
Au même moment, Bob et Greg sont notifiés de l'apparition de cette modification, qui pour eux correspond à la propagation en cours *modifié ailleurs*:

Bob commence lui-même une modification sur le même fichier. Le SAIC montre à Bob et à Alice l'existence de sa propre modification ainsi que l'existence d'une modification parallèle. Le système permet également à chacun de confronter à tout moment ses propres modifications avec celles de l'autre.



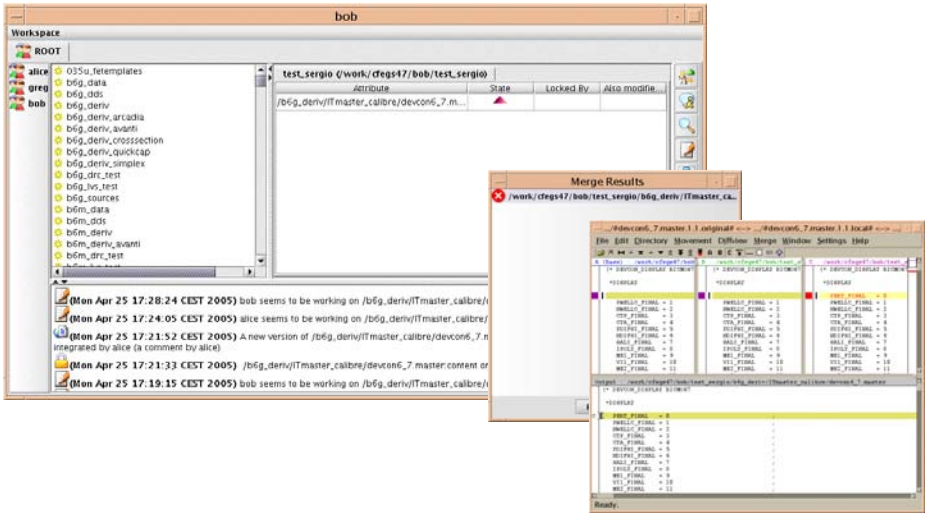
Alice réalise un transfert pour publier ses modifications dans la copie de référence. Pour Alice, la propagation en cours qui avait comme origine son espace de travail et comme objectif celui de la référence disparaît. La propagation modifiée ailleurs (chez Bob) reste pour le moment intacte:

Pour Bob, la modification d'Alice est encore plus proche de son espace de travail, une fusion devra être réalisée lors du transfert Référence -> Bob. C'est l'état que nous avons appelé *conflit* potentiel:



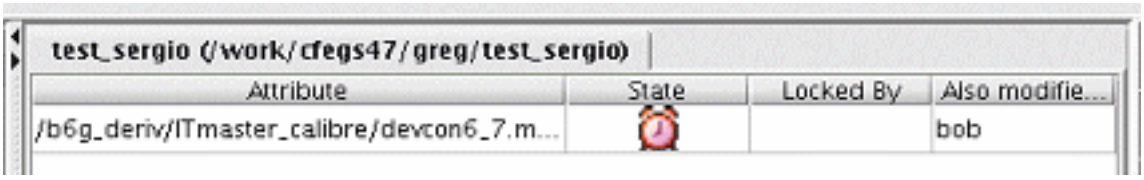
Bob décide de fusionner les nouvelles modifications d'Alice, en réalisant un transfert depuis

l’espace de travail de référence vers son espace de travail.



Le système informe des fusions automatiques réalisées, et demande à Bob de fusionner manuellement les fichiers dont des conflits existent.

1. Apres avoir fusionné les modifications de Alice avec ses propres modifications, Bob transfère le résultat à la copie de référence. Alice est informée que les modifications de Bob sont à distance 1. Sa modification a déjà atteint l’espace de travail et la référence, et elle n’est donc plus en confrontation avec les modifications d’Alice. Le SAIC informe Alice de l’obsolescence de sa copie



7. Un système de règles pour la paramétrisation d'un SAIC

Nous avons illustré comment les concepts du modèle de procédé, élargies avec la notion de distance peuvent être utilisés dans la représentation d’une vision sur le procédé assez riche. Des SAIC spécialisés peuvent être construits pour des contextes différents en définissant des « états » tels que ceux que l’on a montré dans les exemples précédents.

Il serait convenable d’avoir un SAIC universel, adaptable à plusieurs contextes de façon flexible. Dans cette section nous présentons un langage de règles permettant la

paramétrisation d'un SAIC à partir des concepts que nous avons définis. Dans ce langage, pour chaque rôle, nous définissons un ensemble de règles, où chaque règle représente une famille de *propagations en cours* dont les espaces de travail qui jouent le rôle doivent être informés.

Une paramétrisation est une liste des blocs qui suivent la syntaxe suivante:

```
Nom du rôle {
      regle 1
      regle 2
      regle 3
      ....
}
```

Chaque règle correspond à l'un de trois types différents de propagation: INCOMING, OUTGOING ou DIVERGENCE, qui représentent trois préoccupations différentes: les modifications qui arrivent, les modifications qui partent, et les modifications concurrentes.

7.1. INCOMING

Les règles « INCOMING » servent à sélectionner parmi les modifications des autres celles dont l'utilisateur doit être au courant. La syntaxe des règles d'incoming est la suivante:

```
INCOMING : rôle–source* : [distance maximale et nom | noms - distances]
```

La liste de « rôles source » définit quels sont les rôles des participants à l'origine des modifications dont l'utilisateur désire être informé. Ils peuvent être multiples, et dans ce cas le système utilisera la même façon de présenter la modification à l'utilisateur, indépendamment du rôle d'origine de la modification.

La « distance maximale» est la distance au-delà de laquelle les utilisateurs ne veulent pas être avertis de l'existence de la modification. Par exemple, il peut être pratique de n‘informer un utilisateur de la proximité d'une nouvelle spécification que lorsqu'elle est entrée dans un cycle de révision.

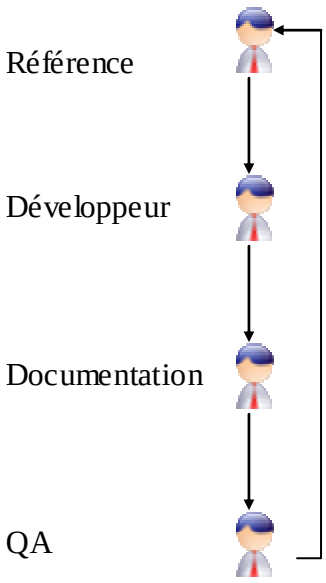
La distance correspond au nombre minimal de transferts qui doivent être effectués pour que l'espace de travail reçoive la modification.

Une distance maximale doit être associée à un nom symbolique et à une icône, afin de rendre l'information plus « lisible » à l'utilisateur.

Si aucune distance maximale n'est déclarée, le système discrimine entre les différentes modifications selon la distance, en donnant un nom différent à chaque distance. Pour cela, des noms différents pour chaque distance doivent être déclarés. Ces noms sont définis comme une liste de couples (distance – nom) ainsi :

```
{
      1:  nom
      2:  nom
      3:  nom
      ...
}
```

Exemple:




```
reference {
      INCOMING : developpeur : 1 : for-integration ;
}
```

Ici, le responsable de l'espace de travail de référence est uniquement averti des modifications faites par les développeurs et il n'est informé que lorsqu'elles sont déjà dans l'espace de travail de QA, à distance 1. Il ne sera pas informé des modifications réalisées pendant la documentation de la modification.

```
Integrateur : developpeur :
{
1:for-integration
2:for-qa
}
```

Dans cet exemple, nous reprenons le scénario antérieur, mais ici l'intégrateur veut être informé dès que les modifications sont dans la phase de documentation.

7.2. OUTGOING

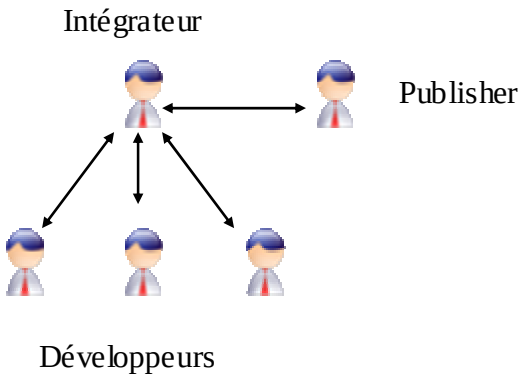
Les règles « OUTGOING » servent à sélectionner les modifications de l'utilisateur qui n'ont pas encore atteint certains espaces de travail particuliers. Ces règles sont définies de la façon suivante:

```
OUTGOING : role-destination * : [noms - distances]
```

La sémantique est semblable à celles des règles d'INCOMING, mais ici l'utilisateur est l'auteur des modifications et le rôle-destination est celui des espaces de travail qui intéressent l'utilisateur. Il s'agit fréquemment de l'espace de référence.

Pour les règles OUTGOING, un nom doit être donné à chaque distance entre la modification et l'espace de travail destination.

Example:




```
developpeur {
      OUTGOING : integrateur : {
```

```
            1: modified
      }
      OUTGOING : publisher : {
            1: for-publish
      }
}
```

Dans l’exemple, les développeurs sont intéressés par deux cas particuliers: ses modifications qui ne sont pas encore intégrées et la proximité de la publication de ses modifications.

DIVERGENCE

Les règles de divergence servent à sélectionner des modifications concurrentes selon la provenance des modifications et selon l'imminence de la fusion dans l'espace de travail de l'utilisateur.

DIVERGENCE : [rôle1 [rôle2]] : distance-minimale, nom

Les règles de divergence permettent la description de plusieurs types de sélection parmi les modifications concurrentes:

Si seul le rôle 1 est mentionné, l'utilisateur est informé lorsqu’il est à l'origine d'une modification qui est en concurrence avec une modification réalisée dans un espace de travail du rôle indiqué, mais seulement si la distance que doivent parcourir les deux modifications pour être fusionnées dans son espace de travail est égale ou inférieure à la "distance-minimale".

Si deux rôles sont mentionnés, l'utilisateur est informé de toute paire de modifications concurrentes entre les deux rôles, pourvu que la distance que doivent parcourir les deux modifications pour être fusionnées dans son espace de travail soit inférieure ou égale à « distance-minimale ».

Si le rôle n'est pas mentionné, l'utilisateur est informé de toute paire de modifications concurrentes qu’il est habilité à fusionner, dès que la distance minimale qui doivent parcourir les deux modifications pour être fusionnées dans son espace de travail est inférieure ou égale

à « distance-minimale ».

Le « nom » indique la dénomination qui sera utilisée pour mentionner la divergence à l'utilisateur.

Exemple:

```
Integrateur {
      DIVERGENCE : développeur, mantainer : 2 : conflit
}
```

Dans cet exemple nous illustrons une topologie dans laquelle un intégrateur reçoit des modifications provenant de deux sources différentes: « mantainers », qui sont en charge de régler des bugs spécifiés par le responsable du produit, et des développeurs. Les « mantainers » peuvent envoyer leurs modifications directement au responsable s'il y a urgence, ou à l'intégrateur. Les développeurs, qui sont en charge de développer de nouvelles fonctionnalités, communiquent uniquement avec l'intégrateur. Sachant que les développeurs travaillent sur des taches bien spécifiées et faciles à contrôler, l'intégrateur n'est pas inquiet de la concurrence entre eux. Par contre, il s'intéresse à de possibles conflits entre les développeurs et les « mantainers », car ces derniers sont contraints de travailler rapidement sans se soucier des conflits potentiels avec les développeurs.

Le comportement du SAIC tel qu'on l'a illustré dans l'exemple de la section 6 (topologie en étoile) peut être défini dans le langage de paramétrisation ainsi:

```
Developpeur {
      INCOMING : développeur : {
            1: obsolete
            2: modifie ailleurs

      OUTGOING : intégrateur : {
            1: modifie

      DIVERGENCE : : 1 : conflit
}
```

C'est en effet l'un de deux comportements possibles du SAIC que nous avons directement implémenté dans notre système. Le deuxième style fait la difference dans l'espace de travail entre la copie de travail est la copie visible, précisant ainsi si une modification a déjà été

rendue visible dans un espace de travail, avec les états suivantes :

En cour de modification: Distance 3 entre la copie du travail du développeur et la version visible dans l'espace de travail de référence. Il indique au développeur qu'il y a des modifications sur sa copie de travail pas encore rendues publiques sur son propre dépôt historique.

Modifié: Distance 2 entre la copie du travail du développeur et la copie visible de l'espace de travail de référence. Il indique que la modification est visible, mais n'est pas encore intégrée dans la copie de référence

En cours de modification ailleurs: Distance 4 entre la copie de travail de quelqu'un d'autre et la copie du travail du développeur concerné. Il indique le développement d'une nouvelle modification, ailleurs.

Modifié ailleurs: Distance 3 entre la copie de travail de quelqu'un d'autre et la copie du travail du développeur concerné.

Presque Obsolète: Distance 2 entre la copie de travail de l'espace de travail référence et la copie de travail du développeur concerné. Il indique qu'une nouvelle modification va être rendue publique

Obsolète: Distance 1 entre la copie historique visible de l'espace de travail de référence et la copie de travail de l'espace de travail concerné. Il indique que l'espace de travail est obsolète par rapport à la copie officielle.

Conflit: Distance 1 pour qu'une modification soit fusionnée dans l'espace de travail du développeur

8. Opportunité

Dans le cadre de l’ingénierie concurrente, l'information contextuelle n'est effective que lorsqu’elle permet aux utilisateurs d’anticiper les évènements à venir, et de réagir à temps. Pour cela, un impératif dans la construction d'un SAIC est de transmettre l'information opportunément.

La notion de distance permet de paramétrer le SAIC pour qu'il n'informe pas les utilisateurs trop tôt, quand l'analyse d'un évènement donné n'est pas encore pertinente à cause du trop grand nombre d’espaces de travail au travers desquels il doit transiter (et qui rendent les données susceptibles être sévèrement modifiées). Une fois dans les limites signalées par la

paramétrisation, le système doit être capable d'informer l'utilisateur le plus tôt possible, pour maximiser sa marge de manœuvre. Pour cela, l'utilisateur doit être informé idéalement de façon immédiate lorsqu'un évènement d'intérêt a lieu. Cela à des conséquences dans implémentation du SAIC:

- Distribution: L'obtention de l'information doit suivre une logique "push" et non "pull": ce n'est pas l'utilisateur qui doit explicitement demander l'information sur l'état du procédé, c'est le système qui doit le tenir continuellement au courant.
- Capture: Le système doit pouvoir collecter l'information automatiquement et continuellement pour éviter des possibles décalages entre l'occurrence d'un évènement et le moment ou le système est au courant.

9. Résumé et conclusions

La modification simultanée d'un système logiciel implique la gestion des risques potentiels pour la cohérence du système. Ces risques se concrétisent lors de la fusion des modifications, une opération nécessaire pour produire un résultat unique, et ainsi aboutir à l'objectif original qui est de réduire le temps de développement.

Afin que l'ingénierie concurrente ne soit pas trop pénalisée par les risques, des mesures préventives doivent être mises en place. Dans les cas les plus drastiques, l'ingénierie concurrente peut être complètement interdite. Dans la plupart des cas, surtout lorsqu'on s'appuie sur des gestionnaires de versionnement classiques, l'ingénierie concurrente est interdite seulement à la granularité du fichier. Nous avons proposé un langage permettant de contrôler l'ingénierie concurrente à un niveau de détail beaucoup plus fin.

Les règles définies dans notre langage ne font que capturer la connaissance et l'expérience des utilisateurs sur le logiciel et sur les besoins de concurrence et de sécurité propres à leur contexte. Cependant, il peut être argumenté que de telles règles ne peuvent capturer que des connaissances très générales, à cause de la nature dynamique et imprévisible des procèdes de Génie Logiciel. Ainsi, seule une vigilance permanente sur le déroulement du procédé peut permettre aux utilisateurs de bien appliquer leur expérience et leur connaissance.

Cette vigilance est possible uniquement si les utilisateurs ont les moyens d'anticiper les risques. Le rôle d'un SAIC (Système d'Augmentation de la Information Contextuelle) est de fournir aux utilisateurs l'information qui peut leur permettre d'identifier ces risques et de prendre les mesures pour les réduire. En particulier, l'information sur les fusions à venir est

nécessaire pour assurer la sécurité et le système doit donc permettre aux utilisateurs de bien les anticiper.

Le problème fondamental dans la conception d'un SAIC vient du fait que la quantité d'information qu'un utilisateur peut absorber est limitée. A cause de cela, l'un des objectifs de base dans la conception d'un SAIC est la minimisation de l'information présentée, toujours en gardant la partie qui maximise les bénéfices pour l'utilisateur.

Dans cette section nous avons montré une façon d'exploiter l'information sur le procédé d'ingénierie concurrente pour filtrer l'information contextuelle. Nous avons montré également comment un SAIC peut être réglé pour montrer la quantité d'information jugée convenable dans chaque situation spécifique, en utilisant les concepts fournis par le modèle du procédé.

La stratégie d'augmentation de l'information contextuelle et la stratégie des politiques correspondent à des visions différentes sur le rôle d'un environnement de support de l'ingénierie concurrente. Ils peuvent cependant cohabiter, laissant à l'utilisateur la flexibilité de définir ses propres solutions mélangeant les deux stratégies. Celles-ci peuvent être vues comme étant complémentaires, l'une capturant et automatisant les stratégiques les plus générales, et l'autre aidant les utilisateurs à assurer la sécurité lors de situations spécifiques.

Chapitre VI. Validation

Tout au long de ce travail de recherche, nous avons développé un environnement pour le support de l'ingénierie concurrente, qui implémente les idées présentées dans cette thèse. Le résultat final est le logiciel CELINE [2]. Ce logiciel a été mis en service dans un environnement industriel au sein de deux équipes de production de la société STMicroelectronics.

La société STMicroelectronics est le cinquième fabricant mondial de produits microélectronique. Elle compte 16 centres de recherche et développement, 39 sites de conception et 16 sites de production, repartis dans quatre continents. Des procédés globaux qui impliquent une interaction très étroite entre un grand nombre de participants, situés dans différents sites, ont lieu en permanence, et cela dans un contexte de marché fortement concurrentiel. L'ingénierie concurrente à STMicroelectronics est une réalité quotidienne. Notre travail s'inscrit dans une démarche générale de cette société pour mieux maîtriser une ingénierie concurrente de plus en plus nécessaire et difficile à gérer.

Il est important de faire deux précisions au sujet de cette démarche. La première est que l'utilisation de CELINE par des équipes de production a été effectuée sur des projets réels et non pas dans des scénarios dédiés à l'évaluation de l'outil. Ceci à nos yeux a rendu l'expérience plus enrichissante. Un gros effort a du ainsi être réalisé pour rendre CELINE assez robuste afin que ses utilisateurs ne soufrent pas de défauts d'implémentation.

La deuxième précision est que seule une partie significative de nos idées, en particulier le SAIC, a pu être utilisée dans cette première expérience industrielle; l'application de politiques jugée moins prioritaire par rapport aux besoins des équipes n'a pas été complètement développée et est restée au niveau de prototype [EG 01][EG 06].

Dans ce chapitre nous faisons la chronique et l'analyse de cette expérience. Nous soulignerons les défis que la mise en production de CELINE a posée et comment ils ont façonné la conception et l'évolution de l'outil.

Finalement, nous analyserons l'impact que CELINE a eu dans le contexte de l'expérience et les leçons que nous en avons tiré.

---

[2] CELINE: **C**oncurrent **E**ngineering **LI**ght **N**ew **E**nvironement

1. Objectifs et méthodologie

Les objectifs généraux que nous nous sommes fixés pour guider l'implémentation, le déploiement et l'évaluation de CELINE dans les équipes ont été:

1. Fournir un bénéfice tangible aux utilisateurs pour leur collaboration. Nous avons donc du faire attention à l'ergonomie de l'outil et à la robustesse de l'implémentation.
2. Etudier les difficultés que l'implémentation de nos idées présente, en particulier la complexité des algorithmes.
3. Définir une architecture permettant d'adapter l'outil à plusieurs gestionnaires de configuration.
4. Avoir des indices sur les fonctionnalités les plus utiles pour les utilisateurs dans un environnement d'ingénierie concurrente tel que le notre.

Notre logiciel CELINE a été développé et déployé en plusieurs itérations courtes. Le tableau ci-dessous présente les dates de livraison des version de notre produit :

Version :	**Date de livraison :**
Version 1.0	Mai 2005
Version 1.1	Septembre 2005
Version 1.2	Novembre 2005
Version 1.3	Février 2006

Les utilisateurs ont été encouragés à donner leur avis sur l'outil et à proposer de nouvelles fonctionnalités qu'ils trouvaient intéressantes. Des réunions avec les utilisateurs ont été effectuées régulièrement.

Les opinions des utilisateurs ont été notre premier critère pour évaluer la pertinence de nos propositions. Nous avons travaillé de près avec les utilisateurs ainsi qu'avec la personne responsable de la gestion du dépôt de référence, également chargée d'établir les politiques d'accès et de modification des données officielles. Ainsi, nous avons profité directement des perceptions de première main sur les bénéfices et faiblesses de l'outil. Les commentaires des utilisateurs ont donnée suite à plusieurs améliorations de CELINE.

Nous nous sommes également basé sur les traces d'utilisation de CELINE et les traces d'accès au dépôt centralisé des données pour observer les patrons d'utilisation. A ce sujet, il est important de noter que l'observation d'un outil dans le cadre d'une expérience industrielle doit faire face à l'impossibilité d'isoler, même partiellement, l'expérimentation de l'influence de facteurs externes. De même, nous avons eu beaucoup de difficultés à mesurer l'impact des facteurs internes tels que l'ergonomie de l'outil, les différents degrés d'expérience des participants, les différences culturelles, les changements dans la gestion de l'équipe, etc.

A la lumière de ces faits, nous ne faisons pas de lecture définitive sur les données extraites des fichiers de trace. Ces données ont été utilisées comme un des indicateurs possibles des tendances, nécessitant une analyse plus approfondie. En conséquence, nous concédons plus d'importance à la perception des utilisateurs sur l'outil qu'aux métriques.

2. Cadre d'expérience

Deux équipes au sein du group DAIS de la société STMicroelectronics ont été choisies pour intégrer CELINE dans leurs environnements de travail.

La première équipe, LVT (Layout Verification Team), est une équipe qui compte environ une dizaine de développeurs (le nombre des développeurs actifs étant variable) en charge de développer des jeux des tests utilisés pour assurer la validité des certaines familles de produits microélectroniques.

La sémantique des données qui constituent les tests est donnée par l'outillage CAD propre à ST. Les jeux de tests peuvent être assimilés à des logiciels traditionnels pour ce qui concerne la collaboration et la concurrence: ils sont constitués d'entités logiques couplées par des relations de composition et de dépendance, ils sont édités avec des outils d'édition de texte, et leur fusion présente les mêmes types de risques pour la cohérence que ceux d'un logiciel écrit dans un langage de programmation traditionnel.

Les membres de équipe LVT qui ont utilisé CELINE se trouvent tous dans un même endroit géographique, dans les installations de STMicroelectronics à Crolles (France).

La deuxième équipe est l'équipe PCELL, chargée du développement des logiciels dans le langage de programmation propriétaire SKILL (Cadence Systems). Les membres de cette équipe (environ 6) sont repartis géographiquement dans deux endroits différents: le centre de recherche et développement à Crolles (France) et le centre de conception à Tunis.

Dans les deux cas les données sur lesquelles les équipes travaillent sont réparties en plusieurs

« projets ». Un projet est une hiérarchie de fichiers *et de projets*. Un projet peut apparaître dans plusieurs hiérarchies différentes, mais il n'existe pas de boucle (un projet ne se contient jamais lui même). Les « projets » sont représentés dans un espace de travail par un ensemble de fichiers de texte, manipulés par les développeurs de façon manuelle à l'aide d'un éditeur de texte traditionnel. Le nombre moyen des copies de produits par utilisateur varie entre cinq et dix, chacun contenant une moyenne proche de 100 fichiers.

Les deux équipes ont été choisies parce que leur méthode de travail se trouve de plus en plus limitée par rapport à une nécessitée croissante de travailler en concurrence. Les deux équipes utilisent le système de gestion de version Synchronicity [Synch], sur lequel ils stockent les données de référence. La synchronisation directe entre deux espaces de travail est interdite, elle doit passer impérativement par le gestionnaire de versions. La politique officielle au moment de l'introduction de CELINE, obligeait les utilisateurs à verrouiller chaque fichier avant de pouvoir faire n'importe quelle modification, interdisant ainsi tout concurrence *par fichier*.

Dans la réalité, la plupart des développeurs ont reconnu avoir eu besoin d'ignorer le verrouillage (en copiant le fichier dans un répertoire non contrôlé) pour pouvoir travailler sur un fichier déjà modifié ailleurs afin de ne pas rester bloqué. La politique de verrouillage se révélait dans ces cas non seulement inadéquate, mais même négative, car elle rend l'utilisation des outils de fusion difficile (parce que le travail concurrent doit être effectué *en dehors* de la gestion du système) et elle peut donner une fausse sensation de sécurité à ceux qui assument la validité du verrou.

Malgré le besoin des utilisateurs de travailler en concurrence, au point de « tromper » le système, le responsable des jeux de test et les développeurs eux mêmes insistent sur le fait que la concurrence présente un risque important pour les données, au point de ne pas pour pouvoir banaliser la concurrence en enlèvent la politique de verrouillage sans un mécanisme pour réduire ce risque.

Dans ce contexte, les contraintes pour l'intégration de CELINE dans le cycle de développement ont été définies ainsi:

- Le seul changement au procédé de travail, dans un premier temps, doit être d'éliminer la politique de verrouillage.
- La topologie en étoile, avec un dépôt central passive, doit être gardée.
- L'utilisation de CELINE doit pouvoir être optionnelle, et elle ne doit pas imposer des

efforts additionnels à l'utilisateur.

- CELINE doit s’intégrer avec le gestionnaire de versionement utilisé [Synch].

Vu ces trois contraintes, il a été décidé de privilégier la mise en place des fonctionnalités SAIC. Les fonctionnalités de CELINE comme moteur de politiques ont été prototypées dans un premier temps (avec un langage autre que celui décrit dans ce document), mais ils n'ont pas été mises en production.

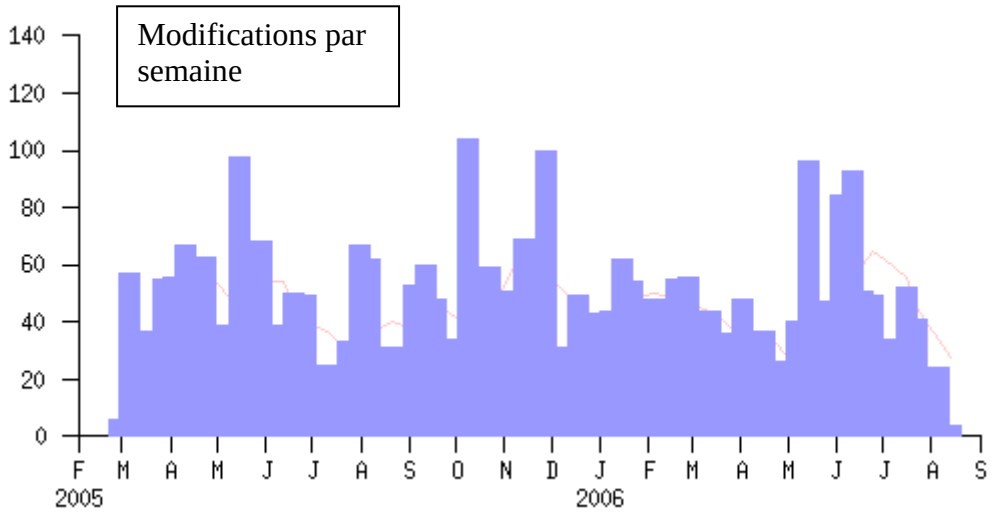
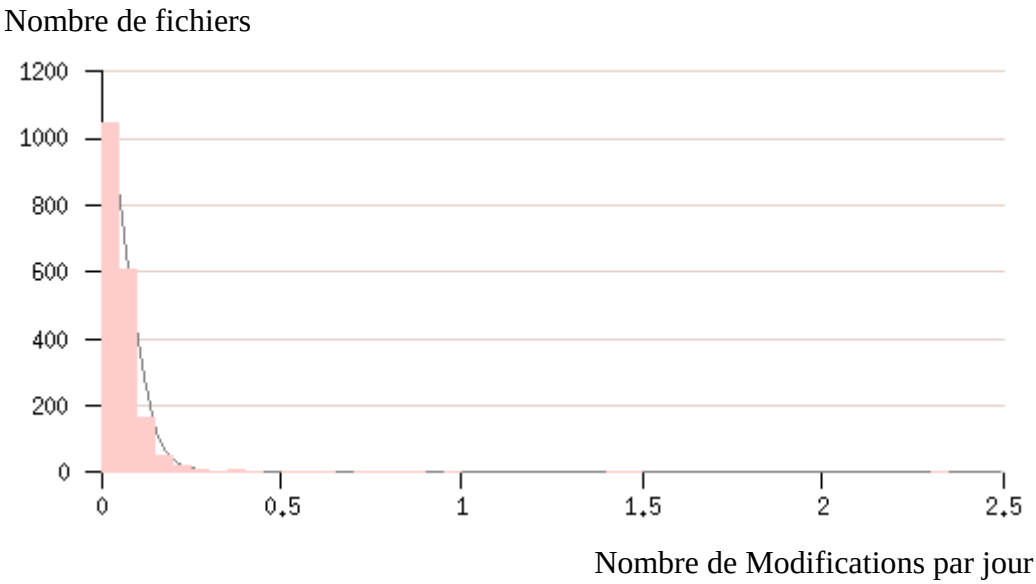


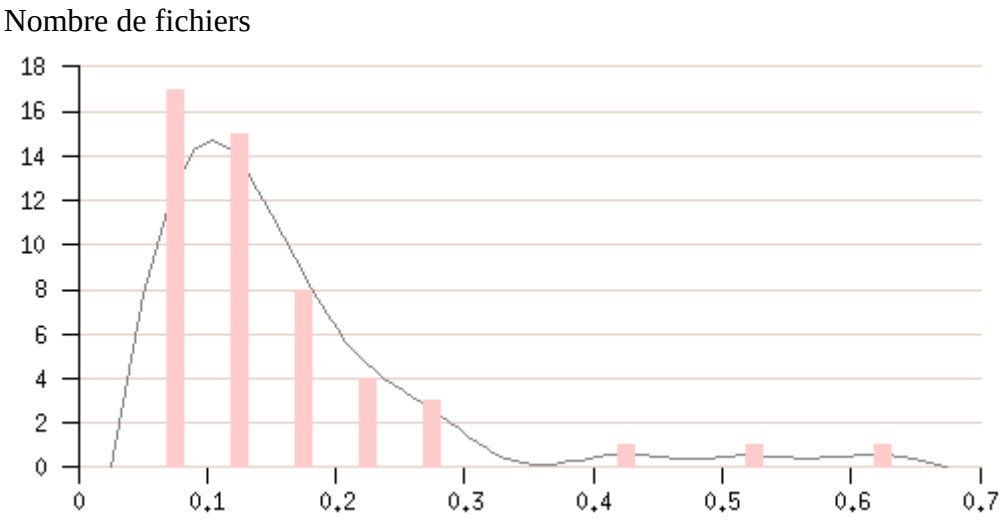
Figure 19 : Modifications publiés par semaine

Nous avons mesuré l’activité dans le dépôt de référence entre les mois de mars 2005 et août 2006 (soit durant 18 mois). Le figure 11 montre le nombre total de modifications qui sont intégrées dans le dépôt par semaine. Il prend en compte la totalité des fichiers modifiés par les deux équipes, et donne une idée du niveau d’activité des équipes.



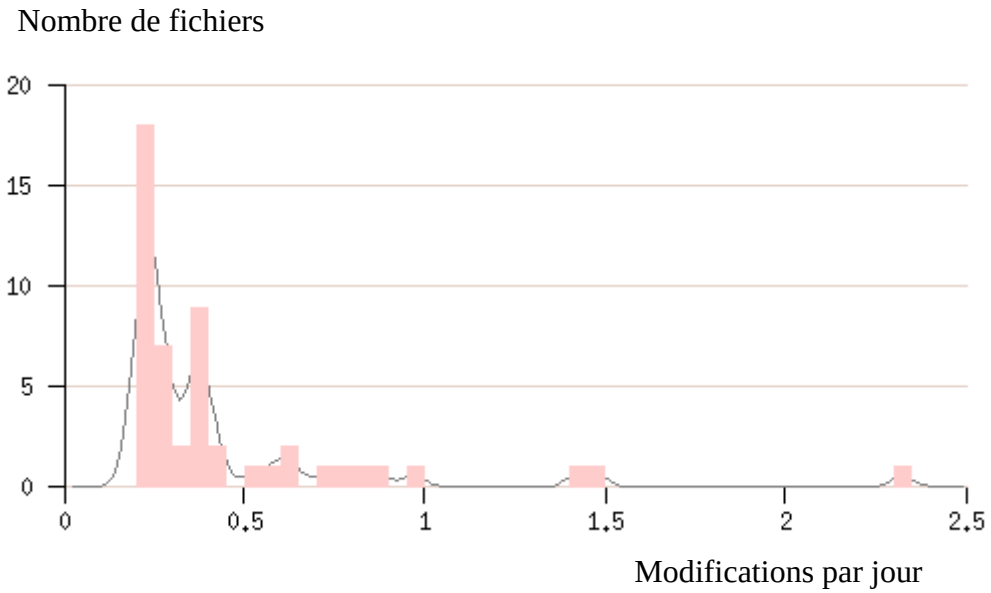
La figure précédente représente la distribution des fichiers selon le nombre de modifications par jour. Comme [Est 96] l'avait déjà montré dans l'étude sur l'activité dans les espaces de travail utilisés pour le développement du logiciel CATIA, il est courant que la grande majorité de fichiers ne soient modifiés que très rarement, tandis qu'un certain nombre sont modifiés en permanence.

Le graphique suivant montre la répartition des 100 fichiers les plus fréquemment modifiés.



En regardant de près les fichiers les plus souvent modifiés, nous remarquons qu'il s'agit des fichiers nommés « fichiers master », qui, en effet, constituent un noyau utilisé par plusieurs « produits » (des jeux des tests correspondant à des technologies différentes). Ces fichiers doivent donc évoluer plus fréquemment que la moyenne et dans plusieurs espaces différents.

Nous avons décidé de circonscrire notre analyse des activités de l'équipe LVT aux fichiers master, pour lesquels l'ingénierie concurrente trouve tout son sens. La répartition des « fichiers masters » selon le nombre des modifications par jour est représenté par le graphique suivant.



3. Implémentation

L'architecture générale de CELINE peut être décrite comme correspondant au paradigme client / serveur, où l'exécution de l'application est comprise comme l'exécution de plusieurs *clients*, qui s'exécutent de façon locale sur les postes de travail des participants, plus l'exécution d'un *serveur* qui gère les données communes et la logique de l'ensemble et qui est le seul interlocuteur des applications clientes.

Grosso modo, la partie cliente de CELINE est en charge de:

- Capturer les ordres de l'utilisateur (avec une interface d'utilisateur), et les exécuter si la politique le permet.
- Empêcher l'utilisateur de réaliser des actions qui violent la politique.
- Présenter à l'utilisateur l'information provenant du serveur.
- Collecter l'information sur les actions de l'utilisateur pour informer le serveur.

La partie serveur en chargée de:

- Maintenir une représentation de l'état du procédé

- Interpréter une politique pour autoriser ou interdire les actions des utilisateurs.
- Répondre aux requêtes des clients sur la légalité d'une opération
- Distribuer l'information contextuelle

A côte de cette division de l'application basée sur le « lieu » d'exécution de chaque partie, nous avons divisé la logique de l'application en plusieurs modules qui capturent des aspects significativement indépendants. Cette division a été fortement guidée par le besoin de maintenir CELINE indépendant du système de gestion de la configuration sous-jacent, et de pouvoir l'adapter à des environnements d'ingénierie concurrente qui utilisent une plateforme existante.

L'exécution de l'application réelle est donc composée de plusieurs processus « serveur » relativement indépendants et d'un processus client qui sert d'interface utilisateur et qui communique avec les différents serveurs.

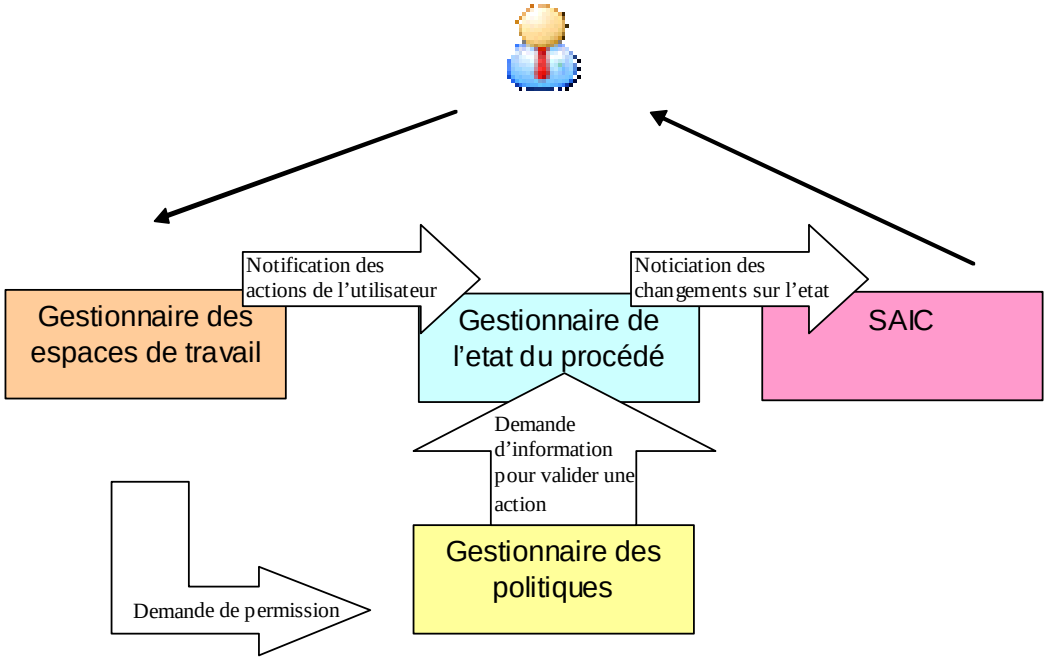


Figure 20 : Architecture de CELINE

3.1. Gestionnaire des espaces de travail

Il s'agit de la partie du système chargé d'implémenter les opérations de base (save, load, synchronize). Ces opérations ont été implémentées en se basant sur un gestionnaire de configurations abstrait, gérant les notions de révision et de branche. Sur cette abstraction, CELINE implémente les trois opérations de base, en cachant les détails de versions, branches et ancêtre commun, qui sont pour nous des concepts appartenant a l'implémentation [ABB

98]. Cela nous permet d'intégrer CELINE avec divers gestionnaires de configuration. En effet, nous avons développé trois implémentations de la couche « gestionnaire de versions ». La première version a été implémentée sur le gestionnaire de versions CVS. Ce prototype a été expérimenté pendant une courte période de temps dans notre équipe de recherche.

Pour STMicroelectronics, nous avons réalisé une implémentation hybride de la couche de gestionnaire de version sur le logiciel Synchronicity Developer Suite [Synch] et notre propre « dépôt » local de révisions basé sur une simple compression zip. Cette version a été mise en production dans les deux équipes.

Finalement, nous avons réalisé une implémentation base sur le gestionnaire de versionnement « distribué » monotone, qui n'a jamais été mis en expérimentation.

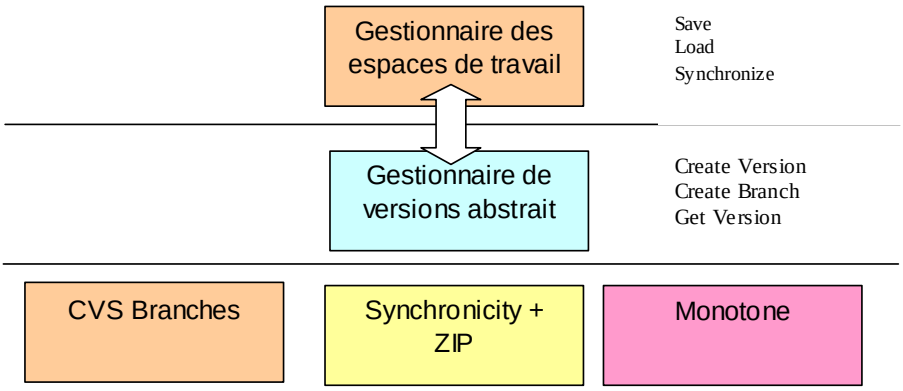


Figure 21 : Gestionnaire d'espaces de travail

Le gestionnaire d'espaces de travail doit informer le gestionnaire de l'état du procédé sur les actions de chaque espace de travail, pour que le deuxième puisse maintenir une représentation de l'état global du procédé.

3.2. Gestionnaire de l'état du procédé

Ce module est en charge de maintenir l'état global du procédé à jour. La simplicité de notre modèle d'exécution des procédés fait de l'implémentation du gestionnaire de l'état du procédé une tache facile en théorie. Dans la pratique, le nombre d'états peut poser des problèmes pour le passage à l'échelle qui doivent être analysés.

En effet, le nombre maximal de Propagations en Cours qui doivent être gardées pour N espaces de travail grandit avec la formule (N * (N^2 – N)) / 2, et ceci pour une seule entité logique. L'entité logique minimale que le système doit prendre en compte est le fichier car la

modification d'un fichier est celle qui est toujours considérée comme étant atomique par notre système: c'est toujours tout le fichier que est modifié, pas telle partie ou telle autre. Le système peut donc ne garder que les propagations en cours *par fichier* et trouver les modifications par entité logique en regardant les propagations en cours sur les fichiers qui en font partie.

Cette représentation peut poser des problèmes lorsque le nombre de fichiers modifiés est élevé. Cependant, il faut noter que dans la plupart des cas [Est 96] seulement un nombre limité des fichiers sont régulièrement modifiés; la majeure partie des fichiers reste immuable la plupart des temps. Des analyses statistiques plus approfondies sur un large nombre de projets logiciel sont nécessaires pour déterminer la faisabilité d'une représentation de l'état du procédé comme celle que l'on propose.

En ce qui concerne la complexité de maintenir à jour l'état du procédé, nous utilisons des algorithmes relativement peu coûteux. Nous décrirons les algorithmes utilisés dans CELINE pour implémenter les conséquences des opérations sur l'état du procédé.

L'édition d'une entité logique (soit le fichier) par un espace de travail peut être implémenté ainsi (pseudo-code):

```
; N étant le nombre d'espaces de travail dans le groupe
; pc étant un vecteur des propagations en cours pour le fichier modifié
; p[i][j][k] = 1, si la propagation en cours (i,j,k) existe, 0 autrement
for (i = 0; i < N ; i++) {
      if (i != k) {
            pc[i][j][i] = 1;
      }
}
```

Et le transfert depuis l'espace de travail *j* à l'espace de travail *k*, ainsi :

```
for (i = 0; i < N; i++) {
      pc[i][j][k] = 0
}
for (i = 0; i < N; i++) {
      for (r = 0; r < N; r++) {
      if (pc[i][j][r]) {
            pc[i][k][r]=1
      }
}
```

Pour trouver les possibles conflits :

```
for (i = 0; i < N; i++) {
      for (r = 0; r < N; r++) {
            if(pc[i][j][k] && pc[r][k][j]) {
                  conflicts[role(i)][role(j)] = 1;
            }
      }
}
```

Le nombre de vérifications qui doivent être faites pour mettre à jour l'état du procédé (pour une E.L. quelconque) lors d'un transfert grandit de manière quadratique par rapport au nombre d'espaces de travail. A nouveau, des analyses statistiques sur plusieurs projets de développement logiciel sont nécessaires pour trouver des nombres typiques et mesurer si notre algorithme est adapté ou non.

3.3. Gestionnaire de politiques

Le gestionnaire de politiques est la partie chargée de confronter les actions des participants à la politique combinée avec l'état actuel du procédé. Pour cela, le gestionnaire des politiques doit communiquer avec le gestionnaire de l'état du procédé pour demander l'information nécessaire, et avec le gestionnaire d'espaces de travail pour permettre ou empêcher une opération de se réaliser.

Nous avons prototypé un gestionnaire des politiques basé sur un langage d'ingénierie concurrente avec une saveur impérative [EG 01]. Pour la mise en production de CELINE à STMicroelectronics, nous l'avons remplacé par une politique prédéfinie directement implantée dans le système.

3.4. SAIC

Le domaine de l'augmentation de l'information contextuelle est chargé de faire parvenir l'information sur l'état du procédé aux utilisateurs.

4. Évaluation

4.1. Introduction

Dans cette section nous exposerons les résultats obtenus avec l'intégration de CELINE dans la chaîne de travail des équipes pilote. Évaluer le résultat d'une expérience comme celle-ci est un exercice difficile, à cause de la difficulté à définir des critères quantitatifs pour mesurer l'accomplissement des objectifs tels que "améliorer la collaboration" ou "fournir une aide a la gestion du travail concurrent". Ceci est particulièrement vrai lorsque l'expérience est menée dans un environnement réel qui n'est donc pas isolée des influences extérieures, comme c'était le cas.

Cependant, nous pouvons nous baser sur des mesures quantitatives pour guider notre évaluation sur les bénéfices apportés par l'outil.

La première valeur que nous pouvons mesurer est tout simplement l'ampleur avec laquelle l'outil a été adopté par les utilisateurs. Ceci peut être mesuré car nous avons décidé d’encourager les utilisateurs à se servir de l'outil sans pour autant employer des mesures techniques externes à l'outil pour privilégier son utilisation ou pour forcer les ingénieurs à l'utiliser. Ainsi, les utilisateurs ont été libres d'utiliser CELINE ou bien de continuer à utiliser leurs outils habituels pour interagir avec le dépôt de référence Synchronicity [Synch]. L'ampleur de l'adoption de CELINE par les utilisateurs peut donc être considérée comme un indicateur important du fait que l'outil fournit vraiment un bénéfice concret. Dans la première partie de cette section nous exposerons les résultats mesurés.

Le deuxième indicateur prétends mesurer "le niveau de concurrence", entendu comme la fréquence avec laquelle les utilisateurs ont travaillé simultanément sur les mêmes données. Nous partons de l'hypothèse qu'un outil qui diminue les risques liés à la concurrence devrait permettre aux utilisateurs de travailler simultanément plus souvent. Malheureusement nous ne disposons pas des traces appartenant à la période antérieure à l'intégration de CELINE qui nous permettrait de comparer cette fréquence avant et après l’expérience, et nous devons nous contenter des estimations données par les utilisateurs.

Ces deux valeurs constituent les deux axes sur lesquels nous avons concentré nos recherches pour évaluer à quel point nos idées apportent un bénéfice tangible aux utilisateurs.

4.2. Niveau d'adoption de CELINE dans les équipes pilote

Pour mesurer l'ampleur de l'adoption de l'outil, nous avons utilisé les fichiers de trace du dépôt synchronicity [Synch] ainsi que les fichiers de trace de CELINE, en comparant le nombre de transferts qui ont été effectués avec CELINE avec le nombre de transferts enregistrés par le serveur synchronicity, qui correspond au nombre totale des transferts.

Pour l'équipe LVT, l'évolution de cette comparaison sur les temps est représentée par le graphique suivant, ou l'axe Y corresponds à la proportion des transferts réalisés avec CELINE par rapport au total de transferts, et l'axe X représente la période dans le temps.

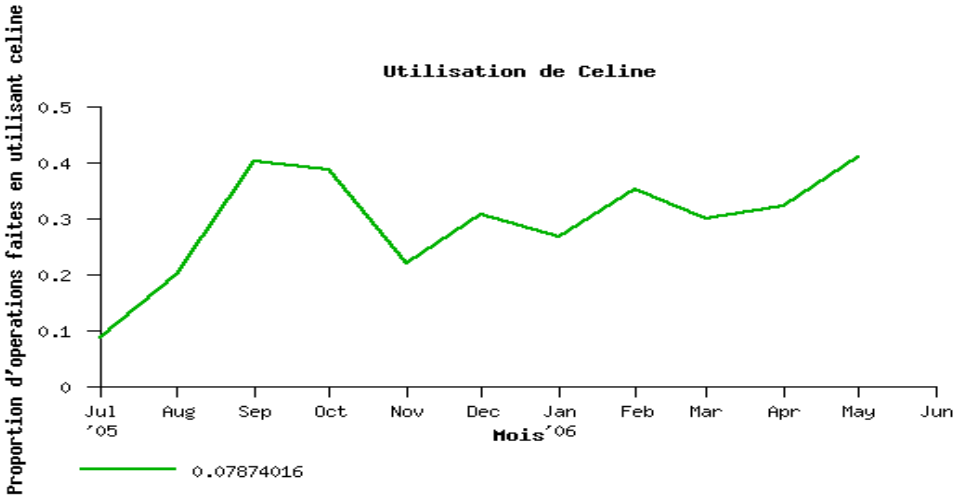


Figure 22 : LVT, Axe Y = opérations faites par CELINE / Totale opérations

L'outil a donc atteint un pic d'utilisation dans le mois de septembre 2005, ce qui correspond à la sortie d'une nouvelle version intégrant plusieurs des suggestions les plus importantes après une première période d'utilisation. Après ce pic, l'adoption de l'outil comme le moyen pour interagir avec le système de versionnement a stagné, se situant autour de 40%.

Pour l'équipe PCELL, qui a commencé à utiliser l'outil en novembre 2005, le graphique correspondant à une période de sept mois est le suivant:

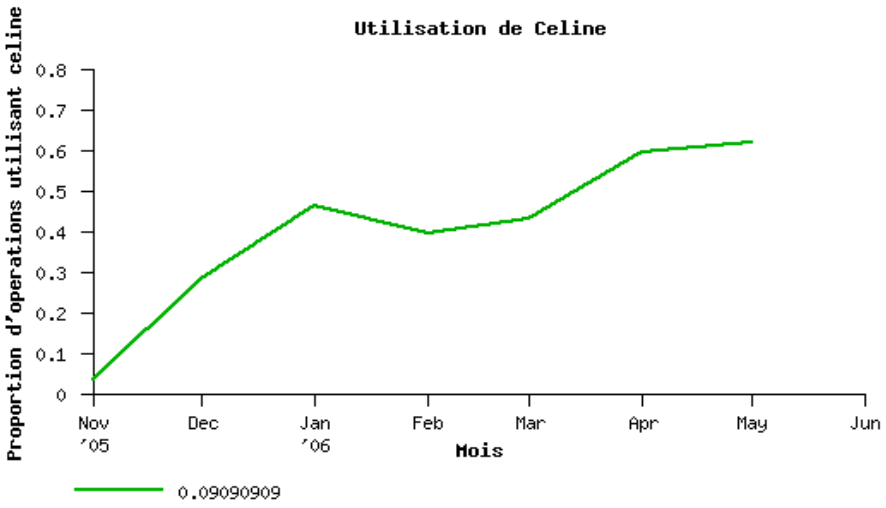


Figure 23 : PCELL, Axe Y = opérations faites par CELINE / Totale opérations

Ici, nous notons un plus fort taux d'utilisation de CELINE, se rapprochant du 70% des actions des utilisateurs passant par CELINE pour la fin de la période que nous avons analysée.

4.3. Analyse

Ces indicateurs sur l'adoption de CELINE ainsi que le dialogue avec les utilisateurs des deux équipes ont fait ressortir les faits suivants:

3. Dans l'équipe LVT, la proximité de tous les développeurs (tous travaillant à quelques mètres les uns des autres) a rendu beaucoup moins significatif le bénéfice pour chaque utilisateur le fait de pouvoir “regarder” le travail des autres depuis son poste de travail. Dans la plupart des cas, une communication verbale informelle est suffisante pour trouver dans quelle mesure le travail d'un autre et incompatible avec le sien. Cette communication informelle facilite également une répartition des taches beaucoup plus fine et « à la volée », qui permet de diminuer le besoin de travailler en parallèle sur les mêmes données. Le faible nombre des participants et la proximité physique facilite ce type de dialogue permanent.
4. En contraste, dans l'équipe PCELL la distribution géographique des utilisateurs rend beaucoup plus significatif le bénéfice d'être informé des activités des autres ainsi que le fait de pouvoir comparer son travail avec les travail des autres depuis son propre poste de travail. En effet, les membres de l'équipe PCELL ont signalé cette deuxième fonctionnalité comme la plus importante de l'outil et l'ont utilisé beaucoup plus souvent que dans l'équipe LVT.

5. L’habitude des utilisateurs à l'environnement du style Unix (lignes de commande) du système de versionnement a été un facteur très important, peut-être sous-estimé, dans l'adoption de l'outil. Le plus grand taux d'adoption de l'outil a été parmi les utilisateurs qui n’avaient pas beaucoup d'expérience avec le gestionnaire de versionement en place et qui trouvaient l'interface graphique plus facile à utiliser, c'est-à-dire les plus nouveaux dans l'équipe. Les utilisateurs plus expérimentés ont déclaré être plus à l’aise en général avec une interface à base de lignes de commandes qu’avec l'interface graphique de CELINE.

6. L'utilisation faite par une partie importante des utilisateurs a été une combinaison de CELINE avec l'environnement habituel du style unix. En effet, plusieurs utilisateurs sont tombé dans un mode d'utilisation où la fenêtre CELINE était laissée dans l'arrière plan de l'écran et les transferts étaient réalisés très souvent avec la ligne de commandes. Dans ce mode là, l'information affiché par le SAIC ne corresponds pas toujours exactement à l'état réel des données, car les transferts réalisés sans l'intervention de CELINE échappent à l'observation du SAIC. Cependant, étant donné que les modifications sur les fichiers sont capturées par CELINE, même lorsque l'édition est réalisée sans passer par l'interface utilisateur de CELINE, le SAIC leur est utile pour informer en temps réel sur l'activité dans l'espace de travail des autres (l'information par rapport à l'état de la copie de référence n'étant pas forcement à jour). L’option de l'interface graphique de CELINE qui permet aux utilisateurs de recalculer l'état des données par rapport à la référence à souvent servi aux utilisateurs pour remplacer ce manque d'information en temps réel causé par les participants qui n'utilisent pas toujours CELINE pour les transferts.

7. Ce mode de fonctionnement hybride (CELINE plus ligne de commandes) a été adopté par presque la totalité des participants de l'équipe LVT, et beaucoup moins par les membres de l'équipe PCELL. Ceci corresponds au plus fort besoin dans l'équipe PCELL d'avoir une information précise sur les activités de chacun (à cause de la difficulté d'avoir une communication informelle permanente) ainsi qu’aux habitudes plus enracinés dans l'équipe LVT de l'utilisation de lignes de commandes.

En résumé, l'outil a été utilisé de manière importante, les participants ayant trouvé des bénéfices tangibles. Cette utilité s’est avéré nettement plus élevé dans l'équipe dont la communication entre les membres est plus difficile, ce qui est naturel et qui indique le point fort de l'outil comme aide à la coordination des ingénieurs.

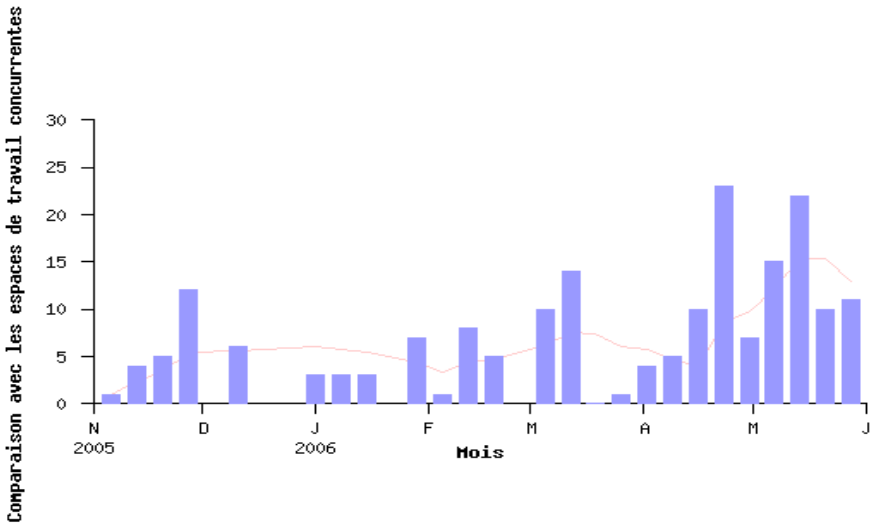
Le fait de ne pas avoir forcé l'utilisation de CELINE a permis de prendre le taux d'adoption de l'outil comme un indicateur de sa réussite. Inversement, le fait que CELINE ne soit pas utilisé par tous les participants a une incidence négative car CELINE ne dispose que des l'informations provenant des participants qui l'utilisent.

4.4. Concurrence

Nous avons mesuré, en utilisant les traces de CELINE, le nombre de modifications concurrentes par jour (entendue comme le nombre des fichiers modifiés simultanément par plusieurs participants). Malheureusement les fichiers de trace du dépôt synchronicity ne permettent pas de récupérer cette information pour la période avant CELINE. Nous n'avons donc pas pu comparer le nombre des modifications concurrentes par jour *avec* et *sans* l'utilisation de CELINE.

Cependant, durant toute la période d'utilisation de CELINE, le nombre de modifications concurrentes par jour (en relation à l'activité totale) n'a guère évolué et il a coïncidé *grosso modo* à l'estimation initiale faite par les utilisateurs, chaque utilisateur étant confronté en moyenne à deux ou trois modifications concurrentes par semaine dans l'équipe PCELL, et une ou deux pour l'équipe LVT. A ce sujet, les utilisateurs ont confirmé avoir continué à limiter par eux mêmes le plus possible les modifications concurrentes sur les mêmes fichiers, en affectant, autant que possible, des taches indépendantes à des personnes différentes.

Les développeurs ont déclaré cependant avoir été plus à l'aise sachant qu'ils seraient informés rapidement des modifications en cours. Dans le cas de l'équipe PCELL, la possibilité de regarder en temps réel l'évolution des modifications concurrentes dans les espaces de travail des autres a été considère comme une aide précieuse. Le graphique suivant montre le nombre d'appels par semaine et par personne à la fonctionnalité de comparer un fichier dans son propre espace de travail avec celui dans un autre espace de travail, pour l'équipe PCELL.



Des membres de l'équipe PCELL ont proposé des améliorations qui ont été implémentés, tels que le fait de pouvoir faire des comparaisons à trois voies (intégrant l'ancêtre commun) à tout moment, et pas seulement au moment de la fusion.

Pour résumer, l'utilisation de CELINE n'a donc pas entraîné un changement quantitatif du niveau de concurrence dans le travail des équipes, mais le fait d'avoir facilité la gestion de chaque utilisateur sur la concurrence permet de penser que CELINE faciliterai de façon importante un élargissement des équipes dans le but de réduire le temps de développement. Dans le cas particulier de l'équipe LVT, cette conclusion est de plus raisonnable lorsqu'on considère que l'équipe est à la limite de la taille permettant un rassemblement des utilisateurs dans un espace unique permettant une communication verbale informelle presque permanente.

4.5. Conclusions

CELINE est un environnement de support de l'ingénierie concurrente que nous avons construit sur les idées présentées dans ce document. Cet environnement fournit les fonctionnalités de base pour travailler collectivement selon la définition de coopération exposée dans le chapitre 3. CELINE est capable également de maintenir une représentation de l'état global de cette collaboration, tel que nous l'avons définit dans le chapitre 4.

Nous avons conçu CELINE pour pouvoir l'intégrer facilement avec plusieurs gestionnaires de configuration. La conception de CELINE vise également son extensibilité par des modules qui peuvent "regarder" l'état de la collaboration et instrumenter les actions des utilisateurs. C'est le cas d'un gestionnaire des politiques, qui permet ou refuse les actions des utilisateurs en se basant sur l'état de la collaboration, pour faire respecter une politique définit par les

utilisateurs. Également, sur cette structure il est possible de brancher des systèmes dits d'augmentation de l'information contextuelle. Nous avons étendue en effet CELINE avec un SAIC visant un style de collaboration correspondant à une topologie en étoile, sans verrouillage. Nous avons par la suite mis CELINE à l'épreuve dans deux équipes de la société STMicroelectronics et nous avons suivi l'utilisation de l'outil pendant douze mois.

L'implémentation de CELINE nous a permis d'identifier les problèmes que la construction d'un système basé sur notre modèle de « état de la collaboration » suppose en terme de complexité algorithmique. Elle nous a permis également de raffiner une architecture dont les différentes composantes semblent aujourd'hui être largement réutilisables.

L'adoption de CELINE par les utilisateurs ainsi que leur retour ont montré que CELINE fournit une aide tangible aux utilisateurs.

La manière d'utiliser CELINE dans les équipes pilotes corresponds à notre perception selon laquelle un environnement de support de l'ingénierie concurrente doit tenir compte de l'existence d'une politique souhaitée (dans ce cas, l'exclusion des modifications concurrentes au niveau fichier) et également supporter les utilisateurs lorsque la politique doit être ignorée (le SAIC et d'autres supports a la coordination).

Conclusion

Clausewitz, cité par [Sch 91], dans son célèbre travail sur la guerre, affirme que même le meilleur plan ne peut tout prévoir, et qu'il peut être mis en l'échec lorsqu'il fait face aux contingences qui apparaissent « in the fog of war » (dans la fumée de la guerre). Le développement des systèmes logiciels est une activité dont l'une des rares choses qui peuvent être anticipées est l'apparition de contingences et le besoin de changement.

Dans cet esprit nous avons proposé une façon de piloter le développement, qui n'impose pas de prédéfinir un « plan maître » qui considère le développement comme étant une série des taches ordonnées. L'observation de la réalité montre que c'est plus la concurrence et l'itérativité que l'ordonnément des tâches qui caractérisent les procédés de développement logiciel. Le succès des systèmes de gestion de configuration, omniprésents aujourd'hui dans les sociétés de développement, constitue une preuve de l'importance dans un environnement de développement de supporter la concurrence.

Si aujourd'hui les moyens informatiques pour permettre à plusieurs personnes de travailler simultanément sur les mêmes données sont maîtrisés (espaces de travail, systèmes de versionement, outils de fusion), le contrôle de la concurrence lui-même a toujours été négligé au profit d'une stratégie basée sur les idées « classiques » de planification des procédés. Le contrôle de la concurrence, dans la plupart des gestionnaires de la configuration, est réduit à deux choix: la concurrence totale et sans contrôle où l'absence totale de concurrence au niveau des fichiers.

Dans notre travail, les utilisateurs peuvent contrôler la concurrence en définissant les types de fusions permises. Ce choix s'explique par le fait suivant: c'est au moment de la fusion que la cohérence des données peut être compromise. C'est donc en assurant des fusions « sûres » que les utilisateurs contrôlent indirectement la concurrence. Le système lui-même assure une concurrence maximale, en respectant au même temps les règles pour la fusion et la capacité à arriver toujours à un résultat final commun.

Cette proposition se base sur un modèle d'exécution de procédé qui est très simple, mais qui permet de définir des façon précise la sémantique des opérations et du langage de définition des politiques de contrôle de la concurrence.

Dans le même esprit de flexibilité qui a guidé notre proposition sur le langage de contrôle de l'ingénierie concurrente, nous avons proposé un système d'augmentation de l'information

contextuelle. Ce système distribue le contrôle de la concurrence en donnant à chaque utilisateur l'information nécessaire pour prendre localement les mesures adéquates pour éliminer ou diminuer les risques associés à des modifications concurrentes.

Nous avons implémenté CELINE, un système de support de l'ingénierie concurrente basée sur ces idées. CELINE a été mis à disposition des utilisateurs dans deux équipes de la société STMicroelectronics, dans une version implémentant un SAIC. L'adoption de CELINE par ces deux équipes a montré que le SAIC est une véritable aide aux utilisateurs.

Bibliographie

[ABB 98] B. Appleton, S. Berczuk, R. Cabera, and Robert Orenstein "Streamed Lines: Branching Patterns for Parallel Software Development," in *Pattern Languages of Programming* 1998.

[ABG 92] P Armenise, S Bandinelli, C Ghezzi, A Morzenti. Software Process Representation Languages: Survey and Assessment in Proceedings of the *Fourth International Conference on Software Engineering and Knowledge Engineering*, pages 455- 462. Capri, Italy. 1992.

[Arch] The Arch Revision Control System. http://www.seyza.com/gnuarchwiki/

[Ask 94] U. Asklund. Identifying conflicts during structural merge, in Proceedings of the Nordic Workshop Programming Environment Research, 1994.

[Bar 92] NS Barghouti, Supporting cooperation in the Marvel process-centered SDE in *Proceedings of the fifth ACM SIGSOFT symposium on Software Development Environments*, pages 21 – 31, Tyson's Corner, Virginia, United States. 1992.

[BBF 94] S. Bandinelli, M. Barga, A. Fuggetta, C. Ghezzi, and L. Lavazza. SPADE An Environment for Software Process Analysis, Design and Enactment. *Software Process Modeling and Technology*, Research Studies Press, Taunton, 1994.

[BFG 93] S. Bandinelli, A. Fuggetta, and Carlo Ghezzi. Software Process Model Evolution in the SPADE Environment in *IEEE Transactions on Software Engineering. Special Issue on Process Evolution*, December 1993

[BFGR 93] S. Bandinelli, A. Fuggetta, and S. Grigolli. Process Modeling-in-the-large with SLANG. In *Proceedings of the 2nd International Conference on the Software Process*, Berlin, Germany, pages 75-83. 1993

[BK 91] N Barghouti, G Kaiser. Scaling Up Rule-Based Software Development Environments. *Proceedings of the 3rd European Software Engineering Conference*. Pages: 380 – 395. 1991

[BS 89] L. Bannon, K. Schmidt. CSCW: Four Characters in Search of a Context, in *Proceedings of the First European Conference on Computer Supported Cooperative Work* 1989

[Buf 95] J. Buffenbarger, “Syntactic Software Merging”, in Software *Configuration Management: Selected Papers SCM 4 – SCM 5*, pages 153-172, 1995

[CGM 98] G. Canals, C. Godart, P. Molli, and M. Munier, A Criterion to Enforce Correctness of Indirectly Cooperating Applications, in *Information Sciences*, vol. 110/3-4, pp. 279-302, 1998.

[CJ 99] R. Conradi, L. Jaccheri (Editor) Cooperation Control in PSEE. in *Software Process: Principles, Methodology, Technology*. Editor. JC Derniame. Pages 28-52, 1999

[CKO 92] B Curtis, M Kellner, J Over. Process Modeling in *Communications of the ACM* Volume 35 , Issue 9, pages 75 - 90 , 1992

[CMM 93] Capability Maturity Model for Software. M. Paulk, C. Weber, S, Garcia, M.B. Chrissis, M. Bush. Software Engineering Institute, CMU/SEI-93-TR-25, DTIC Number ADA263432, 1993

[Coc 01] A. Cockburn, Agile Software Development. Reading, Massachusetts: Addison Wesley Longman, 2001.

[CW 98] Conradi and B. Westfechtel. Version Models for Software Conguration Management in *ACM Computing Surveys*, vol 30(2), pages 232-282, 1998.

[Darcs] Source code management system DARCS. http://abridgegame.org/darcs

[DB 92] P. Dourish, V. Bellotti, Awareness and Coordination in Shared Workspaces, in *Proceedings of CSCW 92 - Sharing Perspectives* pages 107-114, Toronto, Canada, ACM 9 Press, 1992

[Dou 95] P Dourish. The Parting of the Ways: Divergence, Data Management and Collaborative Work in *Proceedings of the ECSCW'95* pages 215-230, Stockholm, 1995

[DQS 95] D. Decouchant, V. Quint, M.R. Salcedo. Structured cooperative editing and group awareness in *Proceedings of the HCI International'95, 6th International Conference on Human-Computer Interaction* (July 1995)

[DQS 96] D. Decouchant, V. Quint, M.R. Salcedo, Structured and Distributed Cooperative Editing in a Large Scale Network, in *R. Rada, Groupware and Authoring*, Academic Press chapitre 13, pages 265-295. 1996

[EB 92] N. Belkhatir, J. Estublier, Process Centered SEE and Adele in *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering*, 1992.

[EC 94] J. Estublier. The Adele Configuration Manager. In Walter F. Tichy, editor, *Configuration Management*. Trends in Software. John Wiley and Sons, 1994

[EG 01] J. Estublier. S. Garcia. G. Vega. Defining and Supporting Concurrent Engineering Policies in SCM. in Proceedings of the Tenth International Workshop on Software Configuration Management, 2001.

[EG 05] J. Estublier, S. Garcia. Process model and awareness in SCM. in Proceedings of the 12th international workshop on Software configuration management. Pages: 59 – 74. Lisbon, Portugal.

[EG 06] J. Estublier, S. Garcia. Concurrent Engineering Support in Software Engineering. Proceedings of Automated Software Engineering conference. Tokyo, Japan, 2006

[EG 91] W. Emmerich, V. Gruhn. FUNSOFT nets: a Petri-net based software process modeling language. *Proceedings of the Sixth International Workshop on Software Specification and Design*, pages 75-184. 1991.

[ELC 02] J Estublier, D. Leblang, G Clemm, R Conradi, W. Tichy, A. van der Hoek, D. Wiborg-Weber. Impact of the research community on the field of software configuration management. in *Proceedings of the 24rd International Conference on Software Engineering*, 2002.

[Elm 92] A. Elmagarmid (Editor) Database transaction models for advanced applications. ISBN:1-55860-214-3. 1992

[Est 00] J. Estublier. Software Configuration Management: A Roadmap, in *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, ACM Press, 2000.

[Est 01] J. Estublier. Work space management in software engineering environments n I. Sommerville, editor, Proceedings of 6th International Workshop on Software Configuration Management, LNCS, Berlin, Germany, March 25-26 1996

[Est 96] J. estublier “Work Space Management in Software Engineering Environments” in Proceedingsof the 6th International Workshop on Software Configuration Management, 1996

[FKL 88] R. Fish, R. Kraut, M. Leland. Quilt: a collaborative tool for cooperative writing Conference on Supporting Group Work. In *Proceedings of the ACM SIGOIS and IEEECS TC-OA 1988 conference on Office information systems*. Pages: 30 – 37, 1988

[Fug 00] A. Fuggeta. Software Process: A Roadmap, in *Proceedings of 22nd International Conference on Software Engineering, The Future of Software Engineering*, ACM Press, 2000.

[GHB 01] Godart, C., G. Halin, J.-C. Bignon, C. Bouthier, P. Malcurat, P. Molli. Implicit or

Explicit Coordination of Virtual Teams in Building Design in *Computer-Aided Architectural Design Research In Asia (CAADRIA'01)*. Australia, 2001.

[Gobby] The gobby collaborative editor. http://darcs.0x539.de/trac/obby/cgi-bin/trac.cgi

[God 94] C. Godart , "Tutorial : Les outils du travail coopératif. Un point de vue ingénierie des données". *18ème Journées Bases de Données Avancées* – BDA'02, Evry, France, Octobre. 2002.

[God 99] C. Godart (Editor) Cooperation Control in PSEE. in *Software Process: Principles, Methodology, Technology*. Editor. JC Derniame. Pages 117-164, 1999

[GPS 99] C. Godart, O. Perrin, and H. Skaf, COO: A workflow operator to improve cooperation modeling in virtual processes, in *9th International Workshop on Research Issues in Data Engineering Information Technology for Virtual Entreprises* (RIDE VE'99), 1999.

[Gru 02] V. Gruhn. Process-Centered Software Engineering Environments. A Brief History and Future Challenges in *Annals of Software Engineering*, Vol 14; Number 1/4, pages 363-382. 2002

[Grud 94] J. Grudin. Computer Supported Cooperative Work: History and Focus. *Computer*. Volume 27 , Issue 5 pages 19-26. 1994

[Hum 89] W. S. Humphrey. Managing the Software Process. Addison-Wesley, Reading MA, 1989.

[JC 93] M.L. Jaccheri, R. Conradi, Techniques for Process Model Evolution in EPOS. in *IEEE Transactions on Software Engineering*, 19:12, December 1993

[JPS 94] G Junkermann, B Peuschel, W Schafer, S Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment in *Software Process Modelling and Technology*, John Wiley and Sons, New York, 1994, 103-130.

[KBF 88] GE Kaiser, NS Barghouti, PH Feiler, RW Schwanke. Database support for knowledge based engineering environments in *IEEE Expert: Intelligent Systems and Their Applications*, vol. 03, no. 2, pages 18-23, 26-32, 1988.

[KFP 88] G. E. Kaiser, P. Feiler and S. Popovich. Intelligent Assistance for Software Development and Maintenance in IEEE Software 5(3):40-49, May, 1988.

[LCL 94] P. Lowry, A. Curtis, M. Lowry. Building a taxonomy and nomenclature of collaborative writing to improve interdisciplinary research and practice in *Journal of Business Communication vol* 41, 66-99. 1994

[Leb 94] D. B. Leblang. The CM Challenge: Configuration Management that Works in *Configuration Management*, edited by W. Tichy; J. Wiley and Sons. 1994

[Men 02] T. Mens. A state-of-the-art survey on software merging in *IEEE transactions on software engineering.* Volume: 28(5) pages 449-462. 2002.

[MJ 05] K. J. Molokken-ostvold, M. Jorgensen. A Comparison of Software Project Overruns - Flexible vs. Sequential Development Models in *IEEE Transactions on Software Engineering*, 31,9, pages 754-766. 2005

[MO 92] L. McGuffin, L, G. Olson. ShrEdit: A Shared Electronic Workspace in *CSMIL Technical Report*, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, 1992.

[Mon 99] C. Montangero (Editor) Cooperation Control in PSEE. in *Software Process: Principles, Methodology, Technology*. Editor. JC Derniame. Pages 1-13, 1999

[Monotone] Distributed version control system Monotone. http://www.venge.net/monotone/.

[MSB 01] P. Molli, H. Skaf-Molli, C. Bouthier: *State Treemap: an Awareness Widget for Multi-Synchronous Groupware* in 7th International Workshop on Groupware (CRIWG'01). 2001

[MSO 02] P. Molli, H Skaf-Molli, G. Oster. Divergence Awareness for Virtual Team through the Web, in *Integrated Design and Process Technology*, Pasadena, CA, USA. Society for Design and Process Science, June 2002.

[NR 04] S. Noël, and JM Robert, Empirical study on collaborative writing: What do co-authors do, use, and like? in *Computer Supported Cooperative Work: The Journal of Collaborative Computing* vol 13 (1) pages 63-89. 2004

[NRZ 92] M. Nodine, S. Ramaswamy, and S. Zdonik. A cooperative transaction model for design databases. in [Elm 92], pages 53-85, 1992.

[Ost 87] L.J. Osterweil. Software Processes Are Software Too. in *Proceedings of the Ninth International Conference of Software Engineering*, pages 2-13, Monterey CA, US. March 1987

[Per 89] D. Perry (editor). 5th International Software Process Workshop: Experience with Software Process Models. IEEE Computer Society Press, Kennebunkport ME, 1989.

[PKH 88] C. Pu, G. E. Kaiser and N. Hutchinson. Split-Transactions for Open-Ended Activities in *Francois Bancilhon and David J. Dewitt (editor), 14th International Conference*

on Very Large Data Bases, pages 26-37. Los Angeles CA, August, 1988

[PSV 98] D E Perry, H P. Siy, L. G. Votta. Parallel Changes in Large Scale Software Development: An observational Case Study in the *Proceedings of the 20th international conference on software engineering*. 1998.

[SBC 00] H. Schuster, D. Baker, A. Cichocki, D. Georgakopoulos, M. Rusinkiewicz. The collaboration management infrastructure. in *proceedings of the IEEE International Conference on Data Engineering (ICDE)*, San Diego CA, USA, February 2000.

[Sch 91] K. Schmidt. Riding a Tiger, or Computer Supported Cooperative Work in L. Bannon, M. Robinson and K. Schmidt (eds.):ECSCW '91. *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, Kluwer Academic Publishers, Amsterdam. 1991

[SHH 92] N. Streitz, J. Haake, J. Hannemann, A. Lemke, W. Schler, H. Schütt, M. Thüring, SEPIA: A Cooperative Hypermedia Authoring Environment in *Proceedings of the European Conference on Hypertext and Hypermedia ECHT'92*, edited by D. Lucarella, J. Nanard, M. Nanard, P. Paolini, pages 11-22, ACM Press, 1992

[SNV 03] A. Sarma, Z. Noroozi, A. van der Hoek. Palantir: Raising awareness among coniguration management workspaces. in *Proceedings of the International Conf. on Software Engineering*, pages 444-454, 2003.

[Synch] The Synchronicity Developer Suite ™. http://www.synchronicity.com/products/developer/developer.html

[Tic 85] W Tichy,RCS - A System for Version Control in *Software-Practice and Experience*,vol. 15,7, pages 637-654, 1985.

[WebDav] The WebDav specs. http://www.webdav.org/specs/

[Wiki] The wiki websites. http://en.wikipedia.org/wiki/Wiki.

[WS 92] G Weikum, HJ. Schek. Concepts and Applications of Multilevel Transactions and Open Nested Transactions in *Concepts and applications of multilevel transactions and open nested transactions*, pages 515 – 553 Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1992. ISBN:1-55860-214-3 1992

[XML] Extensible Markup Language, The XML Specification. W3C Recommendation, 2004

[XSD] XML Schema, W3C Recommendation, 2004.