N° d'ordre:28/2012-M/Inf

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ETPOPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE
LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ DES SCIENCES ET DE LA TECHNOLOGIE
HOUARI BOUMEDIENE
FACULTÉ D'ÉLECTRONIQUE ET D'INFORMATIQUE

MEMOIRE
Présenté pour l'obtention du diplôme de Magister
En : Informatique

Spécialité : Ingénierie de logiciels et Génie des Procédés.

Par : DJERBI RACHID

Thème

FORMALISATION ET IMPLANTATION D'UNE POLITIQUE DE SECURITE DANS LE SYSTÈME FoCaLiZe

Soutenu publiquement, le 30/06/2012, devant le Jury composé de :

Mr - Belkhir Abdelkader, Professeur à l'U.S.T.H.B Président
Mr - Ben-Yelles Choukri-Bey, Professeur à l'U.P.M.F (Grenoble 2) Directeur de mémoire
Mlle – Selmi Nabila, Maître de conférence A à l'U.S.T.H.B Examinatrice
Mme – Zaouche Djaouida, Maître de conférence B à l'U.S.T.H.B Invitée

Remerciements

Je remercie tous les membres du jury pour avoir accepté d'examiner ma thèse.

Je tiens à remercier mon directeur de thèse, le Professeur BEN-YELLES Choukri-Bey, pour m'avoir fait confiance en me proposant ce sujet.

J'exprime ma profonde reconnaissance et mes vifs remerciements aux membres de l'équipe FoCaLiZe de l'USTHB : Mr BELKHIR Abdelkader, Mme DAHMANI Djaouida, Mr KADDOURI Lies et Mr ABBAS Messaoud pour leurs précieux conseils jusqu'à l'aboutissement de ce travail.

Je remercie également les membres du forum FoCaLiZe et en particulier : Mr Renaud Rioboo, Mr Mathieu Jaume, Mr Lionel Habib et Mr Charles Morisset.

Merci à toute ma famille pour leur soutien indéfectible, à mon ami et collègue BOUBEZARI Abderrahim et à toutes les personnes qui ont participé de près ou de loin à la réalisation de cette thèse.

Enfin, une pensée à Mr BENABADJI Karim, ancien membre de l'équipe FoCAL de l'USTHB, et qui nous a tragiquement quitté depuis quelque temps. Un grand merci pour son encadrement qui m'a permis de réaliser ce travail.

Résumé

Tout système informatique doit avoir une politique de sécurité gérant les différents contrôles et assurant la confidentialité, l'intégrité et la disponibilité de l'information. La mise en place d'une politique de sécurité se base sur la formalisation mathématique et logique de ses règles. Pour cela, nous disposons de l'atelier FoCaLiZe, un environnement de programmation certifiée basé sur la notion de preuve formelle. Dans ce document, nous proposons une démarche de formalisation et d'implantation de la politique de sécurité Clark-Wilson dans le système FoCaLiZe. Notre démarche permet de spécifier les contraintes et règles de sécurité de cette politique en propriétés et théorèmes FoCaLiZe, et de prouver la sureté de l'évolution du système modélisé, et ce en utilisant l'outil de preuve automatique Zenon associé à FoCaLiZe.

Mots-clés :

Politique de sécurité, Clark-Wilson, méthodes formelles, preuve, Coq, Zenon, FoCaLiZe

Sommaire

Introduction 5

Chapitre 1: Le système FoCaLiZe 7
Introduction 7
I. Concepts de Focalize 8
II. Le langage FoCaLiZe 19
II.1- Compilation et génération de documentations 19
II.2- Outils de preuve dans FoCaLiZe 21
COQ 22
Zenon 23
Conclusion 24

Chapitre 2: La Sécurité informatique 25
Introduction 25
I. Politiques de sécurité 26
I.1 - La politique de sécurité à matrice d'accès (HRU) 27
I.2 - La politique de sécurité de Bell et LaPadula 30
I.3 - La politique de sécurité de Biba 33
I.4 - La politique de sécurité de Clark-Wilson 34
Conclusion 40

Chapitre 3: Formalisation de Clark-Wilson dans FoCaLiZe 42
Introduction 42
I. Modèle générique 43
I.1. Paramètres 43
I.2. Structure générale (Framework) 45
I.3. Rôles 46
I.4. Contrôles 47
I.5. Modèles 48
I.6. Systèmes 49
I.7 Aspect preuve 63
II- Tableau récapitulatif 73
Conclusion 75

Chapitre 4: Étude de cas 76
Introduction 76
I. Scénario 76
II. Démarche 77
III. Eléments à concrétiser 78
IV- Application dans FoCaLiZe 83
Conclusion 90

Conclusion générale 91

Bibliographie 93

Introduction

Tout système informatique doit avoir une politique de sécurité adjacente décrivant les règles de sécurité qui doivent être respectées lors de sa mise en place.
Une politique de sécurité est l'élaboration d'un ensemble de règles d'accès en lecture et/ou écriture aux données. Ces règles doivent déterminer quelles sont les données à sécuriser, de quelle façon elles seront protégés et comment les gérer. Les politiques de sécurité se divisent entre deux grandes catégories : de la confidentialité de données (HRU [HRU76], BLP [BLP73]) qui protègent les données contre toute violation et lecture non autorisée, et de l'intégrité de données (Biba [Bib77], Clark-Wilson [CW87]) qui interdisent toute modification malveillante ou accidentelle des données.
La mise en place des politiques de sécurité demande une grande discipline, depuis les spécifications formelles réalisées, jusqu'aux preuves des propriétés et des théorèmes qui doivent être réalisées. Pour atteindre ces buts, les équipes du LIP6, de l'INRIA et du CNAM ont développé et utilisé un atelier d'aide au développement de "bibliothèques certifiées", appelé "FoCaLiZe" [FOC06]. Cet atelier permet de spécifier, programmer et prouver que ces programmes sont corrects vis-à-vis des spécifications du système en question.
Notre objectif est d'utiliser le système FoCaLiZe pour formaliser et implanter une politique de sécurité. Pour cela, nous avons adopté une démarche pour montrer comment, avec cet atelier nous vérifions les propriétés correspondant aux différentes règles de la politique de sécurité et comment nous appliquons des preuves formelles sur la sûreté, la cohérence et la validité de notre formalisation.
Cette démarche passe d'abord par la traduction des différentes règles de la politique de sécurité choisie (Clark-Wilson) dans le système FoCaLiZe, puis par l'utilisation des techniques de vérification de preuve formelle offertes par le système. Pour cela, nous modélisons l'ensemble des entités (passives et actives) du système par une hiérarchie d'espèces où toute règle de sécurité sera transcrite en une propriété ou un théorème à prouver dans les espèces dérivées. Ce processus de preuve se base essentiellement sur les outils d'aide à la preuve de FoCaLiZe, Zenon [DOL04] et COQ [COQ97].

Après avoir obtenu un cadre générique constitué d'une hiérarchie d'espèces, elles seront instanciées par des collections pour implanter un exemple réel. Nous avons montré comment le système modélisé peut passer d'un état à un autre en exécutant des ''requêtes'' émises par des ''sujets'' sur des ''objets'', et en calculant les ''décisions'' correspondant aux paramètres de sécurité.

Notre étude se décompose en quatre chapitres. Le premier donne une présentation de l'atelier FoCaLiZe, ses concepts de base, avec quelques exemples de la syntaxe et la sémantique pour chaque concept. Le deuxième chapitre présente un état de l'art sur la sécurité informatique, en introduisant quelques politiques de sécurité parmi les plus connues dans la littérature, avec une présentation détaillée de la politique de sécurité que nous avons choisie de formaliser : Clark-Wilson. Dans le troisième chapitre, la conception de la politique de sécurité Clark-Wilson dans l'atelier FoCaLiZe a été réalisée et un cadre générique proposé. Enfin, le quatrième chapitre concrétise notre démarche par une étude de cas sur un exemple réel.

Chapitre I

Le système FoCaLiZe

Introduction

L'intérêt principal de la programmation certifiée [BOU00] est de donner des preuves mathématiques sur la consistance, la sûreté et la cohérence d'un système informatique. Pour cela, on doit disposer d'un environnement de spécification, programmation et de preuve que ces programmes sont corrects vis-à-vis les spécifications du système en question. Pour atteindre ce but, les équipes du laboratoire LIP6, de l'INRIA et du CNAM ont développé un atelier d'aide au développement de "bibliothèques certifiées". Cet atelier porte le nom "**FoCaLiZe**" [FCLr09].

L'atelier **FoCaLiZe** est un système qui permet à la fois d'écrire des programmes et de prouver certaines propriétés qu'ils vérifient. Il s'appuie sur le langage **OCaml** [REM02] (un environnement de programmation fonctionnelle) pour la partie calculatoire et sur **Coq** [COQ97], [Coq10] (un environnement pour développer des preuves formelles) pour la partie certification.

L'objectif principal de l'atelier FoCaLiZe est de concevoir un environnement de développement intégré logiquement permettant de prouver les propriétés et les théorèmes introduits par le programmeur pour obtenir des implantations correctes et cohérentes.

Le développement de l'atelier FoCaLiZe a été initié en 1998 par T.Hardin et R.Rioobo avec S.Boulmé sous le nom de projet FOC (Formel OCaml et Coq) [FOC03], [RIO03], [PRE03] pour réaliser d'une manière incrémentale un environnement de calcul formel certifié en partant d'une spécification abstraite jusqu'à d'une implantation concrète formellement vérifiée, dans cette version les preuves des différentes propriétés et théorèmes étaient 'prédestinées' spécifiquement au Coq. La version suivante porte le nom FOCAL [FOC06] où un prouveur automatique appelé Zenon [DOL04], [BDD07], [DOL10], a été introduit et qui utilise une spécification de première ordre en entré et retourne une preuve formelle complète en sortie directement vérifiable en COQ. La dernière version du compilateur, appelé FoCaLiZe [FCLr09], a été distribuée et est disponible à l'adresse http://focalize.inria.fr/. Dans

cette version, des améliorations syntaxiques et sémantiques ont été introduites afin d'alimenter la bibliothèque par des nouvelles structures 'réutilisables' comme la notion des ensembles finis avec la preuve de toutes les propriétés et théorèmes définis.

Dans ce chapitre, nous allons présenter les principaux concepts du langage FoCaLiZe, en utilisant plusieurs exemples de développement concret.

I. Concepts de Focalize

Les deux principaux constructeurs de l'environnement FoCaLiZe sont l'espèce et la collection. Les espèces sont utilisées au niveau conception (spécification), et les collections servent à implémenter des espèces complétées.

Plus précisément, un développement FoCaLiZe est une hiérarchie d'espèces qui sont soumis à une succession de raffinements (principalement héritage et paramétrage) pour aboutir à des espèces complétement définies.

Une espèce regroupe un ensemble de méthodes de trois types (type support, méthodes calculatoires et méthodes logiques) :

- le type support donne la représentation des entités manipulées par l'espèce ;
- des déclarations de fonctions, ou signatures, dont on ne connait que le type et qui seront définies plus tard. Ces déclarations peuvent être utilisées dans la définition d'autres méthodes ;
- des définitions de fonctions, correspondant à l'implémentation d'opérations sur les entités de l'espèce ;
- des propriétés exprimant des conditions que les entités de l'espèce doivent satisfaire. Comme dans le cas des signatures, c'est une méthode déclarée qu'il faudra ensuite compléter par une preuve ;
- des théorèmes qui correspondent à des propriétés munies de leurs preuves.

On remarque donc que les méthodes peuvent être soit déclarées (signatures et propriétés) soit définies (fonctions et preuves).

I.1- Notion de type dans FoCaLiZe

L'atelier FoCaLiZe se base dans ses calculs sur la vérification des types des données et leur compatibilité. Les types suivants sont prédéfinis dans la bibliothèque FoCaLiZe :

- int : les entiers naturels.
- float : les nombres réels.
- bool : les booléens.
- char : les caractères.
- string : les chaines de caractères.

L'atelier FoCaLiZe dispose aussi de mécanismes de définition de nouveaux types :

1- Alias : consiste à donner un 'surnom' (ou une abréviation) à une combinaison de types déjà définis.

   Par exemple, pour exprimer le type 'date' on pourra utiliser la notion d'alias comme étant un tuple de cinq composants (année, mois, jours, heure, minute) de type 'int' chacun, la syntaxe est donc comme suit :

```
type date = alias (int* int* int* int* int) ;;
```

   Cette instruction définit une abréviation pour remplacer toutes les occurrences de type (int* int* int* int* int) par le nom date choisi par le programmeur, et ce dans un souci de lisibilité et de facilité d'écriture. Il faut l'introduite au niveau top-level de la spécification (en dehors des espèces) pour pouvoir utiliser le type date dans la spécification des espèces ultérieurement.

2- Types énumérés : Cette définition de type permet de rassembler une liste bien déterminée de valeurs différentes pour bâtir de nouveaux types. Par exemple, nous pouvons définir la notion de couleur, en se basant sur les trois valeurs rouge, vert et jaune :

```
type couleur = | Rouge | Vert | Jaune ;;
```

3- Types structurés (type d'enregistrement) : Cette notion permet de définir un nouveau type permettant de disposer d'une structure de données typées et *ordonnées* en nommant les différents composants:

```
type etat_point = {x : int ; y : int ; z : int ; temperature : int ; pression : int } ;;
```

I.2. Type support

Le type support (ou représentation) d'une espèce définit le type des 'entités' manipulées par l'espèce. Sa définition utilise le mécanisme d'alias, qui lui associe un type. Il est introduit par le mot clé *representation* :

```
representation = int*int ;
```

Sa définition peut être différée :

« Le type support peut ‘’ne pas être encore défini’’ dans une espèce, ce qui signifie que la structure de données réelle encapsulée par l'espèce n'a pas besoin d'être connue à ce point de raffinement du développement» [Phil 11].

Comme pour les autres méthodes, le type support n'est pas polymorphique (il ne contient pas de variables de type), mais on verra plus tard qu'il peut utiliser des paramètres de collection.

Pour terminer, on notera que chaque espèce possède une représentation unique.

I.3- Les espèces

L'entité primitive et basique d'un développement FoCaLiZe est l'espèce. Les espèces sont les nœuds de la hiérarchie des structures définissant une librairie d'un développement FoCaLiZe [FOC06].

Une espèce peut être, en première approche, assimilée à une classe dans la programmation orientée objet. En effet, une espèce spécifie une collection d'entités qui portent les mêmes propriétés.

L'implémentation des bibliothèques dans le développement FoCaLiZe peut être effectuée étape par étape et d'une manière progressive, on commence par la définition d'un ensemble d'espèces à partir d'une première abstraction d'un système donné, puis nous raffinons au fur et à mesure ces espèces pour produire d'autres espèces plus concrètes.

Nous présentons maintenant, à travers un exemple, les différents concepts qui décrivent le corps d'une espèce. L'espèce 'Point' nous permettra d'illustrer tout au long de notre présentation, et de manière moins formelle, les méthodes composant une espèce.

Exemple 1 :

```
species Point =
        representation = int*int ;
        signature x : Self -> int ;
        signature y : Self -> int ;
        signature deplace : Self -> int -> int -> Self ;
        signature distance : Self -> Self -> float ;
        let  egale (a : Self, b : Self) : bool =
```

```
        if ((x(a) = x(b)) && (y(a) = y(b)) then true else false;
    signature  appartient_axe_x : Self -> bool ;
    property appartient_axe_x_spec :  all p : Self ,
        appartient_axe_x (p) <-> (y(p) = 0 ) ;
    theorem egale_reflexive : all x : Self, egale (x , x)
        proof  = by definition of egale
    …
end ;;
```

Dans cet exemple, l'espèce **Point** permet de spécifier les points du plan dans un repère orthogonal.

La représentation (int*int) de l'espèce Point définit la structure de données des instances de l'espèce, elle permet de représenter des paires d'entiers modélisant l'abscisse (le composant gauche) et l'ordonnée (le composant droit) d'un point.

Dans les autres méthodes, qui seront décrites dans la suite, on remarquera l'utilisation du mot clé 'Self' pour faire référence au type support.

Dans le processus de développement FoCaLiZe, on a le choix entre déclarer et définir les méthodes. Précisons que l'intérêt principal de déclarer des méthodes est de retarder leurs définitions tant qu'elles ne sont pas indispensables, tout en donnant au développeur la possibilité de les utiliser dans la déclaration ou même la définition d'autres méthodes, sans avoir à connaître son mécanisme de calcul.

I.4- Fonctions et signatures

Chaque espèce dispose d'un ensemble de fonctions (méthodes calculatoires) manipulant ses instances (entités). Quand une fonction n'est que déclarée, elle prend le nom de signature, si elle est définie, on parlera de fonction.

Signature

La déclaration d'une fonction, introduite par le mot clé **signature** est un énoncé de son nom, le type de ses paramètres (s'ils existent) et du type du résultat. C'est une fonction qui doit encore être définie et implantée lors du raffinement des espèces.

```
signature distance : Self -> Self -> float ;
```

La signature distance représente donc une fonction, qui à deux points (paires de int), associe un réel. Self représente le type support de l'espèce courante Point.

Fonctions

Une définition de fonction, introduite par le mot clé **let,** fournit l'implémentation d'une fonction. Elle peut être nouvellement définie ou être une implémentation d'une signature précédemment donnée. Dans le cas d'une redéfinition d'une méthode, la dernière définition donnée sera prise en compte et donc toutes les définitions précédentes seront supprimées.

Le corps de la fonction peut utiliser d'autres méthodes (définies ou déclarées).

L'intérêt principal de ce type de définition est d'énoncer un algorithme.

```
let  egale (a : Self, b : Self) : bool =
          if ((x(a) = x(b)) && (y(a) = y(b)) then true else false;
```

Dans cet exemple, nous avons défini la fonction 'egale' en donnant son type et son corps. Elle est paramétrée par deux éléments de type 'Self' et retourne un résultat de type booléen vérifiant la superposition des deux points en paramètre, la valeur de l'égalité entre deux points quelconques dépend de l'égalité de leurs ordonnées et de leurs abscisses.

Donc la fonction définie 'egale' permet de décider de l'égalité entre deux points de notre espèce Point. On remarque à ce niveau que les méthodes 'x' et 'y' ne sont pas encore définies.

I.5- Propriété, Théorème et preuve

Après avoir introduit les méthodes calculatoires, nous allons nous intéresser aux méthodes logiques (non calculatoires) que sont les propriétés et les théorèmes.

Propriétés

Une *propriété* est une formule du premier ordre, introduite par le mot clé **property**. A ce niveau la preuve n'est pas encore fournie, c'est une déclaration qui devra être complétée ultérieurement.

```
species  Point  =
     …
     signature  appartient_axe_x : Self -> bool ;
     property appartient_axe_x_spec :  all p : Self ,
          appartient_axe_x (p) <-> (y(p) = 0 ) ;
     …
end;;
```

Avant d'introduire la propriété appartient_axe_x_spec, nous avons enrichi l'espèce 'Point' par la déclaration de la fonction appartient_axe_x permettant de décider de l'appartenance d'un point à l'axe des abscisses.

Théorème

Un théorème est introduit par le mot clé *theorem* suivi par un énoncé et une preuve formelle que cet énoncé est vérifié dans le contexte de l'espèce. La preuve sera traitée par FoCaLiZe et ultimement vérifiée par Coq.

```
species  Point =
        …
        theorem egale_reflexive : all x : Self, egale (x , x)
        proof  = by definition of egale ;
        …
End ;;
```

Cet exemple exprime la propriété de réflexivité, avec sa preuve.

L'écriture d'une preuve peut être faite de plusieurs manières :

- **assumed** : la preuve est omise, elle représente les propriétés qui peuvent être considérées comme des axiomes pour prouver d'autres propriétés ou théorèmes.
- En **'FoCaLiZe proof language**' : script de preuve produit par Zenon [BDD07], [DOL10], un prouveur automatique, développé par D. Doligez, son objectif est d'automatiser et d'aider le programmeur à mener la preuve à son terme.
- En '**Coq**' : script Coq intégré dans le code source de FoCaLiZe, il demande de la part du concepteur une maitrise de l'outil, et surtout d'élaborer la preuve soi-même.

I.6- Dépendances

Les méthodes d'une espèce peuvent se référencer entre elles. Lorsqu'une méthode m1 contient un appel à une méthode m2 de l'espèce courante (ou d'une autre espèce), on dira que m1 dépend de m2. Dans le développement FoCaLiZe, la vérification des dépendances est obligatoire (automatiquement réalisée par le système pendant la compilation FoCaLiZe) pour assurer la cohérence et l'absence de cycles.

On distingue deux types de dépendances :

- La decl-dépendance : une méthode m1 decl-dépend de m2 si la définition de m1 ne dépend que du type de m2.

- La def-dépendance : une méthode m1 def-dépend de m2 si la définition de m1 dépend de la définition de m2.

Il faut interdire les def-dépendances dans la déclaration d'une méthode, en effet :

«La redéfinition d'une méthode risque d'invalider les preuves utilisant des propriétés de l'ancien corps de cette méthode. Donc, toutes les preuves dépendants réellement de la définition initiale seront effacées par le compilateur et devront être refaites dans le contexte de la nouvelle définition» [phil 11].

I.7- Héritage

Les espèces peuvent ne pas être isolées les unes par rapport aux autres, mais regroupées dans une hiérarchie, pouvant avoir plusieurs racines, et qui exprime le fait qu'une structure mathématique est le plus souvent définie à partir de structures préexistantes, en l'enrichissant de nouvelles opérations et/ou propriétés. Ce mécanisme s'apparente ainsi à l'héritage des langages orientés objet. Avec ce mécanisme d'héritage, on pourra définir à n'importe quel moment une nouvelle espèce à partir d'une ou plusieurs autres espèces (héritage multiple). Dans ce cas, on pourra utiliser directement les méthodes et propriétés déclarées et/ou définies sans les redéclarer et/ou redéfinir. Le mécanisme d'héritage permet de redéfinir des méthodes à condition de conserver leurs typages. De même on peut modifier la preuve d'un théorème, dont on conserve l'énoncé.

Les nœuds de plus haut niveau de la hiérarchie correspondent à l'étape de spécification du modèle tandis que les niveaux inferieurs correspondent à l'étape de l'implémentation. Avec ce mécanisme, on se rapproche étape par étape et d'une manière progressive de l'implémentation complète.

La notion d'héritage dans le système FoCaLiZe a deux objectifs principaux à savoir :

- La définition des nouvelles espèces plus complexes avec des nouvelles fonctionnalités à définir au fur et à mesure tout en gardant celles qui ont été définies au niveau de ses parents avec la possibilité d'en redéfinir certaines.
- Avec ce mécanisme d'héritage, on se rapproche de plus en plus vers un modèle ne contenant que des espèces complètes où toutes les fonctions sont définies et toutes les propriétés sont prouvées.

La contrainte la plus importante que le développeur FoCaLiZe doit respecter est que le type des méthodes héritées ne doit pas être modifié pendant l'opération de redéfinition, « cette exigence est le prix à payer pour s'assurer de la cohérence du modèle FoCaLiZe et donc du logiciel développé» [Phil 11].
Soit l'espèce Point_concret suivante :

Exemple 2 :

```
species Point_concret =
        inherit Point ;
        let new_point( a:int , b:int) = (a , b) ; (* constructeur *)
        let x(p) = fst(p) ;
        let y(p) = snd(p) ;
        let deplace (p : Self , a:int , b:int) = ( x(p) + a , y(p) + b ) ;
        let affiche_point ( p : Self) = " X = " ^ string_of_int(x(p)) ^ " Y = " ^
                string_of_int(y(p)) ;
        let egale ( p1 , p2 ) =  ( x(p1) = x(p2) ) && ( y(p1) = y(p2) ) ;
…
end ;;
```

Dans cet exemple, l'espèce ''Point_concret'', tout en héritant de l'espèce ''Point'',

- est enrichie par la fonction new_point, pour permettre la création de nouvelles instances, et la fonction affiche_point pour afficher les cordonnées d'un point ;
- et fournit l'implémentation des fonctions précédemment déclarées dans l'espèce parente Point.

Comme le système FoCaLiZe permet l'héritage multiple, une méthode peut être définie plusieurs fois dans des parents différents, dans ce cas l'espèce en question hérite de la méthode de l'espèce parente la plus à droite dans la clause '**inherit**' pendant la définition de l'espèce.

I.8- Interfaces, collections

Nous avons vu que dans le cadre d'un développement FoCaLiZe, certaines méthodes pouvaient ne pas être déclarées jusqu'à obtenir par raffinements successifs la définition de chaque méthode. Les situations extrêmes correspondent aux interfaces (aucune méthode n'est définie) et aux espèces complètes (toutes les méthodes sont définies)

Espèce complète

Une espèce est dite compète si toutes ses méthodes sont définies, en d'autres termes :

- la représentation est concrète
- toutes les déclarations sont définies
- chaque propriété est prouvée

Interface

Chaque espèce est associée à une interface dont toutes les méthodes ne sont que déclarées. Elle permet de disposer de toutes les fonctions et propriétés de l'espèce, sans se préoccuper des détails d'implémentation. Elle est donc obtenue en :

- rendant la représentation abstraite
- conservant les signatures et propriétés
- transformant les fonctions en signatures, et les théorèmes en propriétés.

Bien qu'une interface ait le même nom que l'espèce sous-jacente, il ne peut y avoir de confusion dans la mesure où une interface n'est utilisée que pour déclarer un paramètre de collection.

Collection

Une collection joue le rôle dual d'une interface, il s'agit d'une instance d'une espèce complète.

Une collection est construite à partir d'une espèce complète par abstraction de sa représentation. Les entités d'une collection ne peuvent être directement manipulées ou créées (type non accessible), elles ne peuvent être manipulées que par les méthodes des espèces implémentées.

Notons aussi, que contrairement aux interfaces, on peut créer plusieurs collections pour la même espèce. On impose cependant deux restrictions aux collections :

- On ne peut pas hériter d'une collection. Celle-ci représentant une structure mathématique particulière, elle ne peut en effet pas être raffinée.

- Une collection est vue à travers l'interface sous-jacente. On peut appeler des méthodes de la collection, mais on n'en connaît pas la définition.

```
collection Point_concret_collection =
                    implements Point_concret; end ;;
```

I.9- Paramétrage

Le mécanisme de paramétrisation permet de construire de nouvelles espèces à partir d'espèces existantes. On distingue deux types de paramétrage, le paramétrage par collection et le paramétrage par entité :

Paramétrage par collection

Le paramétrage par collection permet de simuler le polymorphisme, à défaut de pouvoir disposer de méthodes polymorphiques.

Nous définissons une troisième espèce, nommée Cercle, modélisée à partir de son centre et de son rayon. Nous utilisons l'espèce Point précédemment donnée comme paramètre de l'espèce Cercle.

Exemple 3 *:*

```
species Cercle (P is Point ) =
        representation = (P * float) ;
        signature centre : Self -> P ;
        signature rayon : Self -> float ;
        …
End ;;
```

Le paramètre de collection P est introduit par son nom suivi un nom d'interface Point (portant le même nom que l'espèce correspondante).

Dans cet exemple, le paramètre P nous permet de construire la signature de la méthode centre, qui à tout argument de l'espèce cercle retourne le point correspondant à son centre.

De même, la fonction rayon de signature permet de récupérer son rayon.

Paramétrage par entités

Dans ce second cas de paramétrage, le paramètre est instancié par une entité d'une collection.

Imaginons par exemple, que nous voulons créer une espèce pour mettre en œuvre des opérations géométriques sur les points, comme par exemple la projection des points par rapport un point bien déterminé. Pour s'assurer que l'espèce utilise toujours le même centre de projection, ce dernier doit être inclus dans l'espèce Point. Toutefois, l'espèce elle-même ne doit pas dépendre d'un point particulier de projection, ce point doit être un paramètre de l'espèce qui peut prendre des valeurs

différentes. Dans ce cas on aura besoin non seulement de sa valeur mais aussi sa représentation et des méthodes et opérations agissant sur le type de ce paramètre. Un tel paramètre est un paramètre appelé entité, le paramètre sera instancié par une entité de collection, collection qui doit également être déclarée en tant que paramètre de collection dans l'espèce.

Exemple 4 :

```
species Point_projete (P is Point, p1 in P)=
            inherit Point ;
      signature projeter : Self -> Self ;
…
end;;
```

Avec cet exemple, nous avons défini l'espèce des points projetés 'Point_projete' dotée de la fonction 'projeter' de type 'Self -> Self' qui projette un point par rapport l' ''entité'' en paramètre et retourne un autre point.

I.10- Mode de développement en FoCaLiZe

Dans le développement FoCaLiZe on doit respecter une certaine structure hiérarchique. On commence par créer les différentes espèces nécessaires, puis au fur et à mesure on les enrichit par les déclarations et définitions des méthodes qui manipulent les entités de ces espèces. A chaque nouvelle fonctionnalité on déclare les propriétés qui doivent être conformes avec les exigences de la spécification, ainsi que la déclaration et la démonstration des différents théorèmes qui peuvent être définis. Finalement on passe à l'implantation du modèle tout en définissant l'ensemble des collections dont on aura besoin pour instancier le monde des entités pour faire des calculs concrets.

Après avoir présenté les différentes briques de base de l'atelier FoCaLiZe, et après la mise en place des différents exemples accompagnant ces briques, nous traçons le schéma suivant pour illustrer les notions d'héritage, paramétrage et encapsulation (création de collections) entre les espèces présentées :

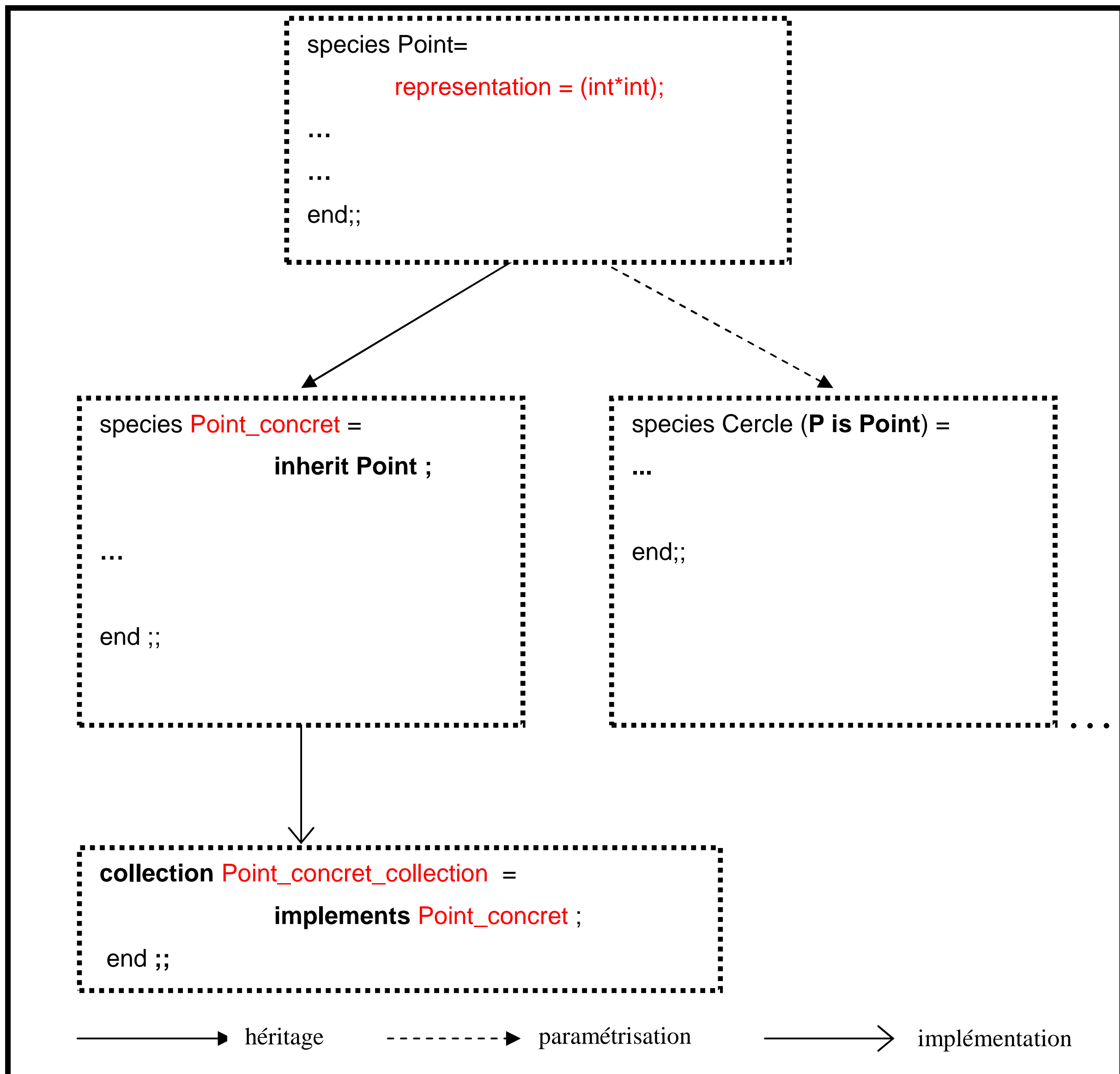

Figure 1.1: Hiérarchie FoCaLiZe..

II. Le langage FoCaLiZe

II.1- Compilation et génération de documentations

Un programme FoCaLiZe est compilé vers deux autres langages : OCAML et COQ avec la commande 'focalizec' qui fait appel aux commandes : Ocamlc, Zvtov et Coqc. Les différents fichiers qui peuvent être générés sont les suivants [FCLt09]:

Fichier généré	Généré par	Utilisé par	Description
.fo	focalizec	focalizec	Fichier object FoCaLiZe
.zv	focalizec	zvtov	Obligations de preuve
.ml	focalizec	ocamlc	Code source Ocaml
.pfc	zvtov	zvtov	Preuves cachées pour Zenon
.v	zvtov	coqc	Preuves Coq
.vo	coqc	coqc	Fichier object Coq
.cmi	ocamlc	ocamlc	Fichier d'interface Ocaml
.cmo	ocamlc	ocamlc	Code objet pour OCaml

Table 1.1 : génération de documentations

De plus, l'atelier FoCaLiZe fournit un code source pour un langage intermédiaire (FocDoc) permettant de créer des fichiers de documentation automatique sous différents formats (HTML, XML, LATEX …).
Il est à noter qu'il est possible de désactiver la génération du code Coq et/ou OCaml avec l'option « **-no-coq-code** » et/ou « **-no-ocaml-code** ».

Une étape préalable à la compilation est nécessaire au bon déroulement de celle-ci, c'est l'analyse du programme FoCaLiZe.
Pour pouvoir faire des preuves des différents algorithmes écrits dans un programme FoCaLiZe, celui-ci doit respecter un certain nombre de contraintes, elles ne sont pas forcément suffisantes pour assurer la correction des programmes, mais elles permettent de simplifier l'expression des propriétés à démontrer, et donc l'écriture des preuves correspondantes. Ces contraintes sont :

- Le typage **:** toutes les expressions doivent être bien typées. Les arguments donnés aux espèces paramétrées doivent avoir le type (ou l'interface) attendu.
- la redéfinition d'une méthode au cours d'un héritage ne doit pas en modifier le type.
- Le type support ne peut pas changer au cours de l'héritage.
- Lorsqu'une collection est créée à partir d'une espèce, toutes les méthodes de l'espèce doivent être définies (et non simplement déclarées) et toutes les propriétés doivent être prouvées.

L'analyse d'une espèce s'effectue autour de trois grands axes :

- l'héritage et en particulier la résolution des conflits en cas d'héritage multiple.
- l'analyse de dépendances et la détection de cycles.
- Le typage des méthodes définies.

La figure 1.2 suivante montre la chaîne de compilation d'un programme FoCaLiZe en précisant les fichiers produits à chaque étape.

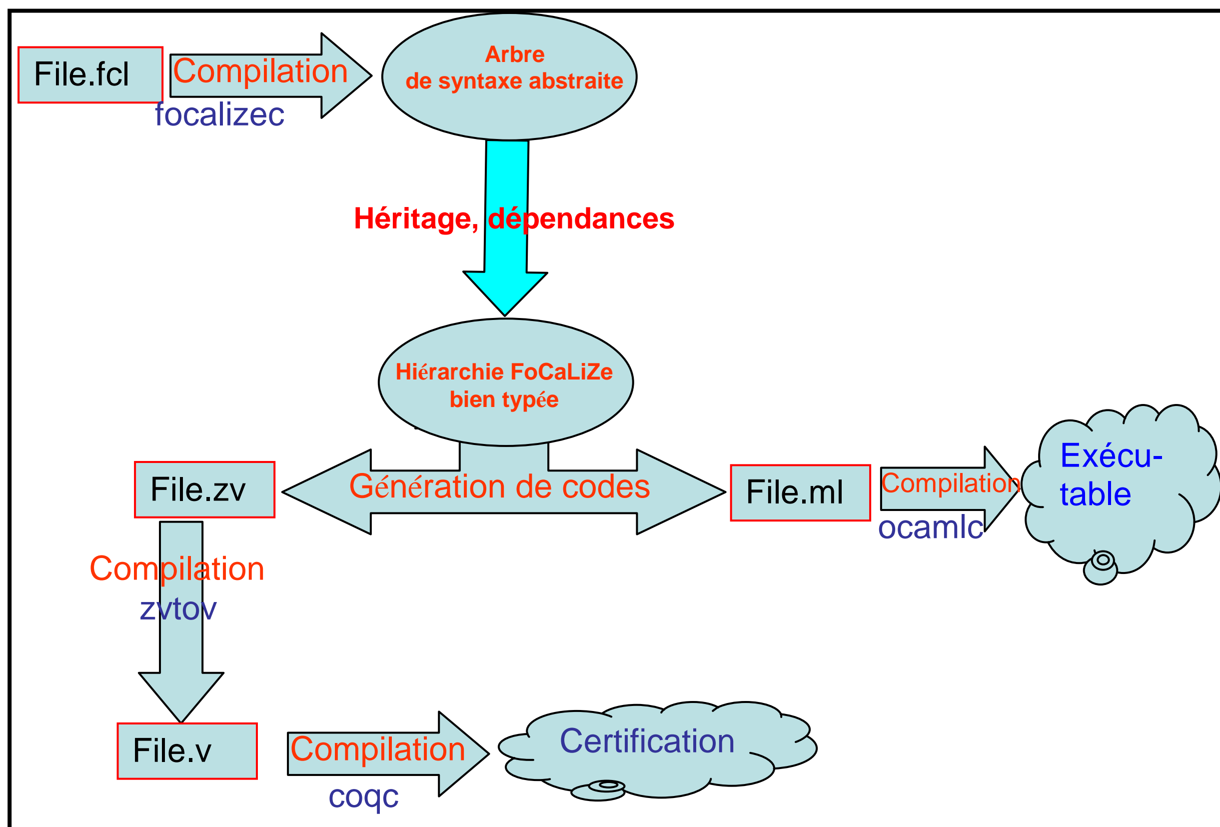

Figure 1.2: La chaîne de compilation d'un programme FoCaLiZe [Rio05].

II.2- Outils de preuve dans FoCaLiZe

Pour certifier une spécification FoCaLiZe, il est nécessaire de prouver les propriétés énoncées dans cette spécification. Pour ce faire, l'atelier FoCaLiZe est équipé de deux outils de preuve que le programmeur utilise dans la démonstration de ses propriétés et théorèmes :

- Coq : qui permet de prouver manuellement des théorèmes
- Zenon : un prouveur automatique de théorèmes pour la logique du premier ordre, basé sur la méthode des tableaux [DOL04]. Zenon est destiné à être le

prouveur de l'environnement FoCaLiZe, il produit des preuves directement vérifiables en Coq.

Dans ce qui suit, nous allons présenter succinctement ces deux outils de preuves associés à FoCaLiZe et donner un exemple pour chacun.

COQ

Développé par le projet LogiCal à INRIA et au LRI (université d'Orsay), il est basé sur la théorie des types (types dépendants et types inductifs). COQ est un système d'aide à la preuve, c'est une extension du noyau purement fonctionnel d'OCAML. Il était donc assez naturel de le choisir comme langage cible pour la partie spécification et certification de FoCaLiZe. C'est à COQ que revient la tâche de vérifier les preuves fournies dans un code source FoCaLiZe.

Un script Coq est la transcription du modèle défini en FoCaLiZe. Il est accompagné de l'ensemble des propriétés énoncées munies de suggestions de preuves, ces preuves doivent être vérifiées par Coq pour être certifiées correctes.

Exemple

Démontrons avec Coq la propriété suivante [FCLt09]:

```
property finite :
        all f : ( Self -> basics # bool ),
        f(nil) ->
        (all l : Self , f(l) -> all v : Val , f( cons (v, l))) ->
        all l : Self , f(l) ;
```

Cette propriété est définie dans le tutorial du FoCaLiZe-0.6.0, plus exactement dans l'espèce '**FiniteList**' des listes finies. Elle exprime le principe de la démonstration par récurrence. La démonstration avec Coq est comme suit :

```
proof of finite =
        coq proof definition of nil , cons
        {* Proof .
        intros .
        unfold abst_cons , abst_nil , cons , nil in *.
        induction l.
        trivial .
        apply H0; apply IHl.
        Qed. *} ;
```

Le mot clé '*coq proof*' indique que la démonstration sera attachée directement au Coq sans aucune assistance par Zenon.

Pour avoir plus d'explications sur cet exemple et même sur les notions des listes et listes finies et leurs propriétés, le tutorial de FoCaLiZe0.6.0 est disponible sur le site Web de INRIA (http://focalize.inria.fr/documentation/focalize-0.6.0_tutorial.pdf)

Zenon

La production de preuve en Coq s'avère difficile. Il faut d'une part maitriser l'outil, et il faut ensuite construire la preuve 'à la main'.

L'introduction de l'outil Zenon a permis de simplifier la tâche du programmeur, en générant automatiquement les scripts de preuve, qui seront ensuite vérifiés par Coq.

Une preuve Zenon est exprimée en FoCaLiZe Proof Langage, sous forme d'une suite d'indications des preuves, semblables aux mécanismes dans Coq, qui sont de trois types :

1. by hypothesis : utilisation d'une hypothèse
2. by property : utilisation d'une propriété déjà prouvée
3. by step : utilisation d'un niveau de preuve précédent

L'échec de Zenon, suite à de mauvaises indications de la preuve, nécessite l'intervention de l'utilisateur pour réécrire ces indications et reprendre le processus de vérification.

Exemple :

```
Theorem prop_K : A -> (B -> A)
Proof =
        <1>1   assume h1: A,
               Prove B -> A
           <2>1  assume h2 : B,
                   Prove A
              by hypothesis h1
           <2>2  qed
              by step  <2>1
        <1>2   qed
           conclude
```

Conclusion

L'atelier FoCaLiZe est un environnement de développement de logiciels sûrs. C'est un outil d'automatisation de la mise en place des méthodes formelles pour spécifier formellement des besoins, implanter des programmes correspondant à ces besoins et exprimer des propriétés et prouver des théorèmes que ces programmes sont correctes vis-à-vis les spécifications définies précédemment.
Grace aux traits objets (héritage multiple, liaison tardive, redéfinition, ...) qu'il possède, l'atelier FoCaLiZe permet aux développeurs, non seulement de structurer leurs codes afin de faciliter son réutilisation, mais d'obtenir un cadre générique à instancier par un passage progressif de la spécification à l'implantation par étapes successives de raffinement.
Avec ce chapitre, nous avons présenté les aspects de bases que l'atelier FoCaLiZe dispose avec des exemples de développement concret pour chaque concept.
Nous avons montré, aussi, l'esprit mathématique et logique du langage FoCaLiZe tout en expliquant son mode de développement, la compilation et génération des différentes documentations et ses outils de preuve qu'il possède (Coq et Zenon).

Chapitre II

La Sécurité informatique

Introduction

La sécurité est une problématique majeure en informatique, il devient aujourd'hui très important de pouvoir contrôler la circulation de l'information dans un système d'information. Tout en garantissant l'accès aux utilisateurs autorisés, il faut protéger les ressources contre toute révélation, modification ou destruction tant accidentelle que malintentionnée. D'une manière générale, la sécurité informatique assure que les ressources matérielles et logicielles ne sont utilisées que dans le cadre prévu. Sécuriser un système informatique consiste donc à mettre en place l'ensemble des moyens techniques, organisationnels, juridiques et humains nécessaires pour conserver, rétablir, et garantir la sécurité de l'information.
La sécurité informatique se base généralement sur les concepts suivants :

- **La confidentialité**: assure que l'information n'est accessible qu'à ceux dont l'accès est autorisé.
- **L'intégrité:** assure que les données du système ne doivent subir aucune modification non autorisée.
- **La disponibilité**: garantir qu'un service doit être disponible à n'importe quel moment pour l'utilisateur.
- **L'authentification**: identifier et authentifier toutes les entités passives et actives du système.

Afin d'atteindre ces objectifs de sécurité, il est nécessaire de mettre en place une **politique de sécurité** applicable à l'ensemble des entités (passives et actives) du système.

I. Politiques de sécurité

Une politique de sécurité est un ensemble de règles de gestion, protection et diffusion de l'information pour assurer la sécurité d'un système. Des modèles de sécurité proposent une représentation formelle de ces politiques, permettant les preuves sur la sécurité d'une application.

Il existe dans la littérature plusieurs types de politique de sécurité selon le domaine d'application (militaire, gouvernemental, commercial et industriels …etc) à savoir [LUD07]:

Politiques basées confidentialité :

- Politique de contrôle d'accès à l'information :
  - High water mark (marée haute), Weissman, 1969
  - Matrice d'accès, Harrizon, Ruzzo et Ullman (HRU), 1973 ->1976
  - Multi-niveaux (ou militaire), Bell et LaPadula (BLP), 1975
  - Muraille de Chine, Brewer et Nash, 1989
- Politique de contrôle du flux d'information :
  - Denning, 1976
- Politique de non-interférence :
  - Goguen et Meseguer, 1982

Politiques basées intégrité :

- Biba, 1977
- Clark et Wilson, 1989

Politique basée disponibilité :

- Ressource allocation, Millen, 1992

Dans les politiques de sécurité de contrôle d'accès, il est donc primordial de définir :

- D'une part l'ensemble des entités passives (objets) manipulables du système et l'ensemble des entités actives (sujets) qui changent le système d'un état à un autre tout en manipulant les objets.
- D'autre part de quelle façon les sujets accèdent aux objets. Dans ce contexte on cite les deux types de politiques de sécurité suivantes [HAD05] :
  - Les politiques discrétionnaires (**DAC** : *Discretionary Access Control*) Ces politiques accordent au sujet propriétaire

de l'information (généralement le créateur) tous les droits d'accès ainsi que la possibilité de les propager aux autres selon sa discrétion.

- Les politiques obligatoires (**MAC** : *Mandatory Access Control*)

  Ces politiques décrètent des règles incontournables qui régissent et gèrent les droits des sujets et des objets. Elles permettent de restreindre les privilèges que possèdent les sujets sur les objets qui leur appartiennent.

Dans ce qui suit nous allons présenter succinctement deux politiques de sécurité de la confidentialité de données (HRU et BLP) et deux autres de l'intégrité de données (Biba et Clark-Wilson).

I.1. La politique de sécurité à matrice d'accès (HRU)

Cette politique de sécurité a été proposée par Harrison, Ruzzo et Ullman [HRU73->76], elle a comme but de vérifier si l'entité active 'Sujet' (utilisateurs, processus, …) demandant l'accès à une entité passive 'Objet' (information, programme, fichier, …) possède les droits nécessaires ou non, elle consiste donc à garantir la confidentialité des données dans un système d'information, elle est basée essentiellement sur la politique de sécurité de Lampson [LAM71] qui représente le système comme étant une machine à états où chaque état est le triplet (S, O, M) où :

- **S** : est l'ensemble des sujets,
- **O** : l'ensemble des objets,
- **M** : la matrice d'accès dont les cases correspondent aux autorisations (lecture, écriture, exécution, contrôle et possession) d'un sujet pour un objet. Par exemple, la case M[s1, o1] correspond à l'ensemble des droits d'accès du sujet ''s1'' sur l'objet ''o1''.

La matrice d'accès est donc comme suit :

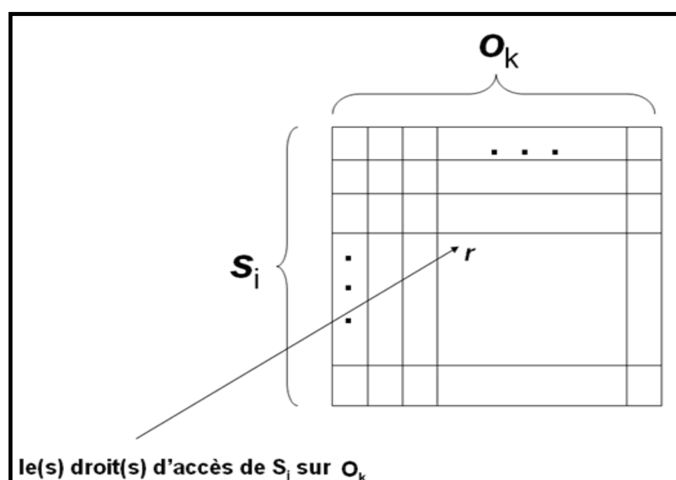

Figure 2.1 : La matrice d'accès des modèles Lampson et HRU

La politique de sécurité HRU est une extension de la politique Lampson avec l'ajout de la notion de '*commandes*' qui peuvent être appliquées sur la matrice d'accès. Les seules opérations 'élémentaires' possibles sont les suivantes [ABO03]:

Enter *a* ***into*** *M(s, o): l'ajout d'un droit d'accès 'a' au sujet 's' sur l'objet 'o'.*
Delete *a* ***from*** *M(s, o) : la suppression d'un droit d'accès 'a' du sujet 's' sur l'objet 'o'.*
Create subject *s: la création d'un nouveau sujet 's'.*
Destroy subject *s : la suppression du sujet 's'.*
Create object *o : la création d'un nouveau objet 'o'.*
Destroy object *o: la suppression de l'objet 'o'.*

Le tableau suivant [SD01] décrit les différentes opérations élémentaires avec leurs conditions d'application et les nouveaux états du système résultants après exécution:

opérations	Conditions	Nouvel état
Enter **a** into **M(s, o)**	$s \in S$ $o \in O$	$S' = S$ $O' = O$ $M'[s,o] = M[s,o] \cup \{a\}$ $M'[si,oj] = M[si,oj] \forall (si,oj) \neq (s,o)$
Delete **a** from **M(s, o)**	$s \in S$ $o \in O$	$S' = S$ $O' = O$ $M'[s,o] = M[s,o] - \{a\}$ $M'[si,oj] = M[si,oj] \forall (si,oj) \neq (s,o)$
Create subject **s'**	$s' \notin S$	$S' = S \cup \{s'\}$ $O' = O \cup \{s'\}$ $M'[s,o] = M[s,o] \forall s \in S \forall o \in O$ $M'[s',o] = \Phi \forall o \in O$ $M'[s,s'] = \Phi \forall s \in S$
Create object **o'**	$o' \notin O$	$S' = S$ $O' = O \cup \{o'\}$ $M'[s,o] = M[s,o] \forall s \in S \forall o \in O$ $M'[s,o'] = \Phi \forall s \in S$
Destroy subject **s'**	$s' \notin S$	$S' = S - \{s'\}$ $O' = O - \{s'\}$ $M'[s,o] = M[s,o] \forall s \in S' \forall o \in O'$
Destroy object **o'**	$o' \notin O$	$S' = S$ $O' = O - \{o'\}$ $M'[s,o] = M[s,o] \forall s \in S' \forall o \in O'$

Table 2.1: Les seules opérations élémentaires du HRU.

Une Commande d'exécution d'une de ces opérations change le système d'un état à un autre tout en passant par la vérification des droits. Une commande HRU est de la forme suivante, avec la partie conditionnelle qui est facultative :

command c(x_1,…,x_k)
if a_1 in M [s_1,o_1] and a_2 in M [s_2,o_2] .. a_n in M[s_m,o_m]
then op_1; op_2;..op_k;
end

Telle que :
c : est le nom de la commande ;
x_i : les paramètres de la commande ;
a_i : les droits d'accès.
op_j : les opérations élémentaires.

Nous donnons un exemple de commande HRU dans laquelle le sujet 'Bob' veut donner le droit de lecture d'un objet 'Dictionnaire' à un autre sujet 'Jaume', la seule condition qui doit être satisfaite est que le sujet 'Bob' doit être le propriétaire de l'objet 'Dictionnaire', formellement on écrit :

command DONNER-Lecture (Bob, Jaume, Dictionnaire)
if own in M [Bob, Dictionnaire]
then enter read into M [Jaume, Dictionnaire]
end

Cette politique de sécurité est très simple à mettre en œuvre, elle définit bien, d'une façon formelle, la notion d'accès des sujets aux objets.

D'autre part, remarquons dans cette politique de sécurité, que tous les objets ont le même niveau de sécurité (pas de différence entre les objets), de même pour les sujets (pas de niveaux hiérarchiques). Par contre la notion de l'intégrité de l'information n'été pas traitée. Et Finalement cette politique de sécurité de type DAC ne permet pas d'exprimer les interdictions et les obligations [ABO03].

I.2 - La politique de sécurité de Bell et LaPadula

Cette politique de sécurité a été proposée en 1973 [BLP73] par Bell et Lapadula. C'est une politique obligatoire (MAC), son objectif principal est de garantir la confidentialité des données dans un système d'information. C'est aussi une politique multi niveaux (chaque entité dans le système a un niveau de sécurité spécifique et selon ce niveau de sécurité les droits d'accès sont définis). Elle est

proposée pour renforcer le contrôle d'accès dans les applications militaires et gouvernementales.
Cette politique de sécurité est composée d'un ensemble d'objets passifs et d'un ensemble de sujets actifs qui peuvent manipuler les objets. Ces deux entités sont classifiées dans plusieurs niveaux de sécurité. Un sujet, peut seulement avoir l'accès aux objets de certains niveaux (en lecture, écriture, exécution …) et un objet ne peut être manipulé que par des sujets de certains niveaux.

Formellement, une classe d'accès est associée à chaque sujet et objet, et l'ensemble des classes est muni d'une relation de dominance (ordre partiel ≥) qui lui confère une structure de treillis [DEN 76].

Un niveau de sécurité **A** domine un autre niveau de sécurité **B** si est seulement si le niveau de classification de **B** est « ≤ » a celui de **A** et la catégorie de **A** contient celle de **B**. Nous avons par exemple, [top-secret, {Nuclear, NATO}] domine [secret, {NATO}] car secret ≤ top-secret et l'ensemble {Nuclear, NATO} contient l'ensemble {NATO}.

Bell et Lapadula respectent les deux règles suivantes [HAD05] :
- No read up : un sujet ne peut lire un objet que si son niveau de sécurité domine celui de l'objet (Un sujet ne doit pas avoir des informations qui ne lui sont pas autorisées).
- No write down : Un sujet ne peut saisir une information dans un objet que si le niveau de sécurité de l'objet domine son niveau de sécurité (Un sujet ne doit pas révéler de secrets).

Formalisation mathématique et logique de Bell et Lapadula [HAD05]
Le système à sécuriser par la politique de Bell et lapadula est composé de

- S : l'ensemble des sujets S,
- O : l'ensemble des objets O,
- A : l'ensemble des actions (droits) A = {lecture, écriture},
- C : l'ensemble des niveaux de sécurité possibles

De plus, à tout moment l'état du système est représenté par :

- M : la matrice d'accès du modèle HRU,

- B= { b = (s, o, a)} l'ensemble des accès courants 'a' que le sujet 's' possède actuellement sur l'objet 'o',
- et la fonction $f: S \cup O \rightarrow C$ de classification qui, à un sujet ou à un objet retourne son niveau de sécurité.

L'ensemble des états du système est défini par les triplets E= {e = (B, M, f)}.
Les deux propriétés suivantes ont été définies afin de formaliser, respectivement, les deux restrictions 'No read up' et 'No write down' :
- Propriété simple : $\forall s \in S, o \in O : (s, o, lire) \in B \Rightarrow f(s) \geq f(o).$
- Propriété étoile : $\forall s \in S, o \in O : (s, o, écrire) \in B \Rightarrow f(o) \geq f(s).$

Une autre propriété (discrétionnaire) définie, consiste à garantir que tout accès courant doit être un élément de la matrice M : $\forall (s_i, o_j, a) \in B \Rightarrow a \in M[i, j].$

Un état du système est dit 'sûr' si et seulement si les propriétés précédentes sont satisfaites.
Ces propriétés permettent d'énoncer le Théorème Basique de la Sécurité (BST), garantissant la sécurité d'un système.
« Les propriétés simple, étoile et discrétionnaire sont vérifiées pour tout triplet de B »

La politique de Bell et Lapadula renforce, d'une façon formelle, la notion d'accès des sujets aux objets par la notion de niveau de sécurité (confidentialité) de chaque entité (renforcement de la politique HRU).

Une formalisation mathématique et logique de cette politique de sécurité a été proposée par E. Gureghian, T. Hardin, et M. Jaume [GHJ03] et une implantation dans FoCAL par M. Jaume et C. Morisset a été réalisée, [JM05].

Néanmoins, plusieurs critiques ont été faites :
Dans cette politique de sécurité, les sujets peuvent lire des informations des objets moins classifiés et écrire ces informations dans des objets plus classifiés, ce qui implique la remontée des différentes informations au plus haut niveau et après un certain temps une sur-classification (changement radical des niveaux de sécurité des informations entre elles-mêmes) de ces informations. Cette notion de sur-classification des informations oblige après un certain temps une dé-classification de ces informations. Bell et Lapadula ont dû introduire la notion de 'sujets de confiance' (trusted subject) qui peuvent déclassifier ces informations mais ils n'ont pas précisé

de quelle façon ces sujets doivent être déterminés [LAN81]. Le contrôle de ses ‘sujets de confiance’ reste, aussi, un point très important à discuter.
John Mclean [MCL90] a critiqué le théorème basique de sécurité défini par Bell et Lapadula. Il a démontré, en utilisant le système de calcul formel Z, que ce théorème ne garantit pas la sécurité du système du fait qu’il n’impose aucune restriction sur l’ensemble des transitions. Mclean a proposé une extension de la politique de Bell et Lapadula en contrôlant l’ensemble des transitions.
Un autre point concernant cette politique de sécurité est que Bell et Lapadula n’ont pas proposé d’axiomes gérants les objets multi-niveaux. La hiérarchisation des objets a été introduite dans [BLP74] mais elle reste toujours très restrictive [LAN81].

I.3 - La politique de sécurité de Biba

Kenneth J. Biba a proposé en 1977 [BIB77] une modification de la politique de sécurité de Bell-LaPadula pour prendre en compte les contraintes **d'intégrité** des données. Il s'agit donc d'assurer le contrôle de l'accès aux données dans le but d'assurer l'intégrité des données : un sujet ne doit pas pouvoir modifier des objets de niveau supérieur.
Biba a introduit la notion de niveau d’intégrité des sujets et objets comme suit :
- Pour les sujets, son niveau d’intégrité reflète la confiance qui lui est accordée pour toute modification, création ou suppression des objets.
- Pour les objets, son niveau d’intégrité représente le danger que peut constituer toute modification de ses informations.

Cette politique de sécurité vise à assurer l'intégrité (plutôt que la confidentialité) au travers de deux règles duales de celles de BLP :
- No write up : un sujet ne pourra pas écrire une information dans un objet de plus haut niveau d’intégrité que le sien, c'est-à-dire que par rapport au BLP on doit interdire l’écriture dans un niveau d’intégrité supérieur ou encore imposer l’écriture dans un niveau d’intégrité inferieur.
- No read down : il faut empêcher qu'un utilisateur puisse lire une donnée de plus faible niveau d'intégrité que le sien, pour ensuite écrire l'information ainsi obtenue dans une donnée de même niveau d’intégrité ou inférieur au sien.

Grace à ces deux règles, Biba a empêché la remontée des informations vers des objets ou des sujets de plus haut niveau. Par contre ces deux règles sont très restrictives, aussi Biba a proposé une alternative plus flexible pour préserver l'intégrité [HAD05] tout en modifiant le niveau d'intégrité des sujets et objets après exécution d'une opération de lecture ou écriture:

- Low water mark pour les sujets : ce principe sert à renforcer le 'No write up' avec une dégradation du niveau d'intégrité d'un sujet s tout en lui donnant la possibilité de lecture des objets o de niveau plus bas que le sien. Le nouveau niveau du sujet est ramené au minimum de f(s) et f(o).

- Low water mark pour les objets : ce principe sert à renforcer le 'No read down' avec une dégradation du niveau d'intégrité d'un objet o tout en donnant à un sujet s de niveau inférieur la possibilité d'écriture dans cet objet. Le nouveau niveau de l'objet ramené au minimum de f(s) et f(o).

L'inconvénient majeur de la politique de Biba est dual à celui de Bell et Lapadula, il s'agit de la sous-classification des niveaux des différentes entités du système qui doivent être remontés après un certain temps. La dégradation de niveau d'intégrité peut perturber le système. Ceci impose l'installation de sujets de confiance et le même problème de gestion de ces sujets se posera.

I.4 - La politique de sécurité de Clark-Wilson

Cette politique a été conçue par David D.Clark et David R.Wilson en 1987 [CW87] pour spécifier et analyser les politiques de sécurité **d'intégrité** de données et surtout dans le domaine commercial. Elle est basée essentiellement sur les points suivants :

- La certification de toute action entreprise dans le système ;
- La traçabilité et journalisation de tout évènement déclenché dans le système ;
- La séparation des pouvoirs entre tous les sujets du système.

Par rapport aux politiques précédemment présentées, les objets ne sont pas caractérisés par leurs niveaux de sécurité mais par l'ensemble des actions (de

confiance) qui peuvent manipuler ces objets, et les sujets sont caractérisés par l'ensemble des actions qu'ils peuvent exécuter.

Cette politique de sécurité se base sur les deux règles suivantes :

- les règles de **certification** qui spécifient les validations qui doivent être entreprises sur les différentes entités du système.
- les règles de **mise en oeuvre** qui spécifient les contrôles qui doivent être effectués par le système lors de toute action entreprise par un utilisateur.

La politique CLARK_WILSON est plus adaptable et applicable aux domaines commerciaux et industriels où l'intégrité du contenu de l'information est contrôlée à tout moment. Elle est basée sur le principe de **transactions bien formées** (Well formed transactions), ce dernier est une suite d'opérations qui changent le système d'un état à un autre. Dans ce cas, l'intégrité des données est basée sur l'intégrité de ces opérations. Avec ce principe de transaction bien formée, Clark et Wilson ont basé leur politique sur le principe de **séparation des pouvoirs** (separation of duty) qui divise l'ensemble des utilisateurs du système en deux grandes catégories:

- **Agents**: des utilisateurs qui exécutent les différentes tâches du système et qu'on doit séparer tout en spécifiant les tâches de chacun,
- **Contrôleurs**: des utilisateurs qui contrôlent ces tâches.

Pour chaque opération, son agent (qui a le droit de l'exécuter) doit être différent de son contrôleur.

Dans cette politique de sécurité de CLARK_WILSON, on doit créer pour chaque action ''**A**'' un triplet (**S, A, O**) signifiant que l'utilisateur **S** a le droit d'exécuter l'action **A** sur l'objet **O**.

Le concept de ***transactions bien formées*** stipule que les utilisateurs n'ont pas le droit de manipuler les données d'une manière arbitraire, mais seulement au travers des procédures de transformation spécifiques, préservant l'intégrité des données.

Le concept de ***séparation des pouvoirs*** est l'un des mécanismes classiques de contrôle de fraudes et des erreurs. Il est fondé sur la répartition des opérations entre plusieurs parties, et l'attribution des droits différents à différentes catégories de personnes [ABO03].

I.4.A. Les éléments de la politique de sécurité

Clark et Wilson définissent :

- Des données soumises à contraintes CDI (Constrained Data Items) qu'on appellera par la suite 'objets intègres', elles correspondent aux données dont on veut assurer l'intégrité.
- Des données non soumises à contraintes UDI (Unconstrained Data Items) qu'on appellera 'objets non intègres', elles correspondent aux données dont on n'a pas besoin d'assurer l'intégrité.
- Des procédures de transformation TP (Transformation Procedures) qu'on appellera « **tâches** » qui correspondent au concept de transaction correcte. Elles constituent les seules entités autorisées à modifier les valeurs des objets intègres (sans altérer leur intégrité), elles préservent l'intégrité des objets manipulés.
- Des procédures de vérification d'intégrité (IVP, pour Integrity Verification Procedure) qui permettent à tout instant de contrôler que tous les objets intègres du système sont conformes aux exigences de la politique d'intégrité.

I.4.B. Les règles de la politique de sécurité [LUD07]

Les règles de la politique de sécurité sont de deux types, des règles de certification afin d'assurer les principes de transformation correcte et de séparation des fonctions, et des règles de mise en œuvre spécifiant au système les contrôles à effectuer.

- Règle de certification 1 (C1)

Toutes les IVP vérifient correctement l'intégrité des CDI.

- Règle de certification 2 (C2)

-*Certifier que toutes les TP accédant à une CDI intègre la laisse intègre.*

-*Construction de listes : (TPi , (CDI1, CDI2, ..., CDIn)).*

Toutes les tâches doivent être valides et toutes les contraintes d'intégrité des objets manipulés doivent être préservées. Il est créé, par tâche, des listes d'objets intègres que cette tâche a droit de manipuler.

- Règle de certification 3 (C3)

- *Construction de listes : (userID, TPi , (CDI1, CDI2, ..., CDIn)).*
- *Certifier que ces listes assurent la séparation des fonctions.*

On associe pour chaque utilisateur une liste de tâches avec les objets intègres a manipuler. Cette règle assure la séparation des fonctions entre utilisateurs.

- Règle de certification 4 (C4)

Certifier que toutes les TP écrivent dans une CDI particulière, de mode d'accès "ajout seul", toutes les informations nécessaires à un audit ultérieur.

- Règle de certification 5 (C5)

Certifier que toute TP vérifie l'intégrité des UDI qu'elle utilise.

Les tâches peuvent manipuler les objets non intègres, les IVP transfèrent ces objets vers des objets intègres (**s'ils respectent** leurs contraintes d'intégrité), sinon elles les rejettent.

- Règle de mise en oeuvre 1 (E1)

Lors de toute manipulation de CDI par une TP, vérification de la liste (TPi , (CDI1, CDI2, ..., CDIn))

Elle sert à vérifier que tout objet intègre est manipulé par sa tâche correspondante.

- Règle de mise en oeuvre 2 (E2)

Lors de toute exécution d'une TP par un utilisateur, vérification de la liste (userID, TPi , (CDI1, CDI2, ..., CDIn))

Elle sert à vérifier que toute tâche est exécutée par son utilisateur correspondant.

- Règle de mise en oeuvre 3 (E3)

La règle de mise en œuvre 2 n'a de sens que si le système identifie et authentifie correctement tout utilisateur.

Pendant la demande d'exécution d'une Tâche par un utilisateur, le système doit identifier ce dernier (utilisateur ou intrus) et vérifier s'il a le droit d'exécuter la tâche demandée.

- Règle de mise en oeuvre 4 (E4)

Seul un utilisateur habilité à certifier une TP peut changer la liste associée à cette TP. De plus, un utilisateur habilité pour certifier une TP ou une IVP ne doit pas disposer de droit d'exécution sur celles-ci.

Chaque tâche doit être exécutée par son agent habilité et contrôlée par son contrôleur habilité, ce dernier a le droit de changer la liste associée à cette tâche sans l'exécuter.

I.4.C- Représentation graphique de la politique de sécurité de Clark_Wilson

Après avoir défini la structure générale de la politique de sécurité de Clark_Wilson, ses éléments et paramètres de base, ses règles de certification et de mise en œuvre. Nous avons tracé ce schéma illustratif qui montre le scénario qui doit être suivi par les différentes entités (passives ou actives) et l'intervention de chaque règle :

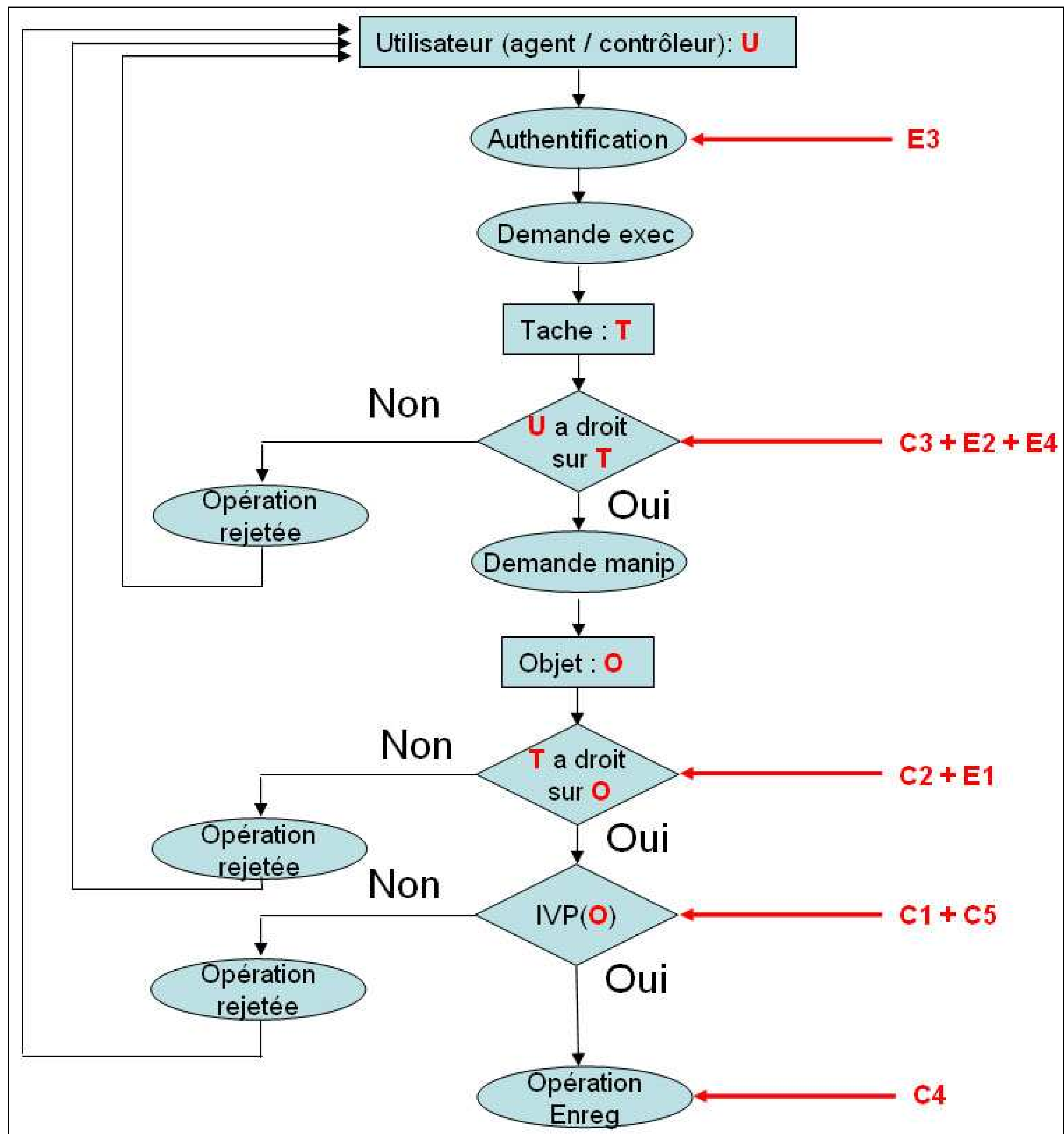

Figure 2.2 : la politique de sécurité de CLARK_WILSON

Un utilisateur 'u' (agent ou contrôleur) s'authentifie et la règle E3 intervient pour vérifier l'existence de cet utilisateur dans le système, il demande l'exécution d'une tâche 't' sur un objet 'o'. Les règles C3, E2 et E4 interviennent afin de vérifier si 'u' a le droit d'exécuter la tâche 't'. Si la réponse est négative, l'opération sera rejetée et nous retournons au début, sinon les règles C2 et E1 vérifient si 't' a le droit de manipuler 'o' **pour le compte** de l'utilisateur 'u'. Si la réponse est négative, l'opération sera rejetée et nous retournons au début, sinon les règles C1 et C5 vérifient si l'objet 'o' respecte toujours ces contraintes d'intégrité tout en exécutant

systématiquement la fonction de vérification d'intégrité IVP sur cet objet. Si la réponse est négative, l'opération sera rejetée et nous retournons au début, sinon la règle C4 enregistre cette opération dans le journal d'évènement du système.

Conclusion

Pour sécuriser un système informatique il faut mettre en œuvre une politique de sécurité décrivant toutes les règles de gestion de contrôle toutes les propriétés qui doivent être satisfaites pour que système soit saint, cohérant et sûr. Les politiques de sécurité se devisent, selon le mode d'accès, entre deux grandes catégories : DAC pour les modèle discrétionnaires et MAC pour les modèle obligatoires.

Comme les politiques discrétionnaires n'ordonnent pas des règles incontournables pour obliger le respect des exigences de sécurité, elles présentent plusieurs inconvénients, surtout les fuites d'informations et la vulnérabilité aux chevaux de Troie. Pour régler ce problème, les politiques obligatoires ont été introduites.

Ainsi, les politiques multi-niveaux affectent aux objets et aux sujets des niveaux non-modifiables par les sujets du système, et donc qui limitent leur pouvoir de gérer les accès aux données qu'ils possèdent.

En termes d'objectif, les politiques de sécurité se devisent entre deux familles : les politiques de sécurité qui s'intéressent à la confidentialité des données pour protéger ces dernières contre toute violation, et des politiques de sécurité d'intégrité de données qui s'intéressent à la protection des ces dernières contre toutes modification malveillante ou accidentelle. Les politiques présentées, de HRU et Bell-LaPadula visant à assurer la confidentialité, tandis que celles de Biba et Clark-Wilson assurent l'intégrité.

La politique de sécurité de Bell et Lapadula renforce la politique HRU avec l'introduction de la notion des niveaux de sécurité pour chaque entité (active ou passive) dans le système, une formalisation mathématique et logique de cette politique de sécurité a été proposée et une implantation dans le système FoCal a été réalisée.

Nous présenterons dans le chapitre suivant, une étude plus détaillée de la politique de Clark-Wilson, ainsi qu'une formalisation dans l'atelier FoCaLiZe. Ce choix est motivé pour les raisons suivantes :

- Il est à noter que la politique de sécurité du Bell et lapadula (qui est une politique de confidentialité de données) a été bien formalisée et implantée dans l'atelier FoCaL. On a préféré de choisir une politique basée intégrité de données, qui est celle de Clark et Wilson.
- Cette politique de sécurité est plus apte et plus adaptable aux domaines commerciaux par rapport à celles citées avant.

Chapitre III

Formalisation de Clark-Wilson dans FoCaLiZe

Introduction

La politique de sécurité de Clark-Wilson met en jeu des ''agents'' (utilisateurs, programmes,…) susceptibles d'exécuter certaines ''tâches'' (création, suppression, modification, …) sur certaines ''ressources'' (fichiers, données,…) tout en assurant les contraintes d'intégrité mises sur ces dernières.

Pour la formalisation de cette politique de sécurité dans l'environnement FoCaLiZe, nous avons défini un ensemble d'espèces qui pourrait être par la suite implanté selon les modèles envisagés. Quelques espèces définies ici héritent des espèces prédéfinies de la librairie FoCaLiZe (présentées en annexe 'A').

Dans ce chapitre nous allons proposer une formalisation de la politique de sécurité de Clark-Wilson et décrire l'ensemble des espèces de notre modèle générique, leurs comportements essentiels, propriétés, méthodes et quelques théorèmes les plus importants avec leurs preuves.

Pour cela, la démarche suivante a été suivie :

1- L'ensemble des entités passives et actives (agents, tâches et ressources) du système seront modélisées par des espèces FoCaLiZe.
2- L'assemblage de ces entités dans une nouvelle espèce modélisant la structure générale du système.
3- Toutes les relations et interactions entre ces entités seront modélisées par d'autres espèces.
4- Toute demande d'exécution d'une tâche émise par un agent est une 'requête' qui sera modélisée par une nouvelle espèce, ces requêtes seront lancées par une fonction 'transition' définie dans l'espèce du 'système' puis les réponses sont modélisées par l'espèce des 'décisions' qui changent le système d'un 'état' à un autre tout en préservant quelques conditions et contraintes émises sur la cohérence des états du système.
5- Des propriétés et théorèmes seront définis et prouvés pour modéliser l'ensemble des règles de sécurité définies par Clark et Wilson.

I. Modèle générique

Dans cette section, nous allons spécifier un cadre générique de notre politique de sécurité choisie. L'ensemble des espèces créées dans notre démarche s'organise comme suit :

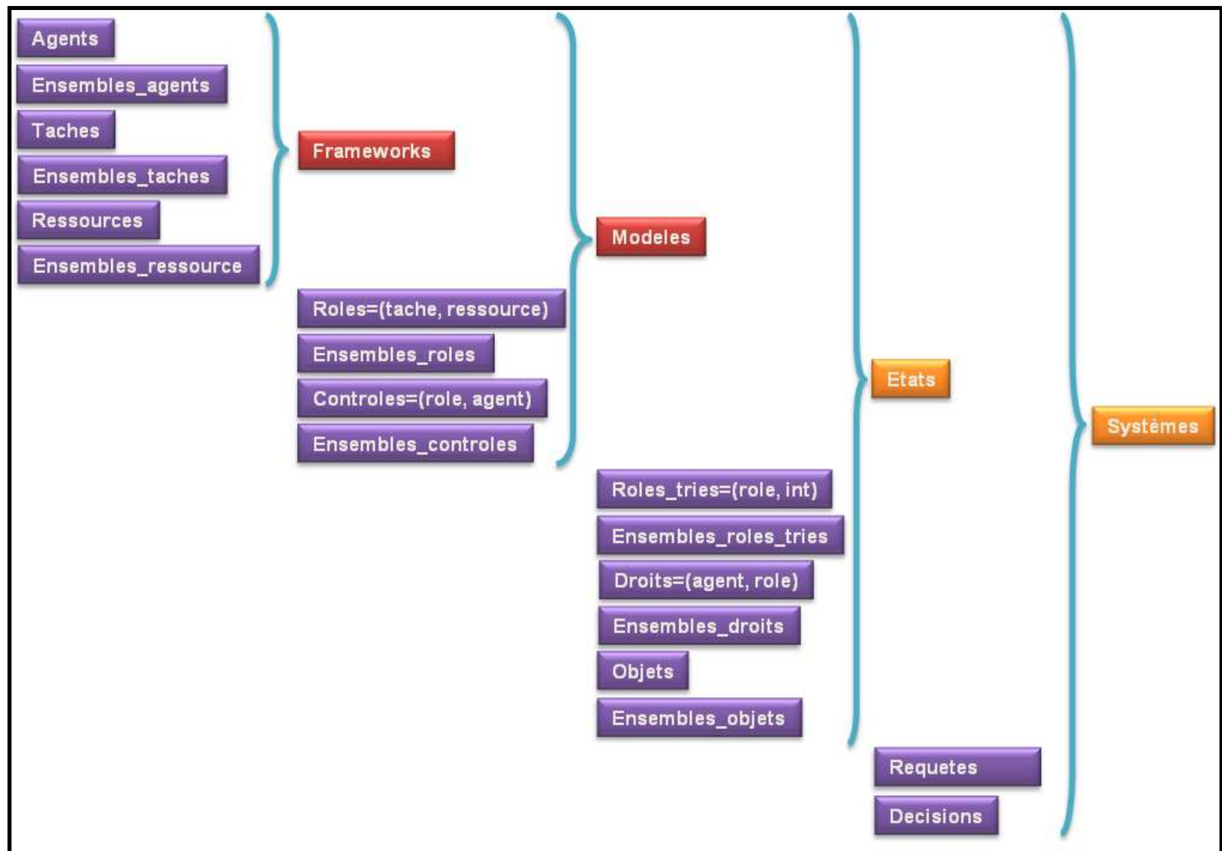

Figure 3.1 : Organisation des espèces.

I.1. Paramètres

L'espèce utilisée la plus basique dont vont hériter de nombreuses espèces, est *Setoid*. C'est un ensemble non vide d'éléments, muni d'une relation d'équivalence « transitive, réflexive, symétrique » égale. Nous utilisons aussi l'espèce *S_setoid* qui hérite de l'espèce *Setoid*.

```
species Setoid = inherit Basic_object;
  ...
end ;;
species S_Setoid = inherit Setoid;
representation = int * string ;
  ...
end ;;
```

Nous avons modélisé la notion d'agent dans l'environnement FoCaLiZe par l'espèce *Agents* qui hérite de l'espèce *S_setoid*. De la même façon, nous avons

modélisé les tâches par l'espèce *Taches* et les ressources par l'espèce *Ressources*. Chaque entité de ces espèces peut être nulle ou non. Pour cela la fonction *agent_nul* (resp. *Ressource_nul*, *tache_nul*) a été définie.

L'espèce *Agents* est spécifiée comme suit :

```
species Agents = inherit S_Setoid;
let num_de_agent (x in Self) in int= basics#fst(x);
let nom_de_agent (x in Self) in string= basics#snd(x);
let create ( num in int , nom in string) in Self= (num , nom);
let agent_nul = Agents!create(0 , "agent_nul");
let equal (s1 in Self, s2 in Self) in bool =  fst (s1) = fst (s2) ;
...
(*  Si deux 'agents' a1 et a2 sont égaux alors
num_de_agent(a1)= num_de_agent(a2)  *)
property agent_equal :
all a1 a2 : Self,
!equal(a1, a2) <-> (!num_de_agent(a1)= !num_de_agent(a2));
end
;;
```

La fonction *agent_nul* est définie pour avoir le même type de résultat de recherche d'un agent si ce dernier n'existe pas, dans ce dernier cas, nous devons avoir un résultat de type *Agent* qui est *agent_nul*.

La même remarque pour les deux autres espèces *Taches* et *Ressources* où les fonctions *tache_nul* et *ressoure_nul* ont été également définies.

Une fonction *create*, qui est un constructeur du type *Self*, a été définie dans chacune de ces espèces.

Quand un agent demande l'exécution d'une tâche sur une ressource, on doit vérifier si cet agent, tâche ou ressource appartient à notre système ou non. Pour cela, les espèces *Ensembles_agents*, *Ensembles_taches* et *Ensembles_ressources* ont été également définies.

L'espèce *Ensembles_agents* est définie dans FoCaLiZe comme suit :

```
species Ensembles_agents (Agent is Agents) = inherit S_set_of(Agent);

(** Impression des résultats *)
  let elements_liste (l : Self) =
    let rec aux (l) =
      match l with
      | basics#[] -> "]"
      | x :: l -> Agent!nom_de_agent (x) ^ "," ^ aux (l) in
    "[" ^ (aux (l));

let afficher_liste (l : Self)= ...
```

L'espèce *S_set_of, qui* hérite des deux espèces *Liste* et *Set_of*, représente la notion des ensembles finis, et sera détaillée dans l'annexe 'A'.

Un agent pourra exécuter de différentes tâches, le système demande à ce dernier de s'identifier en saisissant son nom (qui est un string).

La fonction *agent_de_ce_nom_exist* de type booléen vérifie si cet agent existe ou non dans l'ensemble des agents (actuel). S'il existe, la fonction *agent_de_nom* retourne l'agent du nom saisi. Sinon, elle retourne *agent_nul*. Les deux espèces *ensembles_taches* et *ensembles_ressources* ont été définies de la même façon.

Le système demande après identification, la tâche et la ressource que cet agent veut manipuler. Pour cela, nous avons défini les fonctions *tache_de_ce_nom_exist*, *tache_de_nom*, *ressource_de_ce_nom_exist* et *ressource_de_nom*. La fonction *tache_de_nom* (resp. *ressource_de_nom*) retourne la tâche (resp. la ressource) du nom saisi si elle existe, sinon elle retourne le résultat *tache_nul* (resp. *ressource_nul*).

I.2. Structure générale (Framework)

Un Framework est la donnée des paramètres présentés dans la section précédente. Plus formellement, un Framework est le triplet : (Agents, Tâches, Ressources) où :

- Agents : est un ensemble fini d'agents.
- Tâches : est un ensemble fini de tâches.
- Ressources : est un ensemble fini de ressources.

On a modélisé la notion de Framework dans FoCaLiZe par l'espèce *Frameworks*.

Cette notion de Framework doit être à présent raffinée selon les cas envisagés afin de préciser quelles seront les tâches autorisées à être exécutées par un agent bien spécifique et les ressources autorisées à être manipulées par une tâche bien déterminée.

L'espèce Frameworks a été implantée dans FoCaLiZe comme suit:

```
species Frameworks(
Agent is Agents, Ensemble_agents is Ensembles_agents(Agent),
Tache is Taches, Ensemble_taches is Ensembles_taches(Tache), Ressource is
Ressources, Ensemble_ressources is Ensembles_ressources(Ressource)
) = inherit Setoid ;

representation =  Ensemble_agents * (Ensemble_taches *
                                     Ensemble_ressources);

let agents_de_framework ( x : Self ) : Ensemble_agents = basics#fst(x);
```

```
let taches_de_framework ( x : Self ) : Ensemble_taches =
                                    basics#fst(basics#snd(x));
let ressources_de_framework ( x : Self ) : Ensemble_ressources =
basics#snd(basics#snd(x));
...
end ;;
```

Pour récupérer les éléments d'un framework donné, les fonctions *agents_de_framework*, *taches_de_framework* et *ressources_de_framework*' ont été définies.

I.3. Rôles

Nous avons défini l'ensemble des couples rôles= {(tâche, ressource)} tel que si le couple (ta1, rs1) est un élément de l'ensemble *rôles*, alors l'exécution de la tâche *ta1* sur la ressource *rs1* est un rôle qui peut être manipulé par un agent bien défini. L'espèce *Roles* a été définie pour modéliser la notion des rôles**.** A ce stade, la notion du rôle nul (composé de la tâche nulle et de la ressource nulle) été introduite.

De la même façon que la section précédente, nous avons défini l'espèce *ensembles_roles* pour modéliser des ensembles de rôles. Cette espèce est définie pour vérifier si un rôle demandé par un agent appartient à l'ensemble **courant** des rôles du système.

L'espèce *Roles* à été définie comme suit:

```
species Roles (Tache is Taches, Ressource is Ressources) =
inherit Setoid ;
representation = (Tache* Ressource);

let tache_de_role (x : Self) : Tache= basics#fst(x);
let ressource_de_role (x : Self) : Ressource = basics#snd(x);
...
end ;;
```

L'espèce des ensembles de rôles *Ensembles_roles* a été définie comme suit:

```
  species Ensembles_roles (Tache is Taches, Ressource is Ressources, Role is
           Roles(Tache, Ressource)) = inherits  Set_of(Role), Liste (Role);
...
end ;;
```

L'exécution des différents rôles doit suivre un certain ordre (pour simplifier le travail, on a opté pour un ordre linéaire où il n'y a pas de parallélisme). Pour cela, on a créé l'espèce des rôles triés *Roles_tries* tout en donnant un certain ordre aux rôles

qui doit être respecté pendant leur exécution.

L'espèce des rôles triés a été définie de la façon suivante :

```
species Roles_tries (Tache is Taches, Ressource is Ressources, Role is
Roles(Tache, Ressource)) = inherits Setoid;
representation= Role * int;

let role_de (x : Self) : Role= basics#fst(x);
let ordre_de (x : Self) : int = basics#snd(x);

let est_suivant ( x2 : Self, x1 : Self)  : bool = if (!ordre_de(x2) =
!ordre_de(x1)+ 1) then true else false;
...
end ;;
```

Le dernier paramètre du type support *int* est défini afin de fixer l'ordre d'exécution de chaque rôle. La fonction *est_suivant* est définie pour savoir si un rôle *x2* est bien le suivant du rôle *x1* ou non.

I.4. Contrôles

Nous avons défini également, l'ensemble des couples contrôles= {(role, agent)} où si le couple (r1, ag1) est un élément de l'ensemble *contrôles* alors l'agent *ag1* est le contrôleur du rôle *r1* (séparation des pouvoirs).

Ce contrôleur n'a pas la permission d'exécuter ce rôle mais le contrôler. Cependant, il peut désigner un autre agent pour l'exécution du rôle. Les espèces *Controles* et *Ensembles_controles* ont été définies pour modéliser la notion du contrôle. Ces deux espèces ont été définies en FoCALiZe comme suit:

```
species Controles (Agent is Agents, Tache is Taches, Ressource is
Ressources, Role is Roles(Tache, Ressource)) = inherit Setoid ;
representation= Role * Agent;

let role_controle (x : Self) : Role = basics#fst(x);
let agent_controleur (x : Self) : Agent = basics#snd(x);
...
end ;;

species Ensembles_controles (Agent is Agents, Tache is Taches, Ressource is
Ressources, Role is Roles(Tache, Ressource), Controle is Controles(…)) =
inherit  Set_of(Controle), Liste (Controle) ;

(* pas de rôle avec deux contrôleurs différents *)
let ensemble_controles_coherent (ec : Self)=
let rec aux  (l)=
match l with
| basics#[] -> true
| x :: q ->
```

```
let rec aux2 (ll)=
match ll with
| basics#[] -> true
| y :: p -> if not_b( Controle!role_controle(x) =
              Controle!role_controle(y)) then aux2(p) else false
in aux2(q)
in aux (ec);

let ajoute_controle (c : Controle, ec in Self)=
let ec1= !ajoute_element(c, ec) in
if !ensemble_controles_coherent(ec1) then ec1 else ec;
...
end ;;
```

Ensemble de contrôles cohérent

Dans l'espèce *Ensembles_controles* nous avons défini la notion de cohérence pour éviter qu'un rôle quelconque soit contrôlé par deux agents différents. La fonction *ensemble_controles_coherent* de type booléen a été définie afin de vérifier qu'aucun rôle n'est contrôlé par deux agents différents.

La fonction *ajoute_controle* a été créée pour vérifier si le contrôle qu'on veut rajouter ne modifie pas la cohérence de l'ensemble actuel des contrôles. Dans le cas contraire, l'ajout sera rejeté et par la suite l'ensemble des contrôles reste le même.

I.5. Modèles

Un modèle pour le framework Framework = (Agents, Taches, Ressources) est la donnée d'un élément de l'espèce *Controles*. L'espèce *Modeles* a été définie pour modéliser la notion de modèles. La notion de la cohérence est aussi introduite dans cette espèce où un modèle est dit cohérent si et seulement si son *ensemble_controles* est cohérent. L'exemple suivant illustre cette propriété:

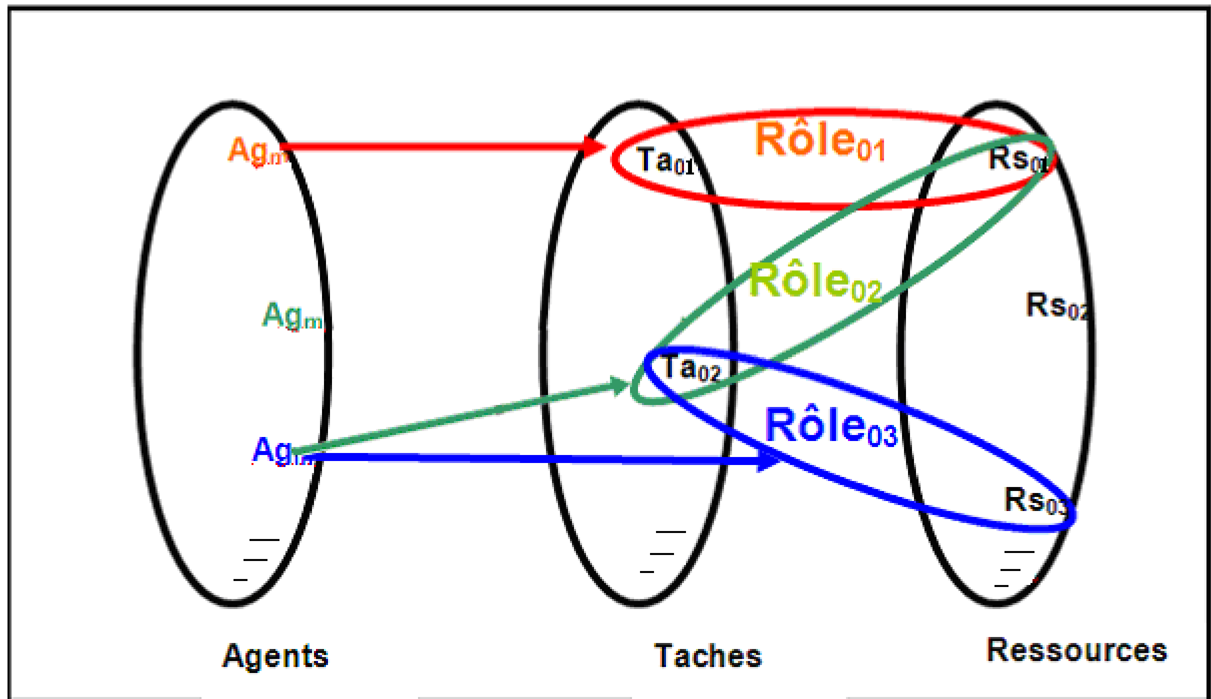

Figure 3.2 : Un Modèle CLARK-WILSON 'cohérent'

Dans cet exemple, nous avons un agent Ag_m qui contrôle le rôle $Role_{01}$ c'est-à-dire contrôlant l'exécution de la tâche Ta_{01} sur la ressource Rs_{01}. Tandis qu'un autre agent (différent du premier) contrôle les deux rôles $Role_{02}$ et $Role_{03}$ où il contrôle l'exécution de la tâche Ta_{02} sur la ressource Rs_{01} et celle de Ta_{02} sur la ressource Rs_{02}. Dans ce cas, comme il n'existe aucun rôle contrôlé par deux agents différents, le modèle est dit *cohérent*.

L'espèce des modèles a été définie de la manière suivante :

```
species Modeles (...) = inherit Setoid ;
representation = Framework* ((Ensemble_roles* Ensemble_roles_tries)*
Ensemble_controles);

let modele_coherent (m in Self) in bool = if
Ensemble_controles!ensemble_controles_coherent(!controles_de_modele(m))
then true else false;
...
end ;;
```

I.6. Systèmes

Un système est composé de l'ensemble des paramètres présentés précédemment (avec les différentes relations et interactions qui peuvent exister entre eux à n'importe quel moment), mais aussi d'autres paramètres que nous allons définir dans ce qui suit:

I.6.a. Etats

Un état contient toutes les informations relatives à l'ensemble des entités du système. Le système peut être changé d'un état à un autre. Ce changement d'états se fait par des requêtes lancées par les agents du système. Un changement d'état peut créer, supprimer ou manipuler des objets. Ces objets sont de type *ressource*.

I.6.b. Objets

Chaque objet dans le système est soit soumis à contraintes, *objet_contraint* qu'on doit protéger, ou non soumis à contraintes *objet_non_contraint*. Pendant la création d'un objet (la saisie de ses paramètres), il n'est soumis à contraintes que pendant son utilisation avec obligation de respecter les contraintes d'intégrité, c'est-à-dire que la création des objets est totalement libre. Un objet peut avoir n'importe quelle valeur. Si cette valeur respecte les contraintes d'intégrité mises sur l'objet, ce dernier sera changé en *objet_contraint* et toute manipulation doit être contrôlée et vérifiée. A cet effet, l'espèce des objets *Objets* a été définie.

Nous avons défini l'espèce des objets comme suit:

```
species Objets (Ressource is Ressources) = inherit Setoid ;
representation=((string* int)* Ressource)* (int* int);

let nom ...
let valeur ...
let ressource_de_objet ...
let val_min ...
let val_max ...
let create  ...
let element=  ...
let objet_nul =  ...

let ivp_o (o : Self) in bool=
if
and_b(
          basics#int_geq(!val_max(o), !valeur(o)),
          basics#int_geq(!valeur(o), !val_min(o)))
then true else false;
...
end ;;
```

Dans cette espèce, on a introduit la notion des *IVP* de Clark_Wilson, où le système doit contrôler la valeur courante de l'objet si elle est entre val_max et val_min. Pendant la création d'un objet, l'utilisateur peut saisir n'importe quelle valeur pour ses paramètres (initialement objet_non_contraint) et il peut être changé en *objet_contraint* à condition de respecter les contraintes d'intégrité, dans le cas contraire il sera rejeté.

L'espèce *Ensembles_objets* a été définie et doit être consultée (resp. mise à jours) avant (resp. après) chaque changement d'état. L'ensemble des objets est dit cohérent si tous ses objets respectent leurs contraintes d'intégrité.

Le type support de cette espèce contient les informations relatives à l'objet suivantes :

- Le nom de l'objet de type string ;
- La valeur de l'objet de type int ;
- La ressource de l'objet de type ressource ;
- Les deux derniers paramètres du type support (int*int) correspondent aux val_max et val_min de l'objet,

De la même façon que les espèces des agents, tâches et ressources, les deux méthodes *element* et *objet_nul* ont été définies.

L'espèce *Ensembles_objets* a été définie comme suit :

```
species Ensembles_objets (Ressource is Ressources, Objet is Objets
(Ressource)) = inherit Set_of(Objet), Liste(Objet) ;

let ivp_eo (eo : Self)= ...
let eo_coherent (eo : Self)= !ivp_eo(eo);
let ajoute_objet (o : Objet, eo : Self) : Self= ...
let modifier_objet (eo : Self, o : Objet, v : int) : Self= ...
...
end ;;
```

On fait appel à la fonction *ivp_o* des objets pour tester si tous les éléments d'un *Ensembles_objets* respectent bien leurs contraintes d'intégrité, pour cela nous avons défini la fonction *ivp_eo*.

La fonction *objet_de_ce_nom_exist* a été définie afin de vérifier si un objet d'un nom donné appartient à l'ensemble **courant** des objets du système. Dans ce cas, la fonction *objet_de_nom* retourne l'objet du nom saisi, sinon objet_nul sera retourné.

La fonction *ajoute_objet* a été définie afin de pouvoir rajouter un nouvel objet dans l'ensemble courant des objets du système. Avant l'ajout, le système vérifie si ce nouvel objet respecte ses contraintes d'intégrité sinon l'opération sera rejetée.

Comme l'objectif principal de notre travail est de préserver l'intégrité de nos objets, nous avons donné aux utilisateurs du système, la possibilité de modifier la valeur d'un objet quelconque tout en exécutant la fonction *modifier_objet* qui a été définie dans cette espèce.

I.6.c. Droits

Pour définir quel agent possède quel(s) rôle(s), nous avons défini l'ensemble "droits" suivant: droits= {(agent, rôle)} où si le couple (ag1, r1) est un élément de l'ensemble "droits" alors l'agent "ag1" a la permission d'exécuter le rôle "r1" (dans certaines conditions).

Nous avons défini dans le système FoCaLiZe la notion de droit par l'espèce *Droits* et l'ensemble des droits par l'espèce *Ensembles_droits* comme suit :

```
species Droits (...) = inherit Setoid ;

representation= Agent* Role;
let agent_de_droit ...
let role_de_droit ...
let create ...
let element ...
...
end;;


species Ensembles_droits (...) = inherits  Set_of(Droit), Liste (Droit) ;

let elements_liste(l : Self) = ...

let afficher_liste(l)= basics#print_string(!elements_liste(l));

(* pas de rôle avec deux agents différents qui ont le droit de le manipuler *)
let ensemble_droits_coherent (ed : Self)=
let rec aux  (l)=
match l with
| basics#[] -> true
| x :: q ->
            let rec aux2 (ll)=
            match ll with
            | basics#[] -> true
            | y :: p -> if not_b( Droit!role_de_droit(x) =
Droit!role_de_droit(y)) then aux2(p) else false
            in aux2(q)
in aux (ed);

let ajoute_droit (d : Droit, ed : Self)=
let ed1= !ajoute_element(d, ed) in
if !ensemble_droits_coherent(ed1) then ed1 else ed;
...
end ;;
```

Ensemble de droits cohérent

Nous avons introduit la notion de cohérence dans l'ensemble des droits afin d'éviter qu'un rôle quelconque ne soit exécuté par deux agents différents. Pour cela, la fonction *ensemble_droits_coherent* a été définie.

L'exemple suivant illustre cette propriété:

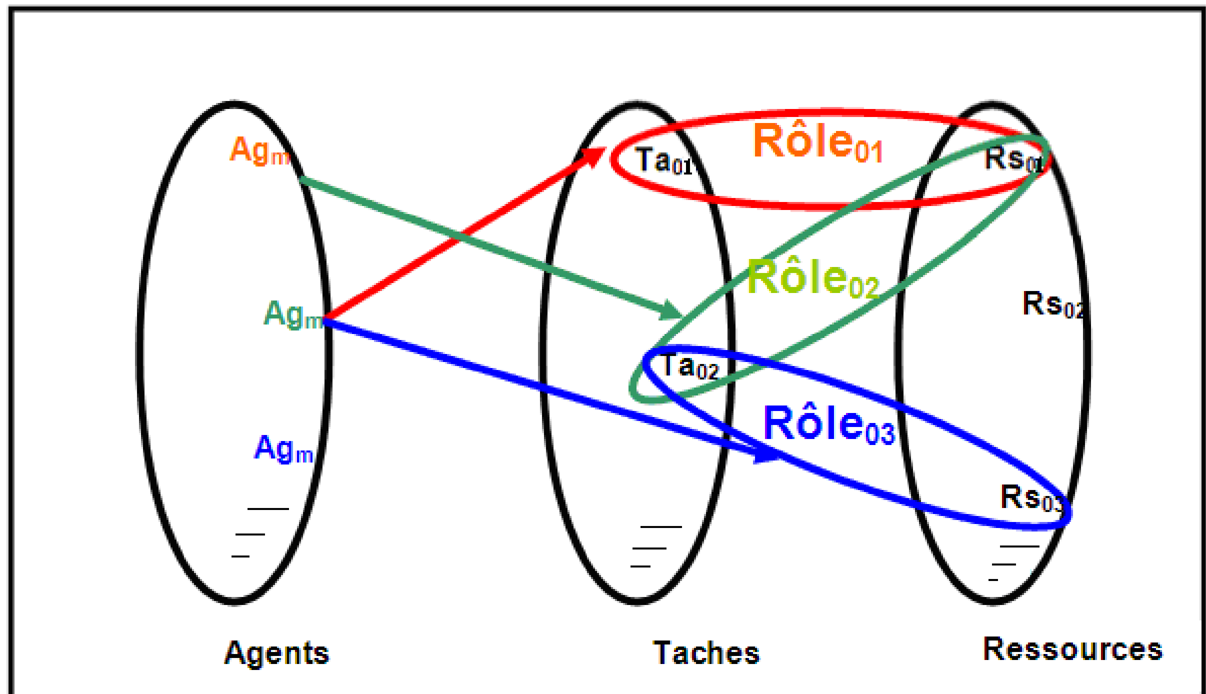

Figure 3.3 : ensemble droits CLARK-WILSON 'cohérent'

Dans cet exemple, nous avons un agent Ag_m qui exécute le rôle $Role_{02}$ c'est-à-dire qu'il exécute la tâche Ta_{02} sur la ressource Rs_{01}. Tandis qu'un autre agent (différent du premier) exécute les deux rôles $Role_{01}$ et $Role_{03}$ ou il exécute la tâche Ta_{01} sur la ressource Rs_{01} et Ta_{02} sur la ressource Rs_{03}. Dans ce cas, comme il n'existe aucun rôle exécuté par deux agents différents, l'ensemble des droits est dit *cohérent*.

A partir d'un framework F= (agents, tâches, ressources), nous avons défini l'ensemble des états d'un système par le quintuplet e= (te, m, eo, ed, r) où :

_ te: le type de l'état "e", initial ou non.

_ m: le modèle correspondant au framework "*F*".

_ eo: l'ensemble des objets courants dans l'état "e".

_ ed: l'ensemble des droits courants dans l'état "e".

_ r: un élément de l'ensemble *ensemble_rôles_tries*. C'est le dernier rôle exécuté qui a créé l'état "e". Ce dernier paramètre a pour but de savoir quel est le rôle suivant à exécuter. Notons que le dernier rôle de l'état initial est le rôle nul. Pour modéliser la notion d'état, nous avons défini l'espèce Etats.

La création d'un nouvel état se fait sans condition par une fonction de transition où avant de confirmer l'état on vérifie si ce dernier est cohérent ou non. La cohérence d'un état est équivalant à la cohérence de l'ensemble de ses objets, de

son modèle et de l'ensemble de ses droits. Etant donné que c'est la première espèce composée par l'assemblage des deux ensembles DROITS et CÔNTROLES, nous devons vérifier, pendant la création d'un nouvel état, qu'aucun rôle n'est contrôlé par un agent qui a le droit de l'exécuter.

Nous avons créé le type énuméré *type_etat* :

```
type type_etat =
   | Tetat_init
   | Tetat_non_init
;;
```

L'espèce *Etats* a été définie comme suit:

```
species Etats (...) = inherit Setoid ;

representation= type_etat* (((Modele* Ensemble_objets)* Ensemble_droits)*
Role_trie);
...
let etat_coherent(e : Self)= ...

let historiser (c : out_channel, s : string )= ...
...
end ;;
```

Dans cette espèce, la fonction *historiser* a été définie, elle sera appelée dès que le système reçoit un événement quelconque. Avec cette fonction, le système sauvegarde toutes les traces de changement ou de tentative de changement d'état dans un fichier journal log.txt.

I.6.d. Requêtes

Une requête est une demande d'exécution d'une tâche émise par l'utilisateur. Elle peut être administrative ou non administrative :

1- Requêtes administratives

C'est une demande d'ajout d'un rôle à un agent ou de suppression d'un rôle d'un agent. Dans ce type de requête, l'utilisateur qui l'a lancé doit être le contrôleur du rôle demandé et aussi l'agent concerné ne l'est pas, d'où la modification de l'ensemble des droits et par conséquent modification du comportement administratif du système.

2- *Requêtes non administratives*

L'utilisateur qui l'a lancé ne doit pas être le contrôleur du rôle demandé, ou bien, il doit être son agent responsable (qui a le droit de l'exécuter), ainsi il n'y aura pas de modification du comportement administratif du système. Dans ce type de requête, nous préconisons deux cas de figure :

a- Demande d'exécution d'un rôle : Dans ce type de requête, nous devons respecter

l'ordre d'exécution des rôles.

b- Demande de modification d'un objet : Avec ce type de requête on doit garantir l'intégrité de nos données. Dans ce cas, l'agent qui initie la requête doit être le créateur de l'objet en question, et ne pourrait le modifier qu'après avoir directement procédé à sa création (avant sa livraison et utilisation).

En conséquence, nous avons modélisé la notion de requête Clark-Wilson sous la forme rq= (tr, ag1, ta, rs, ob, ag2) où :

- tr: type de la requête (exécution d'un rôle donné, ajout un rôle à un agent donné, suppression d'un rôle d'un agent donné ou modification de la valeur d'un objet donné).

- ag1: un élément de "agents", est l'agent qui initie la requête (contrôleur ou non contrôleur).
- ta: un élément de "tâches", est la tâche demandée par la requête.
- rs: un élément de "ressources", est la ressource concernée par la requête.
- ob: un élément de "objets", est l'objet concerné par la requête.
- ag2: un élément de "agents", est l'agent concerné par la requête si la requête est administrative, sinon il doit être nul.

L'espèce des requêtes "Requetes" a été définie pour modéliser la notion de requête dans le système FoCaLiZe.

Avant d'exécuter une requête, le système vérifie que l'un des deux paramètres à savoir 'rs' ou 'ob' (et pas les deux à la fois) n'est pas nul. Quand la requête est de type '' Demande d'exécution d'un rôle'', elle est donc une demande d'exécution d'une tâche sur une ressource (création d'un nouveau objet) ou sur un objet (déjà créé). Pour la création des objets, nous avons préféré donner la possibilité aux utilisateurs de créer leurs objets d'une façon libre en saisissant le nom de l'objet, sa valeur…

Nous avons défini le type énuméré *type_requete* suivant:

```
type type_requete =
  | Trequete_exec
  | Trequete_add
  | Trequete_del
  | Trequete_modif_objet
;;
```

L'espèce *Requetes* a été définie de la manière suivante :

```
species Requetes (Agent is Agents, Tache is Taches, Ressource is
Ressources, Objet is Objets(Ressource)) =inherit Setoid;
representation= ((type_requete* Agent)* (Tache* (Ressource* Objet)))*
Agent;
```

```
...
let requete_est_sur_objet  (rq : Self) : bool=
if (and_b (not_b( Objet!equal(!objet_concerne(rq), Objet!objet_nul)),
Ressource!equal(!ressource_concernee(rq), Ressource!ressource_nul))) then
true else false;
...
let requete_est_sur_ressource  (rq in Self) : bool= ...
let requete_ni_sur_objet_ni_sur_ressource  (rq : Self) : bool= ...
let requete_sur_objet_et_sur_ressource  (rq : Self) : bool= ...
...
let ressource_generale_de_requete (rq : Self) : Ressource=
if (!requete_est_sur_objet(rq)) then
Objet!ressource_de_objet(!objet_concerne(rq)) else
if (!requete_est_sur_ressource(rq)) then !ressource_concernee(rq) else
Ressource!element;
...
let est_exec (r : Self) : bool = if (!type_de_requete(r)= #Trequete_exec)
then true else false;
let est_del (r : Self) : bool = ...
let est_add (r : Self) : bool = ...
let est_modif_objet (r : Self) : bool = ...
let est_ajout_objet (r : Self) : bool = if and_b(!type_de_requete(r)=
#Trequete_exec, !tache_demandee(r)= Tache!element) then true else false;
...
end;;
```

I.6.e. Décision

C'est la réponse d'une requête. Quand un agent demande l'exécution d'une requête sur un état, le système vérifie les paramètres de sécurité et les droits d'accès et génère l'une des décisions suivantes :

- *Oui* : s'il existe une règle autorisant l'exécution de la tâche demandée par l'agent demandant.
- *Non* : sinon.

Nous avons défini l'espèce *Decisions* afin de modéliser la notion de décision dans le système FoCaLiZe, elle est définie de la manière suivante :

```
species Decisions (...)= inherit Setoid;

representation= Requete* Etat;
...
let numero_de_decision (d : Self) : int =
if (*si requete non admin: Trequete_exec *)
(Requete!type_de_requete(!requete_de_decision(d))= #Trequete_exec)
then
(* il faut que l'un des deux  soit nul: ressource ou objet*)
if and_b(((Requete!objet_concerne(!requete_de_decision(d)))=
Objet!objet_nul),((Requete!ressource_concernee(!requete_de_decision(d)))=
Ressource!ressource_nul))
then 1 (* err: et la ressource concernée et l'objet, les deux sont nuls*)
else (* l'un des deux n'est pas nul, ou les deux ne le sont pas!!*)
...
Else 25 (* créer un nouvel état *)
...
```

```
let val_de_decision  (d : Self ) : string =
if (!numero_de_decision(d) = 1)
then "Erreur N°:01, la ressource et l'objet concernés sont tous les deux
nuls!!!"
else
if (!numero_de_decision(d) = 2)
then "Erreur N°:02, la ressource et l'objet concernés: les deux ne sont pas
nuls!!!"
else
...
else
if (!numero_de_decision(d) = 25)
then "La création d'un nouvel état sera lancée\n"
else "Erreur inconue";
...
let oui (d : Self) : bool =
if or_b(or_b(or_b(or_b(!numero_de_decision(d)= 5, !numero_de_decision(d)=
10), !numero_de_decision(d)= 18), !numero_de_decision(d)= 20),
!numero_de_decision(d)=  25) then true else false;

let non (d : Self) : bool = not_b(!oui(d));
...
end;;
```

La fonction *numero_de_decision* nous permet de récupérer le numéro de la décision générée. Avec la fonction *val_de_decison* du type chaîne de caractères, le système nous affiche si la décision est oui ou non et dans ce dernier cas, l'explication de l'erreur (pour quelle raison elle est "non") sera affichée.

Avec la fonction *numero_de_decision* de type int on pourra décider si on lance le changement de l'état ou non.

Comme il a était expliqué précédemment, avant de créer un nouvel état (validation de l'action demandée par la requête), le système doit vérifier si tous les objets *objet_contraint* respectent leurs contraintes d'intégrité ou pas.

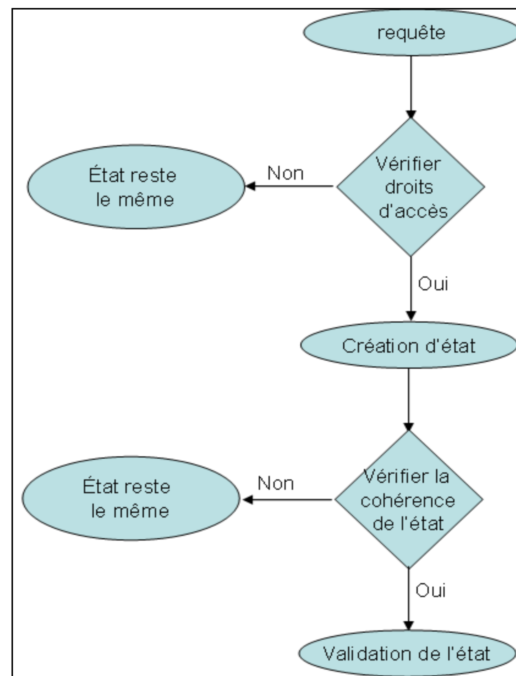

Figure 3.4 : Création, validation et rejet d'un état.

Une demande d'exécution d'une requête est lancée par un agent, le système vérifie les paramètres de sécurité et droits d'accès. S'il trouve une anomalie, l'opération sera rejetée et l'état reste le même. Sinon, il (le système) crée un nouvel état avec les éventuelles modifications apportées sur l'ancien état. Il vérifie la cohérence de l'état résultant. S'il trouve une anomalie, l'opération sera rejetée et l'état reste le même, sinon il valide l'état et l'enregistre.

I.6.f. Transitions

Une fonction de transition permet de faire passer le système d'un état à un autre. Pour cela, un agent effectue une requête sur un état, une décision est alors renvoyée par le système, ainsi qu'un nouvel état correspondant à l'ancien état avec les modifications éventuellement provoquées par la requête.

Nous avons défini donc :

- ''Requetes'' l'ensemble des requêtes possibles,
- ''Decisions'' l'ensemble des décisions possibles,
- ''Etats'' l'ensemble des états possibles,

La fonction de transition d'un état à un autre est donc donnée comme suit:

Transition: Requetes $_X$ Etats → Decisions $_X$ Etats.

Une fonction de transition est dite *sûre* si elle garantit que si un agent *ag1* envoie une requête d'exécution d'un rôle *rl1* et si les conditions nécessaires sont vérifiées alors la décision émise par le système est ''oui''. Dans le cas contraire elle

est ''non''.

Etant donné un framework F = (Agents, Taches, Ressources), un modèle ''M'' pour ''F'', un ensemble ''E'' des états, un ensemble ''R'' des requêtes, un ensemble ''D'' des décisions, une fonction de transition ''T'' et un état initial ''e_0'', alors un système est défini comme suit: Ş = (F, M, E, R, D, T, e_0)

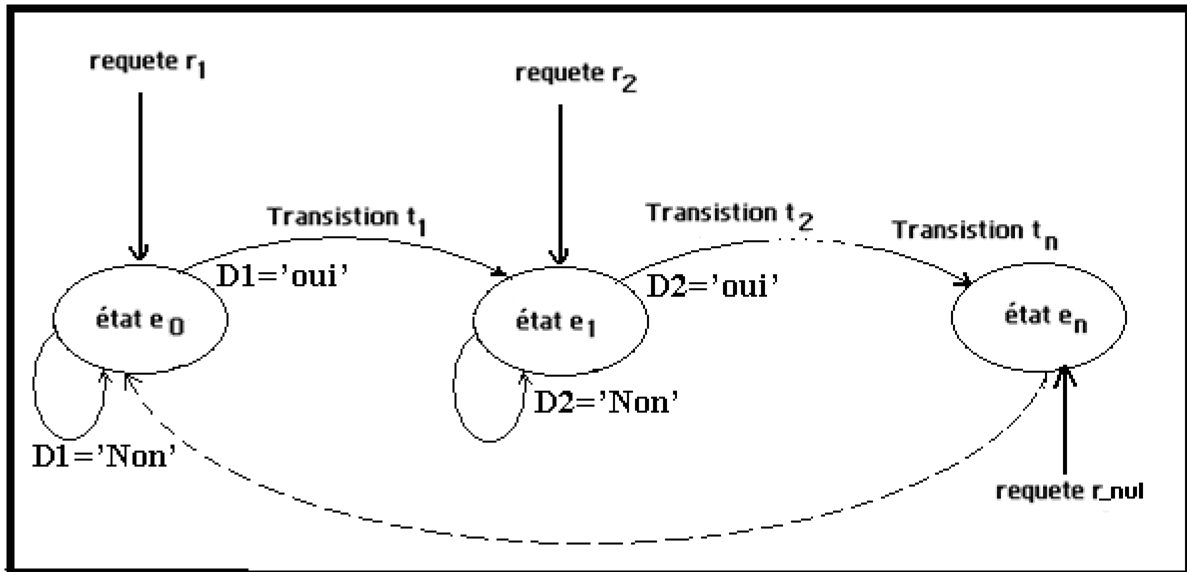

Figure 3.5 : Transition d'états d'un système CLARK-WILSON

Pour modéliser la notion de systèmes, nous avons défini l'espèce *Systèmes* de la façon suivante :

```
species Systeme (...)= inherit Setoid;
...

 (* la fonction Transition *)
let transition (st : Self) : Etat=
 if Requete!est_exec (!requete_de(st))
   then(*1.0*)
       if Objet!equal(Requete!objet_concerne(!requete_de(st)),
Objet!element)
          then(*1.1*)
               if
Ressource!equal(Requete!ressource_concernee(!requete_de(st)),
Ressource!element)
                   then (*1.2*)
                   !etat_de(st)(*dec_1*)
                   else (*1.2*)
                   if ...
else (*1.0*)
       if Requete!est_del(!requete_de(st))
...

logical let omega (e : Etat) =
Modele!modele_coherent(Etat!modele_de_etat(e)) /\
Ensemble_droits!ensemble_droits_coherent(Etat!droits_de_etat(e)) /\
Ensemble_objets!eo_coherent(Etat!objets_de_etat(e));
```

```
...

theorem transition_sure:
all st1 st2 : Etat, all rq : Requete,
      transition (!create(rq, st1)) = st2 ->
        omega (st1) ->
          omega (st2)
    proof = ...
...

end;;
```

Le schéma général (figure 3.6) des paramétrages entre espèces permet de mieux percevoir les liens entre les différentes espèces de notre système.

De même, avec la commande *depgraph*, nous avons pu créer pour chaque espèce un graphe illustrant son comportement (composition). On a choisi l'espèce des états comme illustration (figure 3.7).

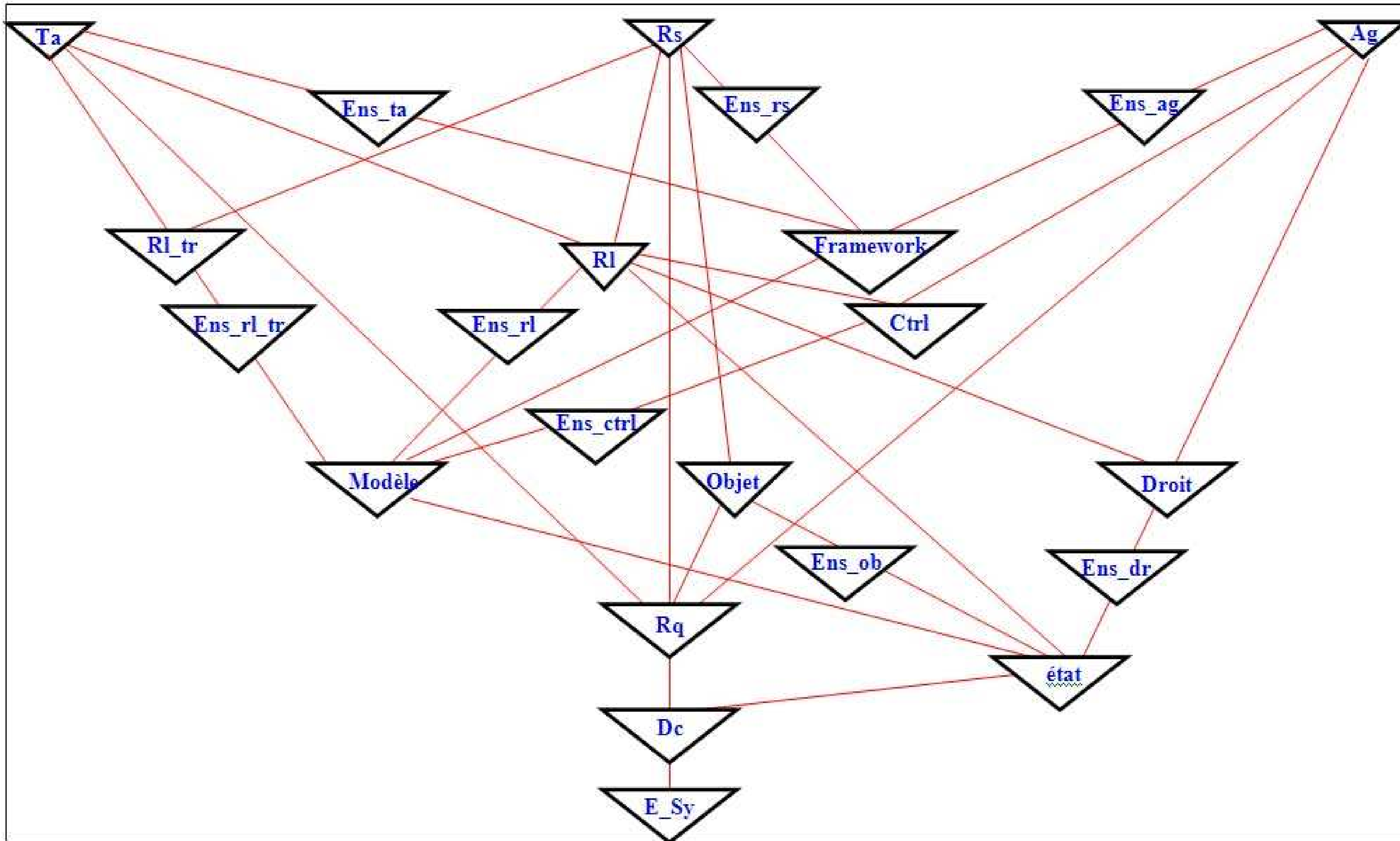

Figure 3.6 : Paramétrage entre espèces

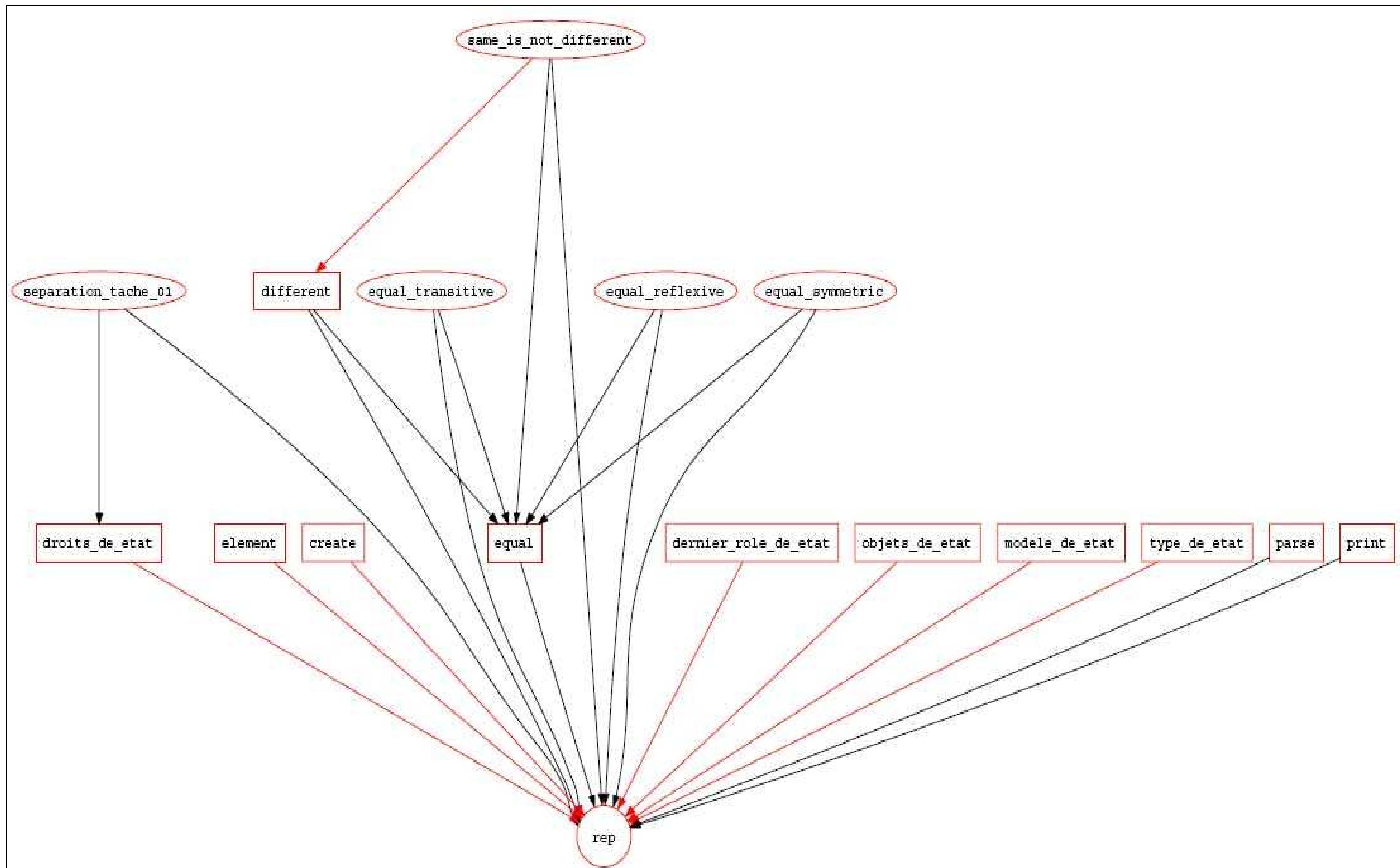

Figure 3.7 : Graphe de dépendance de l'espèce 'Etats', créé par la commande 'depgraphe'

Dans la fonction de transition, selon les paramètres de sécurité et conditions d'exécution de la requête, un nouvel état est créé (où il reste le même). Cet état reste en instance jusqu'à ce qu'on confirme qu'il est cohérent. Par la suite, la fonction *historiser* de l'espèce *Etats* est appelée à la rentrée de la fonction *transition* pour sauvegarder dans le fichier *log.txt* tout événement déclenché dans le système.

Après définition des paramètres du système, nous avons tracé le schéma illustratif suivant (figure 3.8). :

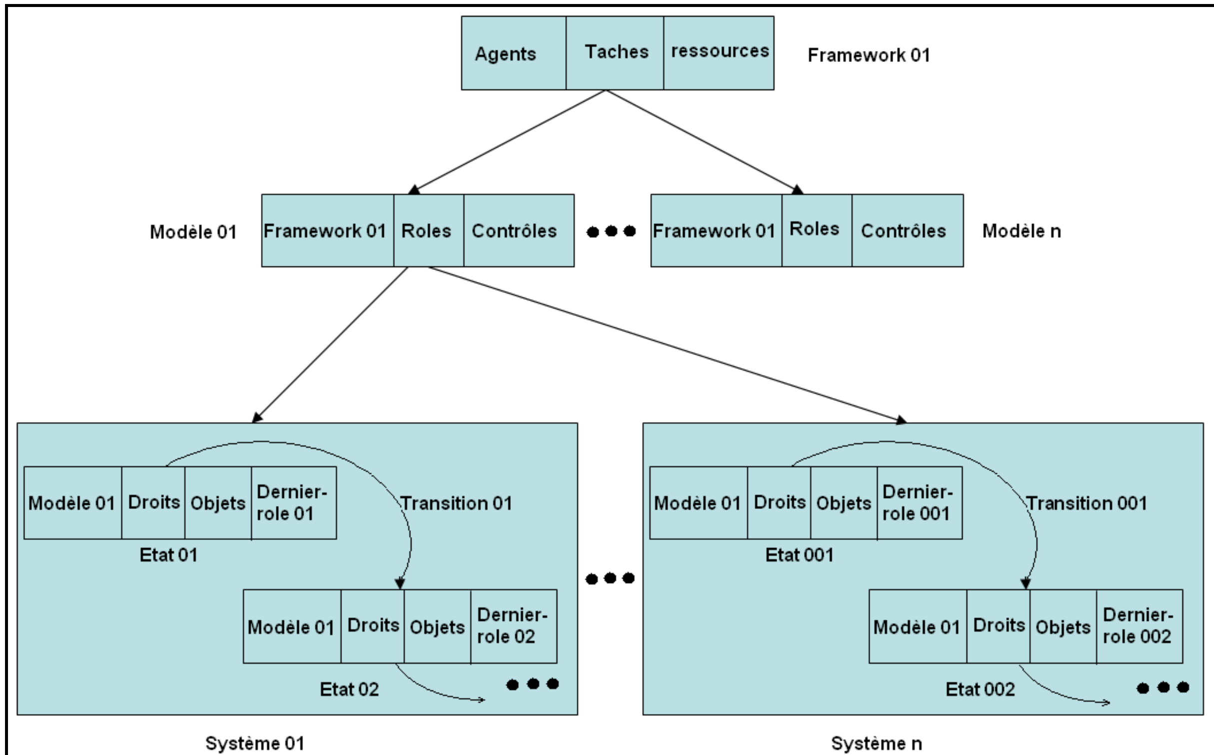

Figure 3.8 : Framework, Modèles, Etats et Systèmes CLARK-WILSON.

I.7 Aspect preuve

Notre objectif principal dans cette étude est de garantir l'intégrité de nos données et la cohérence des différentes associations (droits d'accès) entre les différentes entités du système notamment les deux ensembles droits et contrôles. Pour cela, nous devons nous assurer que la fonction de transition fait changer le système d'un état cohérent à un autre. Pour cet effet, Le théorème *transition_sure* a été spécifié et prouvé dans l'espèce *Systeme*,

La fonction (le prédicat logique) ***omega*** paramétrée par un état quelconque vérifie si ce dernier est cohérent, c'est-à-dire si le modèle de cet état est cohérent, de même que l'ensemble des droits et l'ensemble courant des objets.

La fonction de transition est donnée en détail dans l'annexe B.

Dans cette fonction, nous avons parcouru tous les cas possibles qui peuvent exister en combinant les différentes conditions d'exécution et paramètres d'accès. Pour faire ce parcours sans oublier aucun chemin, nous avons tracé l'organigramme de calcul du numéro de décision suivant :

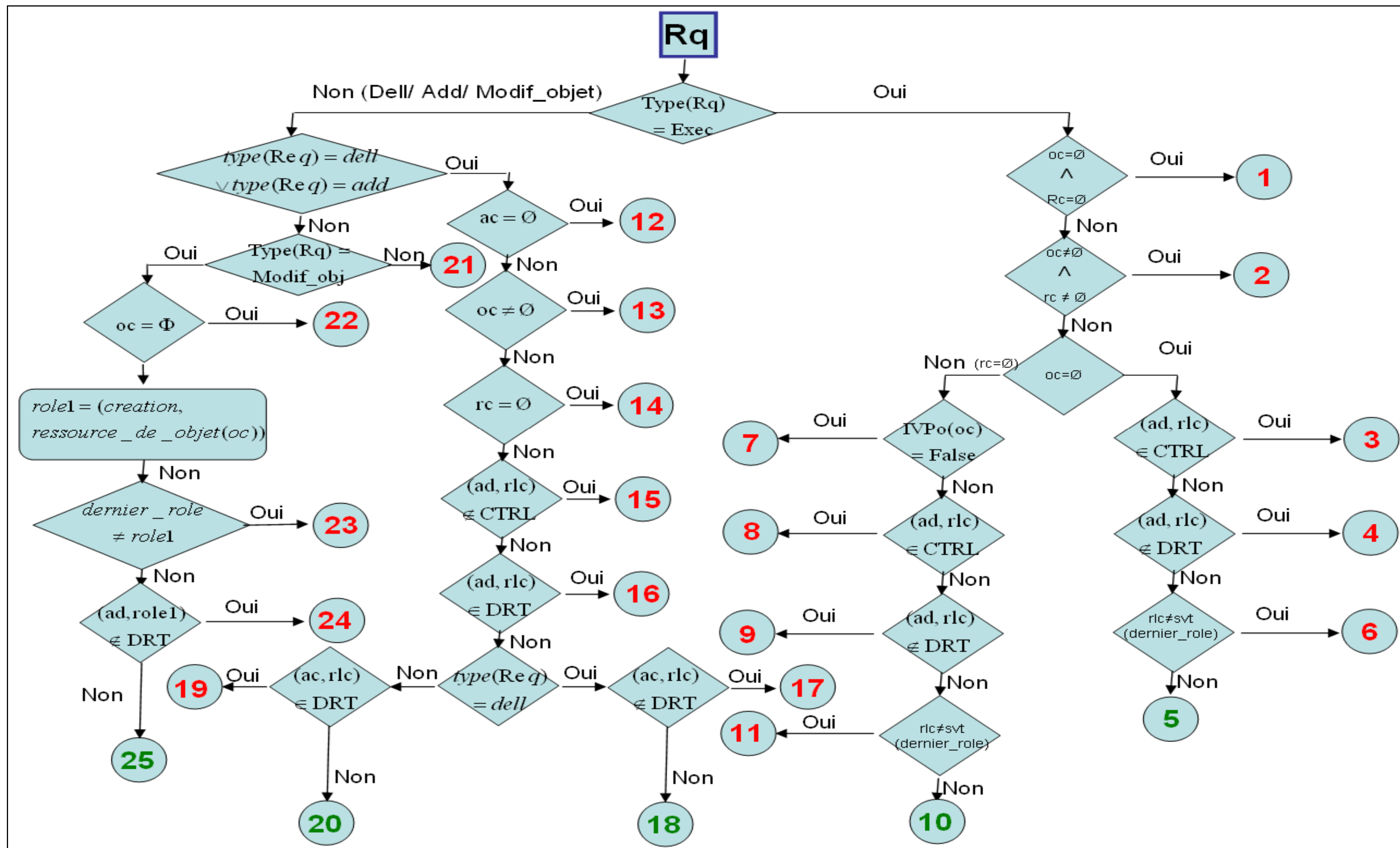

Figure 3.9 : Organigramme de calcul de numéro de décision

Explication de l'organigramme de calcul de la décision

La décision a une valeur entière comprise entre 1 et 25, les cas 5, 10, 18, 20 et 25 correspondent à la valeur 'Oui' de la décision pendant que les autres valeurs indiquent 'Non'.

1- Decision= 01 :

Type de requête= Exec, Objet concerné= nul et Ressource concernée= nul

Une requête doit s'exécuter soit sur une ressource (dans le cas de création d'un nouvel objet) soit sur un objet (dans les autres cas). Donc, l'un des deux paramètres doit être nul mais pas les deux en même temps.

Dans ce cas de décision (decision=01), le système trouve que les deux paramètres sont nuls. Donc la décision est automatiquement 'non', l'état reste donc le même et la requête sera rejetée :

=> Decision='non'

2- Decision= 02 :

Type de requete= Exec, Objet concerné≠ nul et Ressource concernée≠ nul

Les deux paramètres ne sont pas nuls :

=> Decision='non'

3- Decision= 03 :

Type de requête= Exec, Objet concerné= nul, Ressource concernée≠ nul et le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, c'est-à-dire l'agent demandant est un contrôleur du rôle concerné, donc il n'a plus le droit de l'exécuter :

=> Decision='non'

4- Decision= 04 :

Type de requête= Exec, Objet concerné= nul, Ressource concernée≠ nul, le couple (agent_demandant, role_concerné) n'appartient plus à l'ensemble CONTROLES et il n'appartient non plus à l'ensemble DROITS. C'est-à-dire, l'agent demandant n'a pas le droit d'exécuter le rôle concerné :

=> Decision='non'

5- Decision= 06 :

Type de requête= Exec, Objet concerné= nul, Ressource concernée≠ nul, le couple (agent_demandant, role_concerné) n'appartient plus à l'ensemble CONTROLES, il appartient à l'ensemble DROITS et le rôle concerné n'est plus le rôle suivant du dernier rôle qui a créé l'état actuel :

=> Decision='non'

6- Decision= 05 :

Type de requête= Exec, Objet concerné= nul, Ressource concernée≠ nul, le couple (agent_demandant, role_concerné) n'appartient plus à l'ensemble CONTROLES mais à l'ensemble DROITS et le rôle concerné est le suivant du dernier rôle qui a créé l'état actuel :

=> Decision='oui'

7- Decision= 07 :

Type de requête= Exec, Objet concerné≠ nul, Ressource concernée= nul et IVP(objet concerné) = false

=> Decision='non'

8- Decision= 08 :

Type de requête= Exec, Objet concerné≠ nul, Ressource concernée= nul, IVP(objet concerné) = true et le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, c'est-à-dire l'agent demandant est un contrôleur du rôle concerné, donc il n'a plus le droit de l'exécuter :

=> Decision='non'

9- Decision= 09 :

Type de requête= Exec, Objet concerné≠ nul, Ressource concernée= nul, IVP(objet concerné) = true, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES et n'appartient plus à l'ensemble DROITS, c'est-à-dire l'agent demandant n'a pas le droit d'exécuter le rôle concerné :

=> Decision='non'

10-Decision= 11 :

Type de requête= Exec, Objet concerné≠ nul, Ressource concernée= nul, IVP(objet concerné) = true, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, et à l'ensemble DROITS et le rôle concerné n'est plus le rôle suivant du dernier rôle qui a créé l'état actuel :

=> Decision='non'

11-Decision= 10 :

Type de requête= Exec, Objet concerné≠ nul, Ressource concernée= nul, IVP(objet concerné) = true, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, il appartient également à l'ensemble DROITS et le rôle concerné est le rôle suivant du dernier rôle qui a créé l'état actuel :

=> Decision='oui'

12-Decision= 12 :

Type de requête= del ou add et agent concerné=nul:

=> Decision='non'

13-Decision= 13 :

Type de requête= del ou add, agent concerné≠nul et objet concerné≠nul

=> Decision='non'

14-Decision= 14 :

Type de requête = del ou add, agent concerné ≠ nul, objet concerné=nul et ressource concernée = nul

=> Decision='non'

15-Decision= 15 :

Type de requête= del ou add, agent concerné≠nul, objet concerné=nul, ressource concernée≠nul et le couple (agent_demandant, role_concerné) n'appartient plus à l'ensemble CONTROLES, c'est-à-dire l'agent demandant n'est pas le contrôleur du rôle concerné :

=> Decision='non'

16-Decision= 16 :

Type de requête= del ou add, agent concerné≠nul, objet concerné=nul, ressource concernée≠nul, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES et appartient à l'ensemble DROITS (un utilisateur n'a pas le droit d'être agent et contrôleur d'un rôle donné en même temps, il n'a pas le droit de modifier ses rôles, mais c'est son responsable « le contrôleur du rôle qu'il possède » qui a ce droit):

=> Decision='non'

17-Decision= 17 :

Type de requête= **del**, agent concerné≠nul, objet concerné=nul, ressource concernée≠nul, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, il n'appartient plus à l'ensemble DROITS et le couple (agent_**concerné**, role_concerné) n'appartient plus à l'ensemble DROITS (on ne peut supprimer un rôle à un agent qui ne le possède pas !!):

=> Decision='non'

18-Decision= 18 :

Type de requête= **del**, agent concerné≠nul, objet concerné=nul, ressource concernée≠nul, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, il n'appartient plus à l'ensemble DROITS et le couple (agent_**concerné**, role_concerné) appartient à l'ensemble DROITS:

=> Decision='oui'

19-Decision= 19 :

Type de requête= **add**, agent concerné≠nul, objet concerné=nul, ressource concernée≠nul, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, n'appartient plus à l'ensemble DROITS et le couple (agent_**concerné**, role_concerné) appartient à l'ensemble DROITS (on ne peut ajouter un rôle à un agent qui le possède déjà !!):

=> Decision='non'

20-Decision= 20 :

Type de requête= **add**, agent concerné≠nul, objet concerné=nul, ressource concernée≠nul, le couple (agent_demandant, role_concerné) appartient à l'ensemble CONTROLES, n'appartient plus à l'ensemble DROITS et le couple (agent_**concerné**, role_concerné) n'appartient plus à l'ensemble DROITS:

=> Decision='oui'

21-Decision= 21 :

Type de requête≠ **exec**, Type de requête≠ **del,** Type de requête≠ **add** et Type de requête≠ **modif_objet** (type de requête inconnu):

=> Decision='non'

22-Decision= 22 :

Type de requête≠ modif_objet et objet concerné=nul:

=> Decision='non'

23-Decision= 23 :

Type de requête≠ modif_objet, objet concerné≠nul et le dernier rôle qui a créé l'état actuel est **différent** du rôle1=(création, objet_concerné) (la modification d'un objet doit venir directement après sa création):

=> Decision='non'

24-Decision= 24 :

Type de requête≠ modif_objet, objet concerné≠nul, le dernier rôle qui a créé l'état actuel est égal au role1 et le couple (agent_demandant, role1) n'appartient plus à l'ensemble DROITS, c'est-à-dire, l'agent demandant n'est pas le créateur de l'objet concerné, il n'a pas le droit de modifier la valeur d'un objet d'un autre agent, ne pourra modifier que ses objets :

=> Decision='non'

25-Decision= 25 :

Type de requête≠ modif_objet, objet concerné≠nul, le dernier rôle qui a créé l'état actuel est égal au role1 et le couple (agent_demandant, role1) appartient à l'ensemble DROITS:

=> Decision='oui'.

Pour mettre en évidence la preuve du théorème *transition_sure*, nous avons dû énoncer et prouver d'autres théorèmes (présentés en détail dans l'annexe B). Nous allons donner dans ce qui suit, et à titre d'illustration, un de ces théorèmes qui ont été utilisés dans notre preuve générale. Il s'agit du théorème *e1_coh_e1_egal_e2* qui signifie que si deux états sont égaux et que l'un d'eux est cohérent l'autre l'est aussi. Ce théorème avec sa preuve se présentent comme suit :

```
species Systeme (..)
...
theorem e1_coh_e1_egal_e2:
    all e1 e2 : Etat,   Etat!equal(e1, e2) ->   omega(e1) ->  omega(e2)
  proof =
   <1>1 assume e1 : Etat,
      assume e2 : Etat,
      assume H1: Etat!equal(e1, e2),
      assume H2: omega (e1),
      prove omega (e2)
      <2>1 prove Modele!modele_coherent(Etat!modele_de_etat(e2))
        <3>1 prove Modele!equal(Etat!modele_de_etat(e1), Etat!modele_de_etat(e2))
          by hypothesis H1
            property e1_egal_e2_modeles
        <3>2 prove Modele!modele_coherent(Etat!modele_de_etat(e1))
          by hypothesis H2
            property e1_coh_model_coh
        <3>f qed
          by step <3>1, <3>2
            property mdl1_egal_mdl2__mdl1_coh
      <2>2 prove Ensemble_droits!ensemble_droits_coherent(Etat!droits_de_etat(e2))
        <3>1 prove Ensemble_droits!equal(Etat!droits_de_etat(e1), Etat!droits_de_etat(e2))
          by hypothesis H1
            property e1_egal_e2_droits
        <3>2 prove Ensemble_droits!ensemble_droits_coherent(Etat!droits_de_etat(e1))
          by hypothesis H2
            property e1_coh_ed_coh
        <3>f qed
          by step <3>1, <3>2
            property ed1_egal_ed2__ed1_coh
      <2>3 prove Ensemble_objets!eo_coherent(Etat!objets_de_etat(e2))
        <3>1 prove Ensemble_objets!equal(Etat!objets_de_etat(e1),Etat!objets_de_etat(e2))
          by hypothesis H1
```

```
                property e1_egal_e2_objets
            <3>2 prove Ensemble_objets!eo_coherent(Etat!objets_de_etat(e1))
              by hypothesis H2
                property e1_coh_eo_coh
            <3>f qed
              by step <3>1, <3>2
                property eo1_egal_eo2__eo1_coh
        <2>f qed
            by step <2>1, <2>2, <2>3
              definition of omega
    <1>f conclude;
...
end ;
```

Pour prouver que l'état 'e2' est cohérent (ou omega (e2)) nous avons suivi la démarche suivante :

- Il faut prouver que son modèle, son ensemble de droits et ensemble d'objets sont cohérents.
- Les trois étapes **<2>1, <2>2 et <2>3** sont définies pour chaque preuve.
- Pour que le modèle de l'état e2 soit cohérent, on a prouvé qu'il est égal à celui de l'état e1 par la propriété (prouvée) 'e1_egal_e2_modeles' et l'hypothèse H1, pour cela, la sous étape <3>1 a été définie. On a prouvé que le modèle de l'état e1 est cohérent par l'hypothèse H2 et la propriété (prouvée) 'e1_coh_model_coh', pour cela, la sous étape <3>2 a été définie. Avec les deux sous étapes <3>1 et <3>2 et la propriété (prouvée) 'mdl1_egal_mdl2__mdl1_coh', nous arrivons au terme de la preuve que le modèle de l'état e2 est cohérent.
- La même démarche a été suivie pour prouver que l'ensemble de droits et l'ensemble d'objets sont cohérents.
- Avec ces trois étapes et la définition de propriété 'omega' nous avons prouvé que l'état e2 est cohérent.

II- Tableau récapitulatif

Dans le tableau suivant, nous allons reprendre l'ensemble des règles (de certification et de mise en œuvre) définies dans le chapitre précédant et la modélisation dans le système FoCaLiZe correspondante. Pour chaque règle, nous allons expliquerons succinctement la démarche suivie pour sa modélisation et comment elle a été formalisée pour respecter les exigences de la politique de sécurité de Clark-Wilson :

Règle	Modélisation FoCAL
C1 : Toutes les IVP vérifient correctement l'intégrité des objets.	Après chaque changement d'état et avant la validation, la procédure *IVP_eo* de l'espèce *Ensembles_objets* est déclenchée pour vérifier l'intégrité de tous les objets du système.
C2 : - Certifier que toute TP accédant à une CDI intègre la laisse intègre. - Construction de listes : (TPi , (CDI1, CDI2, ..., CDIn)).	- Après déclanchement de la fonction *IVP_eo* nous pouvons savoir si nos objets intègres respectent toujours leurs CI. - L'espèce *Roles* a été définie afin de préciser quels sont les objets autorisés à être manipulés et par quelles tâches.
C3 : - Construction de listes : (userID, TPi , (CDI1, CDI2, ..., CDIn)). - Certifier que ces listes assurent la séparation des fonctions.	L'espèce *Droits* a été créée afin de définir quels sont les rôles autorisés à être manipulés et par quels agents.
C4 : Certifier que toutes les TP écrivent dans une CDI particulière, de mode d'accès "ajout seul", toutes les informations nécessaires à un audit ultérieur.	Le fichier texte ''log.txt'' est créé et utilisé (en écriture seulement) dans la fonction *transition* où elle va saisir tout événement déclenché dans le système.
C5 : Certifier que toute TP vérifie l'intégrité des UDI qu'elle utilise.	La création d'un objet se fait sur deux étapes : - La création par l'utilisateur qui va saisir en

	toute liberté la valeur de l'objet (faire appel à la fonction *create* de l'espèce *Objets*), l'objet n'est pas encore mis à contraintes. - La confirmation de changement d'état où le système vérifie l'intégrité de l'objet, s'il respecte ses CI il sera soumis à contraintes sinon il sera rejeté et l'état reste inchangé.
E1 : Lors de toute manipulation de CDI par une TP, vérification de la liste (TPi , (CDI1, CDI2, ..., CDIn))	La fonction *val_decision* de l'espèce *Decisions* vérifie si l'objet concerné par la requête appartient à l'ensemble des objets autorisés à être manipulés par la tâche concernée (dans l'ensemble des rôles).
E2 : Lors de toute exécution d'une TP par un utilisateur, vérification de la liste (userID, TPi , (CDI1, CDI2, ..., CDIn))	La fonction *val_decision* de l'espèce *Decisions* vérifie si la tâche concernée par la requête appartient à l'ensemble des tâches autorisées à être manipulées par l'agent demandant (dans l'ensemble des droits).
E3 : La règle de mise en œuvre 2 n'a de sens que si le système identifie et authentifie correctement tout utilisateur.	L'espèce des utilisateurs *Agents* hérite de l'espèce *S_setoid* du type support (int*string) où chaque agent dans le système a un unique numéro. La création d'un nouvel agent se fait en deux étapes : - Avec la fonction *create* de l'espèce *Agents*, la création est libre. On pourra créer n'importe quel agent avec n'importe quels paramètres. - Pour ajouter cet agent dans l'ensemble des agents courant du système, la fonction *ajoute_element* de l'espèce *Ensemble_fini* a été redéfini dans l'espèce *Ensemble_agents* pour vérifier si un agent de même numéro existe dans le système.

E4 : Seul un utilisateur habilité à certifier une TP peut changer la liste associée à cette TP. De plus, un utilisateur habilité pour certifier une TP ou une IVP ne doit pas disposer de droit d'exécution sur celles-ci.	L'espèce *Controles* a été créée afin de définir quelles sont les tâches autorisées à être contrôlées et par quels agents. Ces agents contrôleurs (qui sont des agents normaux) ne peuvent pas manipuler les tâches qu'ils contrôlent, mais ils peuvent modifier ses dépendances (les agents qui ont le droit de les manipuler). Ce contrôle est fait par la fonction *val_decision* de l'espèce *Décisions.*

Table 3.1: Tableau récapitulatif

Conclusion

La haute puissance de spécification du système FoCaLiZe, nous a permis de créer l'ensemble des espèces avec leurs comportements (Méthodes, Propriétés, Théorèmes et Preuves). Le théorème le plus important dans cette étude est celui de la sûreté du système. Il s'agit de prouver la cohérence de son comportement qui respecte les exigences émises par Clark et Wilson (les neufs règles citées dans le deuxième chapitre) après chaque changement d'état suite à une exécution d'une requête administrative ou non administrative.

Cette politique de sécurité s'intéresse beaucoup plus à l'assurance de l'intégrité des données. Nous avons pu prouver formellement que notre démarche avec le système FoCaLiZe assure cette propriété pour tout état accéssible par le système.

Nous avons proposé une architecture détaillée, fidèle et cohérente avec toutes les méthodes, propriétés et théorèmes pour modéliser notre politique de sécurité choisie dans le système FoCaLiZe. Il faut noter cependant qu'à titre temporaire nous avons supposé la complétude de l'organigramme de calcul du numéro de décision.
Nous disposons, maintenant, d'un cadre générique que nous allons instancier sur une étude de cas concrétisant un exemple réel, il s'agit du processus d'achat de matériel. Nous allons donc décrire le comportement de l'ensemble des collections et entités correspondante aux éléments (actifs et passifs) du système.

Chapitre IV
Étude de cas

Introduction

Dans ce chapitre, nous allons mettre en pratique la formalisation présentée dans le chapitre précédent sur un exemple concret. Nous avons choisi une procédure d'achat de matériel entre deux entreprises, cliente et vendeuse. Tout d'abord nous allons expliquer le scénario de la procédure. Ensuite, nous allons détailler les étapes de création des collections et entités correspondant aux éléments du système modélisé. Enfin, nous illustrerons notre travail en distinguant deux cas possibles de requêtes, l'une valide et l'autre non valide.

I. Scénario

Nous allons suivre la procédure étape par étape tout en respectant le scénario suivant :

- L'agent des moyens généraux de l'entreprise cliente crée un bon de commande, en envoie une copie à son magasinier et une autre au fournisseur.
- Le fournisseur livre le matériel au magasinier de l'entreprise cliente. Ce dernier le réceptionne, vérifie la conformité (en termes de quantité) avec son bon de commande, et signe un bon de livraison à envoyer au service comptabilité de son entreprise.
- Le fournisseur envoie une facture à la comptabilité de l'entreprise cliente où elle est comparée au bon de commande (quantité), et un chèque est établi au fournisseur.

A travers cet exemple, nous allons mettre en œuvre la politique de sécurité Clark-Wilson, via son implémentation par FoCaLiZe, qui nous permettra à n'importe quelle étape de préserver l'intégrité de nos données.

Nous allons maintenant instancier le modèle générique que nous avons défini dans le chapitre précédent.

II. Démarche

Pour concrétiser cet exemple, nous allons suivre la démarche suivante:

1. Déterminer l'ensemble du personnel de notre scénario, ainsi que l'ensemble des ressources et tâches avec une codification proposée pour chaque entité.
2. Implémenter et concrétiser toutes les espèces par des collections.
3. Pour chaque collection, créer l'ensemble des entités réelles correspondant aux éléments de l'étape 1, pour faire des calculs effectifs et réels sur notre exemple, à savoir :
   a. L'ensemble des personnels qui vont agir dans le scénario précédant : le Directeur Général de l'entreprise cliente (dag_a), son chef de service des moyens généraux de l'entreprise cliente (chef_serv_moy_gen_a)… ;
   b. L'ensemble des ressources manipulables par ce personnel : bon de commande, matériel, chèque …, et l'ensemble des objets concrétisant ces ressources (objet_nul, stock1, …). La cohérence de cet ensemble d'objets est assurée par la fonction ajoute_objet (définie dans l'espèce Ensembles_finis et redéfinie dans l'espèce Ensembles_objets de telle sorte qu'elle interdise l'introduction d'un objet qui ne vérifie pas ses contraintes d'intégrité).
   c. L'ensemble des tâches à exécuter dans ce scénario : création, annulation, comparaison… ;
   d. L'ensemble des relations et interdépendances effectives entre ces entités comme :
      - Les rôles (rl1=<creation, bcmd >, …),
      - Les rôles triés (<rl1, 1>, …),
      - Les droits d'exécution (<chef_serv_moy_gen_a, rl1 >, …),
      - Les contrôles (<rl1, dag_a >, …).

   L'ensemble des droits est cohérent, cette propriété étant assurée par la fonction ajoute_droit de l'espèce Ensembles_droits, de même pour l'ensemble des contrôles.
4. Créer l'état initial du système (qui doit être cohérent : son ensemble d'objets est cohérent ainsi que son ensemble des droits et contrôles).

5. Créer la première requête à exécuter, correspondant à la création d'un bon de commande par le chef_serv_moy_gen_a.
6. Un nouvel état correspondant à la décision générée systématiquement, sera créé, et d'autres requêtes seront exécutées, pour qu'elles créent elles mêmes d'autres états.
7. A la création de chaque nouvel état, le système vérifie s'il est cohérent ou non pour le valider ou le rejeter.
8. Des requêtes administratives, à exécuter par des agents contrôleurs, peuvent être lancées pour modifier les droits correspondants aux rôles qu'ils contrôlent.

   Rappelons que l'exécution des requêtes non administratives doit suivre l'ordre défini pendant la création de l'ensemble des rôles triés. Par contre, les requêtes non administratives peuvent être exécutées à la demande et à n'importe quel moment.
9. Après avoir fini d'exécuter toutes ses requêtes, le système doit revenir à l'état initial (attente d'une nouvelle commande). Pour cela, nous allons introduire un agent appelé agent_systeme, pour exécuter la requête de retour vers l'état initial.

III. Eléments à concrétiser

Dans cette section, nous allons proposer une codification pour toutes les entités (actives et passives) de notre exemple. Cette codification sera utilisée lors de l'application du cadre générique sur cet exemple dans la section suivante :

III.a-Agents

L'ensemble des personnels de notre exemple est comme suit :

Agent	Codification
L'agent nul	agent_nul
L'agent système	Agent_systeme
Le directeur de l'administration générale de l'entreprise acheteuse (cliente)	dag_a
Le chef de service des moyens généraux de l'entreprise acheteuse	chef_serv_moy_gen_a
Chef de service de comptabilité de l'entreprise acheteuse	chef_serv_compt_a
Le magasinier de l'entreprise acheteuse	mag_a
Le comptable de l'entreprise acheteuse	comptable_a
Le directeur générale de l'entreprise fournisseuse (vendeuse)	dg_f
L'agent fournisseur du matériel	fournisseur
Le magasinier de l'entreprise fournisseuse	mag_f

III.b-Ressources

L'ensemble des ressources de notre exemple est comme suit :

Ressource	Codification
La ressource nulle	ressource_nulle
Le bon de commande	bcmd
Le matériel	materiel
Le bon de livraison	bliv
La facture	fact
Le chèque	cheq
Le stock	stock

III.c-Tâches

L'ensemble des tâches de notre exemple est comme suit :

Tâche	Codification
La tâche nulle	tache_nulle
La création	creation
L'annulation	annulation
L'envoie	envoie
La réception	reception
La comparaison avec le bon de commande	comparaison_bcmd
La comparaison avec le hèque	comparaison_cheq

III.d- Rôles

L'ensemble des rôles de notre exemple est comme suit :

Rôles	Codification
Création d'un bon de commande	(creation, bcmd)
Création du materiel	(creation, materiel)
Création d'un bon de livraison	(creation, bliv)
Création d'une facture	(creation, fact)
Création d'un cheque	(creation, cheq)
Annulation d'un bon de commande	(annulation, bcmd)
Annulation du materiel	(annulation, materiel)
Annulation d'un bon de livraison	(annulation, bliv)
Annulation d'une facture	(annulation, fact)
Annulation d'un cheque	(annulation, cheq)
Envoie d'un bon de commande	(envoie, bcmd)
Envoie du materiel	(envoie, materiel)
Envoie d'un bon de livraison	(envoie, bliv)
Envoie d'une facture	(envoie, fact)
Envoie d'un cheque	(envoie, cheq)
Réception d'un bon de commande	(reception, bcmd)

Réception du materiel	(reception, materiel)
Réception d'un bon de livraison	(reception, bliv)
Réception d'une facture	(reception, fact)
Réception d'un cheque	(reception, cheq)
Comparaison du bon de commande avec le stock	(comparaison_bcmd, stock)
Comparaison du bon de commande avec le matériel	(comparaison_bcmd, materiel)
Comparaison du bon de commande avec le bon de livraison	(comparaison_bcmd, bliv)
Comparaison du bon de commande avec la facture	(comparaison_bcmd, fact)
Comparaison du cheque avec la facture	(comparaison_cheq, fact)
Retours à l'état initial du système	(tache_nulle, ressource_nulle)

III.e- Rôles triés

L'ordre de ces rôles est comme suit :

Rôles	Numéro d'ordre
Création d'un bon de commande	1
Création du matériel	5
Création d'un bon de livraison	9
Création d'une facture	13
Création d'un cheque	17
Annulation d'un bon de commande	5
Annulation du matériel	9
Annulation d'un bon de livraison	13
Annulation d'une facture	17
Annulation d'un cheque	21
Envoie d'un bon de commande	2
Envoie du matériel	6
Envoie d'un bon de livraison	10
Envoie d'une facture	14
Envoie d'un cheque	18
Réception d'un bon de commande	3
Réception du matériel	7
Réception d'un bon de livraison	11
Réception d'une facture	15
Réception d'un cheque	19
Comparaison du bon de commande avec le stock	4
Comparaison du bon de commande avec le matériel	8

Comparaison du bon de commande avec le bon de livraison	12
Comparaison du bon de commande avec la facture	16
Comparaison du cheque avec la facture	20
Retours à l'état initial du système	0

Nous remarquons que les deux rôles (Annulation d'un bon de commande, Création du matériel) ont le même numéro d'ordre car ils viennent tous les deux après le rôle numéro 4 qui est (Comparaison du bon de commande avec le stock), nous avons alors deux possibilités (comparaison positive ou négative), et chaque réponse conduit à un chemin (ce qui signifie l'exécution de l'un de ces deux rôles). Même remarque pour les rôles de numéro 9, 13 et 17.

III.f- Objets

L'ensemble **initial** des objets est (objet_nul, stock1).

III.g- Contrôles

L'ensemble des contrôles est comme suit :

- le *directeur de l'administration générale de l'entreprise cliente* contrôle les rôles suivant :

```
- (creation, bcmd)
- (annulation, bcmd)
- (reception, materiel)
- (reception, bliv)
- (reception, fact)
- (creation, cheq)
```

et il pourra à n'importe quel moment modifier les responsables des rôles.

- le *chef de service des moyens généraux de l'entreprise cliente* contrôle les rôles suivant :

```
- (envoie, bcmd)
- (comparaison_bcmd, materiel)
- (comparaison_bcmd, bliv)
- (comparaison_bcmd, fact)
```

- le *directeur général de l'entreprise fournisseuse* contrôle les rôles suivants :

```
- (reception, bcmd)
- (creation, materiel)
- (annulation, materiel)
- (creation, bliv)
- (annulation, bliv)
- (creation, fact)
- (annulation, fact)
```

- (reception, cheq)
- (annulation, cheq)

- l'agent fournisseur contrôle les rôles suivants :

- (envoie, materiel)
- (envoie, bliv)
- (envoie, fact)
- (comparaison_bcmd, stock)

- le *chef de service de comptabilité de l'entreprise cliente* contrôle les rôles suivant :

- (comparaison_cheq, fact)
- (envoie, cheq)

III.h- Droits

L'ensemble des droits est comme suit :

- le *chef de service des moyens généraux de l'entreprise cliente* a le droit d'exécuter les rôles suivant :

- (creation, bcmd)
- (annulation, bcmd)
- (reception, materiel)
- (reception, bliv)
- (reception, fact)

- le *magasinier de l'entreprise cliente* a le droit d'exécuter les rôles suivant :

- (envoie, bcmd)
- (comparaison_bcmd, materiel)
- (comparaison_bcmd, bliv)
- (comparaison_bcmd, fact)

- l'*agent fournisseur du matériel* a le droit d'exécuter les rôles suivant :

- (reception, bcmd)
- (creation, materiel)
- (annulation, materiel)
- (creation, bliv)
- (annulation, bliv)
- (creation, fact)
- (annulation, fact)
- (reception, cheq)

- le *magasinier de l'entreprise fournisseuse* a le droit d'exécuter les rôles suivant :

- (envoie, materiel)
- (envoie, bliv)
- (envoie, fact)
- (comparaison_bcmd, stock)

- le *Chef de service de comptabilité de l'entreprise acheteuse* a le droit d'exécuter les rôles suivants :

- (creation, cheq)
- (annulation, cheq)

- le *comptable de l'entreprise acheteuse* a le droit d'exécuter les rôles suivants :
  - (comparaison_cheq, fact)
  - (envoie, cheq)

- l'*agent_systeme* a le droit d'exécuter les rôles suivants :
  - (tache_nulle, ressource_nulle)

IV- Application dans FoCaLiZe

Pour appliquer cette démarche dans le système FoCaLiZe, nous suivrons les étapes suivantes :

1- L'ensemble des agents cités précédemment, sont créés via les deux collections « c_agents » et « c_ens_agents » suivantes :

```
collection c_agents implements agents= end
collection c_ens_agents implements ensembles_agents(c_agents)= end
```

Ses entités sont été créées, dans le système FoCaLiZe, de la manière suivante :

```
let agent_nul= C_agents!create(0, "agent_nul");;
let dag_a= C_agents!create(1, "dag_a");;
let chef_serv_moy_gen_a= C_agents!create(2, "chef_serv_moy_gen_a");;
let chef_serv_compt_a= C_agents!create(3, "chef_serv_compt_a");;
let mag_a= C_agents!create(4, "mag_a");;
let comptable_a= C_agents!create(5, "comptable_a");;
let dg_f= C_agents!create(6, "dg_f");;
let fournisseur= C_agents!create(7, "fournisseur");;
let mag_f= C_agents!create(8, "mag_f");;
let agent_systeme= C_agents!create(9, "agent_systeme");;
let e_agent_vide=c_ens_agents!vide;;
```

(l'ensemble des agents, est initialement vide).

```
let e_agents =
 C_ens_agents!ajoute_element(#agent_nul,
 C_ens_agents!ajoute_element(#agent_systeme,
 C_ens_agents!ajoute_element(#mag_f,
 C_ens_agents!ajoute_element(#fournisseur,
 C_ens_agents!ajoute_element(#dg_f,
 C_ens_agents!ajoute_element(#comptable_a,
 C_ens_agents!ajoute_element(#mag_a,
 C_ens_agents!ajoute_element(#chef_serv_compt_a,
 C_ens_agents!ajoute_element(#chef_serv_moy_gen_a,
 C_ens_agents!ajoute_element(#dag_a, #e_agent_vide))))))))))));;
```

Pour pouvoir rajouter des agents à l'ensemble des agents du système, nous avons utilisé la fonction ajoute_element, définie dans l'espèce ensembles_finis, et que nous avons redéfinie dans l'espèce ensembles_agents de sorte qu'elle interdise l'introduction d'un agent qui existe déjà. Cette fonction est paramétrée par deux éléments : l'agent à rajouter et l'ensemble dans lequel nous voulons le rajouter.

2- Nous avons procédé de la même façon pour créer les collections : C_taches, C_ens_taches, C_ressources, C_ens_ressources, C_frameworks, C_roles, C_ens_roles, C_roles_tries, C_ens_roles_tries, C_controles, C_ens_controles, C_droits, C_ens_droits, C_objets, C_ens_objets ainsi que leurs entités citées précédemment.

3- La collection C_modeles est créée comme suit :

```
collection C_modeles implements Modeles(C_agents, C_ens_agents, C_taches, C_ens_taches, C_ressources, C_ens_ressources, C_frameworks, C_roles, C_ens_roles, C_roles_tries, C_ens_roles_tries, C_controles, C_ens_controles);;
```

L'entité mdl est donc créée :

```
let mdl= C_modeles!create(#fr, #e_roles, #e_roles_tr, #e_controles);;
```

Où :

- fr: le frameWorck (C_ens_agents, C_ens_taches, C_ens_ressources).
- e_roles: l'ensemble des rôles défini précédemment.
- e_roles_tr : l'ensemble des rôles triés défini précédemment.
- e_controles: l'ensemble des contrôles défini précédemment.

4- Pour concrétiser l'espèce Etats, la collection C_etats est créée :

```
collection C_etats implements Etats(
C_agents, C_ens_agents, C_taches, C_ens_taches, C_ressources,
C_ens_ressources, C_frameworks, C_roles, C_ens_roles, C_roles_tries,
C_ens_roles_tries,  C_controles, C_ens_controles, C_modeles, C_droits,
C_ens_droits, C_objets, C_ens_objets);;
```

L'état initial du système $etat_0$ est créé:

let etat0= C_etats!create(#Tetat_init, #mdl, #e_objets, #e_droits, #role_tr_nul);;

Où :

- Tetat_init : car etat0 est un état initial.
- mdl : Son modèle créé précédemment
- e_objets : Son ensemble d'objets créé précédemment
- e_droits : Son ensemble de droits créé précédemment
- role_tr_nul: par convention, à l'état initial, le paramètre : dernier_role est le rôle nul.

5- Pour concrétiser l'espèce Requetes, la collection C_requetes est créée :

```
collection C_requetes implements Requetes(C_agents, C_taches, C_ressources, C_objets);;
```

La requête initiale à exécuter consiste à la création d'un bon de commande par le chef de service des moyens généraux de l'entreprise cliente :

```
let req0= C_requetes!create(#Trequete_exec, #chef_serv_moy_gen_a,
#creation, #bcmd, #objet_nul, #agent_nul);;
```

Le type de la requête est l'exécution d'une tâche sur une ressource, où :

- `chef_serv_moy_gen_a` : L'agent initiant la requête.
- `creation`: La tâche demandée.
- `bcmd` : la ressource concernée.
- `objet_nul`: L'objet concerné.
- `agent_nul`: L'agent concerné.

6- la collection C_decisions a été créée de la façon suivante :

```
collection  C_decisions implements Decisions(
 C_agents,  C_ens_agents,  C_taches,  C_ens_taches,  C_ressources,
C_ens_ressources,  C_frameworks, C_roles,  C_ens_roles, C_roles_tries,
C_ens_roles_tries,   C_controles,  C_ens_controles,  C_modeles, C_droits,
C_ens_droits,  C_objets,  C_ens_objets,  C_requetes,  C_etats);;
```

La décision d_0 correspondant à l'exécution de la requête req_0 sur l'état $etat_0$ est calculée de la manière suivante :

```
let d0= C_decisions!create(#req0, #etat0);;
```

7- Pour la création d'un nouvel état à partir de l'exécution de la requête req_0 sur l'état $etat_0$, nous déclenchons la fonction transition définie dans l'espèce Systeme. Pour cela, la collection C_systeme a été créée :

```
collection  C_systeme implements Systeme(
C_agents,  C_ens_agents,  C_taches,  C_ens_taches,  C_ressources,
C_ens_ressources,  C_frameworks, C_roles,  C_ens_roles, C_roles_tries,
C_ens_roles_tries,   C_controles,  C_ens_controles,  C_modeles, C_droits,
C_ens_droits,  C_objets,  C_ens_objets,  C_etats, C_requetes,
C_decisions);;
```

L'entité système es_0 suivante est créée :

```
let es0= C_systeme!create(#req0, #etat0);;
```

L'état résultant est donc créé de la manière suivante :

```
let etat1= C_systeme!transition(#es0);;
```

Le système donne la main à l'utilisateur pour saisir les valeurs (nom=bcmd1, val=100, …) de l'objet à créer. Si cet objet respecte ses contraintes d'intégrité, la valeur de la décision sera 'oui' sinon elle sera 'non'.

- Pour afficher le résultat de cette exécution, nous exécutons la ligne de code suivante :

```
basics#print_string(C_decisions!val_de_decision(#d0));;
```

Le résultat suivant s'affiche: `vous êtes dans le cas 5, le nouvel état a été créé.`

- Pour afficher l'ensemble des objets de ce nouvel état, nous exécutons la ligne de code suivante :

```
C_ens_objets!afficher_liste(C_etats!objets_de_etat(#etat1));;
```

Le système nous affiche alors le résultat suivant : (`objet_nul, stock1, bcmd1`).

En effet dans l'état $etat_0$, l'ensemble des objets était (`objet_nul, stock1`), et après l'exécution de cette requête l'ensemble est changé en (`objet_nul, stock1, bcmd1`).

De même, le paramètre 'dernier_role' de l'état $etat_0$ était `(#role_nul=(#tache_nulle,#ressource_nulle),0)` et après l'exécution de cette requête il est changé en `(#rl1=(#creation,#bcmd),1)`.

Comme la requête a été exécutée sans erreurs alors l'objet créé respecte ses contraintes d'intégrité et donc l'état résultant est cohérent.

Cas d'état incohérent :

Le rôle suivant à exécuter sur l'état $etat_1$ est l'envoi du bon de commande par le *magasinier de l'entreprise cliente*, la requête suivante doit être exécutée :

```
let req1=
c_requetes!create(#Trequete_exec,#mag_a,#envoie,#ressource_nulle,#bcmd1,
#agent_nul);;
```

Nous modifierons cette requête en changeant le paramètre 'agent demandeur' (qui est `mag_a` ayant le droit d'envoyer le bon de commande) par l'agent `chef_serv_moy_gen_a` qui est le contrôleur de ce rôle, et qui n'en a pas le droit :

La requête est donc :

```
let req1'=C_requetes!create(#Trequete_exec, #chef_serv_moy_gen_a, #envoie, #ressource_nulle, #bcmd1, #agent_nul);;
```

La décision d_1' correspondante à l'exécution de la requête req_1' sur l'état $etat_1$ est calculée:

```
let d1'= C_decisions!create(#req1', #etat1);;
```

Pour afficher le résultat de cette exécution, nous lançons la ligne de code suivante :

```
basics#print_string(C_decisions!val_de_decision(#d1'));;
```

Le résultat suivant s'affiche:

```
Erreur N°:08, la requête n'est pas administrative et l'agent demandant est un contrôleur du rôle demandé, cet agent n'a pas le droit d'exécuter ce rôle!!! L'état reste le même !!
```

Pour confirmer que l'état n'a pas subi de changements, son dernier rôle sera le suivant :

```
le dernier rôle de l'état suivant est: (creation, bcmd) avec l'ordre d'exécution : 1.
```

Nous rappelons qu'à chaque fois qu'un événement se déclenche dans le système, il sera enregistré dans le fichier **log.txt**.

Comme les rôles sont ordonnés, les requêtes non administratives doivent être exécutées l'une après l'autre. Par contre, les requêtes administratives peuvent être exécutées sur demande, à n'importe quel moment. Pour cela, et pour simplifier l'exemple, nous n'aborderons que l'exécution des requêtes non administratives.

L'exécution des différentes requêtes, le calcul des différentes décisions et la création des différents états résultants ont abouti à la création de l'automate suivant :

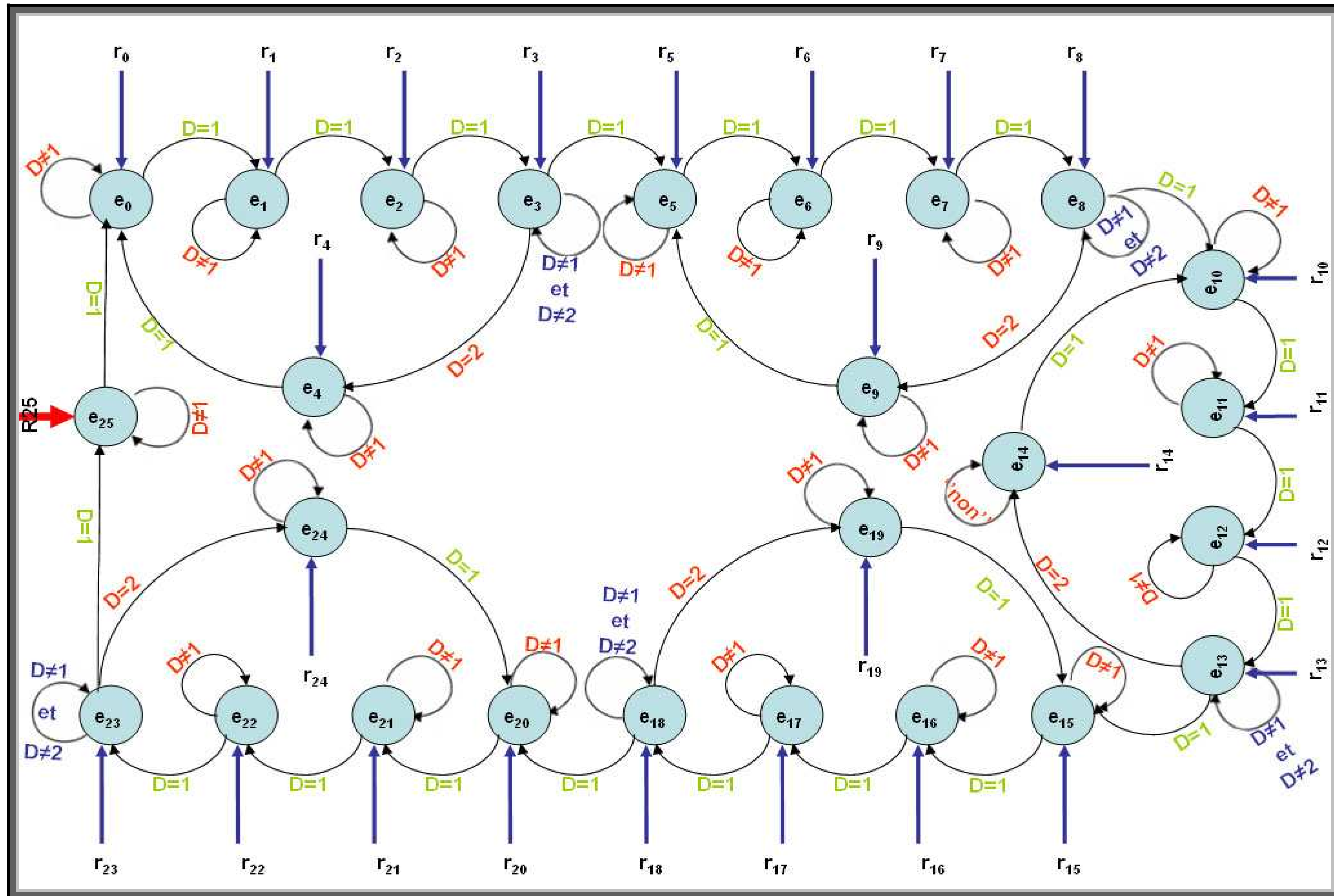

Figure 4.1 : états, requêtes, décisions et transitons de la procédure d'achat de matériel selon la politique Clark-Wilson

Les identifiants e_i représentent les différents états du système qu'il pourra atteindre dans le temps tout en exécutant une succession de requêtes. Alors que les r_i représentent les requêtes (non administratives) correspondant aux rôles triés précédemment. Les D_i représentent les décisions calculées après exécution de chaque requête.

Nous expliquons, dans ce qui suit, l'exécution de deux requêtes : r2 (avec ses deux réponses possibles) et r3 (avec ses trois réponses possibles) :

1- r2 = < type_requete=''Trequete_exec'', fournisseur, Reception, (ressource='Tnul', objet= '$bcmd_1$'), agent_concerne='Tnul'>
signifiant l'exécution de la réception du bon de commande bcmd1 par le fournisseur. Les deux réponses suivantes sont possibles :

1.a- D≠1 : si la réponse est négative, à cause d'une condition quelconque qui n'est pas vérifiée (le bcmd1 n'existe pas, le dernier rôle créant l'état e2 n'est pas la création de ce bon de commande, …), la requête sera rejetée et par la suite l'état reste le même (**E_2**).

1.b- D=1 : si la réponse est positive : toutes les conditions sont vérifiées et tous les paramètres d'accès aussi, l'état suivant sera créé:

E_3= < type_etat='Tnon_init', modele, objets= {objet_nul, $stock_1$, $bcmd_1$}, ensemble_droits, **dernier_rôle= (Reception, bcmd)**>

Ici le seul paramètre qui a été changé est 'dernier_role'.

2- R_3= < type_requete=''Trequete_exec'', mag_f, Comparaison_bcmd, (ressource='Tnul', objet= '$stock_1$'), agent_concerne='Tnul'>
signifiant l'exécution de la comparaison du bon de commande bcmd1 qui a été réceptionné (décision de r2 est positive) avec l'objet stok1 par le magasinier de l'entreprise fournisseuse. Les trois réponses suivantes sont possibles :

2.a- D=2 : si les droits d'accès et paramètres de sécurité sont vérifiés mais la comparaison est négative, l'état suivant sera créé :

E_4 = < type_etat='Tnon_init', modele, objets= {objet_nul, $stock_1$, $bcmd_1$}, ensemble_droits, dernier_rôle= (Comparaison_bcmd, stock)>
et la requête r4 d'annulation de bon de commande avec retour vers l'état initial **E_0** sera exécutée.

2.b- D=1 : si les droits d’accès et paramètres de sécurité sont vérifiés et la comparaison est positive, l’état suivant sera créé :

E_5= < type_etat=’Tnon_init’, modele, objets= {$stock_1$, $bcmd_1$(cdi)}, ensemble_droits, **dernier_rôle= (Comparaison_bcmd, stock)**>

Ici le seul paramètre qui a été changé est ‘dernier_role’.

2.c- (D≠1 et **D**≠2) : si les droits d’accès et paramètres de sécurité ne sont pas vérifiés (l’agent demandant n’a pas le droit de comparer le bon de commande avec le stock, le rôle demandé n’est pas le rôle suivant du paramètre dernier_role de l’état **E_3**, …), la requête sera rejetée et par conséquent l’état reste le même (**E_3**).

Nous rappelons que l’exécution de toutes ces requêtes ne modifiera jamais la composition administrative du système (les ensembles DROITS et CONTROLES définis précédemment), donc elles ne modifieront jamais la cohérence du système. Par contre, les requêtes administratives peuvent changer le comportement (la composition administrative) du système.

Conclusion

La démarche proposée dans ce chapitre nous a permis d’appliquer notre spécification FoCaLiZe de la politique de sécurité Clark-Wilson sur un exemple concret de procédure d'achat de matériel.

Avec cette implantation, nous avons expliqué comment le système contrôle toutes les opérations que ses agents veulent exécuter. Nous avons vu que les opérations permises peuvent être exécutées, alors que celles ne sont pas permises ne doivent jamais être exécutées, jusqu'à ce qu’elles respectent les conditions d’exécution.

Le scénario de la procédure d’achat de matériel a été détaillé, l’ensemble des collections a été créé ainsi que l’ensemble des entités correspondant à notre exemple. Des requêtes administratives et non administratives ont été exécutées sur les différents états du système. Si les conditions d’exécution et paramètres d’accès sont vérifiés, la décision générée est automatiquement ‘**oui**’ sinon elle a la valeur ‘**non**’.

Rappelons pour terminer que la généricité de notre modèle, permet son utilisation sur d’autres exemples de différents domaines.

Conclusion générale

L'atelier FoCaLiZe est un outil très puissant, qui nous a permis de formaliser et d'implanter la politique de sécurité de Clark-Wilson. Avec cet atelier FoCaLiZe, nous avons pu dans un premier temps spécifier cette politique de sécurité, qui était définie en langage naturel. Ensuite, nous avons introduit plusieurs propriétés et théorèmes de sûreté, dont les démonstrations ont été générées par l'outil de preuve automatique ''Zenon'' dont dispose l'atelier FoCaLiZe. On rappelle que ces preuves sont ensuite soumises à vérification en utilisant ''Coq''.

L'objectif principal de notre étude est de définir le théorème ''Transition_sûre'' et de le prouver dans le système FoCaLiZe. Ce théorème consiste à assurer la cohérence de l'état résultant de l'exécution d'une requête sur un état cohérent. Avec ce théorème et si l'état initial du système est cohérent, tous les états résultants après application d'une suite de requêtes sont forcement cohérents.

Il est à noter que notre travail s'est révélé plus long et complexe que prévu pour les raison suivantes, dont il faudra tenir compte pour son évolution :

- Le nombre important des espèces et collections que nous avons créés pour pouvoir spécifier et implanter cette politique de sécurité.
- Le nombre important de paramètres de sécurité et leurs combinaisons pour parcourir tous les cas possibles d'exécution des différentes requêtes.
- Les quatre types de requêtes (exécution d'un rôle, modification d'un objet, ajout d'un rôle à un agent et suppression d'un rôle à un agent) et les vingt cinq possibilités de décisions qui peuvent être générées.
- La définition de la fonction de transition qui change le système d'un état à un autre, avec le parcours de tous les cas possibles d'exécution ou de rejet des requêtes, et avec la préservation de la cohérence du système.
- La preuve du théorème ''Transistion_sûre'' a nécessité la définition d'autres théorèmes avec leurs preuves.

Après avoir combiné les différentes conditions d'exécution des requêtes, nous avons tracé un organigramme de calcul de la décision. Nous nous somme basé pour la

définition de la fonction de transition et la preuve du théorème Transition-sûre sur cet organigramme. Pour cela, nous avons admis comme hypothèse la complétude de cet organigramme. Démontrer cette complétude (ou à défaut compléter notre organigramme) constituera une première perspective

Nous avons conçu notre système de tel sorte qu'il soit **flexible** et **extensible**. Il sera donc aisé d'y introduire des éléments supplémentaires comme : un agent administrateur possédant des droits spécifiques, d'autres tâches comme l'ajout et la suppression de nouveaux agents, la modification de l'ensemble de contrôles …

Disposer d'un SGBD pourrait se révéler intéressant afin de disposer de fonctionnalités supplémentaires comme la sauvegarde des états et la restauration après avoir découvert des anomalies ou des incohérences de l'état en cours, avec la possibilité d'auditer les raisons pour lesquelles cet état est devenu incohérent, et par la suite renforcer notre conception FoCaLiZe.
Ces audits pourraient permettre de déterminer si notre organigramme est complet ou si nous avons oublié certains cas (combinaisons de conditions d'exécution de requêtes) qui n'auraient pas encore été traités.

Bibliographie

[ABO03] A. ABOU EL KALAM. Modèles et politiques de sécurité pour les domaines de la santé et des affaires sociales. Laboratoire d'Analyse et d'Architecture des Systèmes du Centre National de la Recherche Scientifique. décembre 2003
.

[BDD07] David Delahaye Richard Bonichon and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Vol 4790, 2007.

[BIB77] K. J. Biba, Integrity considerations for secure computer systems, Technical Report TR-3153, The Mitre Corporation, Bedford, MA, April 1997.

[BLP73] D. E. Bell and L. J. Lapadula, Secure computer systems: Mathematical foundations, Technical Report ESD-TR-73-278, vol. 1, The Mitre Corp., Bedford, MA, 1973.

[BLP74] D. E. Bell and L. J. Lapadula, Secure computer systems: A refinement of the mathematical model, Technical Report ESD-TR-278, vol. 3, The Mitre Corp., Bedford, MA, 1974.

[BOU00] Sylvain Boulmé, "Spécification d'un environnement dédié à la programmation certifiée de bibliothèque de calcul formel", Thèse de doctorat, Unversité Paris 6,2000.

[COQ97] The COQ development team. The Coq Proof Assistant Reference Manuel V6.1, rapport INRIA N° 0203, 1997.

[COQ10] Coq. The Coq Proof Assistant, Tutorial and reference manual. INRIA –LIP–LRI–LIX–PPS, 2010. Distribution available at: http://coq.inria.fr/.

[CW87] D.D. Clark and D.R. Wilson, A Comparison of Commercial and Military Computer Security Policies, IEEE, 1987.

[DEN 76] D. E. Denning, A lattice model of secure information flow, Communications of the ACM, 19(5): 236-243, May 1976
.

[DOL04] D.Doligez. Zenon, first-order prover with coq-checkable output. 2nd Workshop on Coq and rewriting, September 2004.

[DOL10] Damien Doligez. Zenon reference manual. available at www.inria.fr/zenon, February 2010.

[FCLr09] T. Hardin, F. Pessaux, P. Weis, D. Doligez. FoCaLiZe 0.6.0 Reference Manual, CNAM, INRIA, et LIP6, December 2009.

[FCLt09] FoCaLiZe Team. A Short Tutorial for FoCaLiZe : Implementing Sets, CNAM, INRIA, et LIP6, July 2009.

[FOC03] FOC development team. A brief FoC tutorial, Version 0.0, July 01 2003.

[FOC06] L'équipe de développement FoCAL. Tutoriel FoCAl, Version 0.3, CNAM, INRIA, et LIP6, Septembre 2006.

[GHJ03] E. Gureghian, Th. Hardin, M. Jaume. A full formalisation of the Bell and La Padula security model. SPI-LIP6, University Paris 6, France. 2003.

[HAD05] A. HADDAD. Modélisation et Vérification de Politiques de sécurité. Université Joseph Fourier.2005.

[HRU76] M.H. Harrison, W. L. Ruzzo and J. D. Ullman, Protection in operating systems, Communications of the ACM,19(8): 461-471, 1976.

[JM05] M. Jaume, C. Morisset.Formalisation and implantation of access control models. In Information Assurance and Security (IAS'05). International Confirence on Information Technology, ITCC, 2005.

[JR12] Journées Francophones des Langages Applicatifs. Développement de systèmes sécurisés avec l'atelier FoCaLiZe. M. Jaume & R. Rioboo - février 2012.

[LAM71] B. LAMPSON, "Protection", 5th Princeton Symposium on Information Sciences and Systems, 1971.

[LAN81] Carl E. Landwehr, Formal Models for Computer Security, ACM Computing Surveys, 13(3): 247-278, 1981.

[LUD07] Ludovic Mé, Politiques et modèles de sécurité, ENST-Bretagne, janvier 2007.

[MCL90] J. Mclean, The specification and modeling of computer security, Computer, 23(1): 9-16,January 1990.

[Phil 11] développement de logiciel critique en FoCalize. Philippe AYRAULT, 22 Avril 2011-LIP6 (UPMC).

[PJ 03] V. Prevosto and M. Jaume. Making proofs in a hierarchy of mathematical structures. In Proceedings of the 11th Calculemus Symposium, Rome, sep 2003.

[PRE03] Virgile Prevosto. Conception et implantation du langage FoC pour le développement de logiciels certifiés. PhD thesis, Université Paris VI, 2003.

[REM02] Didier Rémy. Using, Understanding, and Unraveling the OCaml Language. In Gilles Barthe, editor, Applied Semantics. Advanced Lectures. LNCS 2395. Springer Verlag, 2002.

[RIO03] R.Rioboo. Mathematical communications with FoC. Mathematical

Knowledge.

[RIO05] L'Atelier FoCaL, Renaud Rioboo, LIP6, Grenoble, Décembre 2005. Management Symposium, December 2003. Invited Talk, Edinburgh.

[SD01] Pierangela Samarati and Sabrina de Capitani di Vimercati, Access Control: Policies, Models, and Mechanisms, FOSAD 2000, LNCS 2171, pages 137-196, 2001.