UNIVERSITÉ DE MONTRÉAL

MODÉLISATION ET VÉRIFICATION DU FLUX D'INFORMATION POUR LES SYSTÈMES ORIENTÉS OBJETS

CYRIL HENRY NLENG
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

MÉMOIRE PRÉSENTÉ EN VUE DE L'OBTENTION
DU DIPLÔME DE MAÎTRISE ÈS SCIENCES APPLIQUÉES
(GÉNIE INFORMATIQUE)
AOÛT 2014

© Cyril Henry Nleng, 2014.

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Ce mémoire intitulé :

MODÉLISATION ET VÉRIFICATION DU FLUX D'INFORMATION POUR LES SYSTÈMES ORIENTÉS OBJETS

présenté par : NLENG Cyril Henry
en vue de l'obtention du diplôme de : Maîtrise ès sciences appliquées
a été dûment accepté par le jury d'examen constitué de :

M. ADAMS Bram, Doct., président
M. MULLINS John, Ph.D., membre et directeur de recherche
M. KHOMH Foutse, Ph.D., membre

Merci à mes parents et à ma famille, pour tout.

REMERCIEMENTS

Je souhaite remercier en premier lieu mon directeur de recherche, le professeur John Mullins, pour sa patience et sa compréhension. Son enthousiasme pour la recherche, sa curiosité intellectuelle et son souci du détail ont été des inspirations pour moi et je lui suis très reconnaissant de m'avoir toujours encouragé.
Je tiens ensuite à remercier mes amis et collègues, qui m'ont beaucoup soutenu eux aussi et avec lesquels j'ai vécu de très beaux moments. Je pense plus particulièrement à Laurence Gwend, Steve Ndengue. Merci aussi aux membres du CRAC, étudiants et professeurs, que j'ai côtoyés avec beaucoup de plaisir ces dernières années. Sans oublier le personnel du Département d'informatique et de génie logiciel. Enfin, merci à mes parents et à ma famille, pour tout.

RÉSUMÉ

Afin d'assurer la protection de la confidentialité et l'intégrité des systèmes orientés objet, cet article propose une modèle d'étiquettes décentralisées (méfiance mutuelle et sécurité décentralisée) pour le contrôle des flux d'information dans des modèles UML et son intégration dans UML. Le document présente la syntaxe et la sémantique formelle des modèles UML (les diagrammes de classes, diagrammes d'objets et diagrammes d'états) étendus avec des fonctionnalités supplémentaires qui prennent en charge le contrôle des flux d'information. Il prend en charge la vérification du flux d'information à l'exécution et permet de s'assurer que le système n'admet que des flux d'information légaux. Ainsi des politiques de sécurité flexibles sont définies sur des diagrammes UML juste en les étiquetant. Le contrôle du flot de données et des canaux de transmission est effectuée au niveau des diagrammes de classes, diagrammes d'objets et diagrammes d'états. Cela permet de définir le flux d'information lorsque le comportement des objets changent.

ABSTRACT

For protection of the confidentiality and integrity of object-oriented systems, this paper proposes a decentralized label model for control of information flow in UML system models with mutual distrust and decentralized security and its integration into UML. The paper introduces UML (class diagrams, object and statecharts) syntax and formal semantics extensions with additional features that support information flow control. It supports run-time verification of information flow so that the system model can be certified to permit only admissible information flows. Using this, flexible security policies can be easily defined through UML diagrams just by labelling them. The control of data flows and transmitting channels is performed at the level of Class diagrams, Object diagrams and Statechart diagrams. This allows to define information flow when the behaviour of objects change.

TABLE DES MATIÈRES

DÉDICACE iii

REMERCIEMENTS iv

RÉSUMÉ v

ABSTRACT vi

TABLE DES MATIÈRES vii

LISTE DES TABLEAUX x

LISTE DES FIGURES xii

LISTE DES ANNEXES xiv

LISTE DES SIGLES ET ABRÉVIATIONS xv

CHAPITRE 1 INTRODUCTION 1
- 1.1 Définitions et concepts de base 2
  - 1.1.1 Paragdime de programmation orienté objet 2
  - 1.1.2 UML et OCL 2
  - 1.1.3 Les méthodes formelles 2
  - 1.1.4 Sécurité 3
- 1.2 Problématique : UML, propriétés de sécurité et flux d'information 4
- 1.3 Objectifs de recherche 5
- 1.4 Méthodologie 6
- 1.5 Contribution 8
- 1.6 Plan du mémoire 8

CHAPITRE 2 REVUE DE LITTÉRATURE 9
- 2.1 Extension des modèles UML 9
  - 2.1.1 UMLsec 10
  - 2.1.2 SecureUML 11
- 2.2 Formalisation des modèles UML (Syntaxe et semantique) 12
  - 2.2.1 Formalisation des machines à états UML 12

2.2.2 Sémantique opérationnelle structurée pour les diagrammes d'états . . 14
2.3 Formalisation des modèles UML et extension des opérateurs de logique temporelle 16
2.3.1 USVF (UML-based static verification framework) 16
2.3.2 Vérification des failles de sécurité dans la conception des logiciels . . . 18
2.3.3 Modélisation et vérification des politiques de sécurité via des diagrammes UML 20
2.3.4 Extension OCL pour la Modélisation et spécification UML 21
2.4 Méthodes d'analyse de flux d'information basées sur les types 22
2.4.1 Analyse du flux d'information par des langages typé sécurité 22
2.4.2 Modèle des étiquettes décentralisées : DLM 23
2.5 Synthèse comparatrice des approches 26

CHAPITRE 3 DÉFINITION D'UN MODÈLE DE SÉCURITÉ POUR LES SYSTÈMES ORIENTÉS OBJET 31
3.1 Flux d'information dans les systèmes orientés objet 31
3.2 Hiérarchie d'utilisateurs 34
3.3 Syntaxe et caractéristiques des étiquettes 35
3.3.1 Définition formelle d'une étiquette de confidentialité 37
3.3.2 Définition formelle d'une étiquette d'intégrité 39
3.4 Définition d'un flux d'information 42
3.4.1 Relation de restriction entre les étiquettes 43
3.4.2 Étiquettes calculées 46

CHAPITRE 4 SYNTAXE ET SÉMANTIQUE DES MODÈLES UML 48
4.1 Hypothèses de formalisation UML 48
4.1.1 Domaines des valeurs 49
4.1.2 Fragments UML : Diagrammes de classes, diagrammes d'objets et diagrammes d'états 49
4.2 Modélisation UML 51
4.2.1 Syntaxe des diagrammes UML 51
4.2.2 Sémantique des diagrammes UML 72

CHAPITRE 5 SPÉCIFICATION ET VÉRIFICATION DU FLUX D'INFORMATION 79
5.1 Modélisation du système de transition orienté objet 80
5.1.1 Système de transition orienté objet 80
5.1.2 Descriptions des effets des transitions 82

5.1.3 Exemple 83
5.2 Vérification des propriétés de flux d'information 85
5.2.1 Principe de vérification des propriétés de flux d'information 86
5.2.2 Algorithme de vérification des propriétés de flux d'information 88

CHAPITRE 6 ÉTUDE DE CAS 90
6.1 Introduction 90
6.1.1 Spécification informelle du système des Cavéats 91
6.1.2 Différence entre les études de cas 93
6.2 Modélisation UML 94
6.2.1 Étape 1 - Diagramme de classes UML 94
6.2.2 Étape 2 - Diagramme d'objets UML 96
6.2.3 Étape 3 - Diagramme d'états 97
6.2.4 Étape 4 - Définition du flux d'information 100
6.3 Modèle formel du système 107
6.3.1 Modèle formel des classes 108
6.3.2 Modèle formel des objets 109
6.3.3 Modèle formel des diagrammes d'états 110
6.4 Vérification de la sécurité 113
6.4.1 Système de transition 114
6.4.2 Vérification du système de transition 118

CHAPITRE 7 CONCLUSION 120
7.1 Synthèse des travaux 120
7.2 Limitations de la solution proposée 121
7.3 Améliorations futures 121

RÉFÉRENCES 123

ANNEXES 126

LISTE DES TABLEAUX

Tableau 2.1 Tableau de synthèse comparatrice - Extension UML 26
Tableau 2.2 Tableau de synthèse comparatrice - Formalisation diagrammes d'états UML 27
Tableau 2.3 Tableau de synthèse comparatrice - Formalisation UML et sécurité . . 29
Tableau 4.1 Description des déclencheurs des transitions 67
Tableau 4.2 Table des évaluations des gardes 68
Tableau 4.3 Règle de transition entre deux états 75
Tableau 5.1 Description de la fonction labelEffect 82
Tableau 6.1 Tableau des classifications 92
Tableau 6.2 Tableau des caveats 92
Tableau 6.3 Tableau des projets 92
Tableau 6.4 Visibilité, Pays, Rang 96
Tableau 6.5 Niveau, Cavéat, Projet 96
Tableau 6.6 Étiquettes de sécurité - objet u 101
Tableau 6.7 Étiquettes de sécurité - objet u 101
Tableau 6.8 Étiquettes de sécurité - objet u 101
Tableau 6.9 Étiquettes de sécurité - objet $c1A$ 101
Tableau 6.10 Étiquettes de sécurité - objet $c1B$ 102
Tableau 6.11 Étiquettes de sécurité - objet $c1R$ 103
Tableau 6.12 Étiquettes de sécurité - objet $c2R$ 103
Tableau 6.13 Étiquettes de sécurité - objet ceo 103
Tableau 6.14 Étiquettes de sécurité - objet $canus$ 103
Tableau 6.15 Étiquettes de sécurité - objet $usuk$ 104
Tableau 6.16 Étiquettes de sécurité - objet $canusuk$ 104
Tableau 6.17 Étiquettes de sécurité - objet $nato$ 104
Tableau 6.18 Étiquettes de sécurité - objet $unCoalition$ 104
Tableau 6.19 Étiquettes de sécurité - objet ca 104
Tableau 6.20 Étiquettes de sécurité - objet us 105
Tableau 6.21 Étiquettes de sécurité - objet uk 105
Tableau 6.22 Étiquettes de sécurité - objet ger 105
Tableau 6.23 Étiquettes de sécurité - objet egy 105
Tableau 6.24 Étiquettes de sécurité - objet $c1P$ 105
Tableau 6.25 Étiquettes de sécurité - objet $c2P$ 106

Tableau 6.26 Étiquettes de sécurité - méthode *receive* 106
Tableau 6.27 Étiquettes de sécurité - méthode *accessTable* 106
Tableau 6.28 Étiquettes de sécurité - méthode *updateTable* 106
Tableau 6.29 Étiquettes de sécurité - méthode *changeRank* 107

LISTE DES FIGURES

Figure 1.1 Exemple d'un diagramme d'états et une transition 4
Figure 1.2 Diagrammes objets étendus avec des étiquettes de sécurité 7
Figure 2.1 Exemple de diagramme d'états UMLSec 11
Figure 2.2 Architecture USVF 17
Figure 2.3 Méthode de vérification des diagrammes UML avec SPIN 19
Figure 2.4 Méthodologie 20
Figure 3.1 Flux d'information dans un diagramme d'états 33
Figure 3.2 Définition des ensembles d'objets 34
Figure 3.3 Illustration hiérarchie d'utilisateurs 35
Figure 3.4 Définition des ensembles d'étiquettes 36
Figure 4.1 Listes des notations - Domaines de valeurs 49
Figure 4.2 Listes des notations - Diagrammes de classes et d'objets 51
Figure 4.3 Listes des notations - diagramme d'états 52
Figure 4.4 Ensembles des Opérateurs $\mathsf{COp}, \mathsf{BOp}, \mathsf{UOp}$ 55
Figure 4.5 Diagramme de classe 56
Figure 4.6 Diagramme d'objets 60
Figure 4.7 État Simple (SIMPLE) 63
Figure 4.8 État composite (AND) 64
Figure 4.9 État composite (OR) 64
Figure 4.10 Transition vers un état AND 73
Figure 4.11 Transition vers un état OR 73
Figure 5.1 Description de la règle de transition 81
Figure 5.2 Diagramme de classe 83
Figure 5.3 Diagramme d'objets 83
Figure 5.4 Diagramme d'états 84
Figure 5.5 Exemple de système de transition 84
Figure 6.1 Diagramme de classe 95
Figure 6.2 Diagramme d'objets (partie 2) 97
Figure 6.3 Diagramme d'états - $UserStateChart$ 98
Figure 6.4 Diagramme d'états *Document* 99
Figure 6.5 Diagramme d'états - $DocumentStateChart2$ 100
Figure 6.6 Ensemble des attributs de classe 108
Figure 6.7 Ensemble des parametres 108

Figure 6.8 Ensemble des methodes 108
Figure 6.9 Ensemble des classes 109
Figure 6.10 Ensemble des objets 109
Figure 6.11 Ensemble des instances de methodes 109
Figure 6.12 Ensemble des regions 110
Figure 6.13 Ensemble des etats 110
Figure 6.14 Transitions `TabPStateChart1` 111
Figure 6.15 Transitions `DocumentStatechart2` 112
Figure 6.16 Transitions `TabSStateChart1` 112
Figure 6.17 Transitions `TabCStateChart1` 112
Figure 6.18 Transitions `UserStateChart` 113
Figure 6.19 Système de transition simplifié 114

LISTE DES ANNEXES

Annexe A Annexe A - Algorithme de restriction sur des étiquettes de sécurité . 126
Annexe B Annexe B - Validations et contraintes sur des diagrammes UML . . . 130

LISTE DES SIGLES ET ABRÉVIATIONS

IETF	Internet Engineering Task Force
OSI	Open Systems Interconnection
DLM	Decentralized label momdel
ASM	Abstract State Machine
UML	Unified Modelling Language
AVFI	Algorithme de vérification du flux d'information
RBAC	Role-Based Access Control
SAM	Software Architecture Modelling
RDDC	Recherche et Développement de la Défense Canada (RDDC)

CHAPITRE 1

INTRODUCTION

Les environnements technologiques des entreprises se complexifient de plus en plus et requièrent davantage des exigences de fiabilité et de sécurité. Afin d'appréhender cette complexité des systèmes informatiques et logiciels, et simplifer leur développement, la plupart des entreprises ont adopté l'approche orientée objet. Cette approche propose de structurer et analyser séparément les différents composantes d'un système. Ces dernières permettent de décrire séparément et à différents niveaux d'abstraction, les aspects statiques, dynamiques et fonctionnels d'un système. Parmi les langages de conception jouissant d'une grande popularité, UML est devenu le standard pour la modélisation des systèmes orientés objet. La modélisation à l'aide du langage UML demande de développer des techniques qui facilitent la détection et la prévention des erreurs de conception, ceci afin de réduire les coûts et délais des phases de validation. Des méthodes d'analyse ont été développées afin de spécifier et de vérifier des erreurs de conception liées à la sécurité (accès ou modification illégal de données). Parmi les aspects de la sécurité qui peuvent-être considérés, les propriétés de flux d'information revêtent une importance non négligeable. Elles permettent d'indiquer le flot de données légal dans un système.

Les méthodes usuelles d'analyse de modèles UML utilisent des méthodes formelles afin de spécifier et vérifier la sécurité dans les modèles UML. L'ingéniere des systèmes sécurisés propose des métamodèles UML étendus avec de nouveaux artéfacts ou métamodèles UML. Ces modèles étendus représentent les systèmes sécurisés. D'autres méthodes transforment les modèles UML et les requis de sécurité (contrôle d'accès) en modèles formels qui peuvent-être analysés grâce à des techniques de vérification telles que le *model-checking* ou le *theorem-proving.*

Nous proposons dans cette thèse, une méthode de spécification qui utilise les notations graphiques d'UML étendues avec des requis de sécurité. Ces requis de sécurité sont modélisés grace à des étiquettes de sécurité décentralisées. Nous nous intéressons plus précisément à la spécification et vérification du flux d'information qui survient lors de l'exécution des diagrammes d'états UML. Nous développons une syntaxe et une sémantique formel du modèle UML(diagramme de classe, d'objets, d'états) étendu avec les étiquettes de sécurité. Pour la vérification du flux d'information, on dérive à partir du modèle formel un système de transition à états finis.

1.1 Définitions et concepts de base

Nous définissons dans cette section certains concepts qui seront utiles à la bonne compréhension des sections et chapitres subséquents. La plupart de ces concepts, sinon la totalité, sont sûrement bien connus. Nous nous contenterons donc, en substance, de rappeler leur définition générale sans développer davantage.

1.1.1 Paragdime de programmation orienté objet

Les systèmes informatiques orientés objet sont des systèmes informatiques développés sur la base du paragdime orienté objet. Le système est décrit à l'aide de classes et d'objets qui représentent des concepts et des entités du monde réel. Chaque classe possède une structure interne et des caractéristiques, et interagit avec d'autres classes ou objets (instances de classes) du système. L'interaction entre plusieurs classes (ou objets) aboutit à l'obtention d'un modèle conceptuel du système logiciel que l'on souhaite implanter. Les interactions entre les différentes classes (ou objets) permet de réaliser les fonctionnalités logicielles requises par les spécifications. La description du modèle logiciel est réalisée grace à l'utilisation de langage de modélisation.

1.1.2 UML et OCL

Le langage UML est un langage de modélisation qui permet de décrire un logiciel informatique. La modélisation et la description des logiciels à l'aide du langage de modélisation UML (Unified Modelling Language) permet d'obtenir des modèles UML. Le modèle UML décrit uniquement des modèles conceptuels mais ne permet pas de décrire des contraintes sur les entités du modèle. En effet, il est parfois nécessaire de décrire des contraintes supplémentaires sur des modèles UML. Ces contraintes sont souvent décrites en langage naturel et n'éliminent pas les ambiguïtés. Des langages de spécification tels que OCL(Object Constraint Language) ont été ajoutés au langage UML pour spécifier des contraintes (pré-conditions et post-conditions sur des méthodes, invariants de classe).
OCL est un langage d'expression de contraintes sur des modèles UML. Ces expressions indiquent généralement des exigences fonctionnelles et non-fonctionnellees que doivent respecter le système modélisé ou des requêtes sur des objets décrits dans un modèle UML.

1.1.3 Les méthodes formelles

Les méthodes formelles permettent de définir mathématiquement et rigoureusement les propriétés de fonctionnement d'un système. Elles définissent la syntaxe et la sémantique formelles du système. Elles s'appliquent à n'importe quelle phase du cycle de développement

d'un logiciel. L'utilisation de la logique mathématique et la rigueur de ces méthodes permettent de garantir l'impossibilité de comportements indésirables dans un système ou de garantir l'existence de comportements souhaitables.

1.1.4 Sécurité

La sécurité des applications informatiques est un requis non fonctionnel qui n'est pas souvent pris en compte lors des phases de conception et de modélisation et les défauts de sécurité peuvent être difficiles et coûteux à déceler dans le cycle de développement des logiciels. Cette absence de spécification et vérification engendre des défauts de conception qui ne peuvent être détectés que lors des phases de tests et de validation. Les différents aspects de la sécurité sont décrits dans les paragraphes suivants.

Propriétés de sécurité et politiques de sécurité Les propriétés de sécurité sont des requis qui doivent être respectés par un système afin de garantir la sécurité des données qui y circulent. Les propriétés de sécurité qui doivent être garanties sont les suivantes :

1. Confidentialité : propriété d'une donnée dont la diffusion doit être limitée aux seules personnes autorisées.
2. Intégrité : propriété d'une donnée dont la valeur est conforme à celle définie par son propriétaire.
3. Disponibilité : propriété d'un système capable d'assurer ses fonctions sans interruption, délai ou dégradation, au moment même où la sollicitation en est faite.

Une politique de sécurité est un ensemble de règles mises en place dans un système afin de garantir le respect et l'application des propriétés de sécurité.

Modèle de sécurité

Un modèle de sécurité est une description formelle (généralement à l'aide d'une notation formelle ou mathématique) d'une politique de sécurité d'un système. Cette définition inclut souvent la sécurité des fonctionnalités fournies par un système et les relations permises entre les entités en fonction de leurs niveaux de sécurité et/ou des mécanismes de sécurité qui les contrôlent. Les modèles de sécurité sont utilisés dans l'évaluation de la sécurité.

Contrôle d'accès et Sécurité multi-niveau

Le contrôle d'accès est une politique de sécurité pour traiter des données possédant des sensibilités différentes (niveaux de sécurité différents). Elle permet également l'accès simultané

aux données pour des utilisateurs ayant des autorisations de sécurité et des besoins différents. Dans cette approche le système est divisé en sujets (utilisateurs) et objets (données), les sujets demandant l'accès aux objets.

Flux d'information et treillis de sécurité

De façon générale, un treillis de sécurité est défini comme un ensemble de niveaux de sécurité L auquel on associe une relation d'ordre partielle $\preceq$ qui permet de classifier les niveaux de sécurité en catégorie (généralement haut niveau/bas niveau ou privée/publique) Par exemple, si $l_1, l_2 \in L$, $l_1 \preceq l_2$ signifie que l'information est autorisée à circuler de l_1 vers l_2. Le flux d'information se définit de façon générale comme la transmission d'information d'une place à une autre (ou d'une donnée à une autre) dans un environnement de sécurité multi-niveau. Un flux d'information est dit sécuritaire (dans le cas de la confidentialité et de l'intégrité) si la suite des actions réalisées pour transmettre de l'information n'entraine pas de fuites d'informations des données privées vers les données publiques.

1.2 Problématique : UML, propriétés de sécurité et flux d'information

Dans notre étude on recherche les défauts de sécurité (flux d'information illégal) que peut présenter un système orienté objet lors de la phase de conception. La conception est réalisée à l'aide des diagrammes UML suivants : diagramme de classes, diagramme d'objets et diagrammes d'états. On s'intéresse plus précisement au flux d'information qui survient lorsque les transitions des diagrammes d'états s'exécutent. Le langage UML ne possède pas une syntaxe et une sémantique capable d'exprimer et de vérifier fomellement des politiques de flux d'information dans des diagrammes d'états.

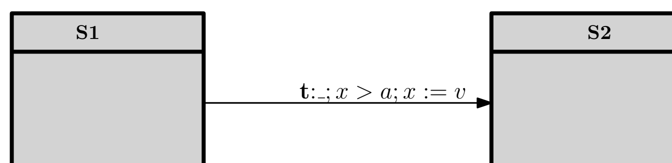

Figure 1.1 Exemple d'un diagramme d'états et une transition

À titre d'illustration (figure 1.1), le diagramme d'état possède une transition qui affecte une variable v à une autre variable x. On considère également un treillis de sécurité $(\{l, h\}, \preceq)$, avec les niveaux de sécurité h et l, et la relation d'ordre $\preceq$ qui donne le sens légal du flux d'information. Les variables x, a et v possèdent chacune un niveau de sécurité. Cette transition peut éventuellement causer des flux d'information illégaux. En effet, l'opération d'affectation de la variable a à la variable x peut causer un flux d'information si le niveau de sécurité de a

(haut niveau) est supérieur à celui de x (bas niveau). De plus si la garde $x > a$ sur la transition est une expression de haut niveau, l'opération $x := v$ ne devrait pas être une expression de bas niveau. De plus, la spécification des modèles de sécurité ne prend pas en considération les contraintes comportementales exprimées à partir des diagrammes d'états. En effet, il est impossible d'obtenir un diagnostic de sécurité lorsque les transitions des diagrammes d'états s'exécutent. Ce diagnostic inclurait la spécification et la vérification des niveaux de sécurité des variables impliquées dans les transitions à chaque pas d'exécution (Les variables x, a et v à la figure 1.1).

Plusieurs approches ont été développées afin de spécifier et vérifier des contraintes de flux d'information sur des modèles UML. Les contributions principales de ces approches sont les suivantes. La première approche consiste à transformer les diagrammes UML en modèles formels (grâce à des notations formelles), et à vérifier les contraintes de sécurité sur ce modèle en utilisant des techniques de vérification telles que le *model-checking* ou le *theorem-proving*. Une syntaxe et une sémantique formelle des diagrammes UML est défini et les propriétés de sécurité sont exprimées à partir de formules de logiques temporelles (LTL ou CTL) étendues avec de nouveaux opérateurs (capable d'exprimer des propriétés de sécurité). La seconde approche consiste à concevoir de nouveaux métamodèles UML. Il s'agit de fragments UML étendus avec de nouveaux artéfacts capable d'exprimer des contraintes de sécurité. Les modèles UML développés à partir de ces métamodèles sont ensuite décrits de façon formelle (syntaxe et sémantique) afin d'appliquer les méthodes de vérification formelles (*model-checking* ou le *theorem-proving*). Néanmoins, aucune de ces approches ne prend en considération la spécification et la vérification du flux d'information lors de l'exécution des diagrammes d'états. En effet, la spécifiation du flux d'information permet d'assurer la sécurité de bout-en-bout. L'information ne doit pas être lue par une opération, peu importe comment elle est utilisée. L'information ne doit pas être modifiée à partir de données moins fiables. De plus, la définition des niveaux de sécurité et des treillis de sécurité n'est pas exprimée de façon formelle au niveau des diagrammes UML. De plus, les contraintes de sécurité sont exprimées très souvent au niveau des diagrammes de structures (diagramme de classe ou diagramme d'objets) et non au niveau des transitions dans les diagrammes à états. Les diagrammes d'états permettent de spécifier des opérations de lecture et d'écriture sur des données (actions des transitions) et leurs conditions d'exécution (gardes des transitions).

1.3 Objectifs de recherche

L'objectif principal de la recherche est de concevoir un modèle de sécurité capable de spécifier et de vérifer des requis de sécurité pour des systèmes orientés objet modélisés avec

le langage UML. Il s'agit de spécifier le flux d'information pour des systèmes orientés objet décrits à l'aide de diagramme de classe, d'objets et d'états et de pouvoir le vérifier à l'exécution des diagrammes d'états. Le diagramme d'objets décrit les objets pour lesquels on veut spécifier et vérifier le flux d'information. Les diagrammes d'états décrivent une séquence de transitions d'états d'un objet (en réponse à des événements). Il permettent d'identifier des attributs d'objet, qui influent sur le comportement de l'objet. Dans notre cas, le comportement à vérifier est la sécurité. Notre étude portera plus précisément sur la modélisation, la spécification et la vérification formelle des politiques de confidentialité et d'intégrité capable de garantir un flux d'information sécuritaire entre les objets lorsque ces derniers interagissent. La vérification des politiques de sécurité permettrait de détecter des défauts de sécurité dans des modèles UML lorsque les transitions des diagrammes d'états s'exécutent. L'accomplissement des sous-objectifs suivants permettra d'atteindre l'objectif principal :

1. Modélisation et spécification formelle des politiques de flux d'information.
2. Modélisation et spécification formelle des modèles UML étendues avec des contraintes de sécurité. Ces contraintes sont exprimées grâce au modèle formel des politiques de flux d'information.
3. Définition d'une méthode de vérification du flux d'information.
4. Validation du modèle de sécurité à l'aide d'une étude de cas.

1.4 Méthodologie

Pour accomplir les différents objectifs fixés à la section 1.3, on suivra les étapes suivantes :

Modélisation et spécification formelle des politiques de flux d'information Pour décrire les politiques de sécurité, on définira de façon formelle les propriétés de confidentialité et d'intégrité sur une donnée dont on veut garantir la sécurité. À partir de la définition des propriétés de confidentialité et d'intégrité, on définira une politique de flux d'information. Notre politique de flux d'information est celle du modèle des étiquettes décentralisées (Myers et Liskov (1998)). Ce modèle est appliqué ainsi sur une donnée.

1. Les contraintes d'intégrité et de confidentialité appliquées sur une donnée sont exprimées à l'aide d'une étiquette de sécurité. Une étiquette de sécurité est un couple constitué d'une étiquette de confidentialité et d'une étiquette d' intégrité, et elle permet de vérifier les usagers qui ont des droits en lecture et en écriture sur la donnée.
2. On définit un treillis de sécurité sur l'ensemble des étiquettes de sécurité. Ce treillis de sécurité indique le flux d'information légal dans le système orienté objet.

Modélisation et spécification formelle des modèles UML On se propose de concevoir un modèle formel capable d'exprimer les paradigmes des modèles UML, ce modèle formel est étendu avec des étiquettes de sécurité.

1. On décrit notre système orienté objet à partir des fragments UML suivants : diagramme de classe, diagramme d'objets, diagramme d'états.
2. On étend l'expressivité des artéfacts UML (diagramme d'états, de classe, d'objets) avec des étiquettes de sécurité. Les étiquettes de sécurité sont associées aux attributs des objets (et classes), aux paramètres des méthodes et aux variables et spécifient leur niveau de sécurité.

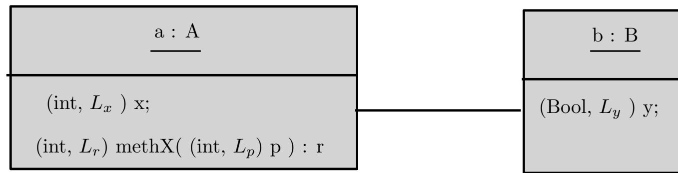

Figure 1.2 Diagrammes objets étendus avec des étiquettes de sécurité

À la figure 1.2, l'attribut x de l'objet a possède l'étiquette de sécurité L_x et la méthode $methX$ possède un paramètre de retour r qui possède une étiquette de sécurité L_r et un paramètre de méthode p qui possède une étiquette de sécurité L_p.

Définition d'une méthode de vérification du flux d'information Afin de vérifier le flux d'information, on suivra les étapes suivantes. Premièrement, on représente le sytème orienté objet sous la forme d'un système de transition étiquetté. Le système de transition est obtenu à partir du modèle formel du système orienté objet. Deuxièmement, on vérifie dans chaque état du système de transition que les affectations survenues sont sécuritaires, c'est à dire que le flux d'information respecte la politique définie par le treillis de sécurité sur l'ensemble des étiquettes de sécurité.

Validation des améliorations apportées à l'outil par une étude de cas On se propose de valider les changements en réalisant une étude de cas. Il s'agit d'un travail de recherche qui a fait l'objet d'une étude dans la thèse (Oarga (2005)) et a été réalisé en collaboration avec RDDC. On proposera un modèle UML simplifié afin de valider notre approche.

1.5 Contribution

La principale contribution de notre recherche est la spécification et la vérification du flux d'information dans les diagrammes d'états des modèles UML. Pour ce faire, on réalise un modèle formel du flux d'information dans des diagrammes d'états, ainsi qu' une analyse dynamique de ces derniers. Les détails de notre contribution sont :

1. On définit un modèle de sécurité dans un contexte orienté objet. Ce modèle de sécurité est défini à partir d'un modèle formel d'étiquettes de sécurité. Les étiquettes de sécurité sont appliquées sur les attributs de chaque objets, les paramètres des méthodes et les variables. On définit la relation d'ordre partielle $\sqsubseteq$ entre deux étiquettes de sécurité qui donne le sens légal du flux d'information. Ainsi l'ensemble des étiquettes de sécurité associé à $\sqsubseteq$, nous permet de définir un treillis de sécurité dans un contexte orienté objet.
2. On définit un modèle formel du système orienté objet. On fournit la syntaxe de notre système orienté objet pour des diagrammes de classes, d'objets et d'états auxquels on associe des étiquettes de sécurité. La sémantique nous permet d'exprimer premièrement les contraintes de sécurité sur les attributs, les variables et les paramètres. Deuxièmement la sémantique permet de donner les contraintes de sécurité sur les gardes et les actions des transitions (diagrammes d'états). Les contraintes de sécurité exprimées à partir de la valeur des étiquettes de sécurité, c'est à dire l'ensemble des objets qui ont des droits en écriture et lecture.

Notre travail fournit, à notre connaissance, la première analyse qui garantit un flux d'information sécuritaire sur des fragments UML (diagramme de classes, d'objets, d'états), notamment lors de l' exécution des diagrammes d'états.

1.6 Plan du mémoire

Le mémoire est structuré de la suivante : Le chapitre 2 présente la revue littérature, qui énumerera les approches existantes dans la littérature, et présente les mécanismes et modèles de sécurité développés pour des modèles UML. Dans le chapitre 3, on développera un modèle de sécurité formel pour les systèmes orientés objets. La syntaxe et la sémantique des modèles UML sont décrites au chapitre 4. Le chapitre 5 décrit le système de transition orienté objet et la méthode de vérification du flux d'information. Le chapitre 5 portera sur une étude de cas. Les résultats de cette étude permettront de valider les modèles de sécurité et les propriétés de flux d'information. Dans le chapitre 6, une conclusion du mémoire et une synthèse de notre étude seront présentée.

CHAPITRE 2

REVUE DE LITTÉRATURE

Ce chapitre présente l'état de l'art en rapport avec les principales méthodes de modélisation et de vérification de la sécurité dans des systèmes orientés objet. On décrit également les méthodes de modélisation des modèles UML et les méthodes classiques d'analyse du flux d'information. La première partie (Section 2.1) présente les approches et méthodes qui étendent le langage UML afin d'exprimer des contraintes de sécurité sur des diagrammes UML. La deuxième partie (Section 2.2) quant à elle couvre les méthodes qui décrivent les modèles formels des diagrammes UML (syntaxe et sémantique). La troisième partie (Section 2.3) décrit les méthodes qui utilisent les formules de logiques temporelles pour spécifier des contraintes de sécurité sur des modèles UML et appliquer des techniques de vérification automatique (*model-checking*). La quatrième partie (Section 2.4) donne une description des méthodes d'analyse du flux d'information à travers les langages typés sécurité. La dernière partie (Section 2.5) est consacrée à une étude comparative des différentes approches décrites dans les sections précédentes. En plus de proposer un modèle formel pour les diagrammes UML (diagramme de classes, d'objets et d'états), notre étude propose des mécanismes d'extension (et leurs formalismes) sur des modèles UML grâce à des étiquettes de sécurité. Le modèle formel des étiquettes de sécurité et des modèles UML fournit un modèle de sécurité formel pour des diagrammes UML et une méthode d'analyse statique du flux d'information dans des diagrammes d'états. En effet, les mécanismes d'extensions sur les variables, les attributs des objets et les paramètres de méthodes permet d'obtenir un type de sécurité (niveau de sécurité) pour chaque élément du programme, ce qui permet de vérifier des erreurs lors de l'exécution des diagrammes d'états.

2.1 Extension des modèles UML

Cette section présente les méthodes qui étendent les métamodèles UML afin de modéliser des politiques de sécurité. Les métamodèles sont des grammaires qui décrivent comment les modèles UML valables sont construits. L'extension sécuritaire de ces modèles consiste en l'ajout de nouveaux artéfacts capable d'exprimer des contraintes de sécurité. Plusieurs approches ont été proposées, néanmoins dans la suite nous présentons uniquement UMLsec et SecureUML qui sont les mécanismes d'extensions les plus utilisées.

2.1.1 UMLsec

UMLsec, proposé par Jürjens (2002b), est une extension du langage UML qui fournit des artéfacts pour le développement de systèmes sécurisés. La modélisation des éléments de sécurité dans les modèles UML se fait grâce à la spécification d'un profil UML. Les modèles ainsi obtenus sont évalués par rapport à des vulnérabilités. UMLsec s'applique aux fragments UML suivants : diagramme d'activité, diagramme de classe, diagramme de séquence, diagramme d'états, diagramme de paquetage et diagramme de déploiement.
UMLsec fournit des mécanismes de sécurité au langage UML en définissant des profils UMLsec. Ces mécanismes de sécurité sont exprimés grâce à des stéréotypes, des balises UMLsec et des contraintes UMLSec. Un profil UMLsec qu'on note pUMLsec est une combinaison d'un stéréotype, d'une balise et d'une contrainte. Il est décrit de façon formelle par :

$$\text{pUMLsec} = (\text{stereotypes},\text{tags},\text{constraints})$$

avec stereotypes $\in \mathsf{Stereotypes}$ (ensemble de stéréotypes), tags $\in \mathsf{Tags}$ (ensembles de balises) et constraints $\in \mathsf{Constraints}$ (ensemble de contraintes). Le flux d'information est exprimé sur des systèmes grâce au stéréotype *no-down flow* et à la balise *secret* et permet de s'assurer qu'il n'y a pas de fuite d'informations lorsque les systèmes s'exécutent.

Pour spécifier une politique de flux d'information sur des diagrammes d'états, un profil UMLsec est défini en combinant le stéréotype *no down-flow* et une balise *secret*. On souhaite que lors de l'exécution d'un diagramme d'états, la contrainte *no down-flow* soit respectée (pas de fuite d'information des variables publiques vers les variables privées). De plus, les variables et méthodes annotées avec la balise *secret* sont considérées *privées* (les autres sont considérées *publiques*). Ainsi, au niveau du diagramme d'états on vérifie qu'aucune information n'est divulguée durant son exécution. On distingue lors de l'exécution d'un diagrammes d'états deux types de méthodes : les méthodes qui écrivent (modifient) dans les variables et les méthodes qui lisent les variables. Les méthodes classées *publiques* peuvent lire des variables classées *publiques*, et les méthodes classées *privées* peuvent lire des variables classées *privées* ou *publiques*. De plus, les méthodes classées *publiques* ne peuvent lire que des variables qui ont été écrites à partir de méthodes classées *publiques*, et les méthodes classées *privées* ne peuvent lire que des variables qui ont été écrites à partir de méthodes classées *privées* ou *publiques*.

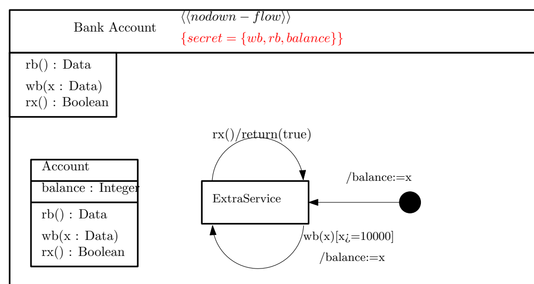

Figure 2.1 Exemple de diagramme d'états UMLSec

L'exemple illustré à la Figure 2.1 présente un objet de type *Account*, qui permet à sa variable *balance* (classée *privée*) d'être lue grâce à la méthode *rb* dont la valeur de retour est classée *privée*. La variable *balance* est modifiée grâce à la méthode *wb* classée *privée*. Dans cet exemple il peut y avoir une fuite d'information puisque l'information de la variable *balance* peut-être déduite à partir de la méthode *rx* classée *publique*.

La sémantique opérationnelle des diagrammes d'états est décrite par le formalisme des ASM dans Jürjens (2002a). Des applications de UMLsec sont la modélisation du contrôle d'accès discrétionnaire et la gestion des permissions sur des objets, méthodes et attributs d'un diagrammes UML (Jürjens (2008), Jürjens *et al.* (2008)). Des outils et des méthodes Jürjens et Shabalin (2007) ont été développés afin de faire de la vérification des modèles UML. Les niveaux de sécurité définis par la méthodologie UMLsec ne se limitent qu'à deux niveaux : *publique* et *privé*. De plus, la sémantique ne prend pas en compte l'exécution simultanée de plusieurs diagrammes d'états. Ces deux manquements sont complétées dans notre étude de deux façons. Premièrement, on utilise le modèle des étiquettes décentralisé pour spécifier les niveaux de sécurité. Deuxièmement, notre syntaxe et sémantique permet de décrire des exécutions simultanées des diagrammes d'états à travers la définition du concept des *locations*, décrites à la section 4.2.2.2.

2.1.2 SecureUML

SecureUML (Brucker *et al.* (2006); Lodderstedt *et al.* (2002); Basin *et al.* (2009)) est une extension du langage UML. SecureUML est utilisé pour la modélisation des mécanismes de contrôle d'accès basé sur les rôles (RBAC) et permet la spécification de politiques de contrôle d'accès. SecureUML définit un modèle de données basé sur le RBAC.

Avec SecureUML, la sécurité est gérée en trois étapes.

Étape 1 : modélisation des environnements d'autorisation À cette étape, on modélise les environnements d'autorisation avec UML et le système UML sur lequel on veut

appliquer la sécurité. L'association entre le modèle d'autorisation et le modèle du système UML permet ainsi de définir le modèle de sécurité que l'on souhaite implanter.

Étape 2 : transformation du modèle du système en contraintes OCL À cette étape, on génère les opérations sécuritaires pour chaque classe, attribut et méthode dans le modèle de conception.

Étape 3 : transformation du modèle de sécurité en contraintes OCL À cette étape, les opérations et les contraintes de sécurité sont spécifées à l'aide du langage OCL, et vérifiées automatiquement avec l'analyseur de modèle HOL-OCL (Brucker et Wolff (2002)). Les politiques de contrôle d'accès spécifiées avec SecureUML sont tranformées en un modèle spécifié en UML/OCL.

SecureUML combine la simplicité d'utilisation graphique de UML et l'expression de politiques de contrôle d'accès sur des diagrammes de classes UML à l'aide de OCL. Cette approche se limite à l'analyse de modèle UML statique (diagrammes de classe), tandis que la notre permet également de spécifier et vérifier la sécurité sur des diagrammes de classes.

La prochaine section 2.2 présente les méthodes qui décrivent les modèles formels des diagrammes UML. La syntaxe et la sémantique des diagrammes d'états UML sont spécifiées.

2.2 Formalisation des modèles UML (Syntaxe et semantique)

Plusieurs formalismes existent pour les modèles UML. Dans cette section, on s'intéresse surtout à la formalisation (syntaxe et sémantique) des diagrammes d'états UML.

2.2.1 Formalisation des machines à états UML

Lilius et Paltor (1999) ont proposé une formalisation des diagrammes d'états UML. L'approche de formalisation s'effectue en deux étapes et est utilisée à des fins de vérification par *model-checking*. Premièrement, la structure des diagrammes d'états UML est traduite en une syntaxe concrète. Dans la deuxième étape, la sémantique opérationnelle des diagrammes d'états est définie. Cette formalisation supporte toutes les fonctionnalités des diagrammes d'états UML et a été mise en oeuvre dans l'outil de vérification de modèles vUML.

La syntaxe est la suivante. Soit un diagramme d'état SM, un diagramme d'état possède un ensemble d'états Σ. Cet ensemble d'états Σ est divisé en trois sous ensembles disjoints, Σ_{fs} un ensemble d'états finaux, Σ_{ss} un ensemble d'états simples et Σ_{cs} un ensemble d'états composés.

Les configurations des diagrammes d'états sont décrites à partir de l'ensemble des termes

T_Σ qui decrivent les ensembles d'états actifs. Une transition est décrite par un six-tuplet $(s,\ t, e, g, a,\ s')$, avec $s, s' \in T_\Sigma$, $t \in Tn$, Tn est l'ensemble des noms des transitions, e est un déclencheur, a est une action, et g est une garde. La syntaxe définit $\mathcal{L}_a$ et $\mathcal{L}_p$ qui sont respectivement un langage pour décrire les actions et les gardes des transitions. La syntaxe complète d'un diagramme d'états UML est donnée par :

$$SM = \langle \Sigma, entryAction, exitAction, activity, \Phi, source, target, trigger, guard, effect \rangle$$

avec :

- Σ est l'ensemble des états du diagramme UML
- *entryAction* : $\Sigma \rightarrow \mathcal{L}_a$, une fonction qui décrit les actions à l'entrée d'un état
- *exitAction* : $\Sigma \rightarrow \mathcal{L}_a$ une fonction qui décrit les actions à la sortie d'un état
- *activity* : $\Sigma \rightarrow \mathcal{L}_a$ une fonction qui décrit les activités qui s'exécutent dans un état
- Φ est l'ensemble des transitions $\Phi \rightarrow T_\Sigma \times T_n \times T_\Sigma$
- *source* : $\Phi \rightarrow \Sigma$, est une fonction qui donne la configuration de départ (ensemble des états actifs)
- *target* : $\Phi \rightarrow \Sigma$, est une fonction qui donne la configuration d'arrivée (ensemble des états actifs)
- *guard* : $\Phi \rightarrow \mathcal{L}_p$, est une fonction qui évalue une garde de la transition
- *effect* : $\Phi \rightarrow \mathcal{L}_a$, est une fonction qui donne les actions que la transition exécutent

Algorithm 1 RTC ()

```
while true do
  if queue ≠ 0 then
    currentevent ← queue.dequeue()
    enabled ← enabled(currentstate, currentevent)
    Select a step Γ
    newstate ← currentstate
    for all γ ∈ Γ do
      newstate = nextstate(γ)currentstate
      execute effect(γ)
    end for
    currentevent ← newstate
  end if
end while
```

Sémantique La sémantique opérationelle des diagrammes d'états est décrite par un algorithme appelé *run-to-completion* (rtc, algorithme 1). Cet algorithme permet de calculer les

éléments suivants lors de l'exécution du diagramme d'états d'un objet. D'abord, il détermine l'ordre et la priorité sur les déclencheurs de transitions. Ensuite, il spécifie l'ensemble des transitions actives, des transitions exécutables, des transitions conflictuelles et la priorité entre les transitions. Enfin, il détermine la nouvelle configuration du système après l'exécution des transitions (ensembles des états UML activés).

Les contributions principales de cette étude sont les suivantes. Premièrement, une syntaxe formelle, simple et déclarative qui permet de décrire les éléments des diagrammes d'états UML. De plus, la syntaxe permet également d'exprimer l'ensemble des configurations possibles d'une machine à états (ensemble des états activés par une transition) La deuxième contribution est l'élaboration d'un algorithme qui donne la sémantique opérationelle des diagrammes d'états UML. Cet algorithme détermine les transitions qui sont effectuées et les nouvelles configurations du système. Comparativement à notre approche, il manque des éléments dans la syntaxe et la sémantique qui seraient capable d'exprimer des requis de sécurité. Plus précisément, on a ajouté un modèle de sécurité formel dans des diagrammes d'‘états UML. La syntaxe et sémantique définies dans notre étude permettent de modéliser simultanément plusieurs diagrammes d'états.

2.2.2 Sémantique opérationnelle structurée pour les diagrammes d'états

von der Beeck (2002)) définit, de façon formelle, une syntaxe et une sémantique opérationelle structurée. Ce travail est basé en partie sur le résultat des travaux de Latella *et al.* (1999a). La syntaxe et la sémantique sont définies à partir d'automates hiérarchiques et prennent en considération les éléments suivants des diagrammes d'états : les transitions entre des états imbriqués (sous-états), les états *historiques*, les actions *entry* et *exit* (respectivement en entrée et en sortie d'un état), la modélisation de séquences de transitions avec des structures de Kripke et finalement la définition des configurations du système (ensemble des états actifs).

La syntaxe des diagrammes d'états est la suivante. Soient $\mathcal{N}$, $\mathcal{T}$, Π, $\mathcal{A}$ les ensembles de noms d'états, de noms de transitions, d'évènements et d'actions respectivement. La syntaxe d'un diagramme d'états UML-SC est définie de façon formelle par :

$$s = [n, (en, ex)], \text{si UML-SC est un état simple (pas d'états internes)}$$
$$s = [n, (s_1, \ldots, s_k), l, T, (en, ex)], \text{si UML-SC est un état OR}$$
$$s = [n, (s_1, \ldots, s_k), (en, ex)], \text{si UML-SC est un état AND}$$

avec :

- n est le nom du diagramme d'états, $n \in \mathcal{N}$
- $s_1, \ldots, s_k$ sont des sous-états imbriqués dans le diagramme s

- l est l'index actif du diagramme s et s_l est le sous-état actif, $l \in \rho = \{1 \dots k\}$
- T est l'ensemble des transitions entre les sous-états $s_1, \dots, s_k$, $T \subseteq \mathcal{T} \times \rho \times 2^{\mathcal{N}} \times \Pi \times \mathcal{A}^{\star} \times \rho \times \{none, deep, shallow\}$
- en est l'ensemble des actions lorsqu'on accède à l'état, $en \in \mathcal{A}^{\star}$
- ex est l'ensemble des actions lorsqu'on quitte un état, $ex \in \mathcal{A}^{\star}$

La sémantique des diagrammes d'états permet de décrire :

- des séquences d'exécution de transition grâce à l'utilisation de structure de Kripke (système de transition étiquetté),
- le non-déterminisme (pris en compte dans la description des actions *entry* et *exit*),
- des états activés lorsqu'une transition survient,
- des règles de transitions (choix et priorité des transitions),

Le non-déterminisme est également pris en compte dans la description des actions *entry* et *exit*. La sémantique statique est définie à deux niveaux. La première sémantique est définie à partir d'un ensemble de fonctions qui donnent les configurations possibles (résultat de l'exécution des transitions) et les actions (en entrée et sortie des états) qui s'exécutent.

- $confAll$: *UML-SC* $\longrightarrow 2^{2^{\mathcal{N}}}$ retourne l'ensemble de tous les sous-états potentiellement actifs dans un état AND ou OR,
- $conf$: *UML-SC* $\longrightarrow 2^{\mathcal{N}}$ retourne l'ensemble de tous les sous-états actifs,
- $subconf$: *UML-SC* $\longrightarrow 2^{\mathcal{N}}$ retourne l'ensemble de tous les sous-états actifs qui sont des états simples,
- $exit$: *UML-SC* $\longrightarrow 2^{\mathcal{A}^{\star}}$ retourne l'ensemble d'actions qui s'exécutent lorsqu'on accède à un état,
- $entry$: *UML-SC* $\longrightarrow 2^{\mathcal{A}^{\star}}$ retourne l'ensemble d'actions qui s'exécutent lorsqu'on quitte un état

Une transition permet de faire passer le système d'une configuration $conf$ à une nouvelle configuration $conf'$, et permet d'exécuter les séquences d'actions *exit* et *entry*, les configurations étant obtenues à partir des fonctions $confAll$, $conf$ et $subconf$. La deuxième sémantique interprète le diagramme d'états comme un système de transitions. La sémantique opérationnelle d'un diagramme d'états associe à un diagramme d'états une structure de Kripke et est donnée par la fonction :

$$[\![.]\!] : UML - SC \longrightarrow \mathcal{K}$$

où $[\![s]\!]$ = K, et K = (S,st,$\longrightarrow$), avec :

- $K \in \mathcal{K}$, est une structure de Kripke
- $S = UML - SC \times \Pi^{\star}$ est l'ensemble des états de K.
- st = (s,ϵ_0), est l'état initial de K, avec $\epsilon_0 \in \Pi^{\star}$
- $\longrightarrow \subseteq S \times S$ est la relation de transition de K.

Cette section propose des approches formelles pour la description des diagrammes d'états, mais ne décrit d'aucune façon des requis de sécurité sur des diagrammes UML. La prochaine suivante 2.3 décrit des travaux qui formalisent des diagrammes d'états UML et la sécurité dans ces mêmes diagrammes. La sécurité est spécifiée sur les systèmes orientés objet en ajoutant des opérateurs de sécurité aux logiques temporelles ou en les exprimant directement sous la forme de formules temporelles.

2.3 Formalisation des modèles UML et extension des opérateurs de logique temporelle

Des études ont été faites sur la syntaxe et la sémantique des modèles UML (Section 2.2). On décrit dans cette section des travaux qui formalisent des diagrammes d'états UML et la sécurité dans ces mêmes diagrammes. La sécurité est spécifiée sur les systèmes orientés objet en ajoutant des opérateurs de sécurité aux logiques temporelles ou en les exprimant directement sous la forme de formules temporelles.

2.3.1 USVF (UML-based static verification framework)

USVF (développée par Siveroni *et al.* (2010)) est une méthode de vérification des modèles UML pour soutenir la conception et la vérification des systèmes logiciels dans les premières phases du cycle de développement de logiciel. Cette approche est une suite des travaux de Siveroni *et al.* (2008) et a été développée afin de prendre en considération la spécification et la vérification de la sécurité d'un système logiciel. Les fragments UML considérés sont les diagrammes de classes et d'états. USVF effectue la vérification statique des modèles UML constitués de diagrammes de classes UML et de diagrammes d'états étendus avec un langage de description d'actions. On définit une sémantique de modèles UML et un langage de spécification de propriété conçu pour raisonner sur les propriétés temporelles et générales des diagrammes d'états UML.

Architecture USVF

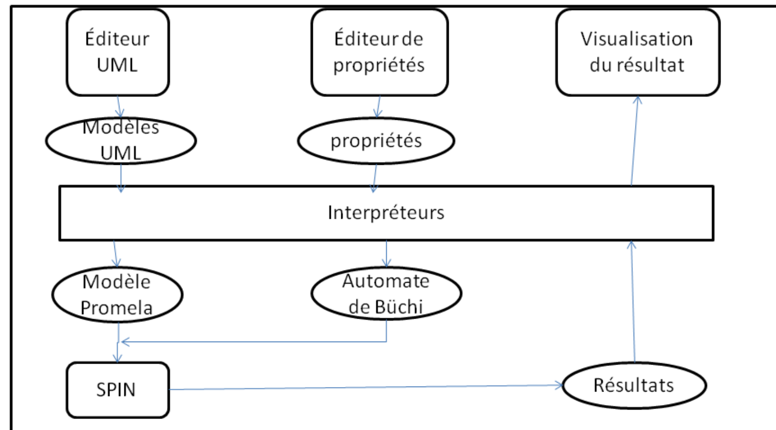

Figure 2.2 Architecture USVF

La méthode USVF décrit la syntaxe des classes UML et leurs attributs, les objets UML (instances des classes), ainsi que les méthodes et leurs paramètres. On décrit également les diagrammes d'états. L'ensemble des diagrammes d'états SM est décrit de façon formelle par :

$$\mu \in SM = State \times State \times \mathcal{P}(State) \times \mathcal{P}(Trans)$$

avec $State$ l'ensemble des états UML et $Trans$ l'ensemble des transitions. Les transitions sont décrites par un déclencheur, une garde, une action, un ensemble d'états actifs source et un ensemble d'états actifs cible.

La méthode USVF décrit la sémantique en trois parties. Premièrement, on présente l'ensemble des valeurs utilisées par la sémantique et on définit un ensemble de fonctions qui permettent d'évaluer les expressions (valeurs des attributs des objets). Ensuite, on présente la sémantique de l'exécution d'une action des transitions. Enfin, on introduit la sémantique opérationnelle des diagrammes d'états qui décrit les configurations possibles des classes et objets lorsque les diagrammes d'états UML s'exécutent. Les configurations correspondent aux différents états activés et aux valeurs des objets UML.

Afin de spécifier et vérifier des requis de sécurité, l'approche USVF fournit également un langage de spécification de propriété sur des systèmes orienté objet et une sémantique opérationnelle permettant l'expression d'un ensemble de propriétés temporelles (sureté et vivacité) impliquant les attributs des objets, les attributs des classes, et les déclencheurs de

transitions. Il s'agit d'une extension de la syntaxe des diagrammes UML avec des opérateurs qui permettent d'exprimer des propriétés de logique temporelle LTL (Banieqbal *et al.* (1987)) pour des objets UML et des diagrammes d'états.

Cette approche permet de vérifier des mécanismes de contrôle d'accès (intégrité et confidentialité) sur les méthodes et les attributs des objets. De nouveaux prédicats *send*, *recv*, *msg*, *write* et *trans* spécifiques aux machines à états UML du modèle sont définis pour pouvoir exprimer les propriétés de sécurité. La différence majeure entre l'approche USVF et notre approche réside dans la modélisation et la vérification des requis de sécurité. La méthode USVF modélise les requis de sécurité comme des propriété de sureté LTL. Ces propriétés sont étendues avec des prédicats capable d'exprimer les paradigmes orientés objet. Dans notre approche, les requis de sécurité sont directement modélisés par les étiquettes de sécurité qui sont rattachées à chaque attribut, variable et paramètres. La syntaxe de ces étiquettes est directement intégrée à la syntaxe des modèles UML. De plus, notre sémantique des diagrammes d'états UML permet de calculer les valeurs des étiquettes de sécurité à chaque transition. Ceci permet d'évaluer la sécurité à chaque pas d'exécution des diagrammes d'états.

2.3.2 Vérification des failles de sécurité dans la conception des logiciels

Cette approche utilise la technique du *model-checking* pour faire de la vérification de modèles UML et est le résultat des travaux de Jinhua et Jing (2010). L'outil utilisé pour faire la vérification est SPIN.

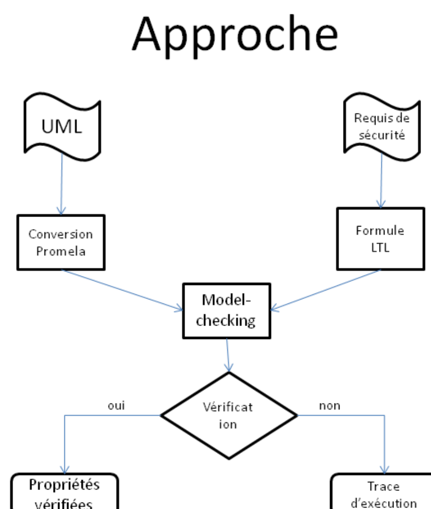

Figure 2.3 Méthode de vérification des diagrammes UML avec SPIN

L'approche pour la vérification de la sécurité dans les modèles UML est illustrée à la figure 2.3. On utilise des diagrammes d'états UML pour décrire les modèles UML. La syntaxe et la sémantique formelle des modèles UML est décrite et est convertie en langage PROMELA. Les propriétés de sécurité sont spécifiées par des formules de logique temporelle LTL. Les modèles formels et les propriétés de sécurité sont vérifiés par *model-checking*.

La syntaxe formelle décrit des diagrammes d'états UML, la syntaxe est décrite à partir d'automates hiérarchiques (Latella *et al.* (1999a)). L'algorithme de conversion de modèles UML en automates hiérarchiques est décrit par (Latella *et al.* (1999b)). La sémantique opérationnelle des diagrammes d'états est décrite dans deux cas. Premièrement, on évalue la sémantique sur un seul diagramme d'états grâce à un système de transition étiquetté. Par la suite, la sémantique est évaluée sur un ensemble de diagrammes d'états. Le système de transition étiquetté décrit l'ensemble des états activés (nouvelle configuration du système), l'ensemble des transitions et la relation de transition qui détermine les nouvelles configurations du système.

Dans cette approche, l'analyse de la sécurité s'effectue de la façon suivante. Les modèles UML sont convertis en langage PROMELA qui sont analysés et vérifiés par SPIN. Le processus de verification de sécurité nécessite la définition d'une politique de sécurité qui décrit les vulnérabilités logicielles en termes de propriétés de sécurité. On exprime chaque propriété de sécurité grâce à une machine à états où les noeuds définissent l'état du système

et les transitions correspondent aux actions du système. Les machines à états sont ensuite transformées automatiquement en modèles PROMELA analysable dans SPIN. Les requis de sécurité (contrôle d'accès) appliqués sur les modèles UML sont spécifiés directement avec des formules de logique temporelle linéaire LTL (Banieqbal *et al.* (1987)).

Notre approche se démarque de cette méthodologie par l'ajout d'étiquettes de sécurité aux diagrammes UML. La syntaxe et la sémantique des étiquettes est directement intégrée à celle des diagrammes UML. De plus notre sémantique des diagrammes d'états UML permet de calculer les valeurs des étiquettes de sécurité à chaque transition. Ceci permet d'évaluer la sécurité à chaque pas d'exécution des diagrammes d'états et d'effectuer une analyse de flux d'information basée sur la valeur des étiquettes de sécurité. Pour se faire, on considère les étiquettes des variables, attributs et paramètres qui sont présente dans les transitions des diagrammes d'états UML.

2.3.3 Modélisation et vérification des politiques de sécurité via des diagrammes UML

(Cheng et Zhang (2011)) propose une méthode qui modélise des politiques de sécurité (modèle de sécurité) via des diagrammes UML(diagramme de classe, diagramme d'états). Le contrôle d'accès est le mécanisme de sécurité qui est modélisé dans cette approche. Les états de sécurité sont décrits à partir des diagrammes UML et les propriétés de sécurité sont exprimées par une formule LTL (Banieqbal *et al.* (1987)) et vérifiées par *model-checking*. Les automates à états finis sont utilisés pour décrire la sémantique des modèles UML.

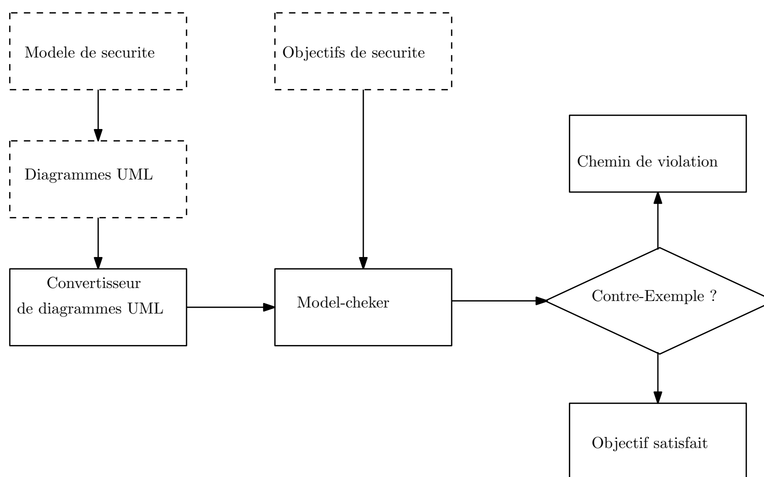

Figure 2.4 Méthodologie

La méthodologie utilisée est représentée à la figure 2.4 et comporte les étapes suivantes :

– le modèle de sécurité est modélisé à partir des diagrammes de classes et d'états

- les objectifs de sécurité sont exprimés à partir de formules de logique temporelle linéaire LTL.
- les modèles UML sont convertis en un modèle formel, analysable par le vérificateur de modèle.
- les objectifs de sécurité sont vérifiés par rapport au modèle formel précédemment obtenu.
- un contre-exemple est retourné si l'objectif de sécurité n'est pas vérifié

La différence majeure entre cette approche et la notre réside dans la modélisation et la vérification des requis de sécurité. Bien que cette méthode modélise le contrôle d'accès comme des propriété de sureté LTL, elle ne modélise pas le flux d'information. En effet, des niveaux de sécurité ne sont pas directement rattachés aux variables utilisées lors des affectations ou de l'exécution des méthodes et aucun treillis de sécurité n'a été formellement défini. Dans notre approche, les requis de sécurité sont directement modélisés par les étiquettes de sécurité qui sont rattachées à chaque attribut, variable et paramètres. La syntaxe de ces étiquettes est directement intégrée à la syntaxe des modèles UML. De plus notre sémantique des diagrammes d'états UML permet de calculer les valeurs des étiquettes de sécurité à chaque transition. Ceci permet d'évaluer la sécurité à chaque pas d'exécution des diagrammes d'états.

2.3.4 Extension OCL pour la Modélisation et spécification UML

Bergeron (2004) propose une technique de validation de la sécurité dans des modèles UML. Les contraintes de sécurité sont exprimées grâce à la notation OCL et sont vérifiées par la technique de *model-checking* à l'aide de l'outil SOCLe. Ce mémoire propose une sémantique statique (décrite à partir des diagrammes de classes et d'objets) et une sémantique dynamique (décrite à travers les diagrammes d'états et les règles de transitions des ASM) des modèles UML. La sémantique statique permet de décrire une configuration du système. Cette configuration correspond à un ensemble d'objets actifs auxquels on associe un ensemble d'instances de méthodes actives. La sémantique dynamique et les règles de transition des ASM permettent de calculer les nouvelles configurations et de déterminer les méthodes en cours d'exécution, et les transitions prioritaires dans les diagrammes d'états. La syntaxe des expressions OCL est étendue avec des opérateurs capables d'exprimer des paradigmes orientés objet (héritage, appels de méthodes sur des objets) et leur sémantique permet d'évaluer des invariants, des postconditions et des préconditions sur les gardes des transitions (diagrammes d'états). Les expressions OCL sont utilisées pour modéliser le contrôle d'accès entre des objets UML. Le contrôle d'accès est modélisé et vérifié au niveau des gardes dans les transitions des diagrammes UML. Pour évaluer la sécurité, on évalue directement les expressions OCL utilisées pour modéliser les gardes. Si une expression est évaluée à vrai l'accès est accordé, sinon

il ne l'est pas.

La différence majeure entre cette approche et la notre réside dans la modélisation et la vérification des requis de sécurité. Bien que cette méthode modélise le contrôle d'accès comme des expressions OCL, elle ne modélise pas le flux d'information. En effet, des niveaux de sécurité ne sont pas directement rattachés aux variables utilisées lors des affectations ou de l'exécution des méthodes et aucun treillis de sécurité n'a été formellement défini. De plus, la vérification se limite à l'évaluation des gardes dans les transitions des diagrammes UML. Dans notre approche, les requis de sécurité sont directement modélisés par les étiquettes de sécurité qui sont rattachées à chaque attribut, variable et paramètres. La syntaxe de ces étiquettes est directement intégrée à la syntaxe des modèles UML. De plus notre sémantique des diagrammes d'états UML permet de calculer les valeurs des étiquettes de sécurité à chaque transition. Ceci permet d'évaluer la sécurité à chaque pas d'exécution des diagrammes d'états.

Dans la prochaine section, on présente les méthodes d'analyse de flux d'information basées sur les types. Ces méthodes se basent sur l'analyse syntaxique et sémantique de programmes annotés avec des niveaux de sécurité.

2.4 Méthodes d'analyse de flux d'information basées sur les types

Cette partie présente les techniques de vérification qui font de l'analyse syntaxique et sémantique de programmes annotés avec des niveaux de sécurité (généralement *haut* ou *bas*).

2.4.1 Analyse du flux d'information par des langages typé sécurité

Une autre méthode de vérification de la sécurité est l'analyse du flux d'information basée sur les langages typés sécurité. Cette technique est le résultat des travaux de Sabelfeld et Myers (2003) et permet d'évaluer le niveau de sécurité de variables présentes dans programme. Plus précisément, il s'agit de l'analyse syntaxique et sémantique de programmes, afin de spécifier et d'appliquer des politiques de sécurité sur des variables. Les variables des programmes sont annotées avec des types de sécurité qui correspondent aux niveaux de sécurité. Le point de départ de l'analyse de flux d'information pour les programmes est la classification des variables du programme en différents niveaux de sécurité. La distinction la plus fondamentale est de classer les données comme *low*, ce qui signifie faible niveau de sécurité, (information publique), et d'autres variables *high*, ce qui signifie de haute sécurité (information privée). L'objectif est de prévenir des fuites d'information des variables *high* vers les variables *low*, lorsque le programme s'exécute. Nous souhaitons faire en sorte que l'information circule uniquement des niveaux de sécurité les moins élevés vers les niveaux de sécurité les plus élevés. Par exemple on permet des flux de *low* à *low*, de *high* à *high*, de *low* à *high*, mais les flux de

high à *low* sont non autorisés. On distingue deux types de flux : les flux explicites qui surviennent lorsque des variables sont affectées directement à d'autres, et les flux implicites qui surviennent dans des instructions de contrôle ou de condition. Si on a une variable *high* dans une instruction, on ne devrait pas avoir d'affectation de valeurs à une variable *low*, puisque cette affectation dépend de la variable *high* présente dans la condition.

```
l ← h
if h < 0 then
  l ← 1
else
  skip
end if
```

Algorithm 2 Fragment de code, variables et niveau de sécurité

L'algorithme 2 montre un fragment de code et les affectations $l \leftarrow h$ et $l \leftarrow 1$ présentent respectivement des flux explicites et implicites. En effet, $l \leftarrow h$ affecte une variable de haut niveau à une variable de bas niveau et $l \leftarrow 1$ est une affectation à une variable de bas niveau alors que la condition d'exécution $h < 0$ implique une variable de haut niveau.

Notre approche tire ses fondements de l'analyse du flux d'information par des langages typés sécurité. On établit une similitude entre l'exécution des diagrammes d'états UML et l'exécution d'un programme. Dans notre étude, les actions effectuées lorsque les transitions s'exécutent (flux explicite), correspondent aux affectations de variables qui surviennent lors de l'exécution des programmes. Les instructions de contrôle des programmes correspondent aux gardes des transitions (flux implicite). La méthode d'analyse de Sabelfeld et Myers (2003) est générale et ne s'applique pas spécifiquement aux contextes orientés objet. Notre approche permet de combler ce manque par l'utilisation des étiquettes de sécurité dans un contexte orienté objet. La syntaxe et la sémantique des étiquettes sont définies pour des attributs d'objets, des variables et les paramètres des méthodes. Une autre différence majeure entre les deux approches est la définition des niveaux de sécurité. Notre définition des niveaux de sécurité ne se contente pas uniquement de deux niveaux (*high* et *low*), mais s'applique dans un contexte où il n'existe aucune autorité centrale et les entités définissent elles même leurs propres politiques de sécurité.

2.4.2 Modèle des étiquettes décentralisées : DLM

Cette section décrit le modèle d'étiquettes décentralisé (DLM : Decentralized Label Model) Myers et Liskov (1998, 2000). Dans ce modèle, les politiques de sécurité ne sont pas

définies par une entité centrale, mais contrôlée par les différentes entités du système. Le système doit ensuite se comporter de manière à respecter ces politiques de sécurité. On utilise des étiquettes pour annoter les données. Les entités principales de ce modèle sont les autorités (*principals*), dont on veut protéger les informations secrètes (données), et les étiquettes (*labels*), qui représentent la manière dont les autorités expriment leurs contraintes.

Autorités Les autorités représentent les entités qui possèdent, modifient et publient les informations. Ils représentent les utilisateurs, groupes ou rôles. Un processus a le droit d'agir au nom d'un ensemble d'autorités. Quelques autorités ont le droit d'agir pour d'autres. Quand une autorité A peut agir pour une autre autorité B. A possède tous les pouvoir et privilèges de B. La relation *agit pour* est réflexive et transitive, et permet de définir une hiérarchie (ordre partiel) entre les autorités. La relation A *agit pour* B est notée A $\succeq$ B. Elle permet la délégation de tous les pouvoirs d'une autorité B à une autre A. La gestion de la hiérarchie des autorités peut également être faite de manière décentralisée. La relation A $\succeq$ B peut être ajoutée à la hiérarchie tant que le processus qui se charge de l'ajout a suffisamment de privilèges pour agir pour l'autorité B. Aucun privilège n'est nécessaire concernant A car cette relation donne plus de pouvoirs à A.

Étiquettes Les autorités expriment leurs contraintes de sécurité en utilisant des étiquettes pour annoter les programmes et les données. Plus précisément, on annote les variables présente dans les programmes. Une étiquette est composée de deux parties : une étiquette de confidentialité et une étiquette d'intégrité. Chacune de ces étiquettes contient un ensemble de politiques de sécurité imposées par des autorités. Une politique est constituée de deux parties : un propriétaire et un ensemble d'autorités, qui représentent des lecteurs dans une étiquette de confidentialité et des modificateurs dans une étiquette d'intégrité. L'objectif d'une politique d'une étiquette est de protéger les données privées du propriétaire de cette politique. Il définit la politique d'utilisation de la donnée. Les utilisateurs font ici référence aux variables du programme.

Étiquettes de confidentialité Une étiquette de confidentialité exprime le niveau de secret désiré par le propriétaire d'une donnée en citant la liste des lecteurs potentiels. Les lecteurs sont les autorités auxquelles cet élément permet de lire la donnée. Ainsi, le propriétaire est la source de la donnée et les lecteurs sont ses destinataires possibles. Les autorités qui ne sont pas listées comme lecteurs n'ont pas le droit de lire les données. Un exemple d'étiquette est :

$$LC = \{o1 \leftarrow r1, r2; o2 \leftarrow r2, r3\}$$

Ici, o1, o2, r1 et r2 sont des autorités. Le point-virgule sépare deux contraintes (éléments) de l'étiquette LC. Les propriétaires de ces politiques sont o1 et o2 et les lecteurs des politiques sont respectivement $\{r1, r2\}$ et $\{r2, r3\}$. La signification d'une étiquette est que chaque politique dans l'étiquette doit être respectée pendant que la donnée circule dans le système, de manière à ce que chaque information étiquetée ne soit divulguée que suite à un consensus entre TOUTES les politiques. Un utilisateur peut lire les données seulement si l'autorité représentant cet utilisateur peut agir pour un lecteur de chaque politique dans l'étiquette. Ainsi, seuls les utilisateurs dont les autorités peuvent agir pour r2 peuvent lire les données étiquetées par L. La même autorité peut être le propriétaire de plusieurs politiques ; les politiques restent renforcées indépendamment l'une de l'autre dans ce cas.

Étiquettes d'intégrité Les politiques d'intégrité sont duales aux politiques de confidentialité. Comme les politiques de confidentialité protègent contre les données lues de manière impropre, les politiques d'intégrité protègent les données contre une modification inappropriée. Une étiquette d'intégrité garde la trace de toutes les sources qui ont affecté sa valeur, même si ces sources l'affectent indirectement. Une politique d'intégrité a un propriétaire, l'autorité pour laquelle la politique est appliquée, et un ensemble d'écrivains, des autorités qui sont autorisées à modifier la donnée. Une étiquette d'intégrité peut contenir un ensemble de politiques d'intégrité avec différents propriétaires. Un exemple d'étiquette est :

$$LI = \{o1 \to w1, w2; o2 \to w2, w3\}$$

avec o1 et o2 les propriétaires de la politique, et w1, w2 et w3 les modificateurs. Une politique d'intégrité est une garantie de qualité. Une politique $\{o1 \to w1, w2\}$ est une garantie fournie par l'autorité o1 que seuls w1 et w2 sont capables de modifier la valeur de la donnée. L'étiquette d'intégrité la plus restrictive est celle qui ne contient aucune politique : $\{\}$. C'est l'étiquette qui ne fournit aucune garantie sur le contenu de la valeur, et peut être utilisée comme donnée d'entrée uniquement quand le destinataire n'impose aucune exigence d'intégrité. Quand une étiquette contient plusieurs politiques d'intégrité, ces politiques sont des garanties indépendantes de qualité de la part des propriétaires de ces politiques. En utilisant une étiquette d'intégrité, une variable peut être protégée contre des modifications impropres. Par exemple, supposons qu'une variable a une politique unique $\{o1 \to w1, w2\}$. Une valeur étiquetée $\{o1 \to w1\}$ peut être introduite dans cette variable car cette valeur a été affectée uniquement par l'autorité w1 et l'étiquette de la variable autorise w1 à la modifier. Si la valeur était étiquetée $\{o1 \to w1, w3\}$, l'écriture ne serait pas autorisée, car la valeur a été affectée par w3, une autorité qui n'est pas mentionnée comme étant un modificateur légal dans l'étiquette de la variable. Cela serait uniquement permis si w3 = w2. Enfin, considérons

une valeur étiquetée $\{o1 \rightarrow w1; o2 \rightarrow w3\}$. Dans ce cas, l'écriture est permise, car la première politique atteste que o croit que seul w1 a altéré la valeur. Le fait que la seconde politique existe de la part de o2 n'affecte pas la légalité de l'écriture dans la variable.

Le modèle des étiquettes décentralisé défini par Myers et Liskov (1998, 2000) est général et s'applique sur des programmes. On étend ce modèle à des contextes orientés objet. Dans nos modèles étendus, les entités correspondent aux objets qui sont définis dans les diagrammes d'objets UML. De plus la syntaxe et la sémantique des étiquettes sont intégrés à celle des diagrammes UML(états et objets).

2.5 Synthèse comparatrice des approches

Différentes méthodes et approches de modélisation de la sécurité ont été décrites dans les sections précédentes (Section 2.1, Section 2.2, Section 2.3 et Section 2.4). Les critères utilisés pour comparer les méthodes de modélisation et de vérification de la securité dans des diagrammes UML sont les suivants. Les mécanismes et/ou les propriétés de sécurité modélisés et vérifiés, les fragments UML modélisés, la syntaxe et la sémantique exprimées. Il est à noter que notre méthode est la première à faire mention de la spécification et de la vérification formelle du flux d'information au niveau de l'exécution des diagrammes d'états UML (Calcul des étiquettes de sécurité des gardes et des données). Notre approche modélise des politiques de sécurité (via l'utilisation d'étiquettes décentralisé) et intègre ce modèle formel à celui des diagrammes UML. Le flux d'information explicite est modélisé et vérifié à travers les actions qui s'effectuent lors les transitions. Les flux explicites peuvent survenir lorsqu'on réalise des opérations d'affectation de variables (ou d'attributs) ou des passages de paramètres aux méthodes. Les flux implicites quant à eux, sont pris en compte lorsque l'exécution d'une action est conditionnée par une garde. On vérifie les étiquettes de sécurité des variables présentes dans la garde.

Tableau 2.1 Tableau de synthèse comparatrice - Extension UML

	Syntaxe et Sémantique	**Mécanisme d'extension** UML	**Spécification de la sécurité**
UMLsec	- Diagrammes de classes UML - Diagrammes d'états UML	- Annotations (stéréotypes,balises) sur les diagrammes UML	- Flux d'information à partir d'annotation sur les variables et les méthodes - Définition de niveaux de sécurité (secret et publique)
SecureUML	Diagrammes de classes UML	- Définition d'environnement de sécurité par des méta-modèles UML	RBAC exprimé par des contraintes OCL
Notre approche	- Diagrammes de classes UML - Diagrammes d'états UML	- Annotations sur les diagrammes UML grâce à des étiquettes de sécurité	- Flux d'information à partir d'annotation sur les variables, les attributs des objets les paramètres des méthodes - Définition de niveaux de sécurité (DLM) - Évaluation des niveaux de sécurité sur les transitions UML

Le tableau 2.1 présente les approches qui font des extensions du langage UML (UML-

sec,SecureUML). Le point fort de ces approches est leur fondement dans une norme standard, largement utilisée pour la modélisation orientée objet. La syntaxe des diagrammes UML est enrichie à l'aide de méta-modèles capable d'exprimer des contraintes. Toutefois, SecureUML ne fournit pas de façon rigoureuse, un modèle formel qui serait capable d'exprimer le flux d'information (propagation de l'information). Dans le cas de SecureUML, l'approche met l'accent sur la spécification formelle du contrôle d'accès basé sur l'attribution des roles (RBAC). La syntaxe et la sémantique des diagrammes d'états n'est pas définie, puisque l'analyse des modèles UML se limite aux modèles de structure (diagrammes de classes, diagrammes d'objets). Dans le cas de UMLsec, un nouveau stéréotype (*no-down flow*) et un ensemble de balises sont définis afin de spécifier des contraintes de flux d'information sur des éléments des diagrammes UML. La sémantique des diagrammes UML ne décrit pas les niveaux de sécurité dans les diagrammes d'états à chaque étape, ainsi que le sens de propagation du flux d'information. Le modèle UMLsec se contente de vérifier que les variables sont lues ou modifiées par des méthodes appropriées (une variable avec un niveau de sécurité *privé* est lue ou écrite par une méthode avec un niveau de sécurité *privé*, et une variable avec un niveau de sécurité *publique* est lue ou écrite par une méthode avec un niveau de sécurité *publique*) La différence majeure entre notre approche et celle de UMLsec est l'utilisation des étiquettes de sécurité décentralisées. Dans notre modèle, les politiques de sécurité ne sont pas définies par une entité centrale, mais contrôlées par les différents objets UML définis dans le diagramme d'objets UML. Lorsque les diagrammes d'états s'exécutent, le système doit ensuite se comporter de manière à respecter les politiques de sécurité imposées par chaque objet.

Tableau 2.2 Tableau de synthèse comparatrice - Formalisation diagrammes d'états UML

	Éléments de syntaxe	**Éléments de Sémantique**
Section 2.2.1	- Description des états UML action qui s'exécute en entrée d'un état action qui s'exécute en sortie d'un état activité qui s'exécute dans un état - Description des transitions UML action qui s'exécute lors d'une transition garde sur la transition activité qui s'exécute dans un état ensemble des états actifs	- Algorithme rtc 1, évaluation priorité des transitions transitions conflictuelles actions qui s'exécutent ensemble des états actifs
Section 2.2.2	- Description des états UML action qui s'exécute à l'entrée d'un état action qui s'exécute à la sortie d'un état ensemble des états actifs description des états simple, composites, historique - Description des transitions UML action qui s'exécute lors d'une transition garde sur la transition ensemble des états actifs	- association d'un diagramme d'état UML à une structure de Kripke - évaluation des états actifs - évaluation des séquences d'actions à exécuter
Notre approche	- Description des états UML action qui s'exécute en entrée d'un état action qui s'exécute en sortie d'un état action qui s'exécute en sortie d'un état ensemble des états actifs pour un objet description des locations (ensemble des états actifs pour plusieurs objets) - Description des transitions UML action qui s'exécute lors d'une transition garde sur la transition déclencheur d'une transition description des niveaux de sécurité des transitions	- association des diagrammes d'états UML à une structure de Kripke - évaluation des états actifs - évaluation des séquences d'actions à exécuter - évaluation des niveaux de sécurité des transitions - exéution simultanée de plusieurs diagrammes d'états UML - définition d'un ensemble de locations UML

Le tableau 2.2 présente les formalismes utilisés pour modéliser les diagrammes UML et les contraintes de sécurité à respecter. La contrainte de sécurité qui est vérifiée est le contrôle d'accès et est exprimée par des formules de logique temporelle linéaire (LTL).
Dans les sous-sections 2.2.1 et 2.2.2 la syntaxe et la sémantique des diagrammes UML est exprimée par des machines à états (automates) finis (Latella *et al.* (1999a)). Contrairement à notre approche les niveaux et les éléments de sécurité ne sont pas décrits par la syntaxe et la sémantique formelle. Notre approche intègre les syntaxes et sémantiques formelles des diagrammes UMLet celles des éléments de sécurité. De plus, notre approche permet l'exécution simultanée de plusieurs diagrammes d'états grâce à la formalisation des locations (produit cartésien sur les ensembles d'états actifs). Notre modèle définit un diagramme d'objets et à chaque objet on associe un diagramme d'états. Nous considérons l'exécution simultanée de plusieurs diagrammes d'états. Ceci permet d'obtenir d'avoir l'ensemble des configurations possibles pour tous les objets du système. Par exemple, l'exécution d'une transition d'un diagramme d'états influencera également les états actifs pour les autres diagrammes d'états du système. L'ensemble des locations correspond à l'ensemble des états UML accessibles dans tous les diagrammes d'états lorsqu'une transition s'exécute. Cette transition peut appartenir à n'importe quel diagramme d'états.

Tableau 2.3 Tableau de synthèse comparatrice - Formalisation UML et sécurité

	Éléments de syntaxe et sémantique	**Spécification de la sécurité**
Section 2.3.1	- Diagrammes de classes UML - Diagrammes d'états UML - Description des transitions UML action qui s'exécute lors d'une transition garde sur la transition - Sémantique statique évaluation des valeurs des attributs des objets évaluation des valeurs des paramètres de méthodes - Sémantique opérationelle évaluation de l'ensemble des états actifs	- Extension de LTL avec de nouveaux opérateurs appel de méthode modification des atributs des objets exécution des transitions déclenchement des messages en objets formalisation du contrôle d'accès, pas de treillis de sécurité
Section 2.3.2	- Diagrammes d'états UML - Description des transitions UML - Syntaxe des diagrammes d'états UML automates hiérarchiques - Sémantique des diagrammes d'états UML exprimée par une structure de Kripke évaluation des ensembles des états UML actifs évaluation des ensembles des déclencheurs UML actifs évaluation des valeurs des paramètres de méthodes	- Expressions LTL - formalisation du contrôle d'accès, pas de treillis de sécurité
Section 2.3.3	- Diagrammes d'états UML - Diagrammes d'objets UML - Transformation des états UMLen états de sécurité - Syntaxe des diagrammes d'états UML automates à états finis	- Expressions LTL - formalisation du contrôle d'accès, pas de treillis de sécurité
Section 2.3.4	- Diagrammes d'états UML - Diagrammes de classes UML - Diagrammes d'objets UML - Syntaxe des diagrammes d'états UML exprimée par des ASM - Sémantique des diagrammes d'états UML exprimée par des ASM évaluation des valeurs des atributs des objets évaluation des valeurs des paramètres des méthodes évaluation des ensembles des états UML actifs	- Expressions OCL - Formalisation du contrôle d'accès, pas de treillis de sécurité
Notre approche	- Description des diagrammes d'états UML - Description des diagrammes de classes UML - Description des diagrammes d'objets UML - Sémantique des diagrammes d'états UML évaluation des valeurs des étiquettes de sécurité des objets évaluation des valeurs des étiquettes de sécurité des transitions évaluation des valeurs des étiquettes de sécurité ds paramètres	- Utilisation des étiquettes décentralisées annotations des attributs des objets annotations des paramètres des méthodes annotations des variables utilisées dans les transitions

Le tableau 2.3 présente les formalismes utilisés pour modéliser les diagrammes UML et les contraintes de sécurité à respecter. La contrainte de sécurité qui est vérifiée est le contrôle d'accès et est exprimée par des formules de logique temporelle linéaire (LTL).

Dans les sous-sections 2.3.2, 2.2.2, 2.3.3, et 2.3.1 la syntaxe et la sémantique des diagrammes UML est exprimée par des machines à états (automates) finis (Latella *et al.* (1999a)). Les sous-sections 2.3.2, 2.2.2, et 2.3.3 utilisent des automates hiérarchiques pour formaliser les diagrammes d'états. La sémantique exprimée ne permet pas de décrire les contraintes de sécurité à chaque évolution du système. Les contraintes de sécurité sont plutôt exprimées à partir des formules de logique temporelle linéaire (LTL). Néanmoins, dans la sous-section 2.3.3, la sémantique permet de faire une correspondance entre les états du système de transition et les *états de sécurité* (les ensembles de droits d'accès, les ensembles de sujets et les ensembles d'opérations possibles) c'est à dire que chaque évolution du système de transition permet d'accéder à de nouveaux états possédant chacun leurs contraintes de sécurité. La section 2.3.1 décrit les états comme un ensemble de classes et un ensemble d'objets initialisés. Les contraintes de sécurité sont exprimées à partir de formules de logique temporelle.

La différence entre l'approche décrite à la section 2.3.3 et notre approche réside dans la

modélisation et la vérification des requis de sécurité. Bien que cette méthode modélise le contrôle d'accès comme des expressions OCL, elle ne modélise pas le flux d'information. En effet, des niveaux de sécurité ne sont pas directement rattachés aux variables utilisées lors des affectations ou de l'exécution des méthodes et aucun treillis de sécurité n'a été formellement défini. De plus, la vérification se limite à l'évaluation des gardes dans les transitions des diagrammes UML. Dans notre approche, les requis de sécurité sont directement modélisés par les étiquettes de sécurité qui sont rattachées à chaque attribut, variable et paramètres. La syntaxe de ces étiquettes est directement intégrée à la syntaxe des modèles UML. De plus notre sémantique des diagrammes d'états UML permet de calculer les valeurs des étiquettes de sécurité à chaque transition. Ceci permet d'évaluer la sécurité à chaque pas d'exécution des diagrammes d'états.

La dernière section 2.4 présente les approches utilisées par les langages typés sécurité. Elle permet de déduire le niveau de sécurité des expressions définies dans un programme à la compilation (analyse statique). Aucune de ces approches ne fait mention de la spécification et de la vérification du flux d'information dans des diagrammes d'états. Dans le cadre de notre étude, on considère l'analyse statique de tous les diagrammes d'états qui constituent le modèles. La déduction des niveaux de sécurité, des flux implicites et explicites mentionnés par 2.4.1 s'effectue en tenant compte des actions et des gardes dans les transitions. Notre étude est la première qui se consacre à la formalisation du flux d'information sur des diagrammes d'états en utilisant des étiquettes de sécurité décentralisées.

CHAPITRE 3

DÉFINITION D'UN MODÈLE DE SÉCURITÉ POUR LES SYSTÈMES ORIENTÉS OBJET

Le chapitre 3 présente la description formelle du modèle des étiquettes de sécurité décentralisées (Myers et Liskov (1998)) dans un contexte orienté objet. Ce modèle repose sur une notion de sécurité multi-niveau et étend les modèles traditionnels de flux d'information avec une politique de flux fixée par chaque utilisateur. En effet, dans ce modèle les politiques de sécurité ne peuvent pas être décidées par une autorité centrale. Au lieu de cela, chaque participant dans le système est capable de définir et de contrôler ses propres politiques de sécurité à l'aide d'étiquettes de sécurité. Dans un contexte orienté objet, les étiquettes sont appliquées sur les variables et les attributs lorsqu'on réalise des opérations d'affectation impliquant des variables et des attributs. Les étiquettes de sécurité sont également appliquées sur les paramètres des méthodes, pour vérifier la sécurité lorsque des paramètres effectifs sont passés à la méthode. De plus les objets du système orienté objet constituent l'ensemble des utilisateurs (participants). L'ensemble des attributs, des variables et des paramètres sera assimilé à l'ensemble des données.

On veut s'assurer qu'il n'y a pas de failles de sécurité lorsqu'on exécute les diagrammes d'états des objets (diagramme d'objets). Les failles de sécurité surviennent dans deux cas. Le premier cas est celui où on affecte des données aux attributs des objets. Le deuxième cas est celui où on passe des données comme paramètres spécifiques à des méthodes des objets. La première partie (Section 3.1) donne une définition du flux d'information telle que définie par Sabelfeld et Myers (2003). Cette définition est utilisée pour la description du flux d'information dans un contexte orienté objet. La seconde partie (Section 3.2) décrit le concept d'*hiérarchie d'utilisateurs* utilisé par le modèle des étiquettes décentralisées. Ensuite, la troisième partie (Section 3.3) décrit la syntaxe et la sémantique des étiquettes de sécurité. Enfin, la dernière partie (Section 3.4) décrit la relation de restriction entre les étiquettes de sécurité.

3.1 Flux d'information dans les systèmes orientés objet

Le résultat des études sur la spécification du flux d'information dans des programmes (réalisées par Sabelfeld et Myers (2003)) ont permis d'établir des règles pour l'évaluation du flux d'information dans des programmes. Le modèle des étiquettes décentralisées est une extension

des travaux de Sabelfeld et Myers (2003). Ce modèle permet de controler le flux d'information dans des programmes à l'aide d'étiquettes décentralisées. Les variables présentes dans les programmes sont annotées avec ces dernières. L'étiquette d'une variable contrôle la manière dont les données stockées dans cette variable peuvent-être diffusées.

Les flux d'information surviennent lorsque les instructions des programmes s'exécutent. Il s'agit des opérations d'affectation impliquant des variables. Si le contenu d'une variable est affectée au contenu d'une autre variable, il y a des flux d'information entre ces dernières. L'affectation d'une variable à une autre variable n'est autorisée que si les flux d'information sont légitimes et sécuritaires. On considère premièrement l'opération d'affectation suivante :

$$x := v$$

La variable x lit l'information à partir de la variable v (ou la variable v modifie l'information contenue dans la variable x) et il y a flux d'information de v vers x. La légitimité du flux est vérifiée en comparant les étiquettes d'intégrité et de confidentialité de la variable affectée x à celles de la variable à affecter v. Il s'agit ici du flux d'information explicite. Ensuite on considère le fragment de code suivant :

$$\begin{array}{c} if\ (v < 0)\ then \quad (1) \\ x = 1 \quad (2) \end{array}$$

Il faut vérifier que toutes les affectations qui surviennent lorsque la condition (1) est vrai n'implique pas de variables ayant des niveaux de sécurité inférieurs à ceux de la variable v. Il s'agit du flux implicite. Le point de départ de l'analyse de flux d'information est la classification des étiquettes de sécurité en différents niveaux de sécurité. La distinction la plus fondamentale est de classer certaines données dans la catégorie de faible niveau de sécurité, (information publique), et d'autres données dans la catégorie de haute sécurité (informations privées). De façon spécifique, on utilise un treillis de niveaux de sécurité (Label,$\sqsubseteq$), où Label est un ensemble d'étiquettes de sécurité (section 3.3) et $\sqsubseteq$ est une relation d'ordre partielle entre les étiquettes de sécurité qui définit le flux légal d'information (section 3.4). Nous souhaitons faire en sorte que l'information circule uniquement des niveaux de sécurité les moins élevés vers les niveaux de sécurité les plus élevés. Si $L_1, L_2 \in \mathsf{Label}$ et $L_2 \sqsubseteq L_1$, on permet des flux de L_1 à L_1, de L_2 à L_2, de L_1 à L_2, mais les flux de L_2 à L_1 sont non autorisés.

Dans notre contexte orienté objet, les flux d'information surviennent lorsque les transitions des diagrammes d'états s'exécutent. Plus précisément, une transition (constituée d'un déclencheur, d'une garde et d'un ensemble d'actions) exécute des actions et est conditionnée par sa

garde. Les actions qui s'exécutent sont, soit des opérations d'affectation, soit des appels ou des retours de méthodes. Les flux explicites peuvent survenir dans deux cas. Premièrement, lorsqu'une opération d'affectation s'exécute, il faut vérifier les niveaux de sécurité des variables impliquées dans cette opération. Deuxièment, lorsqu'un appel (ou retour) de méthode survient il faut vérifier les niveaux de sécurité des paramètres formels qui sont passés (ou retournés) à la méthode.

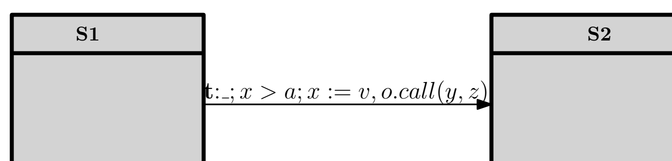

Figure 3.1 Flux d'information dans un diagramme d'états

À titre d'illustration (figure 3.1), le diagramme d'états possède une transition qui affecte une variable v à une autre variable x, et fait un appel à la méthode call de l'objet o. On spécifie le flux d'information en rattachant des étiquettes de sécurité à chacune des données. Cette transition crée des flux d'information à travers les actions $x := v$ et $o.call(y, z)$. En suivant, les règles de flux d'information établies par Sabelfeld et Myers (2003), on doit s'assurer que l'opération d'affectation de la variable v à la variable x ne cause pas un flux d'information illégal. On vérifie que si le niveau de sécurité de v est supérieur à celui de x. De plus, les variables impliquées dans l'opération $x := v$ ne devraient pas avoir un niveau de sécurité inférieur à celui des variables présentes dans la garde $x > a$. L'appel à la méthode *call* implique la vérification des niveaux de sécurité des variables y et z qui sont passé en paramètre à la méthode (on suppose que les paramètres formels de la méthode sont aussi annotées avec des étiquettes de sécurité).

Le modèle des étiquettes décentralisées découle des spécifications de Sabelfeld et Myers (2003) et s'applique dans un contexte où il n'existe aucune autorité. On utilise ce modèle dans un contexte où nos objets (définis dans le diagramme d'objets) constituent les entités du système. Les niveaux de sécurité ne se limitent pas à une simple classification (publique ou privée), mais sont déterminés par chaque objet présent dans le système. De plus les niveaux de sécurité sont évalués par rapport à une hiérarchie d'utilisateurs. Les utilisateurs font référence aux objets du système. La prochaine section 3.2 met l'accent sur définition d'une hiérarchie d'utilisateurs.

3.2 Hiérarchie d'utilisateurs

Dans notre contexte orienté objet, les utilisateurs correspondent à l'ensemble des objets pour lesquels on voudrait spécifier et vérifier le flux d'information. Cet ensemble d'objets est décrit à la figure 3.2.

NOId	:	Ensemble des noms d'objets
OId	:	Ensemble des objets
$\mathcal{H}$	:	ensemble des hiérarchies d'utilisateurs

Figure 3.2 Définition des ensembles d'objets

La figure 3.2 décrit les éléments suivants. L'ensemble NOId correspond à l'ensemble des noms d'objets et OId correspond à l'ensemble des objets. Ces deux ensembles sont décrits au chapitre 4, à la section 4.2. L'ensemble des hiérarchie d'utilisateurs (d'objets) $\mathcal{H}$ est un ensemble d'ensembles de paires d'objets. Les éléments d'un ensemble de paires d'objets doivent vérifier la relation *peut agir pour*. C'est à dire que si un objet a *peut agir pour* b, a possède tous les droits en lecture et écriture de b, en plus de ses propres droits. Soient w_1, w_2 les droits en écriture et r_1, r_2 les droits en lecture des objets $o_1, o_2 \in \mathsf{OId}$. Les notations $w_1 \leq w_2$ signifie que l'objet o_2 a plus de droit en écriture que l'objet o_1. Les notations $r_1 \leq r_2$ signifie que l'objet o_2 a plus de droit en lecture que l'objet o_1. La relation *peut agir pour* peut-être définie par :

$$\begin{aligned} &\mathsf{actFor} \subseteq \mathsf{OId} \times \mathsf{OId} \\ &\mathsf{actFor} \;=\; \{(o_1, o_2) \;:\; o_1, o_2 \in \mathsf{OId},\ \text{et}\ w_1 \leq w_2,\ \text{et}\ r_1 \leq r_2\} \end{aligned}$$

De façon formelle, une hiérarchie d'utilisateurs est une relation binaire h ($\mathsf{h} \in \mathcal{H}$) entre deux objets et est définie par :

$$\mathsf{h} \;=\; \{(a, b) \mid a,\ b \in \mathsf{OId},\ a\ \mathsf{actFor}\ b\}\,,$$

avec OId un ensemble d'objets. $(a,b) \in \mathsf{h}$ si a peut agir pour b et on utilise la notation $a\,\mathsf{h}\,b$ La relation h est une relation d'ordre (réflexive, antisymétrique, transitive). Dans notre modèle, les hiérarchies d'objets sont déterminées à partir des diagrammes d'objets et sont immuables durant l'éxécution des diagrammes d'états.

Exemple

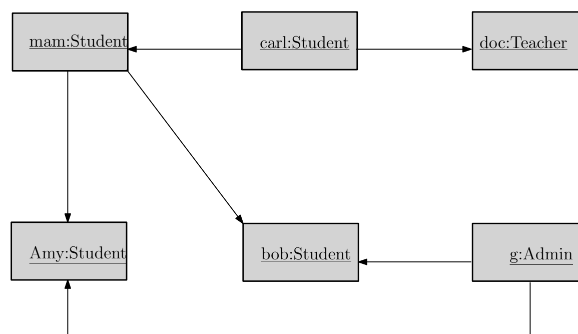

Figure 3.3 Illustration hiérarchie d'utilisateurs

La figure 3.3 illustre un diagramme d'objets (afin de simplifier l'exemple, on n'a pas précisé de diagrammes de classes et la syntaxe formellle des objets n'est pas décrite). L'ensemble de noms d'objets et une hiérarchie d'objets sont donnés respectivement par :

$$\mathsf{OId} = \{doc, carl, mam, bob, g, amy\}$$
$$\mathsf{h} = \{(carl, doc), (carl, mam), (mam, bob), (mam, amy), (g, bob), (g, amy)\}$$

La prochaine section (Section 3.3) décrit la syntaxe formelle des étiquettes de sécurité. Ces étiquettes de sécurité sont utilisées pour spécifier les niveaux de sécurité des données. Les données font référence aux attributs des objets, des variables et des paramètres de méthodes.

3.3 Syntaxe et caractéristiques des étiquettes

Une étiquette de sécurité permet de définir une politique de sécurité sur une donnée. Une étiquette de sécurité est un couple constitué d'une étiquette de confidentialité et d'une étiquette d'intégrité, et elle permet premièrement de définir les objets qui ont le contrôle sur la donnée et deuxièment de définir les usagers qui ont des droits en lecture et en écriture sur la donnée.

Dans notre contexte orienté objet, les données correspondent aux variables, attributs des objets et aux paramètres des méthodes et les objets réalisent des opérations d'écriture et lecture sur ces données via les affectations, les appels et les retours de méthodes.

ILabel	:	Ensemble des étiquettes d'intégrité
CLabel	:	Ensemble des étiquettes de confidentialité
Label	:	Ensemble des étiquettes de sécurité
CComponents	:	Ensemble des composantes d'intégrité
IComponents	:	Ensemble des composantes de confidentialité
IFlux	:	Ensemble des flux d'intégrité
CFlux	:	Ensemble des flux de confidentialité

Figure 3.4 Définition des ensembles d'étiquettes

Pour annoter les données du système avec des étiquettes de sécurité, on utilise les ensembles ILabel, CLabel, Label, CComponents, IComponents, IFlux et CFlux (Figure 3.4). Les contraintes de confidentialité sur une donnée sont exprimées grâce aux éléments de l'ensemble CLabel (ensemble des étiquettes de confidentialité) et les contraintes d'intégrité sont exprimées grâce à l'ensemble ILabel(ensemble des étiquettes d'intégrité).

$$\mathsf{Label} = \mathsf{CLabel} \times \mathsf{ILabel}$$

Dans le cas de la spécification des politiques de confidentialtié, les éléments de l'ensemble CLabel sont des parties de l'ensemble des composantes de confidentialité (CComponents) et permettent de définir des politiques de confidentialité.

$$\mathsf{CLabel} = \{lc : lc \in \mathcal{P}(\mathsf{CComponents})\}$$

L'ensemble des composantes de confidentialité définit des paires *propriétaire-listes d'objets admis en lecture*. Les flux de confidentialité (ensemble CFlux) d'une composante de sécurité sont obtenus à partir d'une fonction d'interprétation qui prend en paramètre une composante de confidentialité et une hiérarchie d'objets (Section 3.2) et associe un objet propriétaire à un autre objet qui a des droits de modifications sur la donnée.

Dans le cas de la spécification des politiques d'intégrité, les éléments de l'ensemble ILabel sont des parties de l'ensemble des composantes d'intégrité (IComponents) et permettent de définir des politiques d'intégrité.

$$\mathsf{ILabel} = \{li : li \in \mathcal{P}(\mathsf{IComponents})\}$$

L'ensemble des composantes d'intégrité définit des paires *propriétaire-listes d'objets admis en écriture*. Les flux d'intégrité (ensemble IFlux) d'une composante de sécurité sont obtenus à partir d'une fonction d'interprétation qui prend en paramètre une composante d'intégrité

et une hiérarchie d'objets (Section 3.2) et associe un objet propriétaire à un autre objet qui a des droits de modifications sur la donnée.

De façon formelle, une étiquette $L_v \in \mathsf{Label}$ est constituée de deux parties :

- une étiquette de sécurité pour assurer la confidentialité de la donnée, notée $L_{C,v} \in \mathsf{CLabel}$
- une étiquette de sécurité pour assurer l'intégrité de la donnée, notée $L_{I,v} \in \mathsf{ILabel}$

De façon formelle, une étiquette sur une donnée v, $L_v \in \mathsf{Label}$ est définie par :

$$L_v \; = \; (\; L_{C,v}, \; L_{I,v} \;)$$

Cette section est organisée comme suit. La première sous-section donne la description formelle d'une étiquette de confidentialité, d'une composante de confidentialité et d'un flux de confidentialité (sous-section 3.3.1) et la seconde partie donne la descritpion formelle d'une étiquette d'intégrité, d'une composante d'intégrité et d'un flux d'intégrité (sous-section 3.3.2).

3.3.1 Définition formelle d'une étiquette de confidentialité

Pour décrire les étiquettes de confidentialité, on définit premièrement l'ensemble des composantes de confidentialité et l'ensemble des flux de confidentialité. Les étiquettes de confidentialité sont exprimées en fonction des éléments de ces deux ensembles. En effet, une composante de confidentialité correspond à une paire *propriétaire-ensemble de lecteurs* sur une donnée et un flux de confidentialité correspond à une paire *propriétaire-lecteur* sur une donnée.

Définition formelle d'un flux de confidentialité Un flux de confidentialité $\mathsf{f}_c \in \mathsf{CFlux}$ correspond à une paire *propriétaire-lecteur* sur une donnée. Il correspond à un objet qui est admis comme lecteur par un autre objet et est défini de façon formelle par :

$$\begin{aligned} &\mathsf{CFlux} \subseteq \mathsf{NOId} \times \mathsf{NOId} \\ &\mathsf{f}_c = (p, q) \end{aligned}$$

où :

- $p \in \mathsf{NOId}$, correspond à un objet propriétaire de la donnée (objet qui a le contrôle sur la variable)
- $q \in \mathsf{NOId}$, correspond à un objet admis comme lecteur par le propriétaire p.

Définition formelle d'une composante de confidentialité Une composante de confidentialité est définie sur une donnée et correspond à une paire *propriétaire-ensemble de lec-*

teurs sur cette donnée. Il correspond à un ensemble d'objets qui sont admis comme lecteurs par un autre objet. L'ensemble des composantes de confidentialité est noté $\mathsf{CComponents}$ et une composante de confidentialtié $\mathsf{C} \in \mathsf{CComponents}$ est définie de façon formelle par :

$$\mathsf{C} = (o, r)$$

où :

- $o \in \mathsf{NOId}$, le nom d'un objet propriétaire de la composante (objet qui a le contrôle sur la composante).
- $r \in \mathcal{P}(\mathsf{NOId})$, $r = \mathsf{readers}(v, o)$ est la liste des noms d'objets qui sont admis comme lecteurs par le propriétaire o, et $v \in \mathsf{Data}$ La fonction $\mathsf{readers}$ permet d'obtenir cette liste.

$$\mathsf{readers} : \mathsf{Data} \times \mathsf{NOId} \longrightarrow \mathcal{P}(\mathsf{NOId})$$

On définit également la fonction d'interprétation $\mathcal{I}_C$ qui associe à une étiquette de confidentialité et une hiérarchie d'utilisateurs, un ensemble de flux de confidentialité :

$$\mathcal{I}_C \; : \; \mathsf{CLabel} \times \mathcal{H} \; \longrightarrow \; \mathcal{P}(\mathsf{CFlux})$$

Définition formelle d'une étiquette de confidentialité Une étiquette de confidentialité correspond à un ensemble de composantes de confidentialité. Une étiquette de confidentialité sur une valeur v, $L_{C,v} \in \mathsf{CLabel}$ est définie de façon formelle par :

$$\begin{aligned} L_{C,v} \; &= \; \{cc \; : \; cc \in \mathcal{P}(\mathsf{CComponents})\} \\ \text{et } L_{C,v} &\subseteq \mathsf{CComponents} \end{aligned}$$

On définit la fonction owners qui associe à une étiquette de sécurité un propriétaire :

$$\mathsf{owners} \; : \; \mathsf{CLabel} \cup \mathsf{ILabel} \; \longrightarrow \; \mathsf{NOId}$$

On définit la fonction $\mathsf{owners}^\star$ qui associe à une composante de sécurité un propriétaire :

$$\mathsf{owners}^\star \; : \; \mathsf{CComponents} \cup \mathsf{IComponents} \; \longrightarrow \; \mathsf{NOId}$$

On définit la fonction $\mathsf{readers}^\star$ qui associe à une composante de sécurité l'ensemble des lecteurs admis par cette composante de confidentialité :

$$\mathsf{readers}^\star \; : \; \mathsf{CComponents} \; \longrightarrow \; \mathcal{P}(\mathsf{NOId})$$

On définit la fonction $\mathsf{readers}^{\star\star}$ qui associe à une composante de sécurité l'ensemble des utilisateurs implicitement admis comme lecteurs pour cette composante :

$$\mathsf{readers}^{\star\star} \ : \ \mathsf{CComponents} \longrightarrow \mathcal{P}(\mathsf{NOId})$$

$\mathsf{owners}^{\star}(I)$ désigne le propriétaire d'une composante I, $\mathsf{readers}^{\star}(I)$ désigne l'ensemble des lecteurs de la composante I. Étant donné une composante de confidentialité I, nous pouvons définir les fonctions $\mathsf{readers}^{\star\star}(I)$ qui donnent l'ensemble des utilisateurs implicitement admis comme lecteurs pour cette politique. Les utilisateurs implicitement admis comme lecteurs sont un ensemble utilisateurs qui satisfont la relation actFor (Section 3.2) dans une hiérarchie d'utilisateurs.

$$\mathsf{readers}^{\star\star}(I) = \{r \mid \exists r' \in \mathsf{readers}^{\star}(I) \ : \ r \,\mathsf{h}\, r', \ \mathsf{h} \in \mathcal{H}\}$$

3.3.2 Définition formelle d'une étiquette d'intégrité

Pour décrire les étiquettes d'intégrité, on définit premièrement l'ensemble des composantes d'intégrité et l'ensemble des flux d'intégrité. Les étiquettes d'intégrité sont exprimées en fonction des éléments de ces deux ensembles. En effet, une composante d'intégrité correspond à une paire *propriétaire-ensemble de modificateurs* sur une donnée et un flux d'intégrité correspond à une paire *propriétaire-modificateur* sur une donnée.

Définition formelle d'un flux d'intégrité Un flux d'intégrité $\mathsf{f}_i \in \mathsf{IFlux}$ correspond à une paire *propriétaire-modificateur* sur une donnée. Il correspond à un objet qui est admis comme modificateur par un autre objet et est défini de façon formelle par :

$$\mathsf{IFlux} \subseteq \mathsf{NOId} \times \mathsf{NOId}$$
$$\mathsf{f}_i = (p, q)$$

où :
- $p \in \mathsf{NOId}$, correspond à un objet propriétaire de la donnée (objet qui a le contrôle sur la variable)
- $q \in \mathsf{NOId}$, correspond à un objet admis comme modificateur par le propriétaire p.

Définition formelle d'une composante d'intégrité Une composante d'intégrité est définie sur une donnée et correspond à une paire *propriétaire-ensemble de modificateurs* sur cette donnée. Il correspond à un ensemble d'objets qui sont admis comme modificateurs par un objet. L'ensemble des composantes d'intégrité est noté $\mathsf{IComponents}$ et une composante

d'intégrité $\mathsf{I} \in \mathsf{IComponents}$ est définie de façon formelle par :

$$\mathsf{I} = (o, w)$$

où :

- $o \in \mathsf{NOId}$, le nom d'un objet propriétaire de la composante (objet qui a le contrôle sur la composante).
- $w \in \mathcal{P}(\mathsf{NOId})$, $w = \mathsf{writers}(v, o)$ est la liste des noms d'objets qui sont admis comme modificateurs par le propriétaire o, et $v \in \mathsf{Data}$. La fonction $\mathsf{writers}$ permet d'obtenir cette liste.

$$\mathsf{writers} : \mathsf{Data} \times \mathsf{NOId} \longrightarrow \mathcal{P}(\mathsf{NOId})$$

On définit également la fonction d'interprétation $\mathcal{I}_I$ qui associe à une étiquette d'intégrité et une hiérarchie d'objets, un ensemble de flux d'intégrité :

$$\mathcal{I}_I \ : \ \mathsf{ILabel} \times \mathcal{H} \ \longrightarrow \ \mathcal{P}(\mathsf{IFlux})$$

Définition formelle d'une étiquette d'intégrité Une étiquette d'intégrité correspond à un ensemble de composantes d'intégrité. Une étiquette d'intégrité sur une valeur v, $L_{I,v} \in \mathsf{ILabel}$ est définie de façon formelle par :

$$L_{I,v} \ = \ \{ii \ : \ ii \in \mathcal{P}(\mathsf{IComponents})\}$$
$$\text{et } L_{I,v} \subseteq \mathsf{IComponents}$$

On définit la fonction owners qui associe à une étiquette de sécurité un propriétaire :

$$\mathsf{owners} \ : \ \mathsf{CLabel} \cup \mathsf{ILabel} \ \longrightarrow \ \mathsf{NOId}$$

On définit la fonction $\mathsf{owners}^\star$ qui associe à une composante de sécurité un propriétaire :

$$\mathsf{owners}^\star \ : \ \mathsf{CComponents} \cup \mathsf{IComponents} \ \longrightarrow \ \mathsf{NOId}$$

On définit la fonction $\mathsf{writers}^\star$ qui associe à une composante de sécurité l'ensemble des lecteurs admis par cette composante de confidentialité :

$$\mathsf{writers}^\star \ : \ \mathsf{IComponents} \ \longrightarrow \ \mathcal{P}(\mathsf{NOId})$$

On définit la fonction $\mathsf{writers}^{\star\star}$ qui associe à une composante de sécurité l'ensemble des

utilisateurs implicitement admis comme lecteurs pour cette composante :

$$\mathsf{writers}^{\star\star} \; : \; \mathsf{IComponents} \longrightarrow \mathcal{P}(\mathsf{NOId})$$

Les notations $\mathsf{owners}^{\star}(I)$ désigne le propriétaire d'une composante I, $\mathsf{writers}^{\star}(I)$ désigne l'ensemble des lecteurs de la composante I. Étant donné une composante d'intégrité I, nous pouvons définir les fonctions $\mathsf{writers}^{\star\star}$ qui donnent l'ensemble des objets implicitement admis comme lecteurs pour cette politique. Les utilisateurs implicitement admis comme modificateurs sont un ensemble utilisateurs qui satisfont la relation actFor (Section 3.2) dans une hiérarchie d'utilisateurs.

$$\mathsf{writers}^{\star\star}(I) \; = \; \{w \,|\, \exists w' \in \mathsf{writers}^{\star}(I) \; : \; w \,\mathsf{h}\, w', \; \mathsf{h} \in \mathcal{H}\}$$

Exemple L'exemple suivant donne la syntaxe formelle d'une étiquette de confidentialité. Considérons un ensemble de données Data, l'ensemble de noms d'objets NOId (qui fait directement références aux objets).

- $\mathsf{Data} = \{x, y\}$
- $\mathsf{NOId} = \{carl, manager, doctor, amy, group, bob, secretary, engineer\}$
- Soit L_x l'étiquette de sécurité sur la variable x

En utilisant la définition d'une composante de l'étiquette de la valeur x, cette dernière peut-être redéfinie comme un ensemble de composantes de sécurité :

$$\begin{aligned} L_{C,x} \; &= \; \{\, (bob, \{bob, doc\}), (amy, \{doc, g\})\} \\ L_{I,x} \; &= \; \{(bob, \{bob, mam, carl\}), (amy, \{bob, g\})\} \end{aligned}$$

$L_{C,x} \; = \; \{K, J\}$, où K $= (bob, \{bob, doc\})$,
J $= (amy, \{doc, g\})$ sont les composantes de l'étiquette de sécurité $L_{C,x}$ et permettent chacune de définir une politique de sécurité sur la donnée x. Ces politiques de sécurité stipulent que les objets *bob* et g sont peuvent modifier la donnée x. Cette permission leur est attribuée par l'objet *bob*. De plus, l'objet g est aussi admis comme lecteur par l'intermediaire de l'objet amy.

La prochaine section 3.4 traite de la définition formelle de la relation de restriction. Cette relation permet de déterminer les niveaux de sécurité et de définir le treillis de sécurité sur l'ensemble des étiquettes.

3.4 Définition d'un flux d'information

Ces étiquettes de sécurité définissent des règles qui permettent de manipuler et modifier les politiques de sécurité définies sur des données. Elles permetttent de vérifier si les affectations exécutées ne révèlent pas de l'information sur les données. Les étiquettes respectives des variables x et v sont :

$$\begin{aligned} L_x &= (L_{C,x}, L_{I,x}) \\ L_v &= (L_{C,v}, L_{I,v}) \end{aligned}$$

Les étiquettes d'intégrité et de confidentialité de la donnée x sont comparées respectivement avec celle de la donnée v, et de façon intuitive, le flux d'information est permis si $L_{C,x}$ est plus restrictive que $L_{C,v}$ et $L_{I,v}$ est plus restrictive que $L_{I,x}$. En d'autres termes il s'agit de vérifier quels sont les utilisateurs qui modifient la donnée x et de vérifier les utilisateurs qui lisent la donnée v, lorsque l'affectation se produit. Pour évaluer le flux d'information, on définit un treillis de sécurité sur l'ensemble des étiquettes $(\mathsf{Label}, \sqsubseteq)$.

Lors de la vérification de la confidentialité, l'expression $L_{C,x} \sqsubseteq L_{C,v}$ signifie que l'étiquette de la donnée x est plus restrictive que l'étiquette de la donnée v (i.e. les utilisateurs qui sont autorisés à lire la donnée x sont également autorisés à lire la donnée x). Les contraintes de confidentialité respectent les propriétés de sécurité simple et l'$\star$-*propriété* telles que définies par Bell et LaPadula (1976). Ces propriétés stipulent qu'aucun usager ne peut lire des données à un niveau supérieur (*no read up*) ou écrire des données à un niveau inférieur (*no write down*).

De façon similaire, $L_{I,v} \sqsubseteq L_{I,x}$ signifie que l'étiquette de la donnée v est plus restrictive que l'étiquette de la donnée v (i.e. les utilisateurs qui sont autorisés à modifier la donnée v sont également autorisés à modifier la donnée x). De plus, les politiques d'intégrité sont le dual des politiques de confidentialité. Cette caractéristique a été démontrée par Bell et LaPadula (1976). De façon intuitive les contraintes d'intégrité restreignent la lecture des données avec un niveau de sécurité plus bas (*no read down*) et restreignent l'écriture des données avec un niveau de sécurité plus haut (*no write up*).

On définit dans cette section la relation de restriction :

$$\sqsubseteq \subseteq \mathsf{Label} \times \mathsf{Label}$$

Cette relation de restriction permet de classer les étiquettes de sécurité selon leurs niveaux de sécurité. Les règles de restriction sur les étiquettes s'appliquent au niveau de l'intégrité et de la confidentialité, ainsi si $L_{C,2} \in \mathsf{CLabel}$ et $L_{I,2} \in \mathsf{ILabel}$

$$L_2 \sqsubseteq L_1 \iff \begin{cases} L_{C,2} \sqsubseteq_C L_{C,1} \\ L_{I,1} \sqsubseteq_I L_{I,2} \end{cases}$$

Cette section est constituée de deux parties. La première partie (sous-section 3.4.1) se consacre à la définition de la relation de restriction $\sqsubseteq$. La seconde partie (sous-section 3.4.2) décrit une méthode pour évaluer des étiquettes de sécurité de données obtenues à partir d'autres données.

3.4.1 Relation de restriction entre les étiquettes

La restriction est une relation entre deux étiquette L_1 et L_2. Elle permet d'indiquer le flux légal de l'information pour une hiérarchie d'objets. On utilise la notation $L_2 \sqsubseteq L_1$ pour indiquer que L_2 est plus restrictive que L_1. La relation de restriction est définie à partir de la relation de restriction pour la confidentialité et la relation de restriction pour l'intégrité.

$$\begin{aligned} \sqsubseteq_C &\subseteq \mathsf{CLabel} \times \mathsf{CLabel} \\ \sqsubseteq_I &\subseteq \mathsf{ILabel} \times \mathsf{ILabel} \end{aligned}$$

Les règles de restriction sur les étiquettes s'appliquent pour la protection de l'intégrité et de la confidentialité, c'est à dire

$$L_2 \sqsubseteq L_1 \Longleftrightarrow \begin{cases} L_{C,2} \sqsubseteq_C L_{C,1} \\ L_{I,1} \sqsubseteq_I L_{I,2} \end{cases}$$

Définition 1. *L'étiquette de confidentialité la moins restrictive possible est notée $L_{C,\perp}$ et correspond au flux qui peut se déplacer partout.*

$$L_{C,\perp} = \{\ \}\,,\ et\, \forall l_c \in \mathit{CLabel},\ l_c \sqsubseteq_C L_{C,\perp}$$

L'étiquette de confidentialité la plus restrictive possible est notée $L_{C,\top}$ et ne possède pas de lecteurs et tous les objets sont propriétaires.

$$L_{C,\top} = \{(o, \{\})\ : o \in \mathit{NOId}\}\,,\ et\, \forall l_c \in \mathit{CLabel},\ L_{C,\top} \sqsubseteq_C l_c$$

Définition 2. *L'étiquette d'intégrité la moins restrictive possible est notée $L_{I,\perp}$ et correspond au flux qui peut se déplacer partout.*

$$L_{I,\perp} = \{(o, \{\})\ : o \in \mathit{NOId}\}\,,\ et\, \forall l_c \in \mathit{ILabel},\ L_{I,\perp} \sqsubseteq_I l_i$$

L'étiquette d'intégrité la plus restrictive possible est notée $L_{I,\top}$ et ne possède pas de lecteurs et tous les objets sont propriétaires.

$$L_{I,\top} = \{\},\ et\, \forall l_i \in \mathit{ILabel},\ l_i \sqsubseteq_I L_{I,\top}$$

Soit Label un ensemble d'étiquettes de sécurité, l'étiquette la plus restrictive dans le treillis de sécurité (Label,$\sqsubseteq$) est

$$L_\top = (L_{C,\top}, L_{I,\top})$$

et l'étiquette la moins restrictive est définie par :

$$L_\bot = (L_{C,\bot}, L_{I,\bot})$$

Relation de restriction pour la confidentialité Considérons l'opération d'affectation :

$$x \ := \ v$$

et les étiquettes $L_v = (L_{C,v}, L_{I,v})$ et $L_x = (L_{C,x}, L_{I,x})$. La vérification de la restriction pour des étiquettes de confidentialité revient à vérifier que la variable v ne sera pas lue de façon non sécuritaire lorsqu'elle est affectée à la variable x. Le flux qui passe de la variable v à la variable x doit être réduit, donc l'ensemble des lecteurs effectifs de la variable x doit être inclus dans celui de la variable v. Une étiquette de confidentialité $L_{C,x}$ doit permettre moins de lecteurs que l'étiquette $L_{C,v}$, et $L_{C,x} \sqsubseteq_C L_{C,v}$ équivaut à :

- $\mathsf{owners}(v) \subseteq \mathsf{owners}(x)$: la variable x doit admettre un nouveau propriétaire capable de spécifier éventuellement de nouvelles contraintes de lecture.
- $\forall o \in \mathsf{owners}(v),\ \mathsf{readers}(x, o) \ \subseteq \ \mathsf{readers}(v, o)$

La restriction entre les étiquettes de confidentialité peut-être exprimée de façon formelle par :

$$\begin{array}{l} L_{C,x} \sqsubseteq_C L_{C,v} \ \equiv \ \forall I \in L_{C,x}, \exists J \in L_{C,v}.\ I \sqsubseteq_C J \\ et\ I \sqsubseteq_C J \ \equiv \begin{cases} (\ \mathsf{owners}^\star(J)\ \mathsf{h}\ \mathsf{owners}^\star(I)\ ) \wedge (\ \mathsf{readers}^{\star\star}(J) \subseteq \mathsf{readers}^{\star\star}(I)\ ) \\ (\ \mathsf{owners}^\star(J)\ \mathsf{h}\ \mathsf{owners}^\star(I)\ ) \wedge \forall r_j \in \mathsf{readers}^\star(J)\ \exists r_i \in \mathsf{readers}^\star(I),\ r_j\ \mathsf{h}\ r_i,\ \mathsf{h} \in \mathcal{H} \end{cases} \end{array}$$

Les algorithmes 4 et 7, permettent d'évaluer les relations de restriction entre les étiquettes de confidentialité et les composantes de confidentialité.

Exemple L'exemple qui suit présente deux variables x et v, et leurs étiquettes de confidentialité respectives $L_{C,x}$ et $L_{C,v}$. Soit l'affectaion suivante :

$$x \ := \ v$$

On veut vérifier si cette affectation est sécuritaire en vérifiant $L_{C,x} \sqsubseteq_C L_{C,v}$ dans la hiérarchie $\mathsf{h} = \{(carl, doc), (carl, mam), (mam, bob), (mam, amy), (g, bob), (g, amy)\}$.
Soient $L_{C,v} \ = \ \{(bob, \{amy, bob\})\}$ et
$L_{C,x} \ = \ \{(bob, \{bob\}), (amy, \{carl\})\}$,

$(bob, \{bob\}) \sqsubseteq_C (bob, \{amy, bob\})$,
car $\mathsf{owners}^\star((bob, \{bob\})) = bob)$, $\mathsf{owners}^\star((bob, \{amy, bob\})) = bob$ et $bob \mathsf{h} bob$
$(amy, \{carl\}) \sqsubseteq_C (bob, \{amy, bob\})$
car $\mathsf{owners}^\star((amy, \{carl\})) = amy)$, $\mathsf{owners}^\star((bob, \{amy, bob\})) = bob$ et $bob \mathsf{h} amy$ n'est pas satisfaite, par conséquent $L_{C,x} \sqsubseteq L_{C,v}$ n'est pas satisfaite.
Une conséquence de la restriction pour la confidentialité est qu'une donnée v peut être affectée à une donnée x, sans causer de flux d'information illégal dans les cas suivants. Premièrement, le retrait d'un lecteur d'une composante de l'étiquette $L_{C,x}$ ne cause pas de flux d'information illégal puisque le propriétaire de la donnée réduit le nombre d'utilisateurs capables de lire la donnée. L'ajout d'une composante de sécurité à l'étiquette $L_{C,x}$ ne modifie pas le flux d'information. Toutes les composantes existantes son toujours présentes et leurs contraintes de sécurité sont toujours appliquées. La troisième façon de modifier une étiquette de sécurité sans modifier les contraintes des sécurité est d'ajouter un lecteur r' à une composante si $r' \mathsf{h} r$ et r appartient à l'ensemble des lecteurs de la variable (r' a tous les privileges de r). La dernière façon de modifier une étiquette est de substituer un propriétaire o d'une composante par un autre utilisateur o', si $o' \mathsf{h} o$, et $h \in \mathcal{H}$.

Relation de restriction pour l'intégrité Considérons l'opération d'affectation :

$$x := v$$

et les étiquettes $L_v = (\ L_{C,v},\ L_{I,v}\ )$ et $L_x = (\ L_{C,x},\ L_{I,x}\ )$. La vérification de la restriction pour des étiquettes d'intégrité revient à vérifier que la variable x ne sera pas modifiée illégalement lorsque la variable v est écrite. Le flux qui passe de la variable v à la variable x doit être réduit, donc l'ensemble des modificateurs effectifs de la variable v doit être inclus dans celui de la variable x. Une étiquette d'intégrité $L_{I,v}$ doit permettre moins de modificateurs que l'étiquette $L_{C,x}$, et $L_{I,x} \sqsubseteq_I L_{I,v}$ équivaut à :

- $\mathsf{owners}(x) \subseteq \mathsf{owners}(v)$: la variable x doit admettre un nouveau propriétaire capable de spécifier éventuellement de nouvelles contraintes d'écriture.
- $\exists o \in \mathsf{owners}(v),\ \mathsf{writers}(v, o) \subseteq \mathsf{writers}(x, o)$

La restriction entre les étiquettes d'intégrité peut-être exprimée de façon formelle par :

$$L_x \sqsubseteq_I L_v \equiv \exists I \in L_x, \forall J \in L_v.\ I \sqsubseteq_I J$$
$$et\ I \sqsubseteq_I J \equiv \begin{cases} (\ \mathsf{owners}^\star(I)\ \mathsf{h}\ \mathsf{owners}^\star(J)\ ) \wedge\ (\ \mathsf{writers}^{\star\star}(I) \subseteq \mathsf{writers}^{\star\star}(j)\ ) \\ (\ \mathsf{owners}^\star(I)\ \mathsf{h}\ \mathsf{owners}^\star(J)\ ) \wedge \forall w' \in \mathsf{writers}^\star(I), \exists w \in \mathsf{writers}^\star(J),\ w\ \mathsf{h}\ w', \mathsf{h} \in \mathcal{H} \end{cases}$$

Les algorithmes 5 et 8 permettent d'évaluer les relations de restriction entre les étiquettes d'intégrité et les composantes d'intégrité.

Dans le cas de l'intégrité, une donnée v peut-être affectée à une donnée x, sans causer de flux d'information illégal, dans les quatre cas suivants. La première façon est d'ajouter un modificateur à une composante d'intégrité. La deuxième est de retirer une composante d'intégrité. En effet, une composante d'intégrité permet de spécifier un ensemble donné de modificateurs capable d'affecter la donnée. Le retrait d'une telle assurance est sûr et limite l'utilisation ultérieure de la valeur. Troisièmement, la substituion d'un modificateur w' par un modificateur w, si $w' \mathsf{h}\, w$ permet de préserver les contraintes de sécurité. Une composante d'intégrité qui admet w comme modificateur admet également w' comme modificateur mais le contraire n'est pas toujours vérifié. Donc un remplacement de w' par w ajoute des modificateurs. Finalement l'ajout d'une composante d'intégrité si la nouvelle composante I' est identique à une composante existante I, si o est le propriétaire de la composante I, si o' est le propriétaire de la composante I', et $o \,\mathsf{h}\, o'$, $h \in \mathcal{H}$.

La prochaine section traite de la façon dont l'étiquette d'une donnée est calculée lorsque cette donnée est calculée à partir d'autres données

3.4.2 Étiquettes calculées

On considère les situations où le flux d'information peut provenir de plusieurs sources ou bien peut se propager vers plusieurs sources. Plus concrètement, on considère la situation suivante. Soit x, y, z des variables, si la combinaison des variables y et z est affectée à la variable x, on se doit de déterminer l'étiquette la moins restrictive capable de respecter le flux d'information imposé à la fois par les variables x, y et z. Une combinaison de variables correspond dans notre cas à une opération arithmétique entre les variables (addition,multiplication, modulo, etc).

$$x \;:=\; y \otimes z, \quad \text{avec } \otimes \text{ un opérateur arithmétique quelconque}$$

Confidentialité

Jointure On considère v_1 et v_2 deux données et L_{C,v_1} et L_{C,v_2} leurs étiquettes de confidentialité respectives. Soit v une valeur obtenue à partir des données v_1 et v_2, c'est à dire $v = v_1 \otimes v_2$. L'étiquette la moins restrictive capable de respecter les contraintes imposées par les deux étiquettes (c'est à dire respecter les contraintes imposées par le treillis de sécurité lorsqu'il y a un flux d'information de v_1, v_2 vers x) L_{C,v_1} et L_{C,v_2} est appelée la jointure des étiquettes L_{C,v_1} et L_{C,v_2}. La jointure permet de calculer l'étiquette de sécurité du résultat d'une opération impliquant plusieurs variables. elle est notée $L_{C,v_1} \sqcup L_{C,v_2}$ et correspond à

l'intersection des étiquettes :

$$L_{C,v_1} \sqcup L_{C,v_2} \equiv L_{C,v_1} \cap L_{C,v_2}$$

La jointure $L_{C,v_1} \sqcup L_{C,v_2}$ est au moins aussi restrictive que les deux étiquettes, c'est à dire $L_{C,x} \sqsubseteq_C L_{C,v_1}$ et $L_{C,x} \sqsubseteq_C L_{C,v_2}$).

Réunion Si on considère v_1 et v_2 deux variables et L_{C,v_1} et L_{C,v_2} leurs étiquettes respectives. L'étiquette la plus restrictive $L_{C,v}$ qui peut être changée (sans causer de flux d'information illégal) pour les étiquettes L_{C,v_1} et L_{C,v_2} ($L_{C,v} \sqsubseteq_C L_{C,v_1}$ et $L_{C,v} \sqsubseteq_C L_{C,v_2}$) est appelée la réunion des étiquettes $L_{C,1}$ et $L_{C,2}$. Elle est notée $L_{C,v_1} \sqcap L_{C,v_2}$ et correspond à l'union des étiquettes :

$$L_{C,v_1} \sqcap L_{C,v_2} \equiv L_{C,v_1} \cup L_{C,v_2}$$

Intégrité

Jointure L'étiquette la moins restrictive capable de respecter les contraintes imposées par les étiquettes L_{I,v_1} et L_{I,v_2} est appelée la réunion des étiquettes $L_{C,1}$ et $L_{C,2}$, elle est notée $L_{I,v} = L_{I,1} \sqcap L_{I,2}$ et correspond à : L'étiquette la plus restrictive capable de respecter les contraintes imposées par les étiquettes L_{I,v_1} et L_{I,v_2} est appelée la réunion des étiquettes $L_{I,1}$ et $L_{I,2}$, elle est notée $L_{I,v} = L_{I,1} \sqcup L_{I,2}$ et correspond à l'union des étiquettes :

$$L_{I,v_1} \sqcup L_{I,v_2} \equiv L_{I,v_1} \cup L_{I,v_2}$$

Réunion L'étiquette la plus restrictive qui peut être changée (sans causer de flux d'information illégal) pour les étiquettes L_{C,v_1} et L_{C,v_2} est appelée la réunion des étiquettes $L_{C,1}$ et $L_{C,2}$, elle est notée $L_{C,v} = L_{v_1} \sqcap L_{v_2}$ et correspond à l'intersection des étiquettes :

$$L_{I,v_1} \sqcap L_{I,v_2} \equiv L_{I,v_1} \cap L_{I,v_2}$$

CHAPITRE 4

SYNTAXE ET SÉMANTIQUE DES MODÈLES UML

On décrit dans le chapitre 4 la syntaxe et la sémantique formelles pour nos modèles UML. On modélise le diagramme de classe, le diagramme d'objets et les diagrammes d'états de chaque objet du système orienté objet. Ce modèle formel est une extension des modèles de von der Beeck (2002) et Bergeron (2004). La syntaxe de notre modèle est une extension des syntaxes de Bergeron (2004). En effet, on utilise les mêmes fragments UML (diagramme de classe, diagramme d'objets et diagrammes d'états) auxquels on rajoute des éléments descriptifs sur les attributs, les paramètres et les variables afin d'exprimer la sécurité des systèmes orientés objet. La sécurité est exprimée grâce aux étiquettes de sécurité décentralisées décrites au chapitre précédent (Chapitre 3). La sémantique de nos modèles UML est exprimée à trois niveaux. Premièrement, la sémantique sur les diagrammes de classes et d'objets exprime des contraintes de sécurité sur des attributs, des variables et des paramètres au niveau des diagrammes de classes et d'objets en donnant la valeur des étiquettes de sécurité (ensemble des objets qui ont les droits en lecture et en écriture sur les données). Ensuite, la sémantique permet également de calculer la valeur des étiquettes de sécurité pour des expressions obtenues à partir de la combinaison d'attributs, de paramètres ou de variables. En effet, elle permet au niveau des diagrammes d'états de calculer la valeur des étiquettes sur les gardes et les actions des transitions. Enfin, comme la sémantique de von der Beeck (2002), elle exprime la sémantique dynamique en donnant les différents ensembles d'états UML actifs et les séquences d'actions qui s'exécutent pour des diagrammes d'états, lorsqu'une transition s'exécute.

On modélise dans ce chapitre les fragments du langage UML qui seront utilisés pour modéliser le système. La première partie (Section 4.1) présente les hypothèses de formalisation utilisées dans la syntaxe et la sémantique. La seconde partie (Section 4.2) décrit la syntaxe et la sémantique formelle des diagrammes classes, d'objets et d'états.

4.1 Hypothèses de formalisation UML

Les diagrammes UML sont utilisés pour simuler la structure statique et dynamique des systèmes orientés objet. Dans notre étude, on utilise des versions simplifiées des diagrammes UML. On utilise exactement un diagramme de classes, un diagramme d'objets et les diagrammes d'états des objets. Les diagrammes de classes et d'objets décrivent la structure

statique du système et les diagrammes d'états décrivent la structure dynamique. L'ensemble des données fait référence à l'union des ensembles d'attributs, de variables et de paramètres de méthode.

4.1.1 Domaines des valeurs

Val	:	ensemble de domaines de valeurs
Int	:	ensemble des nombres entiers
Bool	:	ensemble des valeurs booléennes

Figure 4.1 Listes des notations - Domaines de valeurs

L'ensemble des domaines de valeurs est défini à la figure 4.1. Il s'agit des ensembles des valeurs que peuvent prendre les données. Néanmoins notre étude ne fait pas une vérification sur les types Les types de valeurs utilisés sont : les entiers, les booléens. L'ensemble des valeurs possibles est la réunion des ensembles d'entiers et de booléens :

$$\mathsf{Val} = \mathsf{Int} \cup \mathsf{Bool}$$

où :

- Int est l'ensemble des entiers :

$$\mathsf{Int} = \{\ldots -2, -1, 0, 1, 2 \ldots\}$$

- Bool est l'ensemble des valeurs booléennes :

$$\mathsf{Bool} = \{true, false\}$$

4.1.2 Fragments UML : Diagrammes de classes, diagrammes d'objets et diagrammes d'états

Diagrammes de classes On utilise uniquement les éléments suivants pour représenter une classe du système :

- le nom de la classe
- la liste des attributs (noms des attributs) de la classe
- la liste des méthodes (noms des méthodes) de la classe
- la liste des étiquettes de sécurité sur les attributs et les paramètres des méthodes

La notion d'encapsulation n'est pas modélisée, ainsi la visibilité au niveau des attributs, méthodes et classe sera toujours considérée comme *public*. L'héritage et le polymorphisme ne seront pas pris en compte lors de la formalisation des diagrammes de classes. Également, notre diagramme de classe ne contient que des classes concrètes (les interfaces et les classes abstraites sont proscrites). Les seules associations permises entre les classes sont des associations mono-directionnelles permettant d'indiquer l'invocation d'une méthode d'une classe.

Diagrammes d'objets Il représente les instances de classes (objets) utilisées dans le système. Il spécifie les configurations des objets c'est-à-dire la valeur des attributs des objets. Ceci permet d'exprimer des contextes d'exécution. Les attributs des objets sont représentés avec des noms différents même si les objets sont des instances de la même classe. Ceci, afin d'identifier de façon unique chacun des attributs des objets. Des étiquettes de sécurité sont assignées à chaque attribut des objets.

Diagramme d'états Un diagramme d'états donne une représentation des états d'un objet du système sous forme d'un automate à états finis. On associe à chaque objet son diagramme d'états. Un diagramme d'états représente un graphe du cycle de vie d'un objet. Les diagrammes d'états mettent en évidence les réactions des objets à des évènements déclencheurs. Les évènements supportés par notre modèle sont les appels et retours de méthode sur un objet, les signaux asynchrones envoyés par d'autres objets et les évènements automatiques, qui correspondent à la fin d'exécution de certaines activités.
Une garde est une condition booléenne qui permet d'effectuer une transition si la garde est évaluée à vrai. Un diagramme d'états est constitué d'états reliés entre eux par des transitions. Les transitions contiennent de l'information sur l'évènement déclencheur de la transition, la garde de la transition (condition pour que la transtion s'exécute) et les séquences d'actions qui sont exécutées lorsque la transition survient. Une transition est représentée par un évènement, une garde et une action à exécuter lorsque la transition survient entre deux états simples e_1 et e_2 est représentée par un arc qui les relie. La transition indique qu'une instance dans e_1 peut entrer dans e_2 et exécuter certaines actions, si un évènement déclencheur survient et que les contions imposées par les gardes sont satisfaites. Un état peut être soit simple (aucun état imbriqué interne), soit composé (états imbriqués internes). Un état composé (aussi appelé état composite) est constitué d'une ou plusieurs sous-états à l'intérieur desquels peuvent se trouver d'autres sous-états. La notion de région permet de modéliser les états concurrents (AND-state) et séquentiels (OR-state). Les transitions internes qui surviennent dans un états ne sont pas formalisées dans notre étude, mais on modélise les séquences d'actions qui s'exécutent lorsqu'on entre ou sort d'un état (ou d'une région). Les étiquettes de sécurité au niveau

des gardes sont évaluées à partir des données présentes dans la syntaxe de la garde.

On décrit dans la section suivante (Section 4.2) le modèle formelle des système orienté objet en se basant sur les hypothèses précédentes.

4.2 Modélisation UML

On décrit dans cette section la syntaxe et la sémantique des diagrammes UML. Cette modélisation prend en considération les hypothèses décrites à la section 4.1.

4.2.1 Syntaxe des diagrammes UML

NVar	:	ensemble des noms de variables
NAtt	:	ensemble des noms d'attributs
NCAtts	:	ensemble des noms d'attributs de classe
NPar	:	ensemble des noms de paramètres formels de méthodes
OId	:	ensemble d'objets
NOId	:	ensemble de noms d'objets
NClass	:	ensemble des noms de classes
Class	:	ensemble des classes
Label	:	ensemble des étiquettes de sécurité
NMeths	:	ensemble des noms de méthodes
MId	:	ensemble des instances de méthodes
Vars	:	ensemble des variables
Atts	:	ensemble d'attributs
CAtts	:	ensemble des attributs de classe
Meths	:	ensemble des méthodes de la classe
Pars	:	ensemble des paramètres formels des méthodes

Figure 4.2 Listes des notations - Diagrammes de classes et d'objets

NStateCharts	:	ensemble des noms des diagrammes d'états
StateCharts	:	ensemble des diagrammes d'états
Trans	:	ensemble de transitions
Triggers	:	ensemble de déclencheurs d'une transtion
Guards	:	ensemble des gardes
Actions	:	ensemble d'actions
SActions	:	ensemble des séquences d'actions
NTrans	:	ensemble des noms de transition
NTriggers	:	ensemble des noms de déclencheurs
NActions	:	ensemble des noms d'actions
TType	:	ensemble des types de déclencheurs
AType	:	ensemble des types d'actions
MType	:	ensemble des types de messages
NState	:	ensemble de noms d'état (diagramme d'états)
States	:	ensemble d'états (diagramme d'états)

Figure 4.3 Listes des notations - diagramme d'états

Notre système orienté objet est décrit à l'aide d'exactement d'un diagramme de classe, d'un diagramme d'objets et des diagrammes d'états. Le diagramme de classe donne la structure statique de notre système et les relations entre les classes qui le composent. Le diagramme d'objets donne la configuration de départ du système : les objets qui constituent le système, les valeurs initiales de leurs attributs, ainsi que les valeurs initiales des étiquettes de sécurité des attributs et des paramètres de méthodes. Le diagramme d'états représente les évolutions de l'état d'un objet du système à chaque transition effectuée.

4.2.1.1 Diagramme de Classe

Le diagramme de classes permet de décrire les composantes d'un système et les relations entre ces composantes. Ceci permet de donner une structure hiérarchique du système. Cependant il ne permet pas de définir le nombre, l'état des objets et l'état du système. Ce type de diagramme est une représentation statique du système. On utilise uniquement les éléments suivants pour représenter une classe du système :

- le nom de la classe
- la liste des attributs de la classe
- la liste des méthodes de la classe
- la liste des étiquettes de sécurité sur les attributs et les paramètres des méthodes

Description formelle d'une variable Une variable $v \in \mathsf{Vars}$ est décrite par son nom, sa valeur et l'étiquette de sécurité qui lui est affectée, de façon formelle cela se traduit par :

$$v = (nv, value, L_v)$$

où :

- $nv \in \mathsf{NVar}$: est le nom de la variable
- $value \in \mathsf{Val}$: est la valeur de la variable v
- $L_v \in \mathsf{Label}$: est l'étiquette de sécurité de la variable v

Description formelle d'un attribut de classe Un attribut $a \in \mathsf{CAtts}$ est décrit par son nom, sa valeur et l'étiquette de sécurité qui lui est affectée, de façon formelle cela se traduit par :

$$a = (na, value, L_a)$$

où :

- $na \in \mathsf{NCAtts}$: est le nom de l'attribut de classe
- $value \in \mathsf{Val}$: est la valeur de l'attribut a
- $L_a \in \mathsf{Label}$: est l'étiquette de sécurité de l'attribut a

La valeur et l'étiquette de sécurité ne sont pas pertinentes pour des attributs de classe, puisque chaque instance de la classe les initialise avec des valeurs différentes (Sous-section 4.2.1.2).

Description formelle d'un paramètre Un paramètre $p \in \mathsf{Pars}$ est décrit par son nom, sa valeur et l'étiquette de sécurité qui lui est affectée, de façon formelle cela se traduit par :

$$p = (np, value, L_p)$$

où :

- $np \in \mathsf{NPar}$: est le nom du paramètre.
- $value \in \mathsf{Val}$: est la valeur du paramètre p
- $L_p \in \mathsf{Label}$: est l'étiquette de sécurité du paramètre p

Définition de l'ensemble des données L'ensemble des données Data du système correspond à l'ensemble des variables, des paramètres et des attributs. Il est noté Data et est défini par :

$$\mathsf{Data} = \mathsf{Vars} \cup \mathsf{Pars} \cup \mathsf{Atts} \cup \mathsf{CAtts}$$

La fonction value associe à une donnée sa valeur.

$$\mathsf{value} \ : \ \mathsf{Data} \longrightarrow \mathsf{Val}$$

La fonction label associe à une donnée son étiquette de sécurité.

$$\mathsf{label} \ : \ \mathsf{Data} \longrightarrow \mathsf{Label}$$

La fonction name associe à une donnée son nom.

$$\mathsf{name} \ : \ \mathsf{Data} \longrightarrow \mathsf{NVar} \cup \mathsf{NPar} \cup \mathsf{NAtt} \cup \mathsf{NCAtts}$$

Description formelle d'une méthode Une méthode est décrite par son nom, l'ensemble de ses paramètres formels et un paramètre de retour. De façon formelle, une méthode $m \in \mathsf{Meths}$ de la classe C est décrite par :

$$m = (nm, MPars, MReturn)$$

où :
- $nm \in \mathsf{NMeths}$, est le nom de la méthode.
- $MPars \subseteq \mathsf{Pars}$. La fonction $\mathsf{parameters}$ associe à une méthode les paramètres formels de cette dernière.
$$\mathsf{parameters} \ : \ \mathsf{Meths} \longrightarrow \mathcal{P}(\mathsf{Pars})$$
- $MReturn \in \mathsf{Pars}$, et la fonction return associe à une méthode le paramètre de retour de cette dernière.
$$\mathsf{return} \ : \ \mathsf{Meths} \longrightarrow \mathsf{Pars}$$

Les détails des fonctions qui décrivent les données et les méthodes sont décrites dans l'annexe B.

Opérateurs arithmétiques Les opérateurs arithmétiques permettent de construire des opérations sur une ou plusieurs expressions (par exemple addition des valeurs de deux attributs).

Opérateur de calcul	BOp	=	$\{/, \%, +, \times, \&\&, \|\|\}$
Opérateur de comparaison	COp	=	$\{<=, >=, <, >, ==, !=\}$
Opérateur unaire	UOp	=	$\{+, -, !=\}$

Figure 4.4 Ensembles des Opérateurs $\mathsf{COp}, \mathsf{BOp}, \mathsf{UOp}$

On distingue trois types d'opérateurs. La liste des opérateurs est donnée à la figure 4.4. BOp est l'ensemble des opérateurs de calcul binaire qui permettent de calculer les valeurs de deux expressions.

- Addition. $+$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Int})$
- Soustraction. $-$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Int})$
- Division. $/$: $\mathsf{Expressions} \times \mathsf{IntExpressions}(\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Int})$
- Multiplication. $\times$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Int})$
- Modulo (Reste de la division entière). $\%$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Int})$
- ET logique. $\&\&$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Bool} \times \mathsf{Bool} \longrightarrow \mathsf{Bool})$
- OR logique. $||$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Bool} \times \mathsf{Bool} \longrightarrow \mathsf{Bool})$

COp est l'ensemble des opérateurs de comparaison qui permettent de comparer deux expressions.

- Supérieur ou égal. $>=$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Bool})$
- Inférieur ou égal. $<=$: $\mathsf{Expressions} \times \mathsf{Int} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Bool})$
- Égal. $==$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Bool})$
- Différent. $!=$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Bool})$
- Strictement supérieur. $\%$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Bool})$
- Strictement inférieur. $<$: $\mathsf{Expressions} \times \mathsf{Expressions} \longrightarrow (\mathsf{Int} \times \mathsf{Int} \longrightarrow \mathsf{Bool})$

UOp est l'ensemble des opérateurs unaires qui acceptent un seul argument.

- Plus unaire. $+$: $\mathsf{Expressions} \longrightarrow (\mathsf{Int} \longrightarrow \mathsf{Int})$
- Moins unaire. $-$: $\mathsf{Expressions} \longrightarrow (\mathsf{Int} \longrightarrow \mathsf{Int})$
- Complément logique (négation). $!$: $\mathsf{Expressions} \longrightarrow (\mathsf{Bool} \longrightarrow \mathsf{Bool})$

Syntaxe des expressions arithmétiques Les objets, les variables, les paramètres et les attributs peuvent être combinés à l'aide d'opérateurs pour former des expressions arithmétiques. La syntaxe des expressions arithmétiques est définie récursivement par :

$$e\ (\in \mathsf{Expressions})\ ::=\ v \mid x \mid e_1\ \mathsf{bop}\ e_2 \mid \mathsf{uop}\ e$$

où :

- $x \in \mathsf{Data}$, est une donnée
- $v \in \mathsf{Val}$, une valeur appartenant au domaine des valeurs
- $e_1 \; \mathsf{bop} \; e_2$: $e_1, \; e_2 \in \mathsf{Expressions}$, sont des expressions et bop est un opérateur, $\mathsf{bop} \in \mathsf{COp} \cup \mathsf{BOp}$
- $\mathsf{uop} \; e$: $e \in \mathsf{Expressions}$, uop est une expression $\mathsf{uop} \in \mathsf{UOp}$

Description formelle d'une classe La description d'une classe donne l'information sur les noms des membres (attributs, méthodes et paramètres de méthode) qui la constituent. Le diagramme de classe décrit la structure hiérarchique du système. Par conséquent, Il ne donne aucune information sur la valeur des attributs. Une classe $c \in \mathsf{Class}$ est décrite de façon formelle par :

$$c = (\, nc, \, CAtts, \, CMeths \,)$$

où :

- $nc \in \mathsf{NClass}$ est le nom de la classe.
- $CAtts \subseteq \mathsf{Atts}$ est l'ensemble des attributs de la classe.
  Soit $c \in \mathsf{Class}$, $CAtts \; = \; \mathsf{attributes}(c)$ donne l'ensemble des attributs de la classe c :

$$\mathsf{attributes} \; : \; \mathsf{Class} \; \longrightarrow \; \mathcal{P}(\mathsf{CAtts})$$

- $CMeths \subseteq \mathsf{Meths}$ un ensemble de noms de méthodes.
  Soit $c \in \mathsf{Class}$, $CMeths \; = \; \mathsf{methods}(c)$ donne l'ensemble des méthodes de la classe c :

$$\mathsf{methods} \; : \; \mathsf{Class} \; \longrightarrow \; \mathcal{P}(\mathsf{Meths})$$

Exemple L'exemple suivant décrit la syntaxe formelle du diagramme de classe décrit à la figure 4.5.

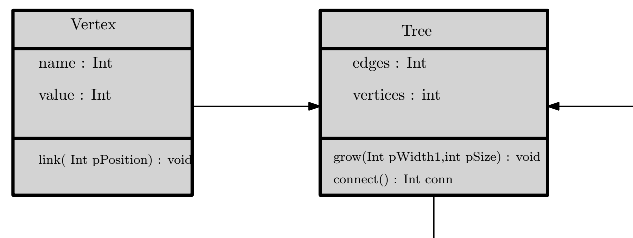

Figure 4.5 Diagramme de classe

Les ensembles des noms de classes, de méthodes, de paramètres et d'attributs sont alors

définis par :

$$\begin{aligned}
&\mathsf{NClass} = \{Tree, Vertex\}\\
&\mathsf{NMeths} = \{link, connect, grow\}\\
&\mathsf{NPar} = \{pWidth, pSize, pPosition, connect\}\\
&\mathsf{NCAtts} = \{edges, vertices, name, value\}
\end{aligned}$$

et les ensembles de classes, de méthodes, de paramètres et d'attributs sont définis par :

$$\begin{aligned}
&\mathsf{Class} = \{c_1, c_2\}\\
&\mathsf{Meths} = \{m_1, m_2, m_3\}\\
&\mathsf{Pars} = \{p_1, p_2, p_3, r_1\}\\
&\mathsf{Atts} = \{a_1, a_2, a_3, a_4\}
\end{aligned}$$

L'ensemble des attributs est constitué des quatre attributs :

$$\begin{aligned}
a_1 &= (edges, 0, L_\top)\\
a_2 &= (vertices, 0, L_\top)\\
a_3 &= (name, 0, L_\top)\\
a_4 &= (value, 0, L_\top)
\end{aligned}$$

L'ensemble des paramètres est constitué des quatre paramètres :

$$\begin{aligned}
p_1 &= (pWidth, 0, L_\top)\\
p_2 &= (pSize, 0, L_\top)\\
p_3 &= (pPosition, 0, L_\top)\\
r_1 &= (connect, 0, L_\top)
\end{aligned}$$

L'ensemble des méthodes est constitué des trois méthodes :

$$\begin{aligned}
m_1 &= (grow, \{p_1, p_2, r_1\}, \{\ \})\\
m_2 &= (connect, \{\ \}, \{r_1\})\\
m_3 &= (link, \{p_3\}, \{\ \})
\end{aligned}$$

L'ensemble des classes est constitué de deux éléments c_1 et c_2 :

$$\begin{aligned}
c_1 &= (Tree, \{p_1, p_2\}, \{m_1, m_2\})\\
c_2 &= (Vertex, \{p_3\}, \{m_3\})
\end{aligned}$$

4.2.1.2 Diagramme d'objets

Le diagramme d'objets permet de représenter les instances de classes, c'est-à-dire des objets. Comme le diagramme de classes, il exprime les relations qui existent entre les composantes, mais aussi les valeurs de leurs attributs, ce qui permet d'exprimer des contextes d'exécution et d'avoir une configuration du système dans un état précis. Le diagramme d'objets est moins général que le diagramme de classes, car il donne l'état des objets (valeurs des attributs,valeurs des étiquettes). Les attributs des objets sont représentés avec des noms différents même si les objets sont des instances de la même classe. Ceci, afin d'identifier de façon unique chacun des attributs des objets. Des étiquettes de sécurité sont assignées à chaque attribut des objets.

Description formelle d'un objet Un objet est une instance d'une classe C et permet d'initialiser les valeurs et les étiquettes de sécurité des attributs. Il est décrit par le nom de l'objet et la classe qu'il instancie De façon formelle un objet o est défini par :

$$o = (no, c)$$

où :
- $no \in \mathsf{NOId}$ est le nom de l'objet,
- $c \in \mathsf{Class}$ est la classe instanciée,

Description formelle d'un attribut d'objet Un attribut $a \in \mathsf{CAtts}$ est décrit par son nom, sa valeur et l'étiquette de sécurité qui lui est affectée, de façon formelle cela se traduit par :

$$a = (na, nc, value, L_a)$$

où :
- $na \in \mathsf{NAtt}$: est le nom de l'attribut de l'objet
- $nc \in \mathsf{NCAtts}$: est le nom de l'attribut de classe que l'objet instancie
- $value \in \mathsf{Val}$: est la valeur de l'attribut a
- $L_a \in \mathsf{Label}$: est l'étiquette de sécurité de l'attribut a

De façon générale, les fonctions $\mathsf{attributes}^*(o)$ et $\mathsf{methods}^*(o)$ permettent d'obtenir respectivement l'ensemble des attributs et des instances de méthodes de l'objet o et

$$\mathsf{attributes}^* \ : \ \mathsf{OId} \longrightarrow \mathcal{P}(\mathsf{CAtts})$$

De plus, si $o \in \mathsf{OId}$, où $o = (no, c)$, on a :

$$\forall a \in \mathsf{attributes}^*(o), \exists a' \in \mathsf{attributes}(c), \mathsf{name}(a) = \mathsf{name}(a'),$$
$$\forall a' \in \mathsf{attributes}(c), \exists a \in \mathsf{attributes}^*(o), \mathsf{name}(a) = \mathsf{name}(a'),$$

Les ensembles d'attributs de la classe et l'ensemble des atttributs d'une instance ont le même nombre d'éléments, les noms des éléments des deux ensembles sont les mêmes et les valeurs des attributs et des étiquettes peuvent être différentes.
La fonction instanceOf associe à un objet la classe qu'il instancie :

$$\mathsf{instanceOf} \ : \ \mathsf{OId} \longrightarrow \mathsf{Class}$$

Description formelle d'une instance de méthode Une instance de méthode permet de spécifier pour un objet $o \in \mathsf{OId}$, l'appel d'une méthode qui est en cours d'exécution. Elle permet également de connaitre la valeur et les étiquettes de sécurité des paramètres spécifiques passés lors de son exécution. De plus une instance de méthode $mid \in \mathsf{MId}$ éxécutée par un objet d'une classe est identifiée par :

$$mid = (o, m, nmid)$$

où :
- $o \in \mathsf{OId}$ l'objet pour lequel la méthode s'exécute,
- $m \in \mathsf{Meths}$ est la méthode qui s'exécute pour l'objet o.
- $nmid \in \mathsf{NMId}$ est le nom de l'instance de la méthode.

À chaque appel de la méthode qui s'exécute les paramètres passés à la méthode peuvent avoir une valeur différente et des étiquettes différentes. De façon générale, la notation $\mathsf{methods}^*(o)$ permet d'obtenir respectivement l'ensemble des instances de toutes les méthodes de l'objet o et :

$$\mathsf{methods}^* \ : \ \mathsf{OId} \longrightarrow \mathcal{P}(\mathsf{MId})$$

De plus, si $o \in \mathsf{OId}$, $o = (no, c)$
$\forall \, mid \in \mathsf{methods}^*(o)$, tel que $mid \ = \ (o, m, i)$, $m \in \mathsf{methods}^*(c)$

$$\mathsf{parameters}^* \ : \ \mathsf{MId} \longrightarrow \mathcal{P}(\mathsf{Pars})$$

De plus, si $mid \in \mathsf{MId}$, $mid = (o, m, i)$, $o = (no, c)$

$$\forall a \in \mathsf{parameters}^*(mid), \exists a' \in \mathsf{parameters}(m), \mathsf{name}(a) = \mathsf{name}(a'),$$
$$\forall a' \in \mathsf{parameters}(m), \exists a \in \mathsf{parameters}^*(mid), \mathsf{name}(a) = \mathsf{name}(a'),$$

Les ensembles de paramètres de méthode et l'ensemble des paramètres d'une instance de méthode ont le même nombre d'éléments et les noms des éléments sont les mêmes.
Si $o \in \mathsf{OId}$, $o = (no, c)$
$\forall\, mid \in \mathsf{methods}^*(o)$, tel que $mid \;=\; (o, m, i)$, $m \in \mathsf{methods}^*(c)$

$$\mathsf{return}^* \;:\; \mathsf{MId} \longrightarrow \mathsf{Pars}$$

est une fonction qui donne le paramètre de retour de l'instance $mid \in \mathsf{MId}$. Les détails des fonctions qui décrivent les objets sont décrites dans l'annexe B.

Exemple L'exemple qui suit décrit la syntaxe formelle du diagramme d'objets présenté à la figure 4.6.

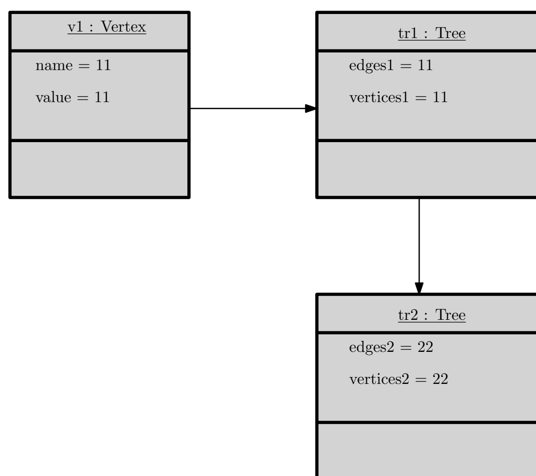

Figure 4.6 Diagramme d'objets

Le diagramme d'objets de la figure 4.6, représente deux instances de la classe *Tree*, $tr_1 = (tr_1, Tree)$ et $tr_2 = (tr_2, Tree)$ et une instance de la classe $Vertex$, $v_1 = (v_1, Vertex)$
On a

$$\mathsf{NOId} = \{tr_1, tr_2, v_1\}$$
$$\mathsf{OId} = \{tr_1, tr_2, v_1\}$$

où :

$$v_1 = (v_1, Vertex)$$
$$tr_1 = (tr_1, Tree)$$
$$tr_2 = (tr_2, Tree)$$

De plus, l'objet tr_1 invoque l'instance de la méthode *connect*, $connect_1$ et l'objet v_1 invoque

une deuxième instance de la méthode $connect_2$:

$$connect_1 = (tr_2, connect, connect_1)$$
$$connect_2 = (tr_2, connect, connect_2)$$

4.2.1.3 Diagramme d'états

Le diagramme d'état représente une vue dynamique d'un modèle UML. Il traduit le comportement et les changements d'états internes des objets du système. On associe à chaque objet de notre système orienté objet un diagramme d'états qui permet de représenter les états internes de cet objet. Les états correspondent aux étapes du cycle de vie de notre objet. Ce diagramme sert à représenter des automates d'états finis, sous forme de graphes d'états, reliés par des arcs orientés qui décrivent les transitions. Les diagrammes d'états permettent de décrire les changements d'états d'une instance d'une classe, en réponse aux interactions avec d'autres instances de classes ou avec des acteurs. On utilise deux types d'états : les états simples et les états composés. Les états simples sont des états qui n'ont pas de structure interne, alors que les états composés sont des états qui possèdent des sous-états. On distingue cinq types d'états :

- BEGIN : Il s'agit d'un état de début. Un état de début est un état spécial qui permet d'activer un état simple. Il est l'état initial dans lequel se trouve l'objet (dans son cycle de vie)
- END : Un état de fin est un état spécial qui permet de désactiver un état composite et tous les états simples qui le constituent. Il est l'état final dans lequel se trouve l'objet
- SIMPLE : Un état simple ne possède pas de structure interne (pas d'états imbriqués) Ces états peuvent se retrouver à l'intérieur d'autres états (états composites).
- OR : est un état qui est constitué d'états imbriqués.L'objet ne peut se trouver que dans un seul des sous-états qui le composent. un état OR peut avoir un état initial BEGIN, qui correspond à l'état de départ d'une transition vers un sous-état de l'état OR. Ce sous-état correspond à l'état initial par défaut de l'état OR. Une transition vers l'état OR correspond à une transition vers son état par défaut.
- AND : est un état qui est contient des états imbriqués. Un état AND est constitué d'une ou plusieurs régions. Chaque région possède un ensemble d'états. L'objet peut se trouver dans deux sous-états à un instant donné. Ces états s'exécutent en parallèle.

Les détails des fonctions qui décrivent les diagrammes d'états sont décrites dans l'annexe B. Les sous-sections qui suivent décrivent les éléments des diagrammes d'états (les régions,les états,les transistions,les actions et les gardes). Les sous-sections suivantes décrivent les régions des diagrammes d'états, les états des diagrammes d'états, les transitions des diagrammes

d'états et la syntaxe et la sémantique des diagrammes d'états.

Description formelle d'une région Les régions d'un diagramme d'états permettent de représenter des états qui s'exécutent en parallèle, et ainsi d'exprimer la notion de concurrence. Une région $r \in \mathsf{Regions}$ est définie de façon formelle par :

$$r = (nr, default, RStates)$$

où :
- $nr \in \mathsf{NRegion}$, le nom de la région
- $default \in \mathsf{States}$, est l'état par défaut de la région
- $RStates \subseteq \mathsf{States}$, l'ensemble des états qui appartiennent à cette région

On définit la fonction regionStates qui associe à une région, tous les états directs qu'elle contient :

$$\mathsf{regionStates} \ : \ \mathsf{Regions} \longrightarrow \mathcal{P}(\mathsf{States})$$

On définit la fonction defaultState qui associe à une région, son état par défaut :

$$\mathsf{defaultState} \ : \ \mathsf{Regions} \longrightarrow \mathsf{States}$$

Description formelle d'un état d'un diagramme d'états Dans les diagrammes d'état d'un objet UML, un état $state \in \mathsf{States}$ permet de représenter une étape du cycle de vie d'un objet UML. L'ensemble des types d'états est noté SType et

$$\mathsf{SType} = \{\mathsf{END}, \mathsf{SIMPLE}, \mathsf{AND}, \mathsf{OR}, \mathsf{BEGIN}\}$$

Un état $s \in \mathsf{States}$ est défini de façon formelle par :

$$s = (tstate, ns, regions, entry, exit)$$

où :
- $tstate \in \mathsf{SType}$, est le type de l'état :
- $ns \in \mathsf{NState}$, est le nom de l'état
- $regions \subseteq \mathsf{Regions}$ est l'ensemble des régions internes à l'état s
- $entry \in \mathsf{SActions}$, est une séquence d'actions effectuées lorsqu'on rentre dans un état
- $exit \in \mathsf{SActions}$, est une séquence d'actions effectuées lorsqu'on quitte un état

Les fonctions entry et exit associe respectivement à un état les séquences d'actions en entrée

et en sortie :

$$\mathsf{entry} : \mathsf{States} \longrightarrow \mathsf{SActions}$$
$$\mathsf{exit} : \mathsf{States} \longrightarrow \mathsf{SActions}$$

Les fonctions stype et regions associe respectivement à un état son type et l'ensemble de ses régions :

$$\mathsf{stype} : \mathsf{States} \longrightarrow \mathsf{TType}$$
$$\mathsf{regions} : \mathsf{States} \longrightarrow \mathcal{P}(\mathsf{Regions})$$

État initial Soit $s \in \mathsf{States}$, si $\mathsf{stype}(s) = \mathsf{BEGIN}$:

$$s = (\mathsf{BEGIN}, ns, \{\}, \langle\rangle, \langle\rangle)$$

Un état initial est un état qui ne contient aucune région. De plus, aucune action n'est effectuée à l'entrée ou à la sortie d'un état initial.

État final Soit $s \in \mathsf{States}$, si $\mathsf{stype}(s) = \mathsf{END}$:

$$s = (\mathsf{END}, ns, \{\}, \langle\rangle, \langle\rangle)$$

Un état final est un état qui ne contient aucune région. De plus, aucune action n'est effectuée à l'entrée ou à la sortie d'un état final.

État simple Soit $s \in \mathsf{States}$, si $\mathsf{stype}(s) = \mathsf{SIMPLE}$:

$$s = (\mathsf{SIMPLE}, ns, \{\}, entry, exit)$$

De plus, l'état simple s ne possède pas d'états internes. La figure 4.7 représente un état simple, un état initial b et un état final e.

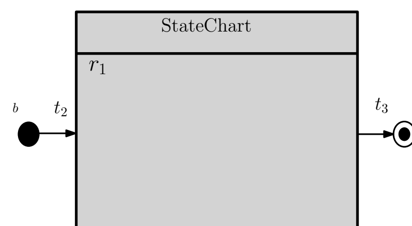

Figure 4.7 État Simple (SIMPLE)

État composite AND Soit $s \in \mathsf{States}$, Si $\mathsf{stype}(s) = \mathsf{AND}$:

$$s = (\mathsf{AND}, ns, regions, entry, exit)$$

L'état composite peut posséder des états internes et la cardinalité de l'ensemble des régions est différent de zéro, et le nombre de région est supérieur à un.

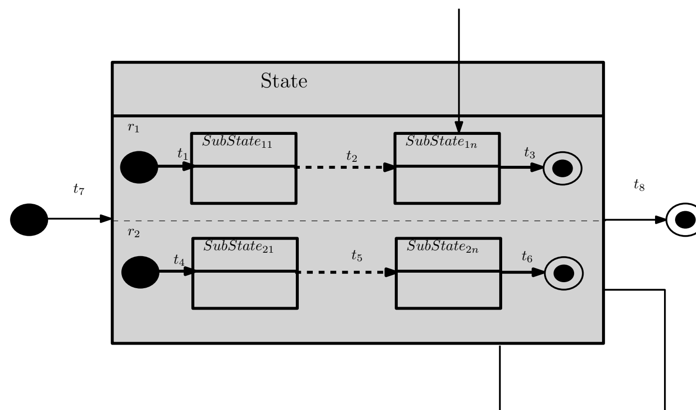

Figure 4.8 État composite (AND)

État composite OR Soit $s \in \mathsf{States}$, Si $\mathsf{stype}(s) = \mathsf{OR}$:

$$s = (\mathsf{OR}, ns, regions, entry, exit)$$

L'état composite peut posséder des états internes et la cardinalité de l'ensemble des régions est différente de zéro, le nombre de région est égal à un.

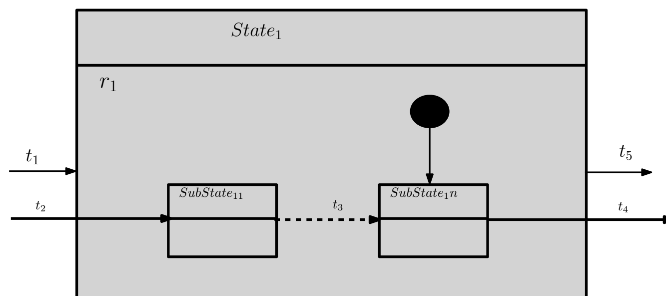

Figure 4.9 État composite (OR)

On définit la fonction $\mathsf{subStates}$ qui associe à un état d'un diagramme d'états l'ensemble de ses sous-états.

$$\mathsf{subStates} : \mathsf{States} \longrightarrow \mathcal{P}(\mathsf{States})$$

On définit la fonction superStates qui associe à un état d'un diagramme d'états tous ses états parents (ensemble de tous ses ancêtres).

$$\mathsf{superStates} \; : \; \mathsf{States} \longrightarrow \mathcal{P}(\mathsf{States})$$

On définit la fonction defaultStates qui associe à un état d'un diagramme d'états, l'ensemble de tous les états par défaut de ses régions.

$$\mathsf{defaultStates} \; : \; \mathsf{States} \longrightarrow \mathcal{P}(\mathsf{States})$$

Description formelle des transitions Une transition représente le passage d'un état vers un autre. Une transition est déclenchée par un événement et permet d'exécuter des actions. Les transitions peuvent aussi être automatiques, lorsqu'on ne spécifie pas l'événement qui la déclenche. En plus de spécifier un événement précis, il est aussi possible de conditionner une transition, à l'aide de gardes : il s'agit d'expressions booléennes. Une transition est décrite par un nom, un évènement qui la déclenche, une garde, une séquence d'actions, un état source et un état destination. De façon formelle, une transition $t \in \mathsf{Trans}$ est définie par :

$$t(\in \mathsf{Trans}) ::= (nt,\ trigger,\ guard,\ sactions,\ tgt,\ src\,)$$

où :
- $nt \in \mathsf{NTrans}$: le nom de la transition
- $trigger \in \mathsf{Triggers}$: l'évènement déclencheur de la transition. Certaines transitions sont automatiques et correspondent au cas où le déclencheur est vide.
- $guard \in \mathsf{Guards}$: la condition à remplir pour que la transition survienne. Il s'agit d'une expression booléenne,
- $sactions \in \mathsf{SActions}$: est la séquence d'actions qui sont effectuées lorsque la transition survient,
- $src \in \mathsf{States}$: est l'état source la transition.
- $tgt \in \mathsf{States}$: est l'état destination de la transition.

On définit les fonctions tgtState et srcState qui associe respectivement à une transtion l'état destination et l'état source.

$$\mathsf{tgtState} \; : \; \mathsf{Trans} \longrightarrow \mathsf{States}$$

$$\mathsf{srcState} \; : \; \mathsf{Trans} \longrightarrow \mathsf{States}$$

On définit les fonctions actions et triggers qui associe respectivement à une transtion sa séquence d'actions, son déclencheur et sa garde :

$$\mathsf{actions} : \mathsf{Trans} \longrightarrow \mathsf{SActions}$$

$$\mathsf{triggers} : \mathsf{Trans} \longrightarrow \mathsf{Triggers}$$

$$\mathsf{guards} : \mathsf{Trans} \longrightarrow \mathsf{Guards}$$

Description formelle des déclencheurs Un déclencheur est décrit par un nom, un type et un évènement. Un déclencheur $trigger \in \mathsf{Triggers}$ est décrit de façon formelle par :

$$trigger = (ntrigger, ttype, event)$$

où :

- $ntrigger \in \mathsf{NTriggers}$, le nom du déclencheur
- $ttype \in \mathsf{TType}$ le type de déclencheur, et

$$ttype \in \mathsf{TType} = \{\mathsf{EMPTY}, \mathsf{MCALL}, \mathsf{MRETURN}\}$$

- $event \in \mathsf{MId}$ est l'évènement qui déclenche la transition.

Les fonctions ttype et event associe respectivement à un déclencheur son type et l'évènement qui le déclenche :

$$\mathsf{ttype} : \mathsf{Triggers} \longrightarrow \mathsf{TType}$$
$$\mathsf{event} : \mathsf{Triggers} \longrightarrow \mathsf{MId}$$

On distingue trois types de déclencheurs :

- EMPTY : déclencheur vide, la transition est automatiquement déclenchée et aucune information n'est envoyée. L'évènement correspondant *event* ne prend donc aucune valeur et $\mathsf{event}(trigger) = \star$
- MCALL : appel de méthode sur un objet. L'évènement déclencheur correspond à l'invocation d'une instance de méthode $mid \in \mathsf{MId}$
- MRETURN : retour de méthode, après l'exécution d'une méthode d'un objet. L'évènement déclencheur correspond au retour d'exécution d'une instance de méthode $mid \in \mathsf{MId}$

Les différents types de déclencheurs possibles sont représentés dans le tableau ci-dessous. Pour un déclencheur vide (automatique), la notation $\star$ indique qu'aucun évènement ne conditionne la transition qui s'exécute automatiquement.

Tableau 4.1 Description des déclencheurs des transitions

Type de déclencheurs	**formel**	**Informel**
EMPTY	$trigger = (ntrigger, \mathsf{EMPTY}, \star)$	déclencheur vide
MCALL	$trigger = (ntrigger, \mathsf{MCALL}, mid)$	appel de méthode
MRETURN	$trigger = (ntrigger, \mathsf{MRETURN}, mid)$	retour de méthode

Description formelle des gardes Une garde est décrite soit par une expression booléenne (*vrai* ou *fausse*) ou de façon récursive par une combinaison de gardes. Les gardes peuvent être combinées entre elles grâce à des opérateurs arithmétiques. Une garde $g \in \mathsf{Guards}$ est une expression booléenne et s'exprime de façon récursive par :

$$g(\in \mathsf{Guards}) \quad ::= \quad true \mid false \mid x\ cmp\ y \mid g_1 \&\& g_2 \mid g_1 || g_2 \mid !g$$

où :
- $true$: la garde est évaluée toujours vraie.
- $false$: la garde est évaluée toujours fausse.
- $x\ cmp\ y$: $x, y \in \mathsf{Expressions}$ et cmp est un opérateur de comparaison $cmp \in \mathsf{COp}$
- $g_1 \&\& g_2$: $g_1, g_2 \in \mathsf{Guards}$ sont des gardes, && est l'opérateur ET booléen.
- $g_1 || g_2$ $g_1, g_2 \in \mathsf{Guards}$ sont des gardes, || est l'opérateur OU booléen.
- $!g$: $g \in \mathsf{Guards}$ est une garde, ! est la négation de la garde.

Une garde est évaluée à vrai ou faux. Si elle est vraie, la transition est exécutée. Si elle est fausse, la transition n'est pas exécutée. Le tableau 4.2 décrit la sémantique d'une garde $g \in \mathsf{Guards}$ à l'aide de la fonction value qui associe à une garde une valeur booléenne.

$$\mathsf{value}^{\star\star} \ : \ \mathsf{Guards} \longrightarrow \mathsf{Bool}$$

Si la garde est évaluée à *vrai*, le résultat retourné est *true*. Si la garde est évaluée à *faux*, le résultat retourné est *false*.

Les différentes évaluations possibles d'une garde sont décrites dans le tableau 4.2. Si la garde est égale à $true$ ou $false$, sa valeur est une constante et est égale à $true \in \mathsf{Bool}$ ou $false \in \mathsf{Bool}$. Si la garde est une comparaison de deux expressions x et y, sa valeur correpsond à la comparaison des valeurs arithmétiques des expressions x et y. Si la garde est une combinaison de deux gardes, sa valeur correspond à la combinaison des valeurs des deux gardes.

Tableau 4.2 Table des évaluations des gardes

Syntaxe, g	**Évaluation**, $\mathsf{value}^{\star\star}(g)$
$true$	$\mathsf{value}^{\star\star}(g) = true$
$false$	$\mathsf{value}^{\star\star}(g) = false$
$x\ cmp\ y$	$\mathsf{value}^{\star\star}(g) = \mathsf{value}^{\star\star}(\mathsf{value}^{\star}(x)\ cmp\ \mathsf{value}^{\star}(y)) = \begin{cases} true, \text{ si } \mathsf{value}^{\star}(x)\ cmp\ \mathsf{value}^{\star}(y) \\ false \text{ sinon} \end{cases}$
$g_1 \&\& g_2$	$\mathsf{value}^{\star\star}(g) = \mathsf{value}(g_1) \&\& \mathsf{value}(g_2)$
$g_1 \vert\vert g_2$	$\mathsf{value}(g) = \mathsf{value}^{\star\star}(g_1) \vert\vert \mathsf{value}^{\star\star}(g_2)$
$!g$	$\mathsf{value}(g) = \begin{cases} true, \text{ si } \mathsf{value}^{\star\star}(g) = false \\ false \text{ sinon} \end{cases}$

De plus, on définit la fonction $\mathsf{label}^{\star\star}$ qui associe à une garde une étiquette de sécurité,

$$\mathsf{label}^{\star\star} : \mathsf{Guards} \longrightarrow \mathsf{Label}$$

$$\mathsf{label}^{\star\star}(g) = \begin{cases} L_{\perp}, & \text{si } g = true \vee g = false \\ \mathsf{label}^{\star}(x), & \text{si } g = x\ cmp\ y \text{ et } \mathsf{label}^{\star}(x) \sqsubseteq \mathsf{label}^{\star}(y) \\ \mathsf{label}^{\star}(y), & \text{si } g = x\ cmp\ y \text{ et } \mathsf{label}^{\star}(y) \sqsubseteq \mathsf{label}^{\star}(x) \\ \mathsf{label}^{\star\star}(g_1), & \text{si } g = g_1 \&\& g_2 \text{ ou } g = g_1 || g_2 \text{ et } \mathsf{label}^{\star}(g_1) \sqsubseteq \mathsf{label}^{\star}(g_2) \\ \mathsf{label}^{\star\star}(g_2), & \text{si } g = g_1 \&\& g_2 \text{ ou } g = g_1 || g_2 \text{ et } \mathsf{label}^{\star}(g_2) \sqsubseteq \mathsf{label}^{\star}(g_1) \\ \mathsf{label}^{\star\star}(g_k) & \text{si } g = !g_k \end{cases}$$

Si la garde est égale aux valeurs $true$ ou $false$, son étiquette de sécurité correspond au niveau de sécurité minimal qui existe dans le treillis $(\mathsf{Label}, \sqsubseteq)$ (La définition du treills est donné au chapitre 3 - Section 3.4). Si la garde est une comparaison de deux expressions x et y, son étiquette correspond à la jointure des étiquettes des deux expressions. Si la garde est une combinaison de deux gardes, son étiquette correspond à la jointure des étiquettes des deux gardes.

Description formelle des actions Une action est décrite par un nom, un type, l'objet qui envoie le message, l'objet qui reçoit le message et des paramètres d'exécution de l'action. On distingue trois type d'actions : les actions d'affectation, les actions d'appel de méthodes et les actions de retour de méthode. Une action $action \in \mathsf{Actions}$ peut être décrite de façon formelle par :

$$action = (na, atype, sndr, rcvr, aParams)$$

où :

- $na \in \mathsf{NActions}$: le nom d'action qui est effectuée
- $atype \in \mathsf{AType}$: le type d'action qui est effectuée

- $sndr \in \mathsf{OId}$ l'objet qui envoit le message, il appartient à l'ensemble des objets
- $rcvr \in \mathsf{OId}$ l'objet qui reçoit le message, il appartient à l'ensemble des objets
- $aParams \in \mathsf{AParams}$ est une liste de paramètres utilisés lors de l'exécution de l'action et varient suivant le type d'action qui est invoqué.

Type d'actions $atype \in \mathsf{AType}$ est le type d'action qui est effectué. AType est l'ensemble des types de messages

$$\mathsf{AType} = \{\mathsf{ASSIGN}, \mathsf{MCALL}, \mathsf{MRETURN}\}$$

On distingue trois types d'actions :
- ASSIGN : affectation d'une valeur à un attribut d'un objet.
- MCALL : appel d'une méthode d'un objet, et affectation de valeurs aux paramètres de la méthode.
- MRETURN : retour d'exécution d'une méthode d'un objet, et affectation de valeurs aux paramètres de la méthode et au paramètres de retour.

La fonction atype associe à une action son type et :

$$\mathsf{atype} : \mathsf{Actions} \longrightarrow \mathsf{AType}$$

La fonction receiver associe à une action, l'objet qui envoie les données :

$$\mathsf{receiver} : \mathsf{Actions} \longrightarrow \mathsf{AType}$$

La fonction sender associe à une action, l'objet qui reçoit les données :

$$\mathsf{sender} : \mathsf{Actions} \longrightarrow \mathsf{AType}$$

La fonction aParams associe à une action, ses paramètres :

$$\mathsf{aParams} : \mathsf{Actions} \longrightarrow \mathsf{AParams}$$

Type de paramètres d'actions Les paramètres $aParams$ utilisés varient également en fonction du type d'action. Ils déterminent les variables, attributs ou paramètres qui seront utilisées par l'action. Ils déterminent également les classes et objets impliquées lorsque l'action s'exécute. L'ensemble des paramètres d'action est défini par :

$$aParams(\in \mathsf{AParams}) ::= (d,e)|(m,(d,e))|(C,(a_1,e_1),\ldots,(a_n,e_n))|(m,(p_1,e_1),\ldots,(p_n,e_n))$$

Les paramètes des actions sont les suivants : une affectation, un appel de méthode ou un retour de méthode.

Paramètre d'affectation Si $\mathsf{atype}(\text{action}) = \mathsf{ASSIGN}$:

$$action = (\ na,\ \mathsf{ASSIGN},\ o_i,\ o_i,\ (\ a,\ e\ )\ )$$

L'objet qui transmet le message est le même que celui qui le reçoit ($\mathsf{receiver}(action) = \mathsf{sender}(action)$), $\mathsf{atype}(action) = \mathsf{ASSIGN}$ et $\mathsf{aParams}(action) = (\ a,\ e\ )$ est une liste constituée de a et e tel que a est l'élément auquel on affecte l'élément e
et $a \in \mathsf{Data}$, e est une expression ($e \in \mathsf{Expressions}$).

Message d'appel de méthode sur un objet Si $\mathsf{atype}(\text{action}) = \mathsf{MCALL}$:

$$action = (\ na,\ \mathsf{MCALL},\ o_i,\ o_j,\ (\ m,\ (p_1, e_1), \ldots, (p_n, e_n)\ )\ )$$

Ce type de message est transmis d'un objet o_i vers un objet o_j ($\mathsf{receiver}(action) = o_j$ et $\mathsf{sender}(action) = o_i$) et invoque l'exécution d'une méthode de o_j, $\mathsf{atype}(action) = \mathsf{MCALL}$ et $\mathsf{aParams}(action) = (\ m,\ (p_1, e_1), \ldots, (p_n, e_n)\ )$ La méthode $m \in \mathsf{methods}^\star(o_j)$ de l'objet o_j est invoquée par l'objet o_i et est exécutée avec les paramètres spécifiques $e_1 \ldots e_n$ et $e_1\ \ldots\ e_n \in \mathsf{Expressions}$.

Message de retour de méthode d'objet Si $\mathsf{atype}(\text{action}) = \mathsf{MRETURN}$:

$$action = (\ na,\ \mathsf{MRETURN},\ o_i,\ o_j,\ (\ m,\ (x, e)\ )\ )$$

Ce type de message est transmis d'un objet o_i vers un objet o_j ($\mathsf{receiver}(msg) = o_j$ et $\mathsf{sender}(msg) = o_i$) et invoque l'exécution d'une méthode de o_j, $\mathsf{atype}(action) = \mathsf{MRETURN}$ et $\mathsf{aParams}(action) = (m,\ (x, e))$ avec la méthode $m \in \mathsf{methods}^\star(o_j)$ de l'objet o_j est invoquée par l'objet o_i et le résultat $e \in \mathsf{Expressions}$ de la méthode est affectée à la donnée $x \in \mathsf{Data}$.

Description formelle d'une séquence d'actions Une séquence d'actions $saction \in$ $\mathsf{SActions}$ peut être décrite de façon formelle par :

$$a(\in \mathsf{Actions})$$
$$saction(\in \mathsf{SActions}) ::= \langle\rangle \mid a \mid a_1 \oplus \ldots \oplus a_n \mid saction_1 \oplus \ldots \oplus saction_n$$

où :
- $a \in \mathsf{Actions}$: correspond à une séquence constituée d'une seule action,
- $\langle\rangle$: correspond à une séquence vide,
- $a_1 \oplus \ldots \oplus a_n$: correspond à une suite d'actions qui constituent une séquence, avec $a_1 \ldots a_n \in \mathsf{Actions}$
- $saction_1 \oplus \ldots \oplus saction_n$: correspond à une suite de séquences d'actions qui s'exécutent une à la suite de l'autre $saction_1 \ldots saction_n \in \mathsf{SActions}$
- $\oplus$ est l'opérateur de concaténation des séquences d'actions et des actions et est défini par :

$$\oplus \ : \ (\mathsf{SActions} \cup \mathsf{Actions}) \times (\mathsf{SActions} \cup \mathsf{Actions}) \longrightarrow \mathsf{SActions}$$

Description formelle d'un diagramme d'états Le diagramme d'états $sc \in \mathsf{StateCharts}$ de l'objet $o \in \mathsf{OId}$ peut-être défini de façon formelle par :

$$sc = (\ nState,\ o,\ SStates,\ STrans\ )$$

où
- $nState \in \mathsf{NStateCharts}$, le nom du diagramme d'états,
- $o \in \mathsf{OId}$, l'objet dont le comportement est modélisé par le diagramme d'état sc,
- $SStates \subseteq \mathsf{States}$, l'ensembles des états contenus dans le diagramme d'états
- $STrans \subseteq \mathsf{Trans}$, l'ensembles des transitions entre états du diagramme d'états, et $\forall t \in STrans$, $\mathsf{srcState}(t) \in SState$, $\mathsf{tgtState}(t) \in SState$

On définit également la fonction $\mathsf{stateChartStates}$ qui associe à un diagramme d'états, l'ensemble de tous les états qu'il contient.

$$\mathsf{stateChartStates} \ : \ \mathsf{StateCharts} \longrightarrow \mathcal{P}(\mathsf{States})$$

et la fonction $\mathsf{stateChartTrans}$ qui associe à un diagramme d'états, l'ensemble de toutes les transitions qu'il contient.

$$\mathsf{stateChartTrans} \ : \ \mathsf{StateCharts} \longrightarrow \mathcal{P}(\mathsf{Trans})$$

4.2.2 Sémantique des diagrammes UML

On décrit dans cette sous-section la sémantique des diagrammes UML. La première partie (partie 4.2.2.1) décrit la sémantique des diagrammes de classes et d'objets et la deuxième partie (partie 4.2.2.2) décrit la sémantique des diagrammes d'états.

4.2.2.1 Sémantique des diagrammes de classes et d'objets

La sémantique des diagrammes de classes et d'objets décrit principalement la valeur des attributs (de classe et d'objets) et la valeur de leurs étiquettes de sécurité.
La sémantique des expressions arithmétiques permet de déterminer la valeur des expressions arithmétiques et le niveau de sécurité des expressions arithmétiques. La fonction $\mathsf{value}^\star$ associe à une expression sa valeur, et $\mathsf{label}^\star$ associe à une expression son étiquette de sécurité.

$$\begin{aligned}\mathsf{value}^\star &: \mathsf{Expressions} \longrightarrow \mathsf{Val}\\ \mathsf{label}^\star &: \mathsf{Expressions} \longrightarrow \mathsf{Label}\end{aligned}$$

Les expressions sont évaluées de la façon suivante :

$$\mathsf{value}^\star(e) = \begin{cases} v, & \text{si } e = v \text{ et } v \in \mathsf{Val}\\ \mathsf{value}(d), & \text{si } e = d \text{ et } d \in \mathsf{Data}\\ \mathsf{value}^\star(e_1)\ \mathsf{bop}\ \mathsf{value}^\star(e_2), & \text{si } e = e_1\ \mathsf{bop}\ e_2 \text{ et } e_1, e_2 \in \mathsf{Expressions},\ \mathsf{bop} \in \mathsf{COp} \cup \mathsf{BOp}\\ \mathsf{bop}\ \mathsf{value}^\star(e_k) & \text{si } e = \mathsf{bop}\ e_k, \text{ et } e_k \in \mathsf{Expressions},\ \mathsf{bop} \in \mathsf{UOp}\end{cases}$$

$$\mathsf{label}^\star(e) = \begin{cases} L_\perp, & \text{si } e = v \text{ et } v \in \mathsf{Val}\\ \mathsf{label}(d), & \text{si } e = d \text{ et } d \in \mathsf{Data}\\ \mathsf{label}^\star(e_1) \sqcup \mathsf{label}^\star(e_2), & \text{si } e = e_1\ \mathsf{bop}\ e_2 \text{ et } e_1, e_2 \in \mathsf{Expressions},\ op \in \mathsf{COp} \cup \mathsf{BOp}\\ \mathsf{label}^\star(e_k), & si\ e = \mathsf{bop} e_k, \text{ et } e_k \in \mathsf{Expressions},\ op \in \mathsf{UOp}\end{cases}$$

4.2.2.2 Sémantique des diagrammes d'états

La sémantique des diagrammes d'états décrit la valeur des étiquettes de sécurité sur les gardes des transitions. Elle décrit également les configurations du système et les actions lorsqu'une transision s'exécute.

Sémantique des diagrammes d'états Dans cette partie, la sémantique dynamique des diagrammes d'états est donnée. Les effets des transitions entre les états d'un diagramme de transition sont interprétées. Cette interprétation correspond aux états (possiblement) activés et aux séquences d'actions qui s'exécutent lorsqu'une transition survient. On utilise la notation suivante pour indiquer qu'une transition survient entre l'état s et l'état s', s est

l'état source et s' est l'état destination, avec $\mathsf{srcState}(t) = s$ et $\mathsf{tgtState}(t) = s'$

$$s \xrightarrow{t} s'$$

L'état s' est explicitement activé et d'autres états peuvent être implicitement activés. on décrit dans le prochain sous-paragraphe la fonction d'évaluation des états actifs qui décrit l'ensemble des états actifs d'un diagramme d'états.

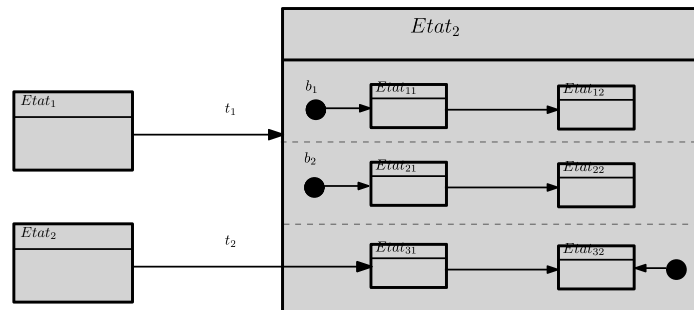

Figure 4.10 Transition vers un état AND

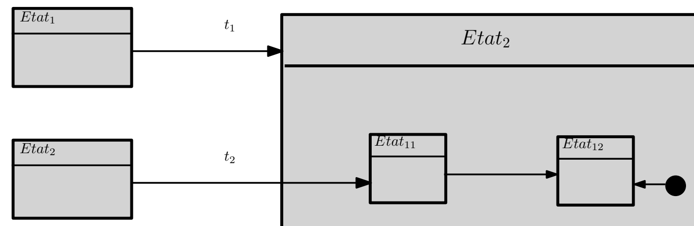

Figure 4.11 Transition vers un état OR

États activés On définit la fonction d'évaluation des états actifs par la fonction ϕ tel que :

$$\phi \in \Phi = \mathsf{StateCharts} \longrightarrow \mathcal{P}(\mathsf{States})$$

Cet ensemble décrit l'ensemble des états actifs. Il correspond aux mises à jour des états actifs dans un diagramme d'états lorsqu'une transition survient entre deux états. Une transition $t \in \mathsf{Trans}$ de l'état s avec l'ensemble des états actifs ϕ vers l'état s' avec l'ensemble d'état actifs ϕ' est notée :

$$\langle\, s,\ \phi \,\rangle \xrightarrow{t} \langle\, s',\ \phi' \,\rangle$$

On définit la fonction $\mathsf{activates}$ qui associe à un ensemble d'états actifs et une transition un nouvel ensemble d'états actifs. Cette fonction permet d'évaluer le nouvel ensemble d'états

actifs ϕ'.

$$\mathsf{activates} : \Phi \times \mathsf{Trans} \longrightarrow \Phi$$

L'évaluation de la fonction activates dépend de l'état de destination de la transition et est donné par :

1. si $\mathsf{stype}(s') = \mathsf{END}$

$$\mathsf{activates}(\phi, t) = \phi'(sc) = \begin{cases} \phi(sc), & \text{si } s = s' \\ \{\} & \text{sinon} \end{cases}$$

   Si une transition survient vers un état final tous les états actifs sont désactivés. Un état de fin END est un état spécial qui permet de désactiver un état composite et tous les états simples qui le constituent. Il correspond à l'état final dans lequel se trouve l'objet et tous les états actifs de l'objets sont désactivés.

2. si $\mathsf{stype}(s') = \mathsf{SIMPLE}$

$$\mathsf{activates}(\phi, t) = \phi'(sc) = \begin{cases} \phi(sc), & \text{si } s = s' \\ \{s'\} \cup \mathsf{superStates}(s') \cup (\bigcup_{s'' \in \mathsf{superStates}(s')} \mathsf{defaultStates}(s'')) & \text{sinon} \end{cases}$$

   Une transition vers un état simple entraine l'activation de tous ses états parents et de leurs états par défaut. Un état SIMPLE est un état simple qui ne possède pas de structure interne (pas d'états imbriqués) Ces états peuvent se retrouver à l'intérieur d'autres états (états composites).

3. si $\mathsf{stype}(s') = \mathsf{AND}$

$$\mathsf{activates}(\phi, t) = \phi'(sc) = \begin{cases} \phi(sc), & \text{si } s = s' \\ \{s'\} \cup \mathcal{S}' \cup \mathsf{superStates}(s') \cup (\bigcup_{s'' \in \mathsf{superStates}(s')} \mathsf{defaultStates}(s'')) & \text{sinon} \end{cases}$$

   où :

$$\mathcal{S}' = \{u : u \in \mathsf{subStates}(s'), \wedge (\exists tr \in \mathsf{Trans}, \mathsf{stype}(srcState(tr)) = \mathsf{BEGIN} \wedge \mathsf{tgtState}(tr) = u)\}$$

   Une transition vers un état composite AND (Figure 4.10) entraine une activation de tous ses états par défaut contenus dans chaque région qui le constitue. Un état AND est un état qui contient des états imbriqués. Un état AND est constitué d'une ou plusieurs régions. Chaque région possède un ensemble d'états. L'objet peut se trouver dans deux sous-états à un instant donné. Ces états s'exécutent en parallèle. Un état OR possède une seule région. Chaque région r d'un état AND possède un seul état de début BEGIN, qui correspond l'état de départ d'une transition vers un sous-état de l'état OR. Ce sous-état correspond à l'état initial par défaut de l'état OR. Une transition vers l'état OR correspond à une transition vers son état par défaut. Une transition vers un état AND

correspond à une transition vers les états par défaut de chacune de ses régions. Si une transition explicite vers un sous-état d'une région (d'un état AND) survient, tous les états par défaut des autres régions sont implictement activés.

4. si $\mathsf{stype}(s') = \mathsf{OR}$

$$\mathsf{activates}(\phi, t) = \phi'(sc) = \begin{cases} \phi(sc), & \text{si } s = s' \\ \{s'\} \cup \mathcal{S}' \cup \mathsf{superStates}(s') \cup (\bigcup_{s'' \in \mathsf{superStates}(s')} \mathsf{defaultStates}(s'')) & \text{sinon} \end{cases}$$

où :

$$\mathcal{S}' = \{u : u \in \mathsf{subStates}(s'), \wedge(\exists tr \in \mathsf{Trans}, \mathsf{stype}(\mathsf{srcState}(tr)) = \mathsf{BEGIN} \wedge tgtState(tr) = u)\}$$

Une transition vers un état composite OR (Figure 4.11) entraine une activation de tous ses états par défaut. Un état OR est un état qui est constitué d'états imbriqués. L'objet ne peut se trouver que dans un seul des sous-états qui le composent. un état OR peut avoir un état initial BEGIN, qui correspond l'état de départ d'une transition vers un sous-état de l'état OR. Ce sous-état correspond à l'état initial par défaut de l'état OR. Une transition vers l'état OR correspond à une transition vers son état par défaut.

Le tableau 4.3 montre la sémantique des transitions entre les états d'un diagramme d'états.

Tableau 4.3 Règle de transition entre deux états

$$\frac{s \xrightarrow{t} s' \quad t \in \mathsf{Trans}, \quad s, s' \in \mathsf{States}}{\langle\, s,\ \phi \,\rangle \xrightarrow{t} \langle\, s',\ \phi' = \mathsf{activates}(\phi, t) \,\rangle}$$

Ensemble des locations d'un diagramme d'états On définit dans cette sous-section l'ensemble des locations d'un diagramme d'états. Cet ensemble correspond à l'ensemble des états accessibles pour l'ensemble des diagrammes d'états-transitions. Il permet d'obtenir toutes les configurations possibles lorsqu'une transition survient et est défini par :

$$\mathsf{Loc} = \{(s_0, s_1, \ldots, s_n) \ : \ s_0 \in \mathcal{P}(\phi(sc_0)),\ s_1 \in \mathcal{P}(\phi(sc_1)) \ldots,\ s_n \in \mathcal{P}(\phi(sc_n))\}$$

où : $sc_0, sc_1, \ldots, sc_n \in \mathsf{StateCharts}$. On définit la fonction $\mathsf{actionSet}$ qui associe à une séquence d'actions, un ensemble d'actions qui sont exécutées par cette séquence :

$$\mathsf{actionSet} \ : \ \mathsf{SActions} \longrightarrow \mathcal{P}(\mathsf{Actions})$$

On définit également la fonction $\mathsf{sequenceActions}$ qui associe à une location et une transition la séquence d'action qui est exécutée :

$$\mathsf{sequenceActions} \; : \; \mathsf{Loc} \times \mathsf{Trans} \longrightarrow \mathsf{SActions}$$

De façon générale, l'évaluation de la fonction $\mathsf{sequenceActions}$ dépend des états de départ et destination, de l'ensemble des objets actifs et est donnée par :

$$\mathsf{sequenceActions}(loc, t) = \mathsf{exit}(s) \oplus \mathcal{A} \oplus \mathsf{actions}(t) \oplus \mathsf{entry}(s') \oplus \mathcal{B} \quad \text{sinon}$$

où :
- $\mathsf{exit}(s)$: séquence d'action lorsqu'on quitte l'état de départ,
- $\mathcal{A}$: séquence d'actions pour tous les états composites à l'intérieur desquels l'état de départ peut se trouver
- $\mathsf{actions}(t)$: séquence d'action exécutée par la transition,
- $\mathsf{entry}(s')$: séquence d'action lorsqu'on accède à l'état destination
- $\mathcal{B}$: séquence d'action pour tous les états composites à l'intérieur desquels les états destination peuvent se trouver.

1. si $\mathsf{superStates}(s') = \{\}$, $\mathsf{superStates}(s) = \{\}$ et $\mathsf{stype}(s') = \mathsf{SIMPLE}$, $\mathsf{stype}(s) = \mathsf{SIMPLE}$

$$\mathcal{A} = \langle\rangle$$
$$\mathcal{B} = \langle\rangle$$

2. si $\mathsf{superStates}(s') = \{\}$, $\mathsf{superStates}(s) = \{\}$ et $\mathsf{stype}(s') = \mathsf{END}$, $\mathsf{stype}(s) = \mathsf{SIMPLE}$

$$\mathcal{A} = \langle\rangle$$
$$\mathcal{B} = \langle\rangle$$

3. si $\mathsf{superStates}(s') = \{\}$, $\mathsf{superStates}(s) = \{\}$ et $\mathsf{stype}(s') = \mathsf{END}$, $\mathsf{stype}(s) = \mathsf{BEGIN}$

$$\mathcal{A} = \langle\rangle$$
$$\mathcal{B} = \langle\rangle$$

4. si $\mathsf{superStates}(s') = \{\}$, $\mathsf{superStates}(s) = \{\}$ et $\mathsf{stype}(s') = \mathsf{AND}$ ou $\mathsf{stype}(s') = \mathsf{OR}$, $\mathsf{stype}(s) = \mathsf{SIMPLE}$

$$\mathcal{A} = \langle\rangle$$
$$\mathcal{B} = \mathsf{permute}(\{\mathsf{entry}(s^\star),\; s^\star \in \phi'\})$$

5. si $\mathsf{superStates}(s') \neq \{\}$, $\mathsf{superStates}(s) = \{\}$ et $\mathsf{stype}(s') = \mathsf{SIMPLE}$, $\mathsf{stype}(s) = \mathsf{SIMPLE}$

$$\mathcal{A} = \langle\rangle$$
$$\mathcal{B} = \mathsf{permute}(\{\mathsf{entry}(s^\star) \mid s^\star \in \phi'\}$$

6. si $\mathsf{superStates}(s') \neq \{\}$, $\mathsf{superStates}(s) = \{\}$ et $\mathsf{stype}(s') = \mathsf{AND}$ ou $\mathsf{stype}(s') = \mathsf{OR}$, $\mathsf{stype}(s) = \mathsf{SIMPLE}$

$$\mathcal{A} = \langle\rangle$$
$$\mathcal{B} = \mathsf{permute}(\{\mathsf{entry}(s^\star) \mid s^\star \in \phi'\}$$

7. si $\mathsf{superStates}(s') = \{\}$, $\mathsf{superStates}(s) \neq \{\}$ et $\mathsf{stype}(s') = \mathsf{SIMPLE}$, $\mathsf{stype}(s) = \mathsf{SIMPLE}$

$$\mathcal{A} = \mathsf{permute}(\{\mathsf{entry}(s^\star) \mid s^\star \in \phi\}$$
$$\mathcal{B} = \langle\rangle$$

8. si $\mathsf{superStates}(s') \neq \{\}$, $\mathsf{superStates}(s) \neq \{\}$

$$\mathcal{A} = \mathsf{permute}(\{\mathsf{entry}(s^\star) \mid s^\star \in \phi\}$$
$$\mathcal{B} = \mathsf{permute}(\{\mathsf{entry}(s^\star) \mid s^\star \in \phi'\}$$

La fonction $\mathsf{permute}$ permet d'obtenir tous les arrangements possibles de séquences dans l'ensemble des séquences pour chaque région de l'état destination.

Exemple Considérons le diagramme de la figure 4.10, le diagramme d'état et les transitions t_1 et t_2, tel que $tgtState(t_1) = s_3$ et $srcState(t_1) = s_1$, et $tgtState(t_2) = s_{31}$ et $srcState(t_2) = s_2$. Les états activés respectivement par les transitions t_1 et t_2 sont donnés par les fonctions ϕ' et ϕ'' :

$$\langle\, s_1,\ \phi \,\rangle \xrightarrow{t_1} \langle\, s_3,\ \phi' \,\rangle$$
$$\langle\, s_2,\ \phi \,\rangle \xrightarrow{t_2} \langle\, s_{31},\ \phi'' \,\rangle$$

où :

$$\phi' = \{s_3, s_{11}, s_{21}, s_{32}\}$$
$$\phi'' = \{s_3, s_{11}, s_{21}, s_{32}\}$$

Considérons le diagramme de la figure 4.11, le diagramme d'état $sc \in \mathsf{StateCharts}$ et les transitions t_1 et t_2, tel que $tgtState(t_1) = s_3$ et $srcState(t_1) = s_1$, et $tgtState(t_2) = s_{11}$ et $srcState(t_2) = s_2$. Les états activés respectivement par les transitions t_1 et t_2 sont donnés par les fonctions ϕ' et ϕ'' :

$$\langle\, s_1,\ \phi \,\rangle \xrightarrow{t_1} \langle\, s_3,\ \phi' \,\rangle$$
$$\langle\, s_2,\ \phi \,\rangle \xrightarrow{t_2} \langle\, s_1,\ \phi'' \,\rangle$$

où :

$$\phi' = \{s_3, s_{12}\}$$
$$\phi'' = \{s_3, s_{11}\}$$

Exemple Considérons le diagramme de la figure 4.11 et la transition
$t_1 = (\ t_1,\ \star,\ \star,\ \oplus act_2 \oplus act_3,\ Etat_1,\ Etat_1)$
avec :

$$act_1 = (\ act_1,\ \mathsf{ASSIGN},\ v_1,\ v_1,\ mparams_1\ )$$
$$act_2 = (\ act_2,\ \mathsf{MCALL},\ v_1,\ v_1,\ mparams_2\ )$$
$$act_3 = (\ act_3,\ \mathsf{MRETURN},\ v_1,\ v_1,\ mparams_3\ )$$

et les paramètres des actions sont respectivement définis par :

$mparams_1 = (\ name,\ edges1\ )$
$mparams_2 = (\ connect_1,\ (\star, \star))$ appel de la méthode *connect* sans aucun paramètres
$mparams_3 = (\ connect_2,\ (r, -1))$ *avec* $r\ =\ \mathsf{return}(connect)$

et les instances de méthodes sont définies par :

$$connect_1 = (\ tr_1,\ connect,\ connect_1\ )$$
$$connect_2 = (\ tr_1,\ connect,\ connect_2\ )$$

CHAPITRE 5

SPÉCIFICATION ET VÉRIFICATION DU FLUX D'INFORMATION

Les chapitres précédents (chapitre 3 et chapitre 4) ont présenté respectivement la spécification formelle du flux d'information et du modèle UML. Le chapitre 5, quant à lui, décrit la méthode de vérification du flux d'information dans les modèles UML. Les contributions de ce chapitre sont les suivantes. La première contribution est la définition d'un système de transition symbolique à états. Contrairement au travail de von der Beeck (2002), le modèle formel n'est pas défini à partir d'automates hiérarchiques (Latella *et al.* (1999a)). Les états se caractérisent par les ensembles des étiquettes de sécurité des données présentes dans l'état et l'ensemble des états UML actifs. La seconde contribution est la vérification du flux d'information pour un système orienté objet en comparant les niveaux de sécurité dans un état avec les niveaux de sécurité dans les états précédents. Un algorithme de vérification a été implanté afin de vérifier le flux d'information pour le système de transition. Le chapitre 5 est constitué de deux sections. La section 5.1 porte sur la description formelle du système de transition d'un système orienté objet. La description formelle du système de transition est déduite directement de la description formelle du système orienté objet du chapitre 3. La sémantique dynamique d'un système orienté objet (décrit avec un diagramme de classe, un diagramme d'objet et des diagrammes d'états) est exprimé sous la forme d'un système de transition symbolique où chaque état est décrit par l'ensemble des états UML actifs et les valeurs des étiquettes de sécurité.
La section 5.2 porte sur la technique d'analyse et de vérification du flux d'information dans un système orienté objet. On réalise une analyse du système orienté objet qui consiste à calculer automatiquement les étiquettes de sécurité des données du système orienté objet. La vérification est réalisée dans chaque état du système. On recherche en particulier des erreurs possibles à l'exécution des transitions des diagrammes d'états. Les erreurs que l'on essaie de détecter sont les violations des contraintes de flux d'information (imposées par les étiquettes de sécurité). Le flux d'information légal est défini à partir du treillis de sécurité défini à la section 3.1.

Le principe de vérification est de comparer les niveaux de sécurité des données entre un état et les niveaux de sécurité dans l'état suivant du système. Pour que le flux d'information soit considéré comme sécuritaire il faudrait que les niveaux de sécuité des données dans l'état suivant soient supérieurs à ceux de l'état précédent.

5.1 Modélisation du système de transition orienté objet

Le système orienté objet est interprété comme un système de transition symbolique à états. Il permet d'exprimer les requis de sécurité grâce à la fonction $\mathsf{labelEffect}$. Le système de transition permettra de simuler l'évolution et les changements d'états du système orienté objet. Les transitions modélisent les actions effectuées par le système orienté objet et les états décrivent les valeurs des étiquettes de sécurité des données du système orienté objet (attributs des objets, paramètres des méthodes, variables) et les états UML activés (Chapitre 4, section 4.2.2.2).

5.1.1 Système de transition orienté objet

La définition formelle d'un système de transition $\mathcal{T}$ d'un système orienté objet est donnée par :

$$\mathcal{T} = (q_0,\ \mathcal{Q},\ \mathcal{R},\ \mathsf{Trans},\ \mathsf{Guards},\ \mathsf{labelEffect})$$

La description des éléments du système de transition sont décrits dans les paragraphes suivants.

Ensemble des états $\mathcal{Q}$

Cet ensemble est noté $\mathcal{Q}$ et $\mathcal{Q} \subseteq \mathsf{Loc} \times \Lambda$. Un état de notre système de transition est caractérisé par une location $l \in \mathsf{Loc}$ à laquelle on associe une fonction d'évaluation qui nous donne l'étiquette de sécurité des données dans l'état courant. De façon formelle, soit $\mathcal{Q}$ l'ensemble des états du système orienté objet, un état quelconque $q \in \mathcal{Q}$ est décrit par :

$$q = (l, \lambda)$$

où :

- $l \in \mathsf{Loc}$, est une location des diagrammes d'états (Les détails de la définition d'une location sont donnés à la section 4.2.1.3).
- $\lambda \in \Lambda$ est une fonction qui donne la valeur des étiquettes de sécurité dans l'état courant.
- q_0 est l'état initial du système et $q_0 \in \mathcal{Q}$

Fonction d'évaluation des étiquettes de sécurité

Soit $q = (l, \lambda)$, l'état courant du système :

$$\lambda \in \Lambda = \mathsf{Data} \longrightarrow \mathcal{P}(\mathsf{Label})$$
$$\lambda(d) = \{\mathsf{label}(d)\}$$

Cet ensemble correspond aux mises à jour des étiquettes de sécurité des données lorsqu'une transition $t \in \mathsf{Trans}$ est exécuté à partir d'une location $l \in \mathsf{Loc}$.

Ensemble de transitions Trans

L'ensemble de transitions permet de passer d'un état à un autre. Il a été défini de façon formelle à la section 4.2.1.3.

Ensemble de gardes Guards

L'ensemble de gardes impose des conditions sur les transitions entre les états. En effet, si une transition est étiquettée avec une garde et que cette dernière est évaluée à vraie, on peut exécuter la transition. Cet ensemble est décrit à la section 4.2.1.3.

Fonction d'évaluation labelEffect

Fonction qui associe à une étiquette de sécurité et une séquence d'actions, une nouvelle étiquette de sécurité.

$$\mathsf{labelEffect} : \mathsf{SActions} \times \Lambda \longrightarrow \Lambda$$
$$\mathsf{labelEffect}(s, \lambda) = \lambda'$$

Relation de transition $\mathcal{R}$

Chaque état correspond à une location à laquelle on associe une évaluation de valeur et une évaluation de valeur des étiquettes de sécurité lorsque des transitions du diagramme d'états surviennent.

$$\mathcal{R} \subseteq \mathcal{Q} \times \mathsf{Guards} \times \mathsf{Trans} \times \mathcal{Q}$$

et $q \xrightarrow{g:t} q'$ signifie que $(q, g, t, q') \in \mathcal{R}$ et la relation de transition est défini par la règle de la figure 5.1, avec $q = (l, \lambda)$ et $q' = (l', \lambda')$. On utilise également la notation suivante pour indiquer un passage de la location l vers la location l' lorsqu'une transition $t \in \mathsf{Trans}$ est executée et une condition est imposée par la garde $g \in \mathsf{Guards}$.

$$\langle\, l,\, \lambda \,\rangle \xrightarrow{g:t} \langle\, l',\, \lambda' \,\rangle$$

$$\frac{l \xrightarrow{g:t} l' \quad \wedge \quad \mathsf{value}(g) = \mathit{true} \wedge \mathsf{guard}(t) = g}{\langle l, \lambda \rangle \xrightarrow{t} \langle l', \lambda' \rangle} \qquad \text{où } \lambda' = \mathsf{labelEffect}(\mathsf{sequenceActions}(l, t), \lambda)$$

Figure 5.1 Description de la règle de transition

La règle de transition de la figure 5.1 traduit les faits suivants : lorsqu'une transition survient et que la garde g est évaluée à *vrai*, on passe d'un état avec des valeurs d'étiquettes

données par la fonction λ à un état suivant avec des valeurs décrites par la fonction λ'.

5.1.2 Descriptions des effets des transitions

La fonction $\mathsf{labelEffect}$ permet d'évaluer les effets d'une transition sur les fonctions d'évaluations (étiquettes des données). Plus précisément, elle associe à une étiquette de sécurité et une séquence d'actions, un ensemble de nouvelles étiquettes de sécurité. Les fonctions d'évaluation sont évaluées en deux parties. On les évalue, premièrement, en fonction des effets des séquences d'actions et deuxièment en fonction des effets du type des actions. La fonction $\mathsf{labelEffect}$ est évaluée pour chacune des actions qui constituent la séquence (en fonction du type de l'action). Le résultat de l'évaluation en fonction de la séquence d'action donne un ensemble d'étiquettes de sécurité qui correspond aux nouvelles valeurs possibles des étiquettes. Ce nouvel ensemble d'étiquettes donne le niveau de sécurité des données dans le nouvel état dans lequel se trouve le système. L'effet des transitions dépend du type de l'action qui est exécutée. De façon formelle, la fonction $\mathsf{labelEffect}$ est définie par :

$$\lambda'(u) = \mathsf{labelEffect}(sa, \lambda) = \begin{cases} \lambda(u), & \text{si } sa = \langle\rangle \\ \lambda''(u), & \text{si } sa = \langle a \rangle \\ \bigcup_{i=0..n} \mathsf{labelEffect}(sa_i, \lambda) \end{cases}$$

$$\lambda'(u) = \mathsf{labelEffect}(sa, \lambda) = \begin{cases} \lambda(u), & \text{si } sa = \langle\rangle \\ \lambda''(u), & \text{si } sa = \langle a \rangle \\ \bigcup_{i=0..n}(\bigcup_{j=0..n} \mathsf{labelEffect}(\langle a_{ij}\rangle, \lambda)), & sa_i = \langle a_{i0}\rangle \oplus \langle a_{i1}\rangle \oplus \ldots \oplus \langle a_{in}\rangle \end{cases}$$

où $sa = \mathsf{sequenceActions}(l, t) = sa_0 \oplus sa_1 \oplus \ldots \oplus sa_n$,
la séquence d'actions qui est générée lorsqu'on exécute la transition $t \in \mathsf{Trans}$ à partir de la location $l \in \mathsf{Loc}$.

Tableau 5.1 Description de la fonction $\mathsf{labelEffect}$

Type d'actions $a \in \mathsf{Actions}$	$\lambda'' = \mathsf{labelEffect}(\langle a \rangle, \lambda)$
si $\mathsf{atype}(a) = \mathsf{ASSIGN}$ $a = (na, \mathsf{ASSIGN}, o_i, o_j, (v, e))$	$\lambda''(u) = \begin{cases} \lambda(u), & \text{si } v \neq u \\ \{\mathsf{label}^\star(e)\} & \text{sinon} \end{cases}$
si $\mathsf{atype}(a) = \mathsf{MCALL}$ $a = (na, \mathsf{MCALL}, o_i, o_j, (m, (p_1, e_1), \ldots, (p_n, e_n)))$	$\lambda''(u) = \begin{cases} \lambda(u), & \text{si } p_i \neq u \\ \{\mathsf{label}^\star(e_i)\} & \text{si } p_i = u \end{cases}$
si $\mathsf{atype}(a) = \mathsf{MRETURN}$ $a = (na, \mathsf{MRETURN}, o_i, o_j, (m, (x, e)))$	$\lambda''(u) = \begin{cases} \lambda(u), & \text{si } x \neq u \\ \{\mathsf{label}^\star(e)\} & \text{sinon} \end{cases}$

Le tableau 5.1 illustre le résultat d'une action a sur λ, en fonction du type de l'action. Pour chaque $d \in dom(\lambda)$, $\lambda(d)$ dénote l'étiquette de sécurité de la donnée d. En fonction des types d'actions exécutées :

– affectation d'une expression à un attribut, la fonction λ donne l'étiquette de sécurité de l'attribut $v \in dom(\lambda)$.
– appel de méthode avec les paramètres formels $p_0, p_1, \ldots, p_n \in dom(\lambda)$ et les paramètres spécifiques $e_0, e_1, \ldots, e_n$, la fonction $\lambda(p_0), \lambda(p_1), \ldots, \lambda(p_n)$ donne les étiquettes de sécurité des paramètres formels.
– retour de méthode avec le paramètre de retour $x \in dom(\lambda)$ la fonction λ donne l'étiquette de sécurité du paramètre de retour de la méthode $x \in dom(\lambda)$.

5.1.3 Exemple

L'exemple qui suit décrit un système orienté objet à partir des diagrammes UMLillustrés dans les figures 5.2, 5.3 et 5.4. Le système de transition de transition est décrit de façon formelle à partir de ces diagrammes UML.

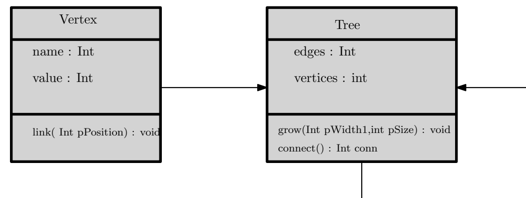

Figure 5.2 Diagramme de classe

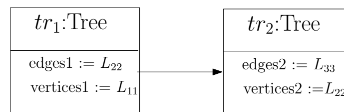

Figure 5.3 Diagramme d'objets

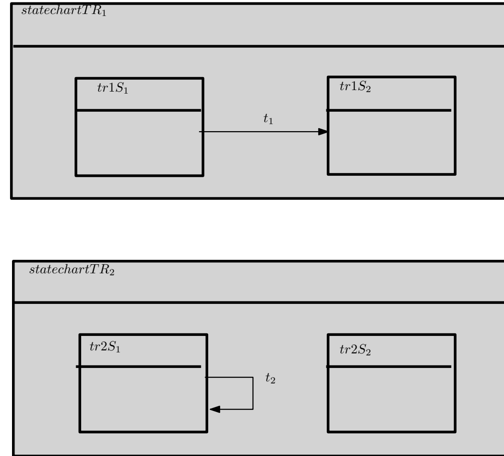

Figure 5.4 Diagramme d'états

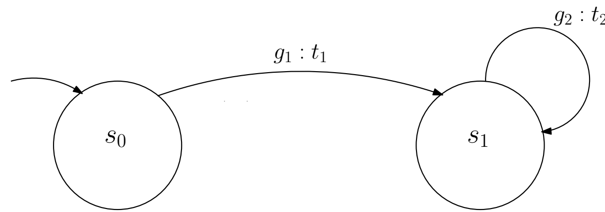

Figure 5.5 Exemple de système de transition

La figure 5.5 illustre un système de transition constitué de deux états : l'état initial S_0 et le second état S_1. Ce système de transition correspond au système orienté objet décrit par le diagramme de classe, d'objets et d'états illustrés dans les figures 5.2, 5.3 et 5.4. Le but de cet exemple est d'illustrer les changements d'étiquettes lorsqu'une transition survient. Ainsi, on décrit uniquement le niveau de sécurité des attributs des objets illustrés à la figure 5.3. Les diagrammes d'états des objets sont illustrés à la figure 5.4. L'ensemble des étiquettes de sécurité est donné par $\mathsf{Label} = \{L_{11}, L_{22}, L_{33}\}$. Le treillis de sécurité est donné par $(\mathsf{Label}, \sqsubseteq)$, avec $L_{11} \sqsubseteq L_{22}$ et $L_{22} \sqsubseteq L_{33}$. Le système de transition illustré à la figure 5.5 est décrit de façon formelle par :

$$\mathcal{T} = (s_0, \{s_0, s_1\}, \mathcal{R}, \{t_1, t_2\}, \{g_1, g_2\}, \mathsf{labelEffect})$$

Le système de transition est constitué de deux états s_0 et s_1. Les transitions sont étiquettées avec les transitions t_1, $t_2 \in \mathsf{Trans}$. La relation $\mathcal{R}$ est donnée par l'ensemble

$$\mathcal{R} = \{(s_0, (g_1, t_1), s_1), (s_1, (g_2, t_2), s_1)\}$$

L'ensemble des transitions est donné par $\mathsf{Trans} = \{t_1, t_2\}$, et

$$t_1 = (t_1, \star, g_1, sa_1, tr1S_1, tr1S_2)$$
$$t_2 = (t_2, \star, g_2, sa_2, tr2S_1, tr2S_2)$$

L'ensemble des gardes est donné par $\mathsf{Guards} = \{g_1, g_2\}$,
L'ensemble des actions est donné par $\mathsf{Actions} = \{a_1, a_2\}$, où :

$$a_1 = (a_1, \mathsf{ASSIGN}, tr_1, tr_1, (vertical1, edges1))$$
$$a_2 = (a_2, \mathsf{ASSIGN}, tr_2, tr_2, (vertical2, edges2))$$

L'ensemble des séquences d'actions est donné par $\mathsf{SActions} = \{sa_1, sa_2\}$,

$$sa_1 = \langle a_1 \rangle$$
$$sa_2 = \langle \ \rangle$$

État S_0 L'état initial $S_0 = (\mathsf{Loc}_0, \lambda_0)$ est représenté par le diagramme d'objets.
$\mathsf{Loc}_0 \ = \ \{(\{tr1S_1\}, \{tr2S_1\})\}$
$dom(\lambda_0) \ = \ \{ \ edges1, edges2, vertices1, vertices2 \ \}$
$\lambda_0(edges1) \ = \{L_{22}\}, \ \lambda_0(vertices1) \ = \{L_{11}\}$
$\lambda_0(edges2) \ = \{L_{33}\}, \ \lambda_0(vertices2) \ = \{L_{44}\}$

État S_1 Dans l'état $s_1 = (\mathsf{Loc}_1, \lambda_1)$, on affecte la valeur de l'attribut $edges1$ à l'attribut $vertices1$ $\mathsf{Loc}_1 \ = \ \{(\{tr1S_2\}, \{tr2S_1\})\}$
$dom(\lambda_0) \ = \ \{edges1, edges2, vertices1, vertices2\}$
$\lambda_1(edges1) \ = \{L_{22}\}, \ \lambda_1(vertices1) \ = \{L_{22}\}$
$\lambda_1(edges2) \ = \{L_{33}\}, \ \lambda_1(vertices2) \ = \{L_{44}\}$ La fonction d'évaluation des transition est donnée par $\mathsf{labelEffect}(sa_1, \lambda_0) = \lambda_1$

5.2 Vérification des propriétés de flux d'information

On présente dans cette section 5.2, la méthode de vérification des systèmes orienté objet. Pour cela on utilise la définition formelle du système de transition orienté objet présenté à la section 5.1 et les définitions du flux d'information établies au chapitre 3. Le principe de vérification est de comparer les niveaux de sécurité des données entre un état et les niveaux de sécurité dans l'état suivant du système. Pour que le flux d'information soit considéré comme sécuritaire il faudrait que les niveaux de sécuité des données dans l'état suivant soient supérieurs à ceux de l'état précédent.

5.2.1 Principe de vérification des propriétés de flux d'information

Dans cette sous-section on décrit comment un système de transition peut-être utilisé pour vérifier la sécurité des systèmes orienté objet. L'évolution d'un système orienté objet est interprété comme une suite d'actions exécutées lorsque des transitions surviennent. Les actions affectent les valeurs des données (variables,attributs des objets et paramètres des méthodes) du système. La syntaxe des données et transitions exécutées par un système orienté objet est donnée en détail au chapitre 4.

La vérification des transitions se base sur les principes suivants qui sont le résultat des études de (Sabelfeld et Myers (2003)) et du modèle d'étiquettes de sécurité défini au chapitre 3 :

- Les expressions et les gardes sont classifiées selon leur niveau de sécurité ; les niveaux de sécurité des expressions et des gardes sont calculés tel que décrit aux sections 4.2.1.3 et 4.2.1.1 et sont représentés par les étiquettes de sécurité.
- pour $L_1, L_2 \in \mathsf{Label}$, si $L_1 \sqsubseteq L_2$, les expressions avec un niveau de sécurité L_1 ne doivent pas être affectées aux expressions avec un niveau de sécurité L_2. Cette règle permet de vérifier les flux explicites. Les flux explicites surviennent lorsque les actions des transitions sont exécutées (affectation, appel de méthode, retour de méthode, création d'objet).
- Les flux implicites sont vérifiés en imposant les restrictions suivantes aux gardes. Une garde avec une étiquette de sécurité L_2 ne doit pas permettre d'affectation à des expressions ayant une étiquette de sécurité L_1, si $L_1 \sqsubseteq L_2$.

De façon spécifique, si on considère un système orienté objet et le système de transition $\mathcal{T}$ qui lui est associé où

$$\mathcal{T} = (q_0,\ \mathcal{Q},\ \mathcal{R},\ \mathsf{Trans},\ \mathsf{Guards},\ \mathsf{labelEffect})$$

Soit $t \in \mathsf{Trans}$ une transition du système orienté objet telle que

$$t = (nt, e, g, \langle a \rangle, tgtState, srcState)$$

où $e \in \mathsf{Triggers}$, $g \in \mathsf{Guards}$, $t \in \mathsf{Trans}$, $\langle a \rangle \in \mathsf{SActions}$ et $tgtState, srcState \in \mathsf{States}$. Les ensembles $\mathsf{Trans}, \mathsf{Triggers}, \mathsf{Guards}, \mathsf{Trans}, \mathsf{Trans}$ et States sont décrits à la section 4. La vérification des transitions dépend du type d'action exécutée et s'effectue dans l'état suivant. On compare le niveau de sécurité d'une donnée $d \in \mathsf{Data}$ dans l'état courant à son niveau de sécurité dans l'état suivant. Ainsi, vérifier la transition $t \in \mathsf{Trans}$ revient à vérifier que pour,

$$\begin{aligned}&((q, g, t, q') \in \mathcal{R},\ d \in dom(\lambda)) \implies \forall s \in \lambda(d), \forall s' \in \lambda'(d),\ s \sqsubseteq s'\\&\text{avec}\ t = (nt, e, g, \langle a \rangle, tgtState, srcState)\end{aligned}$$

La sémantique de la vérification est exprimée sur le modèle formel défini à la section 5.1. Cette sémantique est donnée par la fonction $[\![_]\!]$ et dépend des actions exécutées et du contexte d'exécution de la transition. Le contexte d'exécution de la transition est défini pour une configuration à partir de laquelle cette transition est exécutée. Plus précisément, elle donne une location (ensemble des états UML actifs), un ensemble d'étiquettes de sécurité (donné par la fonction d'évaluation) et la condition d'exécution (relation de transition). Le contexte d'exécution est une fonction définie par :

$$\mathsf{context} : \mathsf{Trans} \rightarrow \mathsf{Loc} \times \Lambda \times \mathcal{R}$$

$$\mathsf{context}(t) \;=\; (l, \lambda, r), \text{ avec } \begin{cases} r = (q, g, t, q') \in \mathcal{R}, \\ t = (nt, e, g, act, tgtState, srcState), \\ q = (l, \lambda),\; q' = (l', \lambda'),\; l \in \mathsf{Loc} \\ \forall act' \in \mathsf{actionSet}(\mathsf{sequenceActions}(l, t)) \end{cases}$$

où

- Loc, Λ, et $\mathcal{R}$ sont respectivement les ensembles des locations, l'ensemble des fonction d'évaluation des étiquettes et la relation de transition. Ils ont les mêmes significations que dans la section 5.1.
- act' correspond à une action qui est effectuée lorsque la transition survient (en tenant compte de la concurrence)

La fonction $[\![_]\!]$ associe à une transition t un état (ensemble d'états UML actifs et une fonction d'évaluation de la sécurité λ) et un élément de la relation de transition. Ainsi, le flux d'information de la transition t est évaluée par rapports à l'ensemble des états UML actifs l, aux valuations de la fonction λ (ensemble de données), aux ensembles des actions exécutées lorsque la transition survient (act') et un élément de la relation de transition. On utilise la notation $[\![t]\!]_{l,\lambda,r}$ pour évaluer le niveau de sécurité de la transition t dans le contexte de $l \in \mathsf{Loc}, \lambda \in \Lambda, r \in \mathcal{R}$ et $[\![t]\!]_{l,\lambda,r}$ retourne la valeur booléenne $true$ si aucun flux d'information illégal n'est détecté, où :

$$\begin{aligned} &(r = (q, g, t, q') \in \mathcal{R},\; d \in dom(\lambda)) \implies \lambda(d) \sqsubseteq \lambda'(d) \\ &\text{avec } t = (nt, e, g, act, tgtState, srcState), \\ &q = (l, \lambda),\; q' = (l', \lambda'),\; l \in \mathsf{Loc} \\ &\forall act' \in \mathsf{actionSet}(\mathsf{sequenceActions}(l, t)),\; \text{et } act \in \mathsf{sequenceActions}(l, t) \end{aligned}$$

Si $\mathsf{atype}(act') = \mathsf{ASSIGN}$:

$$act' = (a, o_i, o_i, \mathsf{ASSIGN}, (a, e)) \text{ et } \mathsf{label}^{\star\star}(g) \sqsubseteq l_a,\ \mathsf{label}^{\star\star}(g) \sqsubseteq l_e$$
$$(l_a \in \lambda(a), l'_a \in \lambda'(a), l_e \in \lambda(e), l'_e \in \lambda'(e), l_a \sqsubseteq l'_a \wedge l_e \sqsubseteq l'_e) \Longrightarrow [\![t]\!]_{l,\lambda,r} = true$$

L'expression $e \in \mathsf{Expressions}$ doit avoir un niveau de sécurité inférieur ou égal à l'attribut $a \in \mathsf{Atts}$.

Si $\mathsf{atype}(act') = \mathsf{MCALL}$:

$$act' = (a, o_i, o_j, \mathsf{MCALL}, (m, (p_1, e_1),\ \ldots\ , (p_n, e_n))) \text{ et}$$
$$\mathsf{label}^{\star\star}(g) \sqsubseteq l_{p_1}, \ldots, \mathsf{label}^{\star\star}(g) \sqsubseteq l_{p_n},$$
$$\mathsf{label}^{\star\star}(g) \sqsubseteq l_{e_1}, \ldots, \mathsf{label}^{\star\star}(g) \sqsubseteq l_{e_n}$$
$$\forall p_i \in \mathsf{Pars}, e_i \in \mathsf{Expressions}\ l_{p_i} \in \lambda(p_i),$$
$$l'_{p_i}\lambda'(p_i),\ l_{e_i} \in \lambda(e_i), l'_{e_i} \in \lambda'(e_i), (l_{p_i} \sqsubseteq l'_{p_i} \wedge (l_{e_i} \sqsubseteq l'_{e_i}) \Longrightarrow [\![t]\!]_{l,\lambda,r} = true$$

Les niveaux de sécurité des paramètres de méthodes doivent-être supérieurs ou égaux à ceux des expressions qui leurs sont affectées.

Si $\mathsf{atype}(act') = \mathsf{MRETURN}$:

$$act' = (a, o_i, o_j, \mathsf{MRETURN}, (m, (x, e))) \text{ et } \mathsf{label}^{\star\star}(g) \sqsubseteq l_x,\ \mathsf{label}^{\star\star}(g) \sqsubseteq l_e$$
$$l_x \in \lambda(x), l'_x \in \lambda'(x),\ l_e \in \lambda(e), l'_e \in \lambda'(e)(l_x \sqsubseteq l'_x \wedge l_e \sqsubseteq l'_e) \Longrightarrow [\![t]\!]_{l,\lambda,r} = true$$

L'expression $e \in \mathsf{Expressions}$ doit avoir un niveau de sécurité inférieur ou égal au paramètre de retour $x \in \mathsf{Pars}$.

5.2.2 Algorithme de vérification des propriétés de flux d'information

Cette section présente l'algorithme pour la vérification des propriétés de flux d'information AVPI. Le pseudo-code de l'algorithme AVPI est une implantation de la vérification du flux d'information (section 5.2.1). L'algorithme AVPI prend en entrée le système de transition $\mathcal{T}$ et la transition $t \in \mathsf{Trans}$ (section 5.1)et doit satisfaire la spécification :

$$func\ AVPI(\mathcal{T}, t)\ \texttt{return True}\ ssi\ [\![t]\!]_{l,\lambda,r} = \texttt{True}$$

L'algorithme 3 permet d'évaluer si le système de transition $\mathcal{T}$ n'admet de flux d'informations illégaux.

Algorithm 3 Calculate $AVPI(\mathcal{T}, t)$

Require: $q \in \mathcal{Q}$, $q' \in \mathcal{Q}$, $t \in \mathsf{Trans}$, $g \in \mathsf{Guards}$, $r \in \mathcal{R}$, $r = (q, q', t, g)$
Require: $q = (l, \lambda)$, $q' = (l', \lambda')$
Ensure: $[\![t]\!]_{q,r} = \texttt{True} \vee [\![t]\!]_{q,r} = \texttt{False}$

```
act ← action(t)
⟦t⟧_{q,r} ← true
if atype(act) == ASSIGN then
  aParams(act) ← (a, e)
  if not ((λ(a) ⊑ λ'(a)) and (λ(e) ⊑ λ'(e))) then
    ⟦t⟧_{q,r} ← false
  end if
end if
if atype(act) == MCALL then
  aParams(act) ← (m, (p_1, e_1) . . . (p_n, e_n))
  for 1 = 1ton do
    if not ((λ(a_i) ⊑ λ'(a_i)) and (λ(e_i) ⊑ λ'(e_i))) then
      ⟦t⟧_{q,r} ← false
    end if
  end for
end if
if atype(act) == MRETURN then
  aParams(act) ← (a, e)
  for 1 = 1ton do
    if not ((λ(a_i) ⊑ λ'(a_i)) and (λ(e_i) ⊑ λ'(e_i))) then
      ⟦t⟧_{q,r} ← false
    end if
  end for
end if
return ⟦t⟧_{q,r}
```

CHAPITRE 6

ÉTUDE DE CAS

Ce chapitre présente une étude de cas qui permet de montrer comment modéliser et vérifier le flux d'information dans un système orienté objet. Il s'agit d'un travail de recherche qui a fait l'objet d'une étude dans la thèse (Oarga (2005)) et a été réalisé en collaboration avec RDDC. Néanmoins, notre étude de cas prend en considération la protection des attributs et des paramètres de méthodes et la propagation de l'information lorsque les diagrammes d'états s'exécutent. De plus, la notion de sécurité multi-niveau est modélisé grâce aux étiquettes de sécurité sur des objets (Section 3), ainsi les contraintes OCL ne sont plus utilisées pour vérifier la sécurité du modèle UML. Les politiques de sécurité imposées par les étiquettes de sécurité permettent de vérifier les contraintes de flux d'information. Il s'agit de spécifier et vérifier le flux d'information pour un sous-ensemble d'objets définis dans notre système (i.e accessibilité des attributs et passages de paramètres lors des appels de méthodes). Le flux d'information est spécifié en utilisant des étiquettes de sécurité. Les objets sur lesquels on veut vérifier le flux d'information sont annotés avec des étiquettes de sécurité (attributs et paramètres des instances de méthodes).

6.1 Introduction

L'objectif principal de cette étude de cas est de modéliser et vérifier les *Cavéats*, aussi connus sous le nom de *Multi-Level Security*. Les *Cavéats* représentent des *catégories de sécurité*, qui s'additionnent aux restrictions de classifications sécurisées existantes des documents. Elles sont importantes pour le ministère national de la défense (Departement of National Defense - DND) et leur système de contrôle (Command and Control Information Systems - C2IS). Certains C2IS travaillent dans le mode *System High Mode*, c'est-à-dire chaque personne qui a acces au système, a la possibilité de voir l'information du système or, tout le monde n'est pas obligé de tout savoir. Pour limiter les droits d'accès, le *Multi-Level Security* est proposé. Les types de cavéats les plus communs consistent à une restriction par nation (i.e. Canadian Eyes Only (CEO), Canadian or US Eyes Only (CANUS)).

Un cavéat est associé à un niveau de sécurité. Par exemple, un document peut-être étiqueté *Secret CEO*, *Top Secret CEO* ou quelque chose de similaire.

6.1.1 Spécification informelle du système des Cavéats

Notre modèle décrit les usagers qui essaient d'accéder aux documents sécurisés. L'objectif de notre modèle est simple. Il s'agit de concevoir et vérifier un système de *Cavéats* simplifié sur la base des spécifications suivantes.

Les usagers. Dans ce système les usagers ont les attributs suivants :

- un niveau de sécurité. Le niveau de sécurité peut prendre les valeurs suivantes *Enhanced*, *Confidential*, *Secret* ou *Top Secret*.
- un pays d'origine. Le pays d'origine peut prendre par exemple les valeurs suivantes *Canada*, *US*, *UK*, *Germany* ou *Egypt*.
- un rang. Le rang peut prendre les valeurs suivantes *General*, *Officer* ou *Civilian*.

Sauf pour le pays d'origine, ces attributs peuvent être modifié dans le temps. Par exemple, un utilisateur particulier peut être mise à niveau de*Civilian* à *Officer* ou de *Secret* à *Top Secret*.

Les documents. Dans ce système les documents ont les attributs suivants :

- un niveau de classification. Le niveau de classification peut prendre les valeurs suivantes *Unclassified*, *Confidential*, *Secret* ou *Top Secret*.
- un caveat. Le caveat peut prendre par exemple les valeurs suivantes *CEO*, *CANUS*, *CANUSUK*, *NATO*, *UN Coalition* ou *USUK*.
- un projet associé. Le projet associé est décrit par le nom du projet. Par exemple, *ProjetA*, *ProjetB* ou *ProjetC*.

Tous les attributs peuvent être modifiés pendant la durée de vie un document, même si certaines modifications ont peu de chance de se produire.

Les tables de contrôle d'accès. Il y a trois tables de contrôle d'accès qui dictent qui peut accéder aux documents : le tableau de classification (tableau 6.1), la table de cavéats (tableau 6.2), et la table de projets (tableau 6.3). Ils sont tous présentés dans les paragraphes qui suivent. Les données sont stockées dans trois tableaux (6.1, 6.2, 6.3). Ils représentent les contraintes imposées au système. La spécification informelle se trouve dans Oarga (2005).

Les tables de contraintes décrivent les contraintes de sécurité qui existent dans le système.

Tableau 6.1 Tableau des classifications

Niv. Classif. /Sec. du doc.	Enhanced	Confidential	Secret	TopSecret
Unclassified	Granted	Granted	Granted	Granted
Confidential	Denied	Granted	Granted	Granted
Secret	Denied	Denied	Granted	Granted
TopSecret	Denied	Denied	Denied	Granted

Tableau 6.2 Tableau des caveats

Caveat/Country	Canada	US	UK	Germany	Egypt
CEO	Granted	Denied	Denied	Denied	Denied
CANUS	Granted	Denied	Denied	Denied	Denied
CANUSUK	Granted	Denied	Denied	Denied	Denied
NATO	Granted	Denied	Denied	Denied	Denied
UN Coalition	Granted	Denied	Denied	Denied	Denied
USUK	Granted	Denied	Denied	Denied	Denied

Tableau 6.3 Tableau des projets

Projet/Country	Canada	US	UK	Germany	Egypt
A				G-O	G-O-C
B	G-O-C	G-O-C			
C	G	G-O	G-O-C	G-O-C	
D	G-O	G-O-C	G-O-C		
E	G-O-C	G-O-C	G		
F			G-O-C		G-O

Legendes
G : General
O : Officer
C : Civilian

Tableau de classification Le tableau 6.1 représente les politiques d'accès et des contraintes strictes imposées par le système. Les colonnes représentent le niveau de classification des usagers et les lignes représentent le niveau de classification du document. Par exemple, un document classé *Secret* ne peut-être consulté que par des usagers qui ont un niveau de visibilité *Secret* ou *TopSecret*.

Tableau des Caveats Le tableau 6.2 représente des contraintes strictes imposées par les cavéats en fonction du pays. Les colonnes représentent des pays et les lignes représentent les cavéats qui sont associés aux pays. Par exemple, un caveat *USUK* n'est accesssible que par les pays *US* et *UK*.

Tableau des Projets Le tableau 6.3 représente une contrainte stricte imposée sur les documents en fonction des projets, du pays et des usagers. Les colonnes représentent des pays et les lignes représentent les projets qui sont associés aux pays. Par exemple, un document faisant partie du projet C peut être lu par des usagers qui ont comme pays d'origine : le *Canada*, *UK US* et *Germany*.

Propriété à vérifier Finalement, la propriété suivante doit être vérifiée pour être valide pour le modèle suivant des cavéats :

> À tout moment, pour chaque instance de classe dans le modèle actuel des cavéats, si une instance de méthode (appel de méthode) est invoquée les étiquettes de sécurité des paramètres formels sont plus restrictives que celles des paramètres spécifiques, et les étiquettes de sécurité des variables auxquelles le paramètre de retour de méthode est affecté. On doit également vérifier que pour chaque instance de classe, si on affecte une variable à un de ses attributs, l'étiquette de sécurité de l'attribut est plus restrictive que celle de la variable. (ou si un de ses attributs est affecté à une variablel'étiquette de sécurité de la variable, est plus restrictive que celle de l'attribut)

6.1.2 Différence entre les études de cas

Notre modèle différera du modèle réalisé par Oarga (2005) sur les points suivants. Premièrement, les propriétés à vérifier sont différentes d'un modèle à un autre. En effet, dans notre étude de cas la propriété à vérifier consiste en une vérification des niveaux de sécurité des données (variables,attributs et paramètres de méthodes). La vérification permet de s'assurer que toutes les affectations aux attributs et les passages de paramètres aux instances de méthodes sont sécuritaires. Dans le modèle de Oarga (2005), par contre, on vérifie si un usager en particulier peut accéder à un document en respectant les contraintes décrites par les tables des cavéats, de classification et de projets. L'étude de cas de Oarga (2005), met l'accent sur la sécurité du système modélisé par les diagrammes UML, tandis que notre modèle met l'accent sur la sécurité des objets proprement dits en garantissant un acces sécuritaire aux attributs des objets et aux paramètres des méthodes. Les niveaux de classifications des usagers et les niveaux de classification des documents ne sont pas représentés par des objets dans les diagrammes d'objets, mais sont des contraintes modélisées par le concept d'étiquettes décentralisées présenté au chapitre 3. Les méthodes de vérification diffère également. En effet, Oarga (2005) spécifiait la sécurité à l'aide d'expressions OCL étendues avec des opérateurs capables d'exprimer les paradigmes orientés objet. Les expressions OCL décrivent les gardes dans les diagrammes d'états. Elles sont fomalisées grâce aux machines à états abstraites

(ASM) et sont évaluées grâce au *model-checking*. De plus, les états décrits dans les systèmes de transition sont différents. Dans notre modèle, les niveaux de sécurité sont directement décrits et évalués dans les états. Dans le modèle de Oarga (2005), les états sont représentés par deux ensembles de fonctions. Le premier donne les valeurs des attributs des objets et le second donne les instances de méthodes actives. La sécurité est évalué au niveau des gardes dans les diagrammes d'états.

6.2 Modélisation UML

Cette section explique comment les entités présentes dans le système sont modélisées à l'aide des diagrammes de classes, du diagramme d'objets et des diagrammes d'états. On utilisera un modèle simplifié du système des cavéats afin de pouvoir modéliser et vérifier le flux d'information. À ce stade de la modélisation, on s'intéresse au comportement dynamique et à la recherche des erreurs de conceptions (détection des flux d'informations illégaux). Ainsi, les modèles UML proposés dans l'étude de Oarga (2005) sont modifiés afin d'inclure la notion d'étiquette de sécurité présentée à la section 3.1.

6.2.1 Étape 1 - Diagramme de classes UML

On prend comme point de départ, le diagramme de classes présenté à la figure 6.1. Il décrit la configuration statique du système. Un usager interagit avec un document en invoquant une méthode du document. Le contrôle passe directement par la classe `Document`. On associe à chaque attribut des classes du diagramme une étiquette de sécurité. Le détail des étiquettes de sécurité est donné à la section 6.2.4.

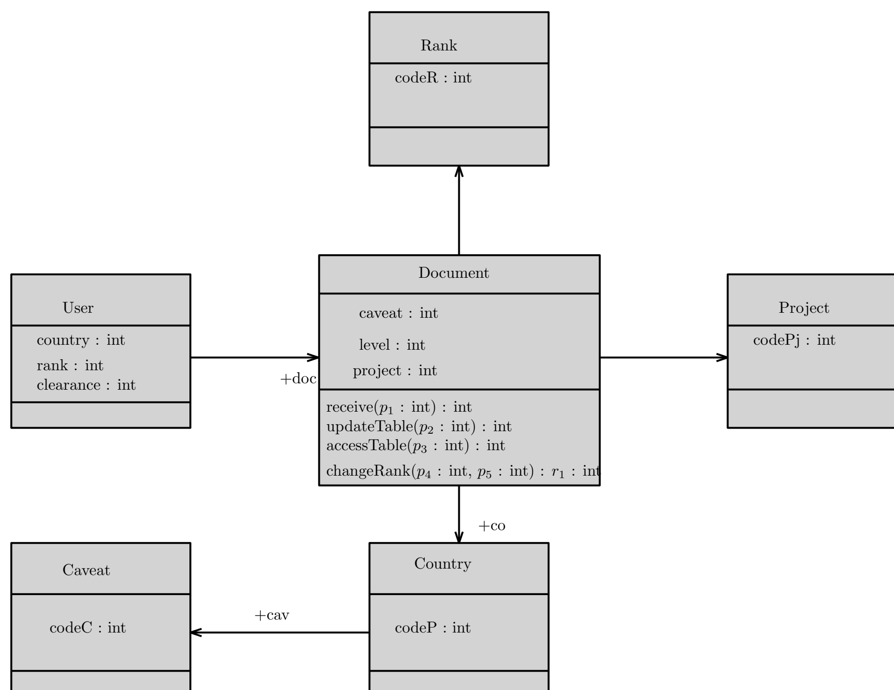

Figure 6.1 Diagramme de classe

Le diagramme de classe comporte les classes suivantes :

- `User`. Cette classe présente trois champs. *clearance*, un entier qui représente le niveau de visibilité d'un usager. *country* qui représente le pays d'origine de l'usager. *rank* qui représente le rang militaire de l'usager.
- `Document`. Cette classe présente trois champs : *level*, un entier qui caractérise le niveau de sécurité des données contenues dans le document. *project*, l'identifiant du projet auquel le document appartient, *caveat* représente le cavéat du document. À ce stade de la modélisation c'est le document qui gère laccès, puisqu'on associe au champ *level* un niveau de sécurité. Il recoit une demande de l'usager à travers un appel de méthode. Cette classe permet la création des documents.
- `Project`. Cette classe présente un champ : *codePj*, entier qui représente le projet.
- `Rank`. Cette classe présente un champ : *codeR*, entier qui représente le rang militaire.
- `Country`. Cette classe présente un champ : *codeP*, entier qui représente le pays.
- `Caveat`. Cette classe présente un champ : *codeC*, entier qui représente le cavéat.

Pour définir l'ensemble des étiquettes de sécurité, il faut prendre en considération le diagramme d'objets de la figure 6.2. L'ensemble des étiquettes donne les objets qui ont des droits en écriture et lecture sur les champs des objets du diagramme d'objets. Les contraintes non formelles sont représentées dans les tableaux 6.1, 6.2 et 6.3.

Tableau 6.4 Visibilité, Pays, Rang

Nom	Code
Enhanced	0
Confidential	1
Secret	2
TopSecret	3

Nom	Code
Canada	0
USA	1
UK	2
Germany	3
Egypt	3

Nom	Code
General	0
Officer	1
Civilian	2

Tableau 6.5 Niveau, Cavéat, Projet

Nom	Code
Unclassified	0
Confidential	1
Secret	2
TopSecret	3

Nom	Code
CEO	0
CANUS	1
CANUSUK	2
NATO	3
UN Coalition	4
USUK	5

Nom	Code
A	0
B	1
C	3
D	4
E	2
F	2

Correspondance entre les types String et Int Compte tenu que le type String n'est pas supporté, les champs des classes de type String seront représentées par des entiers Int. Les tableaux 6.4 et 6.5 présentent les correspondances String-Int pour les atributs utilisés dans l'étude de cas. Le tableau 6.4 donne les correspondances pour la visibilité, le pays et le rang, tandis que le tableau 6.5 donne les correspondances pour le niveau de sécurité, le cavéat et le projet.

6.2.2 Étape 2 - Diagramme d'objets UML

Le diagramme d'objets illusttré à la figure 6.2 permet de modéliser les objets *passifs* (qui encodent les tableaux 6.3, 6.2 et 6.1) et les objets dynamiques (instance de la classe `User` et `Document`) pour lesquels le flux d'information est vérifié.

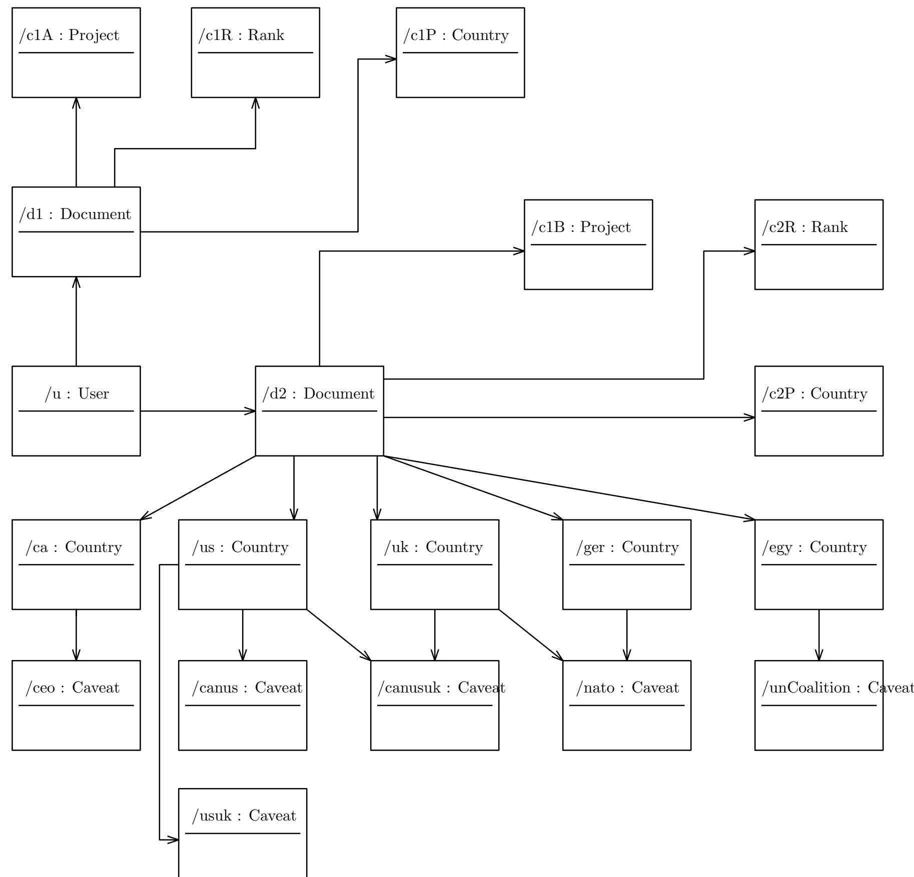

Figure 6.2 Diagramme d'objets (partie 2)

Dans le diagramme d'objets (Figure 6.2), on a initialisé un seul usager et deux documents. L'ensemble des noms d'objets et l'ensemble des objets sont donnés respectivement par :

$$\mathsf{NOId} = \{u, d1, d2, ca, us, uk, ger, egy, ceo, canus, usuk, canusuk, nato, unCoalition\}$$
$$\cup\{c2P, c2R, c1A, c1R, c1P, c1B\}$$
$$\mathsf{OId} = \{u, d1, d2, ca, us, uk, ger, egy, ceo, canus, usuk, canusuk, nato, unCoalition\}$$
$$\cup\{c2P, c2R, c1A, c1R, c1P, c1B\}$$

Tous les champs des objets ont été initialisés avec des valeurs.

6.2.3 Étape 3 - Diagramme d'états

On modélise dans cette sous-section les diagrammes d'états des objets suivants : *u*, *d1* et *d2* (usager et deux documents).

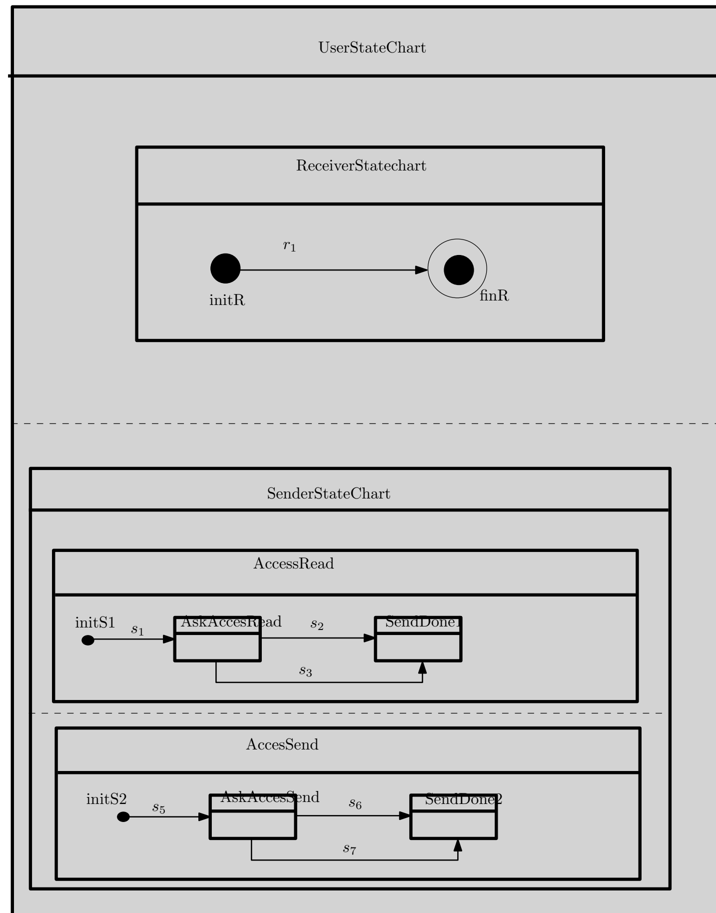

Figure 6.3 Diagramme d'états - $UserStateChart$

6.2.3.1 Diagramme d'états - $UserStateChart$

Le diagramme illustré à la la figure 6.3 modélise le comportement dynamique de la classe `User`. Il est composé de deux états qui s'exécutent en concurrence. Le premier état *ReceiverStateChart*, ne posssède qu'une seule transition. Cette transition permet d'envoyer un document à un usager.
Le second état *SenderStateChart* est constitué également de deux états imbriqués (concurrents) *accessRead* et *accessSend*, qui modélise respectivement les demandes d'accès à un objet de la classe `Document` et les demandes de communications entre deux usagers. Les demandes d'accès correspondent à un appel de la méthode $accessTable(p_3 : int)$, tandis que les demandes de communication correspondent à un appel à la méthode $updateTable(p_1 : int)$.

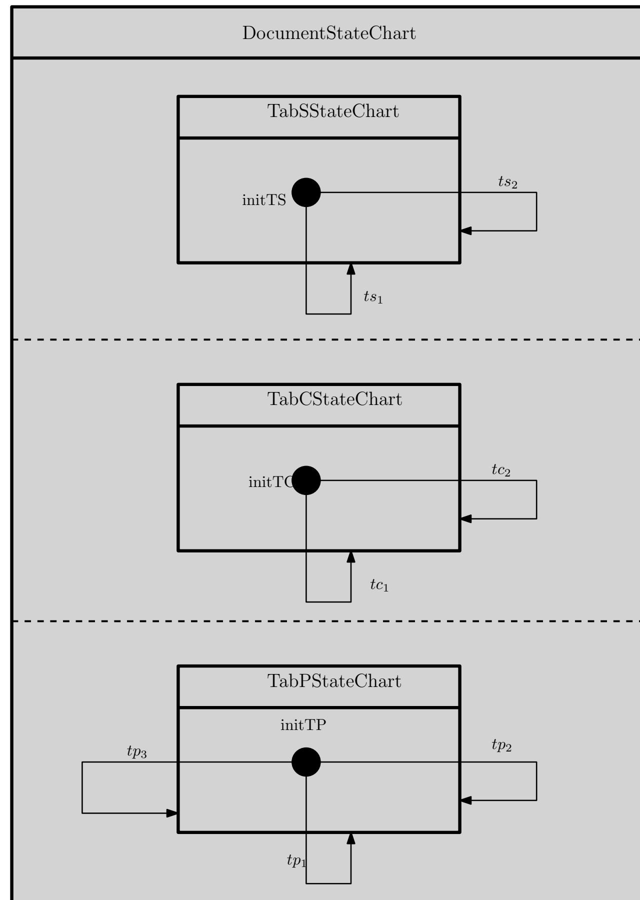

Figure 6.4 Diagramme d'états *Document*

6.2.3.2 Diagramme d'états - *DocumentStateChart*1

Le diagramme illustré à la figure 6.4 modélise le comportement dynamique de l'instance $d1$ de la classe `Document`. Les demandes proviennent des appels faits par une instance de la classe `User`. L'état *DocumentStateChart1* est constitué de trois états : *TabSStateChart1*, *TabCStateChart1* et *TabPStateChart1*. L'état *TabPStateChart1* modélise le comportement dynamique des instances de la classe `Document` lorsqu'une demande de changement de projet survient. Un appel de la méthode $accessTable(p3 : int)$ réveille l'une des deux transitions *tp1* et *tp2*, et vérifie les droits d'accès (niveau de sécurité de l'usager et du document) et retourne une réponse. De plus, l'appel à la méthode $updateTable(p2 : int)$ permet de modéliser une demande de changement de projet (mise à jour des projets) et réveille la transition *tp3*. L'état *TabCStateChart1* modélise le comportement dynamique des instances de la classe `Document` lorsqu'une demande de changement de cavéat survient. Un appel de la méthode $accessTable(p3 : int)$ réveille l'une des deux transitions $tc1$ et $tc2$, et vérifie les droits d'accès (niveau de sécurité de l'usager et du document) en considérant le pays d'origine.

L'état *TabSStateChart1* modélise le comportement dynamique des instances de la classe `Document` lorsqu'une demande de changement de rang survient. Un appel de la méthode $changeRank(p4 : int, p5 : int)$ réveille l'une des deux transitions $tc1$ et $tc2$. La table reçoit une demande de vérification de la part du `Document`, vérifie les droits d'accès (niveau de sécurité de l'usager et du document) et retourne une réponse.

6.2.3.3 Diagramme d'états - $DocumentStateChart2$

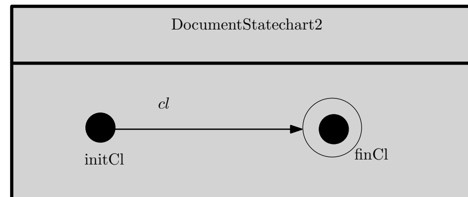

Figure 6.5 Diagramme d'états - $DocumentStateChart2$

Le diagramme illustré à la la figure 6.5 modélise le comportement dynamique de l'instance $d2$ de la classe `Document`. Il est composé de deux états : le premier état $initCl$, qui correspond à l'état initial et le second état $finCL$ qui correspond à l'état final. Cette transition permet d'envoyer un document à un usager.

6.2.4 Étape 4 - Définition du flux d'information

On définit dans cette sous-section le flux d'information légal à partir de l'ensemble des étiquettes de sécurité et d'une relation d'ordre sur les étiquettes de sécurité.

6.2.4.1 Ensemble des étiquettes de sécurité des attributs des objets

Les ensembles d'étiquettes de sécurité sont exprimées à partir des diagrammes d'objets UML (Section 6.2.2) et des tableaux de contraintes décrits à la section (Section 6.1). Les étapes pour l'intégration des étiquettes de sécurité est la définition d'un ensemble d'objets du système qui constitueront la hiérarchie d'objets. Ensuite, on associe à chaque attributs des objets une étiquette de sécurité. La hiérarchie d'objets est définie à partir du diagramme objet UML, et correspond à l'ensemble $\mathcal{H}$ suivant :

$$\mathcal{H} = \{(ca, us), (us, uk), (uk, ger), (ceo, canus), (canus, usuk)\}$$
$$\cup\{(usuk, canusuk), (canusuk, nato), (c1A, c1B), (c1B, c2P)\}$$

Tableau 6.6 Étiquettes de sécurité - objet u

Objet u	
Attributs	Étiquettes
clearance	$L_C = \{(u, \{u\})\}$ $L_I = \{(u, \{u\})\}$
country	$L_C = \{(u, \{u\})\}$ $L_I = \{(u, \{u\})\}$
rank	$L_C = \{(u, \{u\})\}$ $L_I = \{\}$

Tableau 6.7 Étiquettes de sécurité - objet u

Objet $d1$	
Attributs	Étiquettes
caveat	$L_C = \{(d1, \{d1\})\}$ $L_I = \{(d1, \{d2\})\}$
level	$L_C = \{(d1, \{d1\})\}$ $L_I = \{(1, \{d1\})\}$
projet	$L_C = \{(d1, \{d2\})\}$ $L_I = \{\}$

Tableau 6.8 Étiquettes de sécurité - objet u

Objet $d2$	
Attributs	Étiquettes
caveat	$L_C = \{(d2, \{d2\})\}$ $L_I = \{(d1, \{d1\})\}$
level	$L_C = \{(d2, \{dd2\})\}$ $L_I = \{(d2, \{d2\})\}$
projet	$L_C = \{(d1, \{d2\})\}$ $L_I = \{\}$

Tableau 6.9 Étiquettes de sécurité - objet $c1A$

Objets $c1A$	
Attributs	Étiquettes
codeC	$L_C = \{(c1A, \{c1A\})\}$ $L_I = \{(c1A, \{c1A\})\}$

Tableau 6.10 Étiquettes de sécurité - objet $c1B$

Objets $c1B$	
Attributs	Étiquettes
codeC	$L_C = \{(c1B, \{c1B\})\}$ $L_I = \{(c1B, \{c1B\})\}$

et l'ensemble des étiquettes de sécurité pour chaque attribut d'objet est :

Tableau 6.11 Étiquettes de sécurité - objet $c1R$

Objet $c1R$	
Attributs	Étiquette
codeR	$L_C = \{(c1R, \{c1R, c2R, d1\}), (c2R, \{c2R, d1\}), (d1, \{d1\})\}$ $L_I = \{(c1R, \{c1R, c2R, d1\}), (c2R, \{c2R, d1\}), (d1, \{d1\})\}$

Tableau 6.12 Étiquettes de sécurité - objet $c2R$

Objet $c2R$	
Attributs	Étiquette
codeR	$L_C = \{(c2R, \{c2R, d1\}), (d1, \{d1\})\}$ $L_I = \{(c2R, \{c2R, d1\}), (d1, \{d1\})\}$

Tableau 6.13 Étiquettes de sécurité - objet ceo

Objets ceo	
Attributs	Étiquettes
codeC	$L_C = \{(ceo, \{usuk, canus\}), (unCoalition, \{usuk, canus, nato\})\}$ $L_I = \{(ceo, \{nato, canusuk\}), (unCoalition, \{canusuk\})\}$

Tableau 6.14 Étiquettes de sécurité - objet $canus$

Objets $canus$	
Attributs	Étiquettes
codeC	$L_C = \{(canus, \{canusuk, nato\}), (canusuk, \{unCoalition, canus\})\}$ $L_I = \{(unCoalition, \{canusuk\}), (nato, \{canus\})\}$

Tableau 6.15 Étiquettes de sécurité - objet *usuk*

Objets *usuk*	
Attributs	Étiquettes
codeC	$L_C = \{(usuk, \{ceo, nato\}), (canus, \{canusuk, nato\}), (canusuk, \{unCoalition, canus\})\}$ $L_I = \{(unCoalition, \{canusuk\}), (nato, \{canus\})\}$

Tableau 6.16 Étiquettes de sécurité - objet *canusuk*

Objets *canusuk*	
Attributs	Étiquettes
codeC	$L_C = \{(canus, \{canusuk, nato\}), (canusuk, \{unCoalition, canus\})\}$ $L_I = \{(unCoalition, \{canusuk\}), (canusuk, \{ceo\}), (nato, \{canus\})\}$

Tableau 6.17 Étiquettes de sécurité - objet *nato*

Objets *nato*	
Attributs	Étiquettes
codeC	$L_C = \{(canus, \{canusuk, nato\}), (nato, \{unCoalition, canus\})\}$ $L_I = \{(unCoalition, \{canusuk\}), (nato, \{canus\})\}$

Tableau 6.18 Étiquettes de sécurité - objet *unCoalition*

Objets *unCoalition*	
Attributs	Étiquettes
codeC	$L_C = \{(canus, \{canusuk, nato\}), (unCoalition, \{canusuk, canus\})\}$ $L_I = \{(unCoalition, \{canusuk\}), (nato, \{canus\})\}$

Tableau 6.19 Étiquettes de sécurité - objet *ca*

Objets *ca*	
Attributs	Étiquettes
codeP	$L_C = \{(ceo, \{ceo\}), (canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$ $L_I = \{(ceo, \{ceo\}), (canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$

Tableau 6.20 Étiquettes de sécurité - objet us

Objets us	
Attributs	Étiquettes
codeP	$L_C = \{(canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$ $L_I = \{(canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$

Tableau 6.21 Étiquettes de sécurité - objet uk

Objets uk	
Attributs	Étiquettes
codeP	$L_C = \{(canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\}), \}$ $L_I = \{(canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\}), \}$

Tableau 6.22 Étiquettes de sécurité - objet ger

Objets ger	
Attributs	Étiquettes
codeP	$L_C = \{(nato, \{nato\})\}$ $L_I = \{(nato, \{nato\})\}$

Tableau 6.23 Étiquettes de sécurité - objet egy

Objets egy	
Attributs	Étiquettes
codeP	$L_C = \{(egy, \{egy\})\}$ $L_I = \{(egy, \{egy\})\}$

Tableau 6.24 Étiquettes de sécurité - objet $c1P$

Objets $c1P$	
Attributs	Étiquettes
codeP	$L_C = \{(ceo, \{ceo\}), (canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$ $L_I = \{(ceo, \{ceo\}), (canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$

Tableau 6.25 Étiquettes de sécurité - objet $c2P$

Objets $c2P$	
Attributs	Étiquettes
codeP	$L_C = \{(canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$ $L_I = \{(canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\}$

6.2.4.2 Étiquettes de sécurité des paramètres de méthodes

On définit dans cette section les étiquettes de sécurité des paramètres formels des méthodes *receive*, *updateTable*, *accessTable* et *changeRank*.

Tableau 6.26 Étiquettes de sécurité - méthode *receive*

Méthode *receive*	
Paramètres	Étiquette
p1	$L_C = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$ $L_I = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$

Tableau 6.27 Étiquettes de sécurité - méthode $accessTable$

Méthode $accessTable$	
Paramètres	Étiquette
p3	$L_C = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$ $L_I = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$

Tableau 6.28 Étiquettes de sécurité - méthode $updateTable$

Méthode $updateTable$	
Paramètres	Étiquette
p2	$L_C = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$ $L_I = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$

Tableau 6.29 Étiquettes de sécurité - méthode *changeRank*

Méthode *changeRank*	
Paramètres	Étiquette
p4	$L_C = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$ $L_I = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$
p5	$L_C = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$ $L_I = \{(u1, \{u1, d1, d2\}), (d1, \{d1, d2\}), (d2, \{d2\})\}$

6.2.4.3 Treillis de sécurité

La relation $\sqsubseteq_C$ permet de définir un ordre sur l'ensemble des étiquettes de confidentialité. La relation $\sqsubseteq_I$ permet de définir un ordre sur l'ensemble des étiquettes d'intégrité. L'ordre sur $\sqsubseteq_C$ et $\sqsubseteq_I$ est déterminé respectivement par les algorithmes 4 et 5. L'ensemble des étiquettes de sécurité et la relation $\sqsubseteq$ permettent de définir un treillis de sécurité à partir de l'algorithme 6.

L'étiquette de confidentialité la moins restrictive est donnée par :

$$L_{C,\bot} = \{\ \}$$

L'étiquette d'intégrité la moins restrictive est donnée par :

$$L_{I,\bot} = \{(u, \{\}), (d1, \{\}), (d2, \{\}), (c1A, \{\}), (c1R, \{\}), (c1P, \{\}), (c1B, \{\}), (c2R, \{\}), (c2P, \{\}), (ca, \{\}), (us, \{\}), (uk, \{\}), (ger, \{\}), (egy, \{\})\}$$
$$\cup\{(ceo, \{\}), (canus, \{\}), (canusuk, \{\}), (nato, \{\}), (ger, \{\}), (ceo, \{\}), (canus, \{\}), (usuk, \{\}), (unCoalition, \{\}), (usuk, \{\})\}$$

L'étiquette de confidentialité la plus restrictive est donnée par :

$$L_{I,\bot} = \{(u, \{\}), (d1, \{\}), (d2, \{\}), (c1A, \{\}), (c1R, \{\}), (c1P, \{\}), (c1B, \{\}), (c2R, \{\}), (c2P, \{\}), (ca, \{\}), (us, \{\}), (uk, \{\}), (ger, \{\}), (egy, \{\})\}$$
$$\cup\{(ceo, \{\}), (canus, \{\}), (canusuk, \{\}), (nato, \{\}), (ger, \{\}), (ceo, \{\}), (canus, \{\}), (usuk, \{\}), (unCoalition, \{\}), (usuk, \{\})\}$$

L'étiquette d'intégrité la plus restrictive est donnée par :

$$L_{I,\top} = \{\}$$

6.3 Modèle formel du système

On décrit dans cette section, la syntaxe des classes, objets et diagrammes d'états modélisés dans le système des cavéats.

6.3.1 Modèle formel des classes

Ensemble des attributs de classe L'ensemble des attributs de classe est constitué des éléments suivants :

$clearance = (clearance, 0, L_{Clearance})$	$country = (country, 0, L_{country})$
$rank = (rank, 0, L_{rank})$	$level = (level, 0, L_{level})$
$caveat = (caveat, 0, L_{caveat})$	$project = (project, 0, L_{project})$
$codeP = (codeP, 0, L_{codeP})$	$codeR = (codeR, 0, L_{codeR})$
$codeP = (codeP, 0, L_{codeP})$	$codeR = (codeR, 0, L_{codeR})$

Figure 6.6 Ensemble des attributs de classe

Ensemble des paramètres L'ensemble des paramètres est constitué des éléments suivants :

$p_1 = (p_1, 0, L_{p_1}))$	$p_2 = (p_2, 0, L_{p_2})$
$p_3 = (p_3, 0, L_{p_3})$	$p_4 = (p_4, 0, L_{p_4})$
$p_5 = (p_5, 0, L_{p_5})$	$r_1 = (r_1, 0, L_{r_1})$

Figure 6.7 Ensemble des parametres

Ensemble des méthodes L'ensemble des méthodes est constitué des éléments suivants :

$receive = (receive, \{p_1\}, \star)$
$accessTable = (accessTable, \{p_3\}, \star)$
$updateTable = (updateTable, \{p_2\}, \star)$
$changeRank = (changeRank, \{p_4, p_5\}, r_1)$

Figure 6.8 Ensemble des methodes

Ensemble des classes L'ensemble des classes est constitué des éléments suivants :

$project = (\texttt{Project}, \{codePj\}, \{\})$
$rank = (\texttt{Rank}, \{codeR\}, \{\})$
$country = (\texttt{Country}, \{codeP\}, \{\})$
$caveat = (\texttt{Caveat}, \{codeC\}, \{\})$
$user = (\texttt{User}, \{clearance, country, rank\}, \{\})$
$doc = (\texttt{Document}, \{level, caveat, project\}, \{receive, updateTable, accessTable, changeRank\})$

Figure 6.9 Ensemble des classes

6.3.2 Modèle formel des objets

On décrit dans cette section le modèle formel des objets représentés par le diagramme de la figure 6.2.

Ensemble des objets L'ensemble des objets est constitué des éléments suivants :

$u = (u, user)$	$d1 = (d1, doc)$
$d2 = (d2, doc)$	$ca = (ca, country)$
$us = (us, country)$	$uk = (uk, country)$
$ger = (ger, country)$	$egy = (egy, country)$
$ger = (ger, country)$	$egy = (egy, country)$
$ceo = (ceo, caveat)$	$canus = (canus, caveat)$
$usuk = (usuk, caveat)$	$canusuk = (canusuk, caveat)$
$nato = (nato, caveat)$	$unCoalition = (unCoalition, caveat)$
$c1A = (c1A, project)$	$c1R = (c1R, rank)$
$c1B = (c1B, project)$	$c1P = (c1P, country)$
$c2R = (c2R, rank)$	$c2P = (c2P, country)$

Figure 6.10 Ensemble des objets

Ensemble des instances de méthodes L'ensemble des instances de méthodes est constitué des éléments suivants :

$receive_1 = (d1, receive, receive_1)$
$accessTable_1 = (d1, accessTable, accessTable_1)$
$updateTable_1 = (d1, updateTable, updateTable_1)$
$changeRank_1 = (d1, changeRank, changeRank_1)$
$changeRank_2 = (d2, changeRank, changeRank_2$

Figure 6.11 Ensemble des instances de methodes

6.3.3 Modèle formel des diagrammes d'états

Ensemble des régions L'ensemble des régions est constitué des éléments suivants :

$rg_1 = (rg_1, \{ReceiverStateChart\})$
$rg_2 = (rg_2, \{SenderStateChart\})$
$rg_3 = (rg_3, \{initR, finR\})$
$rg_4 = (rg_4, \{accessRead\})$
$rg_5 = (rg_5, \{accessSend\})$
$rg_6 = (rg_6, \{intS1, AskaccessRead, SendDone\})$
$rg_7 = (rg_7, \{initS2, AskaccessSend, SendDone2\})$
$rg_8 = (rg_8, \{initTS\})$
$rg_9 = (rg_9, \{initTC\})$
$rg_{10} = (rg_{10}, \{initTP\})$
$rg_{11} = (rg_{11}, \{initCl, finCl\})$

Figure 6.12 Ensemble des regions

Ensemble des états L'ensemble des états est constitué des éléments suivants :

$UserStateChart = (\mathsf{AND}, UserStateChart, \{\}, entry, exit)$
$SenderStateChart = (\mathsf{AND}, SenderStateChart, \{\}, entry, exit)$
$ReceiverStateChart = (\mathsf{OR}, ReceiverStateChart, \{\}, entry, exit)$
$DocumentStateChart1 = (\mathsf{AND}, DocumentStateChart1, \{\}, entry, exit)$
$DocumentStateChart2 = (\mathsf{AND}, DocumentStateChart2, \{\}, entry, exit)$
$initR = (\mathsf{BEGIN}, initR, \{\}, entry, exit)$
$finR = (\mathsf{END}, finR, \{\}, entry, exit)$
$accessRead = (\mathsf{OR}, accessRead, \{\}, entry, exit)$
$accessSend = (\mathsf{OR}, accessSend, \{\}, entry, exit)$
$AskaccessRead = (\mathsf{AND}, AskaccessRead, \{\}, entry, exit)$
$SendDone = (\mathsf{AND}, SendDone, \{\}, entry, exit)$
$SendDone2 = (\mathsf{AND}, SendDone2, \{\}, entry, exit)$
$initS1 = (\mathsf{BEGIN}, initS1, \{\}, entry, exit)$
$initS2 = (\mathsf{BEGIN}, initS2, \{\}, entry, exit)$
$TabSStateChart1 = (\mathsf{SIMPLE}, TabSStateChart1, \{\}, entry, exit)$
$TabCStateChart1 = (\mathsf{SIMPLE}, TabCStateChart1, \{\}, entry, exit)$
$TabPStateChart1 = (\mathsf{SIMPLE}, TabPStateChart1, \{\}, entry, exit)$
$initTC = (\mathsf{BEGIN}, initTC, \{\}, entry, exit)$
$initTS = (\mathsf{BEGIN}, initTS, \{\}, entry, exit)$
$initTP = (\mathsf{BEGIN}, initTP, \{\}, entry, exit)$
$initCl = (\mathsf{BEGIN}, initCl, \{\}, entry, exit)$
$finCl = (\mathsf{END}, finCl, \{\}, entry, exit)$

Figure 6.13 Ensemble des etats

tp1
trigger($tp1$) : $(trgtp1, \mathsf{MCALL}, accessTable)$
guard($tp1$) : c2P.codeP != u.country
action($tp1$) : $(acttp1, \mathsf{MRETURN}, u, d1, (accessTable, (p_3, u.country)))$

tp2
trigger($tp2$) : $(trgtp2, \mathsf{MCALL}, accessTable)$
guard($tp2$) : c2P.codeP == u.country
action($tp2$) : $(acttp2, \mathsf{MRETURN}, u, d1, (accessTable, (p_3, u.country)))$

tp3
trigger($tp3$) : $(trgtp3, \mathsf{MCALL}, updateTable)$
guard($tp3$) : True
action($tp3$) : $(acttp3, \mathsf{MCALL}, tabP, c3, (changeRank, (p_4, u.rank), (p_5, c1R.rank))); (acttp3, \mathsf{MRETURN}, u, d1, (updateTable, (p_2, d2.level)))$

Figure 6.14 Transitions `TabPStateChart1`

Ensemble des transitions

cl
trigger(cl) : $(trgs1, \mathsf{MCALL}, changeRank)$
guard(cl) : True
action(cl) : $(acts1, \mathsf{ASSIGN}, u, u, (u.rank, c2R.rank)); (acts1, \mathsf{ASSIGN}, u, u, (u.country, c2P.country))$

Figure 6.15 Transitions `DocumentStatechart2`

ts1
trigger($ts1$) : $(trgts1, \mathsf{MCALL}, accessTable_1)$
guard($ts1$) : u.clearance >= d1.level
action($ts1$) : $(actts1, \mathsf{MRETURN}, u, d1, (accessTable_1, (p_3, u.country)))$

ts2
trigger($ts2$) : $(trgts2, \mathsf{MCALL}, accessTable_1)$
guard($ts2$) : u.clearance < d1.level
action($ts2$) : $(actts2, \mathsf{MRETURN}, u, d1, (accessTable_1, (p_3, u.country)))$

Figure 6.16 Transitions `TabSStateChart1`

tc1
trigger($tc1$) : $(trgtc1, \mathsf{MCALL}, accessTable_1)$
guard($tc1$) : ceo.codeC == d1.caveat && (u.country == uk.codeP || u.country == ger.codeP)
action($tc1$) : $(acttc1, \mathsf{MRETURN}, u, d1, (accessTable_1, (p_3, u.country)))$

tc2
trigger($tc2$) : $(trgtc2, \mathsf{MCALL}, accessTable_1)$
guard($tc2$) : canus.codeC == d2.caveat && (u.country == uk.codeP || u.country == ger.codeP)
action($tc2$) : $(acttc2, \mathsf{MCALL}, u, d2, (accessTable_2, (p_3, d2.caveat))); (acttc2, \mathsf{ASSIGN}, d1, d1, (d1.level, u.clearance))$

Figure 6.17 Transitions `TabCStateChart1`

s1
trigger($s1$) : $\star$
guard($s1$) : $\star$
action($s1$) : $(acts1, \mathsf{MCALL}, u, d1, (accessTable, (p_3, u.country)))$

s2
trigger($s2$) : $(trgs2, \mathsf{MRETURN}, access)$
guard($s2$) : True
action($s2$) : $(acts2, \mathsf{ASSIGN}, u, u, (u.clearance, u.country))$

s3
trigger($s3$) : $(trgs3, \mathsf{MRETURN}, access)$
guard($s3$) : False
action($s3$) : $(acts3, \mathsf{ASSIGN}, u, u, (u.clearance, u.rank))$

s5
trigger($s5$) : $\star$
guard($s5$) : True
action($s5$) : $(acts5, \mathsf{MCALL}, u, d1, (receive, (p_1, u.rank)))$

s6
trigger($s6$) : $(trgs6, \mathsf{MRETURN}, accessSend)$
guard($s6$) : u.accessSend==True
action($s6$) : $(acts6, \mathsf{ASSIGN}, u, u, (u.rank, u.rank))$

s7
trigger($s7$) : $(trgs6, \mathsf{MRETURN}, accessSend)$
guard($s7$) : u.accessSend==False
action($s7$) : $(acts7, \mathsf{ASSIGN}, u, d1, (accessSend, true))$

r1
trigger($r1$) : $(trgr1, \mathsf{MCALL}, receive)$
guard($r1$) : True
action($r1$) : $(actr1, \mathsf{ASSIGN}, u, u, (u.rank, c1R.codeR))$

Figure 6.18 Transitions `UserStateChart`

6.4 Vérification de la sécurité

Les sections précédentes (Section 6.2 et section 6.3) ont présenté respectivement les spécifications informelles et formelles des modèles UML simplifié des cavéats. La section , quant à elle, décrit la méthode de vérification du flux d'information pour les objets représentés dans le système des cavéats. On applique notre vérification sur le premier état de notre système de transition et l'état suivant, ceci dans le but de ne pas cosntruire le système de transition entier. Cette section est organisée comme suit. La première sous-section (section 6.4.1) décrit porte sur la description formelle du système de transition d'un système de transition. La sous-section 6.4.2 applique les algorithmes de vérification du flux d'information dans le système des cavéats.

6.4.1 Système de transition

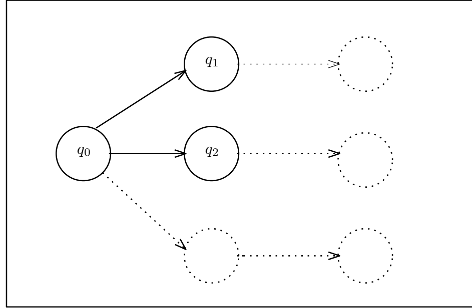

Figure 6.19 Système de transition simplifié

On définit dans cette sous-section le système de transition pour notre système des cavéats. On ne décrit pas le système de transition complet, mais uniquement trois états pour lesquels on veut vérifier le flux d'information. Une partie de notre système de transition est illustré à la figure 6.19. Il est décrit de façon formelle par :

$$\mathcal{T} = (q_0,\ \mathcal{Q},\ \mathcal{R},\ \mathsf{Trans},\ \mathsf{Guards},\ \mathsf{labelEffect})$$

où :

- q_0 est l'état initial du système,
- $\mathcal{Q}$ est l'ensemble des états du système,
- $\mathcal{R}$ est la relation de transition entre les états du système,
- Trans est l'ensemble des transitions
- Guards est l'ensemble des gardes
- $\mathsf{labelEffect}$ est la fonction d'évaluation des étiquettes de sécurité

État initial L'état initial de notre système de transition est donné par :

$$q_0 = (l_0, \lambda_0)$$

où l_0 appartient à l'ensemble des locations et est défini par

$$l_0 \ = \ (\{initR, initS1, initS2\}, \{initTS, initTP, initTC\}, \{initCL\})$$

et λ_0 correspond à la valeur des étiquettes de sécurité des données dans l'état initial. Ces valeurs sont décrites à la section 6.2.4.

Ensembles des états L'ensemble des états est défini à partir des figures 6.4, 6.5, 6.18, 6.17, 6.16 et 6.14 de la section 6.3 et est donné par : Premièrement, on définit pour le diagramme d'états *DocumentStateChart1* l'ensemble des états UML actifs. Cet ensemble est calculé à partir de la sémantique des diagrammes d'états définis à la section 4.2.2.2.

$$\langle initTP, \phi\rangle \xrightarrow{tp1} \langle TabPStateChart1, \phi_1\rangle$$
$$\langle initTP, \phi\rangle \xrightarrow{tp2} \langle TabPStateChart1, \phi_2\rangle$$
$$\langle initTP, \phi\rangle \xrightarrow{tp3} \langle TabPStateChart1, \phi_3\rangle$$

où :

$$\phi = \{initTP\}$$
$$\phi_1 = \{initTP, TabPStateChart1\}$$
$$\phi_2 = \{initTP, TabPStateChart1\}$$
$$\phi_3 = \{initTP, TabPStateChart1\}$$

$$\langle initTC, \phi\rangle \xrightarrow{tc1} \langle TabCStateChart1, \phi_4\rangle$$
$$\langle initTC, \phi\rangle \xrightarrow{tc2} \langle TabCStateChart1, \phi_5\rangle$$

où :

$$\phi = \{initTP\}$$
$$\phi_4 = \{initTC, TabCStateChart1\}$$
$$\phi_5 = \{initTC, TabCStateChart1\}$$

$$\langle initTS, \phi\rangle \xrightarrow{ts1} \langle TabSStateChart1, \phi_6\rangle$$
$$\langle initTS, \phi\rangle \xrightarrow{ts2} \langle TabSStateChart1, \phi_7\rangle$$

où :

$$\phi = \{initTP\}$$
$$\phi_6 = \{initTS, TabSStateChart1\}$$
$$\phi_7 = \{initTS, TabSStateChart1\}$$

Pour le diagramme d'états *DocumentStateChart2*, l'ensemble des états actifs est défini par :

$$\langle initCl, \phi\rangle \xrightarrow{cl} \langle TabPStateChart1, \phi_8\rangle$$

où :

$$\phi_8 = \{finCl, DocumentStateChart2\}$$

Pour le diagramme d'états *UserStateChart*, l'ensemble des états actifs est défini par :

$$\langle initR, \phi \rangle \xrightarrow{r1} \langle finR, \phi_9 \rangle$$
$$\langle initS1, \phi \rangle \xrightarrow{s1} \langle AskaccessRead, \phi_{10} \rangle$$
$$\langle initS2, \phi \rangle \xrightarrow{s5} \langle AskaccessSend, \phi_{11} \rangle$$
$$\langle AskaccessRead, \phi \rangle \xrightarrow{s2} \langle SendDone1, \phi_{12} \rangle$$
$$\langle AskaccessSend, \phi \rangle \xrightarrow{s6} \langle SendDone2, \phi_{13} \rangle$$
$$\langle AskaccessRead, \phi \rangle \xrightarrow{s3} \langle SendDone1, \phi_{14} \rangle$$
$$\langle AskaccessSend, \phi \rangle \xrightarrow{s7} \langle SendDone2, \phi_{15} \rangle$$

où :

$$\phi_{10} = \{AskaccessRead, accessRead, UserStateChart\}$$
$$\phi_{12} = \{SendDone1, accessRead, UserStateChart\}$$
$$\phi_{14} = \{SendDone1, accessRead, UserStateChart\}$$
$$\phi_{11} = \{AskaccessSend, accessSend, UserStateChart\}$$
$$\phi_{13} = \{SendDone2, accessSend, UserStateChart\}$$
$$\phi_{15} = \{SendDone2, accessSend, UserStateChart\}$$
$$\phi_9 = \{finR, ReceiverStateChart, UserStateChart\}$$

L'ensemble des locations (Chapitre 4, sous-section 4.2.2.2) est défini à partir des ensembles d'états UML activés. Le premier état de notre système de transition qui correspond à une exécution de la transition $tp1$. Cet état est décrit par :

$$q_1 = (l_1, \lambda_1)$$

où l_1 appartient à l'ensemble des locations et est défini par

$$l_1 \ = \ (\{initR, initS1, initS2\}, \{initTS, initTP, initTC\}, \{initCL\})$$

et λ_1 correspond à la valeur des étiquettes de sécurité des données dans le premier état, avec

$$\langle l_1, \lambda_0 \rangle \xrightarrow{gtp1:tp1} \langle l_1, \lambda_1 \rangle$$
$$\lambda_1 = \mathsf{labelEffect}(\mathsf{action}(tp1), \lambda_0)$$

On considère également un deuxième état qui correspond à une exécution de la transition $tc2$. Cet état est décrit par :

$$q_2 = (l_2, \lambda_2)$$

où l_2 appartient à l'ensemble des locations et est défini par

$$l_2 = (\{initR, initS1, initS2\}, \{initTS, initTP, initTC\}, \{initCL\})$$

et λ_2 correspond à la valeur des étiquettes de sécurité des données dans le second état, avec

$$\langle l_0, \lambda_0 \rangle \xrightarrow{gtc2:tc2} \langle l_2, \lambda_2 \rangle$$
$$\lambda_2 = \mathsf{labelEffect}(\mathsf{action}(tc2), \lambda_0)$$

Dans notre étude de cas, pour l'évaluation de λ, on considère uniquement les données qui sont impliquées lors de la transition (garde et séquence d'actions). Ainsi, d'après la description de la séquence d'actions qui s'exécute lors de la transition $tp1$, on a un appel de la méthode $accessTable$, avec les données $u.country$, $c2P.codeP$, p_3 et $u.clearance$. λ_1 correspond aux nouvelles valeurs des étiquettes de sécurité pour les données $u.country$, $c2P.codeP$, p_3 et $u.clearance$.

$$\lambda_1(u.country) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_1(c2P.codeP) = \{(\{(canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\},$$
$$\{(canus, \{canus\}), (canusuk, \{canusuk\}), (nato, \{nato\}), (usuk, \{usuk\})\})\}$$
$$\lambda_1(p_3) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_1(u.clearance) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$

Ainsi, d'après la description de la séquence d'actions qui s'exécute lors de la transition $tc2$, on a un appel de la méthode $accessTable$, avec les données $canus.codeC$, $d2.caveat$, $u.country$, $uk.codeP$, $ger.codeP$ et p_3. λ_2 correspond aux nouvelles valeurs des étiquettes de sécurité pour les données $canus.codeC$, $d2.caveat$, $u.country$, $uk.codeP$, $ger.codeP$, $d1.level$ et p_3.

$$\lambda_2(u.country) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_2(canus.codeC) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_2(uk.codeP) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_2(ger.codeP) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_2(d2.caveat) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_2(d1.level) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$
$$\lambda_2(p_3) = \{(\{(u, \{u\})\}, \{(u, \{u\})\})\}$$

Ensembles des transitions L'ensemble des transitions est défini à partir des figures 6.4, 6.5, 6.18, 6.17, 6.16 et 6.14 de la section 6.3 et est donné par :

$$\mathsf{Trans} = \{tp1, tp2, tp3, tc1, tc2, ts1, ts2, s1, s2, s3, s5, s6, s7, r1, cl\}$$

Ensembles des gardes L'ensemble des gardes est défini à partir des figures 6.4, 6.5, 6.18, 6.17, 6.16 et 6.14 de la section 6.3 et est donné par :

$$\mathsf{Guards} = \{gtp1, gtp2, gtp3, gtc1, gtc2, gts1, gts2, gs1, gs2, gs3, gs5, gs6, gs7, gr1, gcl\}$$

Dans la prochaine sous-section (sous-section 6.4.2), on utilise l'algorithme de vérification pour vérifier le flux d'information dans les états q_0, q_1 et q_2.

6.4.2 Vérification du système de transition

Dans cette sous-section, on vérifie le flux d'information pour les états q_0, q_1, q_2. On s'assure que lorsque les transitions $tp1$ et $tc2$ s'exécutent, les nouvelles étiquettes de sécurité dans les états q_1 et q_2, sont plus restrictives que celles dans l'état q_0.

On fait cette vérification en trois étapes. Premièrement, on vérifie les flux explictes, c'est à dire les flux qui proviennent directement d'affectations ou de passage de paramètres.

$$\llbracket tp1 \rrbracket_{l_0,\lambda_0,r_1} = true$$
$$\llbracket tc2 \rrbracket_{l_0,\lambda_0,r_2} = false$$

avec,

$$r_1 = (q_0, gtp1, tp1, q_1)$$
$$r_2 = (q_0, gtc2, tc2, q_2)$$

Dans le cas de la transition $tc2$ et du contexte d'exécution $\mathsf{context}(tc2) = (l_0, \lambda_0, r_2)$ (Chapitre 5, Section 5.2.1)

$$L'_{u.country} \in \lambda_1, \text{ et } L_{u.country} \sqsubseteq L'_{u.country}$$
$$L'_{canus.codeC} \in \lambda_1, \text{ et } L_{canus.codeC} \sqsubseteq L'_{canus.codeC}$$
$$L'_{d2.caveat} \in \lambda_1, \text{ et } L_{d2.caveat} \sqsubseteq L'_{d2.caveat}$$
$$L'_{uk.codeP} \in \lambda_1, \text{ et } L_{u.country} \sqsubseteq L'_{uk.codeP}$$
$$L'_{ger.codeP} \in \lambda_1, \text{ et } L_{ger.codeP} \sqsubseteq L'_{ger.codeP}$$
$$L'_{d1.level} \in \lambda_1, \text{ et } L_{d1.level} \sqsubseteq L'_{d1.level}$$
$$L'_{p3} \in \lambda_1, \text{ et } L_{p3} \sqsubseteq L'_{p3}$$

De plus, les données $u.country$, $uk.codeP$, $ger.codeP$, $canus.codeC$ et $d2.caveat$ sont présentes dans la garde $gtc2 = \mathsf{guard}(tc2)$, et $L_{d1.level} \sqsubseteq L_{d2.caveat}$. Le niveau de sécurité de la donnée $d2.caveat$ est inférieur à celui de la donnée $d1.level$. Par conséquent, $\llbracket tc2 \rrbracket_{l_0,\lambda_0,r_2} = false$ et le flux d'information n'est pas sécuritaire lorsque la transition $tc2$ s'exécute.

Dans le cas de la transition $tp1$ et du contexte d'exécution $\mathsf{context}(tp1) = (l_0, \lambda_0, r_1)$

(Chapitre 5, Section 5.2.1)

$$
\begin{aligned}
&L'_{u.country} \in \lambda_1, \text{ et } L_{u.country} \sqsubseteq L'_{u.country} \\
&L'_{c2P.codeP} \in \lambda_1, \text{ et } L_{c2P.codeP} \sqsubseteq L'_{c2P.codeP} \\
&L'_{p3} \in \lambda_1, \text{ et } L_{p3} \sqsubseteq L'_{p3}
\end{aligned}
$$

De plus, les données $u.country$ et $c2P.codeP$ sont présentes dans la garde $gtp1 = \mathsf{guard}(tp1)$, et $L_{u.country} \sqsubseteq L_{p3}$, et $L_{c2P.codeP} \sqsubseteq L_{p3}$. Les niveaux de sécurité des données $u.country$ et $c2P.codeP$ sont supérieurs à celui de la donnée $p3$. Par conséquent, $[\![tp1]\!]_{l_0,\lambda_0,r_1} = true$ et le flux d'information est sécuritaire lorsque la transition $tp1$ s'exécute.

On a ainsi vérifié le flux d'information dans notre système orienté objet. La vérification utilise un système de transition étiquetté avec les transitions et les gardes. Les transitions et les gardes correspondent aux transitions présentent dans les diagrammes d'états.

CHAPITRE 7

CONCLUSION

Nous avons présenté dans cette thèse un cadre pour la spécification et la vérification du flux d'information dans des diagrammes d'états en utilisant une méthode d'analyse statique (Vérification des niveaux de sécurité dans tous les états du système de transition).

La modélisation et la vérification des systèmes orientés objets à l'aide du langage UML demande de développer des techniques qui facilitent la détection et la prévention des défauts de sécurité, plus précisément le flux d'information. En effet, les objets du système interagissent entre eux à travers des appels de méthodes et des affectations de variables à leurs attributs respectifs. Le comportement dynamique du système est modélisé grâce aux diagrammes d'états. L'idée générale est de détecter les affectations illégales de variables et de paramètres de méthodes. Les appels de méthodes et les affectations sont déclenchées par les transitions des diagrammes d'états. Les méthodes usuelles d'analyse de modèles UML utilisent des méthodes formelles afin de spécifier et vérifier la sécurité dans les modèles UML. L'ingéniere des systèmes sécurisés propose des métamodèles UML étendus avec de nouveaux artéfacts ou métamodèles UML. Ces modèles étendus représentent les systèmes sécurisés. D'autres méthodes transforment les modèles UML et les requis de sécurité (contrôle d'accès) en modèles formels qui peuvent-être analysés grâce à des techniques de vérification telles que le *model-checking*. Une stratégie de vérification du flux d'information est donc nécessaire dans la modélisation des diagrammes d'états.

7.1 Synthèse des travaux

La modélisation et la vérification du flux d'information dans un système orienté objet requiert les étapes suivantes. On définit premièrement un modèle de sécurité. Deuxièment, on définit un modèle formel pour notre système orienté objet et finalement on implante des algorithmes qui vérifient les erreurs de flux d'information.

Le modèle de sécurité utilisé est une extension du modèle des étiquettes décentralisées de (Myers et Liskov (1998, 2000)). Notre modèle est étendu à un système orienté objet, c'est à dire que les utilisateurs du système correspondent aux objets du système. Les politiques de sécurité sont appliquées sur les attributs des objets, les paramètres des méthodes et les variables en les annotant avec des étiquettes de sécurité. Une étiquette de sécurité est constituée d'une étiquette de confidentialité (définition des objets qui ont les droits en lecture) et

d'une étiquette d'intégrité (définition des objets qui ont les droits en écriture). Le sens légal du flux d'information (relation $\sqsubseteq$, $\sqsubseteq_C$ et $\sqsubseteq_I$) est déterminé par les algorithmes de restriction définis dans les annexes 6, 4 et 5.

Le système orienté objet est décrit par un diagramme de classe, un diagramme d'objets et les diagrammes d'états de chaque objet. La syntaxe formelle décrit les paradigmes orientés objet étendus avec des étiquettes de sécurité sur les attributs, les variables et les paramètres des méthodes. Les valeurs des étiquettes de sécurité sur les gardes et les transitions des diagrammes d 'états sont calculées. La sémantique formelle est donnée par un système de transition à états finis. Il s'agit d'un système de transition symbolique où l'information est concentrée dans les états et sur les transitions. Un état est décrit par une paire constituée d'un ensemble d'états UML et d'une fonction qui donne la valeur des étiquettes de sécurité. Les transitions donnent de lìnformation sur les affectations et les appels de méthodes qui surviennent.

Après avoir exprimé la syntaxe et la sémantique des systèmes orientés objets, on a présenté un algorithme de vérification du flux d'information dans un système orienté objets. Cet algorithme prend en paramètre une transition et un système de transition.

7.2 Limitations de la solution proposée

Selon nous, le travail présenté dans cette thèse constitue un début prometteur. Cependant, comme pour tout travail qui débute, beaucoup reste encore à faire. Nous discutons ici des limitations proposées par notre approche. La première limite de notre approche est l'impossibilité d'exprimer des contraintes sur les systèmes grace à des formules de logiques temporelles (von der Beeck (2002); Cheng et Zhang (2011); Siveroni *et al.* (2010); Jinhua et Jing (2010)). On pourrait ainsi vérifier de façon automatique les modèles de sécurité obtenus à partir des des diagrammes UML. Une deuxième limite est le nombre d'approximation et hypothèse que l'on a émises sur les diagrammes UML, notamment les diagrammes d'états. De nombreux éléments n'ont pas été pris en compte lors de la modélisation des diagrammes. Il s'agit des états historiques, des priorités des transtions et des évènements. Ces éléments introduisent du non déterminisme. Une troisième limite est reliée au nombre d'objets qui peuvent être modélisés dans le système. On considère un système possédant un nombre fini d'objets. Aucun nouvel objet ne peut être créé durant l'exécution du système.

7.3 Améliorations futures

Selon nous, le travail présenté dans cette thèse constitue un début prometteur. Cependant, comme pour tout travail qui débute, beaucoup reste encore à faire. Nous discutons

ici des travaux futurs envisagés. Nous projetons tout d'abord d'intégrer la possibilité de spécifier des propriété de sécurité. Nous souhaiterions notamment étendre la syntaxe et la sémantque de OCL avec de nouveaux opérateurs. Ces nouveaux opérateurs devraient être capable d'exprimer des requis de sécurité directement sur les objets et les classes du système UML. L'utilisation des expressions OCL nous permettrait de réaliser des requêtes sur des objets spécifiques. Ceete nouvelle syntaxe et sémantiqueserait une extension des travaux décrits par Bergeron (2004).

Une autre orientation des futurs travaux serait de spécifier le flux d'information pour d'autres fragments UML. On s'intéressera plus précisément aux diagrammes de séquences, d'interactions, de collaboration, d'activités ou de temps. C'est un travail qui s'ali

RÉFÉRENCES

BANIEQBAL, B., BARRINGER, H. et PNUELI, A., éditeurs (1987). *Temporal Logic in Specification*. Springer-Verlag.

BASIN, D., CLAVEL, M., DOSER, J. et EGEA, M. (2009). Automated analysis of security-design models. *Information and software technology*, vol. 51, pp. 815–831.

BELL, D. E. et LAPADULA, L. (1976). Secure computer systems : Unified exposition and multics interpretation. *MITRE technical report, MITRE Corporation, Bedford Massachusetts*, 2997, ref A023 588.

BERGERON, M. (2004). *Model-checking UML designs using a temporal extension of the object constraint language*. Mémoire de maîtrise, École polytechnique de Montréal.

BRUCKER et WOLFF (2002). A proposal for a formal OCL semantics in isabelle/HOL. *IWHOLTP : 15th International Workshop on Higher Order Logic Theorem Proving and Its Applications*. LNCS.

BRUCKER, A. D., DOSER, J. et WOLFF, B. (2006). A model transformation semantics and analysis methodology for secureUML. *Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy, October 1-6, 2006, Proceedings*. Springer, vol. vol. 4199 de *Lecture Notes in Computer Science*, pp. 306–320.

CHENG, L. et ZHANG, Y. (2011). Model checking security policy model using both UML static and dynamic diagrams. M. A. Orgun, A. Elçi, O. B. Makarevich, S. A. Huss, J. Pieprzyk, L. K. Babenko, A. G. Chefranov et R. Shankaran, éditeurs, *Proceedings of the 4th International Conference on Security of Information and Networks, SIN 2011, Sydney, NSW, Australia, November 14-19, 2011*. ACM, 159–166.

JINHUA, L. et JING, L. (2010). Model checking security vulnerabilities in software design. *Wireless Communications Networking and Mobile Computing (WiCOM), 2010 6th International Conference on*. pp. 1–4.

JÜRJENS, J. (2002a). A UML statecharts semantics with message-passing. *SAC*. ACM, 1009–1013.

JÜRJENS, J. (2002b). UMLsec : Extending UML for secure systems development. J.-M. Jézéquel, H. Hußmann et S. Cook, éditeurs, *UML 2002 - The Unified Modeling Language, 5th International Conference, Dresden, Germany, September 30 - October 4, 2002, Proceedings*. Springer, vol. vol. 2460 de *Lecture Notes in Computer Science*, pp. 412–425.

JÜRJENS, J. (2008). Model-based run-time checking of security permissions using guarded objects. M. Leucker, éditeur, *Runtime vérification, 8th International Workshop, RV 2008,*

Budapest, Hungary, March 30, 2008. Selected Papers. Springer, vol. vol. 5289 de *Lecture Notes in Computer Science*, pp .36–50.

JÜRJENS, J., SCHRECK, J. et YU, Y. (2008). Automated analysis of permission-based security using UMLsec. J. L. Fiadeiro et P. Inverardi, éditeurs, *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*. Springer, vol. vol. 4961 de *Lecture Notes in Computer Science*, pp. 292–295.

JÜRJENS, J. et SHABALIN, P. (2007). Tools for secure systems development with UML. *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 9, pp. 527–544.

LATELLA, D., MAJZIK, I. et MASSINK, M. (1999a). Automatic vérification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11, 637–664.

LATELLA, D., MAJZIK, I. et MASSINK, M. (1999b). Towards a formal operational semantics of UML statechart diagrams. *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*. 465.

LILIUS, J. et PALTOR, I. P. (1999). Formalising UML state machines for model checking. R. France et B. Rumpe, éditeurs, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*. Springer, vol. 1723 de *LNCS*, 430–445.

LODDERSTEDT, T., BASIN, D. et DOSER, J. (2002). SecureUML : A UML-based modeling language for model-driven security. *Lecture Notes in Computer Science*, 2460, 426–441.

MYERS et LISKOV (1998). Complete, safe information flow with decentralized labels. *RSP : 19th IEEE Computer Society Symposium on Research in Security and Privacy*. pp. 186–197.

MYERS, A. C. et LISKOV, B. (2000). Protecting privacy using the decentralized label model. *ACM Trans. Softw. Eng. Methodol*, 9, 410–442.

OARGA, R.-M. (2005). *Vérification à la volée de contraintes OCL étendues sur des modèles UML*. Mémoire de maîtrise, École polytechnique de Montréal.

SABELFELD, A. et MYERS, A. C. (2003). Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21, 5–19.

SIVERONI, I., ZISMAN, A. et SPANOUDAKIS, G. (2008). Property specification and static vérification of UML models. *3rd International Conference on Availability, Security, and Reliability (ARES)*. IEEE Computer Society, pp. 96–103.

SIVERONI, I., ZISMAN, A. et SPANOUDAKIS, G. (2010). A UML-based static vérification framework for security. *Requirements Engineering*, vol. 15, pp. 95–118.

VON DER BEECK, M. (2002). A structured operational semantics for UML-statecharts. *Software and System Modeling*, 1, 130–141.

ANNEXE A

Annexe A - Algorithme de restriction sur des étiquettes de sécurité

Cet annexe expose les détails sur les algorithmes de restriction.

Algorithm 4 Calcul $\mathit{isRestrictive}(\mathsf{CLabel}\, l_1,\ \mathsf{CLabel}\, l_2,\ \mathcal{H}\, h)$

Ensure: $\mathit{isRestrictive}(l_1, l_2, h) = \textbf{false} \lor \mathit{isRestrictive}(l_1, l_2, h) = \textbf{true}$
  $\mathit{isRestrictive} \leftarrow$ **false**
  $\mathit{components}_1 \leftarrow \mathsf{Components}(l_1)$
  for all $c_1 \in \mathit{components}_1$ **do**
    $\mathit{isRestrictive} \leftarrow$ **false**
    $\mathit{components}_2 \leftarrow \mathsf{Components}(l_2)$
    for all $c_2 \in \mathit{components}_2$ **do**
      if $\mathit{isRestrictive}(c_1, c_2, h)$ **then**
        $\mathit{isRestrictive} \leftarrow$ **true**
      end if
    end for
  end for
  return isRestrictive

Algorithme de comparaison des étiquettes de confidentialité L'algorithme 4 permet d'évaluer deux étiquettes de confidentialité l_1 et l_2. Si l'étiquette l_1 est plus restrictive que l'étiquette l_2 , le résultat retourné par l'algorithme est *true* (vrai), sinon le résultat est $false$ (faux). L'algorithme utilise l'alogrithme de comparaison des composantes de confidentialité, pour comparer respectivement les composantes de l'étiquette l_1 et les composantes de l'étiquette l_2.

Algorithme de comparaison des étiquettes d'intégrité L'algorithme 5 permet d'évaluer deux étiquettes d'intégrité l_1 et l_2. Si l'étiquette l_1 est plus restrictive que l'étiquette l_2 , le résultat retourné par l'algorithme est *true* (vrai), sinon le résultat est $false$ (faux). L'algorithme utilise l'alogrithme de comparaison des composantes d'intégrité, pour comparer respectivement les composantes de l'étiquette l_1 et les composantes de l'étiquette l_2.

Algorithme de comparaison des étiquettes de sécurité L'algorithme 6 permet d'évaluer deux étiquettes de sécurité l_1 et l_2. Si l'étiquette l_1 est plus restrictive que l'étiquette

Algorithm 5 Calcul $isRestrictive(\mathsf{ILabel}\ l_1,\ \mathsf{ILabel}\ l_2,\ \mathcal{H}\ h)$

Ensure: $isRestrictive(l_1, l_2, h) = \textbf{false} \vee isRestrictive(l_1, l_2, h) = \textbf{true}$
  $isRestrictive \leftarrow \textbf{false}$
  $components_1 \leftarrow \mathsf{Components}(l_1)$
  for all $c_1 \in components_1$ **do**
    $isRestrictive \leftarrow \textbf{false}$
    $components_2 \leftarrow \mathsf{Components}(l_2)$
    for all $c_2 \in components_2$ **do**
      if $isRestrictive(c_1, c_2, h)$ **then**
        $isRestrictive \leftarrow \textbf{true}$
      end if
    end for
  end for
  return isRestrictive

Algorithm 6 Calcul $isRestrictive(\mathsf{Label}\ l_1,\ \mathsf{Label}\ l_2,\ \mathcal{H}\ h)$

Require: $clabel_1, clabel_2 \in \mathsf{CLabel}$, $ilabel_1, ilabel_2 \in \mathsf{ILabel}$
Ensure: $isRestrictive(l_1, l_2, h) = \textbf{false} \vee isRestrictive(l_1, l_2, h) = \textbf{true}$
  $isRestrictive \leftarrow \textbf{false}$
  $clabel_1 \leftarrow \mathsf{CLabel}(l_1)$
  $clabel_2 \leftarrow \mathsf{CLabel}(l_2)$
  $ilabel_1 \leftarrow \mathsf{ILabel}(l_1)$
  $ilabel_1 \leftarrow \mathsf{ILabel}(l_2)$
  return $isRestrictive(clabel_1, clabel_2, h) \wedge isRestrictive(ilabel_2, ilabel_1, h)$

l_2 , le résultat retourné par l'algorithme est $true$ (vrai), sinon le résultat est $false$ (faux). Cet algorithme utilise les algorithmes 4 et 5 qui comparent respectivement les étiquettes de confidentialité et des étiquettes d'intégrité.

Algorithm 7 Calcul $isRestrictive(\mathsf{CComponents}\ c_1,\ \mathsf{CComponents}\ c_2,\ \mathcal{H}\ h)$

Ensure: $isRestrictive(c_1, c_2, h) = \textbf{false} \vee isRestrictive(c_1, c_2, h) = \textbf{true}$
  $readers_1 \leftarrow \mathsf{readers}(c_1)$
  $actFor \leftarrow \textbf{false}$
  for all $r_1 \in readers_1$ **do**
    $readers_2 \leftarrow \mathsf{readers}(c_2)$
    $actFor \leftarrow \textbf{false}$
    for all $r_2 \in readers_2$ **do**
      if $actFor(r_2, r_1, h)$ **then**
        $actFor \leftarrow \textbf{true}$
      end if
    end for
  end for
  return $actFor(owners^{\star}(c_2), owners^{\star}(c_1), h) \wedge \mathsf{readers}^{\star}(c_2) \subseteq \mathsf{readers}^{\star}(c_1) \wedge actFor$

Algorithme de comparaison des composantes de confidentialité L'algorithme 7 permet d'évaluer deux composantes de confidentialité c_1 et c_2. Si la composante c_1 est plus restrictive que la composante c_2, le résultat retourné par l'algorithme est $true$ (vrai), sinon le résultat est $false$ (faux). Cet algorithme utilise l'alogrithme $actFor$ (algorithme 9)qui permet de vérifier si la paire (r_1, r_2) appartient à une hiérarchie particulière, avec $r_1, r_2 \in \mathsf{Old}$.

Algorithme de comparaison des composantes d'intégrité L'algorithme 8 permet d'évaluer deux composantes d'intégrité i_1 et i_2. Si la composante i_1 est plus restrictive que la composante i_2, le résultat retourné par l'algorithme est $true$ (vrai), sinon le résultat est $false$ (faux). Cet algorithme utilise l'alogrithme $actFor$ (algorithme 9)qui permet de vérifier si la paire (r_1, r_2) appartient à une hiérarchie particulière, avec $r_1, r_2 \in \mathsf{Old}$.

Algorithme d'évaluation de la hiérarchie L'algorithme 9 permet d'évaluer deux objets o_1 et o_2. Si la composante (o_1, o_2) appartient à la hiérarchie h, le résultat retourné par l'algorithme est $true$ (vrai), sinon le résultat est $false$ (faux).

Algorithm 8 Calcul $isRestrictive(\mathsf{IComponents}\ i_1,\ \mathsf{IComponents}\ i_2),\ \mathcal{H}\ h$

Require: $h \in \propto$
Ensure: $isRestrictive(i_1, i_2, h) = \mathbf{false} \vee isRestrictive(i_1, i_2, h) = \mathbf{true}$
  $writers_1 \leftarrow \mathsf{readers}(i_1)$
  $actFor \leftarrow \mathbf{false}$
  for all $r_1 \in writers_1$ **do**
    $writers_2 \leftarrow \mathsf{readers}(i_2)$
    $actFor \leftarrow \mathbf{false}$
    for all $r_2 \in writers_2$ **do**
      if $actFor(r_2, r_1, h)$ **then**
        $actFor \leftarrow \mathbf{true}$
      end if
    end for
  end for
  return $actFor(owners^\star(i_2), owners^\star(i_1), h) \wedge \mathsf{writers}^\star(i_2) \subseteq \mathsf{writers}^\star(i_1) \wedge actFor$

Algorithm 9 Calcul $actFor(\mathsf{OId}\ o_1,\ \mathsf{OId}\ o_2,\ \mathcal{H}\ h)$

Ensure: $actFor(o_1, o_2, h) = \mathbf{false} \vee actFor(o_1, o_2, h) = \mathbf{true}$
  if $(o_1, o_2) \in h$ **then**
    return **true**
  else
    return **false**
  end if

ANNEXE B

Annexe B - Validations et contraintes sur des diagrammes UML

Cet annexe expose les détails sur le modèle formel des diagrammes UML.

Diagrammes de classe

Fonctions sur les données La fonction value associe à une donnée $d \in \mathsf{Data}$, $d = (nd, val, L_d)$ sa valeur.

$$\mathsf{value} : \mathsf{Data} \longrightarrow \mathsf{Val}$$
$$\mathsf{value}(d) = val$$

La fonction label associe à une donnée son étiquette de sécurité.

$$\mathsf{label} : \mathsf{Data} \longrightarrow \mathsf{Label}$$
$$\mathsf{label}(d) = L_d$$

La fonction name associe à une donnée son étiquette de sécurité.

$$\mathsf{name} : \mathsf{Data} \longrightarrow \mathsf{NVar} \cup \mathsf{NPar} \cup \mathsf{NVar}$$
$$\mathsf{name}(d) = nd$$

Fonctions sur les méthodes La fonction $\mathsf{parameters}$ associe à une méthode $m \in \mathsf{Meths}, m = (nm, mpars, mreturn)$, (ou une instance de méthode $mid \in \mathsf{MId}, mid = (o, m, nmid)$) les paramètres formels de cette dernière.

$$\mathsf{parameters} : \mathsf{Meths} \longrightarrow \mathcal{P}(\mathsf{Pars})$$
$$\mathsf{parameters}(m) = mpars$$

La fonction return associe à une méthode (ou instance de méthode) le paramètre de retour de cette dernière.

$$\mathsf{return} : \mathsf{Meths} \longrightarrow \mathsf{Pars}$$
$$\mathsf{return}(m) = mreturn$$

La fonction name associe à une méthode (ou instance de méthode) son nom.

$$\mathsf{name} : \mathsf{Meths} \longrightarrow \mathsf{Pars}$$
$$\mathsf{name}(m) = nm$$

Fonctions sur les diagrammes de classes Les fonctions suivantes sont utilisées sur une classe $c \in \mathsf{Class}$, $c = (nc, CAtts, CMeths)$:

$$\begin{aligned}&\mathsf{attributes} \; : \; \mathsf{Class} \longrightarrow \mathcal{P}(\mathsf{Atts})\\&\mathsf{attributes}(c) = CAtts\end{aligned}$$

Les fonctions suivantes sont utilisées pour déterminer l'ensemble des méthodes d'une classe :

$$\begin{aligned}&\mathsf{methods} \; : \; \mathsf{Class} \longrightarrow \mathcal{P}(\mathsf{Meths})\\&\mathsf{methods}(c) = CMeths\end{aligned}$$

Les fonctions suivantes sont utilisées pour déterminer le nom d'une classe :

$$\begin{aligned}&\mathsf{name} \; : \; \mathsf{Class} \longrightarrow \mathsf{NClass}\\&\mathsf{name}(c) = nc\end{aligned}$$

Diagrammes d'objets

Fonctions sur les diagrammes d'objets La fonction suivante est utilisée sur objet $o \in \mathsf{OId}$, $o = (no, c)$

$$\begin{aligned}&\mathsf{attributes}^\star \; : \; \mathsf{OId} \longrightarrow \mathcal{P}(\mathsf{Atts} \times \mathsf{NOId})\\&\mathsf{attributes}^\star(o) = \{(a, no) \; : \; a \in \mathsf{attributes}(c), \; no \in \mathsf{NOId}\}\end{aligned}$$

Les fonctions suivantes sont utilisées sur les diagrammes d'objets :

$$\begin{aligned}&\mathsf{name}^\star \; : \; \mathsf{OId} \longrightarrow \mathsf{NOId}\\&\mathsf{name}^\star(o) = no\end{aligned}$$

Les fonctions suivantes sont utilisées sur les diagrammes d'objets :

$$\begin{aligned}&\mathsf{instanceOf} \; : \; \mathsf{OId} \; \longrightarrow \; \mathsf{Class}\\&\mathsf{instanceOf}(o) = c\end{aligned}$$

La fonction suivante est utilisée sur objet $o \in \mathsf{OId}$, $o = (no, c)$, pour déterminer l'ensemble des isntances de méthodes

$$\begin{aligned}&\mathsf{methods}^\star \; : \; \mathsf{OId} \longrightarrow \mathcal{P}(\mathsf{MId})\\&\mathsf{methods}^\star(o) = \textstyle\bigcup_{m \in \mathsf{methods}(c)} \{(o, m, nmid)\}\end{aligned}$$

Les fonctions suivantes sont utilisées sur une instance de méthode $mid \in \mathsf{MId}$, $mid =$

$(o, m, nmid)$

$$\mathsf{methods}^{\star} \;:\; \mathsf{OId} \longrightarrow \mathcal{P}(\mathsf{MId})$$
$$\mathsf{methods}^{\star}(o) =$$

Diagrammes d'états

Fonctions sur les regions On définit la fonction regionStates qui associe à une région $r \in \mathsf{Regions}, r = (nr, RStates, entry, exit)$, tous les états directs qu'elle contient :

$$\mathsf{regionStates} \;:\; \mathsf{Regions} \longrightarrow \mathcal{P}(\mathsf{States})$$
$$\mathsf{regionStates}(r) = RStates$$

On définit la fonction name qui associe à une région $r \in \mathsf{Regions}$,
$r = (nr, RStates, entry, exit)$, le nom de la région :

$$\mathsf{name} \;:\; \mathsf{Regions} \longrightarrow \mathsf{NRegion}$$
$$\mathsf{name}(r) = nr$$

On définit la fonction entry qui associe à une région $r \in \mathsf{Regions}$,
$r = (nr, RStates, entry, exit)$, la séquence d'actions qui est exécutées lorsqu'on rentre dans la région :

$$\mathsf{entry} \;:\; \mathsf{Regions} \longrightarrow \mathsf{SActions}$$
$$\mathsf{entry}(r) = entry$$

On définit la fonction exit qui associe à une région $r \in \mathsf{Regions}$,
$r = (nr, RStates, entry, exit)$, la séquence d'actions qui est exécutées lorsqu'on rentre dans la région :

$$\mathsf{exit} \;:\; \mathsf{Regions} \longrightarrow \mathsf{SActions}$$
$$\mathsf{exit}(r) = exit$$

On définit la fonction defaultState qui associe à une région $r \in \mathsf{Regions}$,
$r = (nr, RStates, entry, exit)$, son état par défaut :

$$\mathsf{defaultState} \;:\; \mathsf{Regions} \longrightarrow \mathsf{States}$$
$$\mathsf{defaultState}(r) \;=\; \begin{cases} \star, \text{si } \mathsf{regionStates}(r) = \emptyset \\ s, \text{ si } \exists s, s' \in \mathsf{regionStates}(r), \exists t \in \mathsf{Trans},\ \mathsf{srcState}(t) = s' \wedge \mathsf{stype}(s') = \mathsf{BEGIN} \end{cases}$$

Fonctions sur les états On définit la fonction name qui associe à un état, $s \in \mathsf{States}$, $s = (tstate, ns, regions, entry, exit)$

$$\begin{aligned} &\mathsf{name} : \mathsf{States} \longrightarrow \mathsf{NState} \\ &\mathsf{name}(s) = ns \end{aligned}$$

On définit la fonction stype qui associe à un état, $s \in \mathsf{States}$, $s = (tstate, ns, regions, entry, exit)$

$$\begin{aligned} &\mathsf{stype} : \mathsf{States} \longrightarrow \mathsf{TType} \\ &\mathsf{stype}(s) = tstate \end{aligned}$$

On définit la fonction $\mathsf{regions}$ qui associe à un état, $s \in \mathsf{States}$, $s = (tstate, ns, regions, entry, exit)$

$$\begin{aligned} &\mathsf{regions} : \mathsf{States} \longrightarrow \mathsf{Regions} \\ &\mathsf{regions}(s) = regions \end{aligned}$$

On définit la fonction entry qui associe à un état, $s \in \mathsf{States}$, $s = (tstate, ns, regions, entry, exit)$

$$\begin{aligned} &\mathsf{entry} : \mathsf{States} \longrightarrow \mathsf{SActions} \\ &\mathsf{entry}(s) = entry \end{aligned}$$

On définit la fonction exit qui associe à un état, $s \in \mathsf{States}$, $s = (tstate, ns, regions, entry, exit)$

$$\begin{aligned} &\mathsf{exit} : \mathsf{States} \longrightarrow \mathsf{SActions} \\ &\mathsf{exit}(s) = exit \end{aligned}$$

On définit la fonction $\mathsf{subStates}$ qui associe à un état, $s \in \mathsf{States}$, $s = (tstate, ns, regions, entry, exit)$ l'ensemble de ses sous-états.

$$\begin{aligned} &\mathsf{subStates} : \mathsf{States} \longrightarrow \mathcal{P}(\mathsf{States}) \\ &\mathsf{subStates}(s) = \begin{cases} \emptyset, \quad \text{si } \mathsf{regions}(s) = \emptyset \vee \mathsf{stype}(s) = \mathsf{END} \vee \mathsf{stype}(s) = \mathsf{BEGIN} \vee \mathsf{stype}(s) = \mathsf{SIMPLE} \\ \bigcup_{r \in \mathsf{regions}(s)} (\bigcup_{st \in \mathsf{regionStates}(r)} \mathsf{subStates}(st)) \quad \text{sinon} \end{cases} \end{aligned}$$

On définit la fonction $\mathsf{superStates}$ qui associe à un état d'un diagramme de statecharts

tous ses états parents (ensemble de tous ses ancêtres).

$$\mathsf{superStates} : \mathsf{States} \longrightarrow \mathcal{P}(\mathsf{States})$$
$$\mathsf{superStates}(s) = \{s' : s' \in \mathsf{States},\ s \in \mathsf{subStates}(s')\}$$

On définit la fonction defaultStates qui associe à un état d'un diagramme l'ensemble de tous les états par défaut de ses régions.

$$\mathsf{defaultStates} : \mathsf{States} \longrightarrow \mathcal{P}(\mathsf{States})$$
$$\mathsf{defaultStates}(s) = \begin{cases} \emptyset \quad \text{si } \mathsf{stype}(s) = \mathsf{SIMPLE} \vee \mathsf{stype}(s) = \mathsf{END} \vee \mathsf{stype}(s) = \mathsf{BEGIN} \\ \mathcal{D} \cup (\ \bigcup_{s' \in \mathcal{D}} \mathsf{defaultStates}(s')) \quad \text{sinon} \\ \text{avec } \mathcal{D} = \bigcup_{r \in \mathsf{regions}(s)} \{\mathsf{defaultState}(r)\} \end{cases}$$

Fonctions sur les déclencheurs On définit la fonction ttype qui associe à un déclencheur $tr \in \mathsf{Triggers}$,
$tr = (ntrigger, ttype, event)$ le type de déclencheur :

$$\mathsf{ttype} : \mathsf{Triggers} \longrightarrow \mathsf{TType}$$
$$\mathsf{ttype}(tr) = ttype$$

On définit la fonction name qui associe à un déclencheur $tr \in \mathsf{NTriggers}$,
$tr = (ntrigger, ttype, event)$ le nom de déclencheur :

$$\mathsf{name} : \mathsf{Triggers} \longrightarrow \mathsf{NTriggers}$$
$$\mathsf{name}(tr) = ttype$$

On définit la fonction event qui associe à un déclencheur $tr \in \mathsf{NTriggers}$,
$tr = (ntrigger, ttype, event)$ le nom de déclencheur :

$$\mathsf{event} : \mathsf{Triggers} \longrightarrow \mathsf{MId}$$
$$\mathsf{event}(tr) = ttype$$

Fonctions sur les actions On définit la fonction name qui associe à une action $act \in \mathsf{Actions}$,
$act = (na, atype, sndr, rcvr, aParams)$ le nom de l'action :

$$\mathsf{name} : \mathsf{Actions} \longrightarrow \mathsf{NActions}$$
$$\mathsf{name}(act) = na$$

On définit la fonction atype qui associe à une action $act \in \mathsf{Actions}$,

$act = (na, atype, sndr, rcvr, aParams)$ le nom de l'action :

$$\mathsf{atype} \;:\; \mathsf{Actions} \longrightarrow \mathsf{AType}$$
$$\mathsf{atype}(act) = atype$$

On définit la fonction receiver qui associe à une action $act \in \mathsf{Actions}$,
$act = (na, atype, sndr, rcvr, aParams)$ le nom de l'action :

$$\mathsf{receiver} \;:\; \mathsf{Actions} \longrightarrow \mathsf{OId}$$
$$\mathsf{receiver}(act) = rcvr$$

On définit la fonction sender qui associe à une action $act \in \mathsf{Actions}$,
$act = (na, atype, sndr, rcvr, aParams)$ le nom de l'action :

$$\mathsf{sender} \;:\; \mathsf{Actions} \longrightarrow \mathsf{OId}$$
$$\mathsf{sender}(act) = sndr$$

On définit la fonction aParams qui associe à une action $act \in \mathsf{Actions}$,
$act = (na, atype, sndr, rcvr, aParams)$ le nom de l'action :

$$\mathsf{aParams} \;:\; \mathsf{Actions} \longrightarrow \mathsf{AParams}$$
$$\mathsf{aParams}(act) = aParams$$

Fonctions sur les transitions On définit la fonction name qui associe à une transition $t \in \mathsf{Trans}$,
$t = (nt, trigger, guard, sactions, tgt, src)$ le nom de la transition :

$$\mathsf{name} \;:\; \mathsf{Trans} \longrightarrow \mathsf{NTrans}$$
$$\mathsf{name}(t) = nt$$

On définit la fonction triggers qui associe à une transition $t \in \mathsf{Trans}$,
$t = (nt, trigger, guard, sactions, tgt, src)$ le déclencheur de la transition :

$$\mathsf{triggers} \;:\; \mathsf{Trans} \longrightarrow \mathsf{Triggers}$$
$$\mathsf{triggers}(t) = trigger$$

On définit la fonction actions qui associe à une transition $t \in \mathsf{Trans}$,

$t = (nt, trigger, guard, sactions, tgt, src)$ la séquence d'action exécutée par la transition :

$$\mathsf{actions} : \mathsf{Trans} \longrightarrow \mathsf{SActions}$$
$$\mathsf{actions}(t) = trigger$$

On définit la fonction guards qui associe à une transition $t \in \mathsf{Trans}$, $t = (nt, trigger, guard, sactions, tgt, src)$ la garde de l'action la transition :

$$\mathsf{guards} : \mathsf{Trans} \longrightarrow \mathsf{Guards}$$
$$\mathsf{guards}(t) = guard$$

On définit la fonction $\mathsf{tgtState}$ qui associe à une transition $t \in \mathsf{Trans}$, $t = (nt, trigger, guard, sactions, tgt, src)$ l'état destiantion la transition :

$$\mathsf{tgtState} : \mathsf{Trans} \longrightarrow \mathsf{States}$$
$$\mathsf{tgtState}(t) = tgt$$

On définit la fonction $\mathsf{srcState}$ qui associe à une transition $t \in \mathsf{Trans}$, $t = (nt, trigger, guard, sactions, tgt, src)$ l'état de départ la transition :

$$\mathsf{srcState} : \mathsf{Trans} \longrightarrow \mathsf{States}$$
$$\mathsf{srcState}(t) = src$$