Description de propriétés de sécurité pour un processus de validation/vérification

Thierry Moutet

Conservatoire National des Arts et Métiers

Centre régional Rhône-Alpes

Centre d'Enseignement de GRENOBLE

MEMOIRE

présenté par Thierry Moutet

en vue d'obtenir

Le Diplôme d'Ingénieur C.N.A.M.
en Informatique

Description de propriétés de sécurité pour un processus de validation/vérification

Soutenu le 30 juin 2010

Jury

Président : M. Eric Gressier-Soudan

Membres : M. Jean-Pierre Giraudin

M. André Plisson

M. Mathias Voisin-Fradin

Mme Marie-Laure Potet

M. Philippe Lassaut

Conservatoire National des Arts et Métiers

Centre régional Rhône-Alpes

Centre d'Enseignement de GRENOBLE

MEMOIRE

présenté par Thierry Moutet

en vue d'obtenir

Le Diplôme d'Ingénieur C.N.A.M.
en Informatique

Description de propriétés de sécurité pour un processus de validation/vérification

Soutenu le 30 juin 2010

Les travaux relatifs au présent mémoire ont été effectués sous la direction de Marie-Laure Potet, au sein de l'équipe de recherche VASCO (VAlidation, Spécification et COnstruction de logiciels), dirigée par Yves Ledru. L'équipe VASCO fait partie de l'axe "Logiciels" du LIG (Laboratoire d'Informatique de Grenoble), UMR (Unité Mixte de Recherche) de l'INRIA, du CNRS, de l'INPG et de l'UJF.
Le LIG regroupe plusieurs équipes autour des quatre domaines suivants : les infrastructures informatiques, les logiciels, les interactions et les connaissances.

Remerciements

Le diplôme d'ingénieur CNAM est souvent une longue épreuve. Pour moi, elle représente l'aboutissement de plus de six ans de formation au CNAM. Entreprendre une telle formation en plus d'une vie professionnelle et familiale n'est pas chose facile. Aussi, un tel parcours n'est pas seulement l'œuvre d'une personne, nombreuses sont celles qui ont contribué à sa réussite.

Je tiens tout d'abord à remercier Madame Marie-Laure Potet, professeur ensimag et tuteur pour ce stage. D'une part, elle m'a non seulement accueilli dans son équipe en me proposant ce sujet mais elle m'a également accordé beaucoup de temps à répondre à mes questions et à me donner de précieux conseils. D'autre part, elle m'a permis de vivre pleinement cette année au sein de son équipe en m'offrant la possibilité de publier à plusieurs reprises les travaux de mes recherches ainsi qu'en me permettant de participer à des conférences scientifiques. Par son engagement, elle a su me passionner pour un domaine qui m'était jusqu'alors inconnu.

Mes remerciements vont également aux membres de l'équipe VASCO. Ils s'adressent notamment à Monsieur Didier Bert, chercheur CNRS, qui a toujours été présent pour m'aider aussi bien sur l'utilisation de la méthode B que dans la mise en œuvre de la bibliothèque java BOB. Ils sont dédiés également à Frédéric et à Amal, respectivement post-doctorant et doctorante, avec qui j'ai réalisé plusieurs publications.

Mes remerciements vont aussi à mes camarades de l'AIPST, l'Association des Ingénieurs de la Promotion Supérieure du Travail-CUEFA/CNAM. Je voudrais d'abord dire un grand merci à son président Eric, grâce à qui j'ai pu réaliser ce rapport au moyen des outils LaTeX sous Linux. Merci encore à toutes les personnes qui ont lu ce rapport et m'ont aidé à le corriger mais aussi à celles qui m'ont entraîné pour la soutenance.

Je tiens à exprimer ma reconnaissance à toutes les personnes du CNAM pour leur dévouement tout au long de ces années, entre autres Monsieur André Plisson, directeur du centre d'enseignement de Grenoble et Monsieur Jean-Pierre Giraudin, professeur à l'Université Pierre-Mendes France de Grenoble et responsable du cycle ingénieur CNAM en informatique à Grenoble. Leur aide précieuse a fortement contribué à la réussite de mon parcours.

Je remercie Monsieur Eric Gressier, professeur au CNAM de Paris, qui me fait l'honneur de présider le jury, ainsi que Monsieur Philippe Lassaut, Ingénieur à Traciel, la société où je travaille, également membre de ce jury.

Je remercie enfin ma famille et mes amis pour leur soutien sans faille durant toutes ces années. Je tiens particulièrement à exprimer ma sincère gratitude à ma femme. Elle a toujours su, par sa patience et son amour, m'aider à trouver la force de continuer dans les moments difficiles.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Contexte : la sécurité des systèmes informatiques

La sécurité informatique est le souci majeur des développeurs de logiciels critiques et des utilisateurs. Par sécurité, on entend généralement cinq propriétés essentielles : l'*authentification*, qui permet de s'assurer de l'identité d'une entité donnée ou de l'origine d'une communication ou d'un fichier, la *confidentialité*, qui empêche la divulgation d'une information tenue secrète, l'*intégrité*, qui vise à garantir qu'une information ne sera pas modifiée sans en avoir eu l'authorisation, la *disponibilité*, qui garantit l'accès aux informations pour les utilisateurs autorisés, et la *non-répudiation* qui empêche le reniement d'actions ou de messages anciens.

Pour garantir la sécurité, plusieurs techniques existent dont le *chiffrement*, l'*authentification* et le *contrôle d'accès*. Dans l'étude qui est présentée ici, nous nous sommes focalisés sur le contrôle d'accès. Cette technique vise à intercepter toutes les tentatives d'accès aux informations critiques. Un contrôle d'accès se définit à différents niveaux de conception : les politiques de sécurité définissent les règles de haut niveau qui régissent les accès, les modèles de sécurité dressent une représentation formelle des politiques de sécurité et enfin les mécanismes de sécurité définissent les fonctions bas niveau (logicielles et matérielles) permettant d'implanter les contrôles imposés par la politique.

Dans l'objectif d'évaluer la sécurité, les *Critères Communs* (CC) ont récemment été introduit. Il s'agit d'une norme (ISO 15408) qui valide la sécurité d'un système du point de vue logiciel et matériel. Les critères communs définissent une cible de sécurité (TOE - Target Of Evaluation) qui doit présenter un certain nombre d'exigences se répartissant en deux catégories : les exigences fonctionnelles de sécurité et les exigences d'assurance de sécurité. Les premières évaluent les mécanismes déployés pour garantir la sécurité, tandis que les secondes visent à garantir la bonne mise en œuvre de ces mécanismes et leur adéquation à la cible de sécurité [CC006a].

La problématique : le contrôle d'accès appliqué aux technologies multi-applicatives des cartes à puces

Dans le domaine des cartes à puce, la spécification et le développement des applications embarquées sont toujours des phases délicates. La validation du contrôle d'accès représente

souvent une lourde étape qui nécessite l'application de nombreux tests et l'utilisation de beaucoup de temps. Si ces tests permettent d'assurer l'obtention d'un produit possédant un très haut degré de sécurité, c'est au prix d'un coût élevé aussi bien pour la fabrication du produit (i.e. puce sécurisée, OS sécurisé, application sécurisée) que pour l'évaluation et la certification de ces différents éléments. Les travaux présentés dans cette étude s'inscrivent dans le cadre du projet RNTL POSÉ[1]. L'objet de ce projet est de réduire ces coûts et d'améliorer encore la sécurité des produits cartes à puce en intégrant les besoins en sécurité le plus en amont possible du cycle de développement, c'est-à-dire dès les phases de conception. L'objectif était aussi de fournir une démarche d'automatisation de ces processus de validation de la sécurité qui puisse s'intégrer dans les développements futurs.

L'implication de ce stage vise d'une part à décrire des propriétés de sécurité pour un processus de validation et de vérification sur des applications cartes à puces de type JavaCard et dans une démarche de certification de haut niveau aux Critères Communs. La description des propriétés de sécurité se fera au moyen de la méthode B qui est une méthode de spécification formelle. D'autre part, l'objectif fondamental de ce stage est d'automatiser cette démarche de validation et de vérification, en développant un outil permettant, à partir de ces modèles de description de politiques de sécurité, de produire automatiquement un moniteur de sécurité visant à garantir le respect de ces politiques ainsi définies.

Organisation de ce rapport

Afin de simplifier la lecture de ce document, un découpage logique a été réalisé. Le but est de proposer des parties indépendantes afin de fournir des accès directs selon l'intérêt de chacun.

Le prochain chapitre présente le contexte global du stage, faisant apparaître les thèmes de recherche de l'équipe d'accueil. Les grandes lignes du projet POSÉ sont également exposées.

Les trois chapitres qui suivent, dressent un état de l'art des normes et technologies utilisées dans le projet. Le chapitre 3 s'intéresse à la technologie JavaCard en partant du monde de la carte à puce pour arriver aux spécificité des cartes utilisant cette technologie. Les recherches effectuées sur ce thème ont constitué une grande partie de mon travail et m'ont valu plusieurs implications dans des projets parallèles qui sont abordées au chapitre 10. La norme des Critères Communs, qui a été le fil conducteur des réalisations que j'ai pu faire, est abordée dans le chapitre 4. La méthode B, qui a été la base pour permettre une formalisation de haut niveau guidée par les Critères Communs, est ensuite introduite dans le chapitre 5.

Après avoir posé le contexte technologique de mon projet, les réalisations qui ont été effectuées sont développées dans les quatre chapitres suivants. Les chapitre 6, 7 et 8 traitent de la partie modélisation des politiques de sécurité alors que le chapitre 9 détaille le développement de l'outil Meca.

[1] Le projet POSÉ est financé par l'ANR et est référencé sous le numéro ANR-05-RNTL-01001 : http ://www.rntl-pose.info. Dernière consultation : 06-jun-07

Enfin, le chapitre 10 est axé sur l'organisation et la gestion du projet. Il essaie de faire apparaître la conformité entre les prévisions initiales et ce qui a été effectivement réalisé en montrant les écarts qu'il a pu y avoir. C'est également l'occasion de présenter les implications diverses auxquelles j'ai pu participer avant de dresser dans la conclusion un bilan de synthèse de cette année de stage.

Première partie

Contexte et Etat de l'art

Chapitre 2

Contexte du stage

Ce mémoire est le fruit d'un stage d'ingénieur CNAM, réalisé au sein de l'équipe de recherche VASCO du laboratoire LIG. Ce stage s'est déroulé dans le cadre du projet RNTL POSÉ relatif au test de conformité de politiques de sécurité de systèmes enfouis. Il est essentiellement orienté vers des systèmes cartes à puce. Ce chapitre a pour but de présenter d'une part le cadre de travail de ce stage, qu'il s'agisse aussi bien du laboratoire que de l'équipe de recherche dans laquelle il a été effectué. D'autre part, le contexte du projet POSÉ est également abordé.

2.1 Le LIG

Le LIG[1] est un Laboratoire public de recherches qui est né en janvier 2007 de la fusion de plusieurs laboratoires de l'agglomération grenobloise travaillant sur des thématiques relatives aux logiciels. Il regroupe notamment plusieurs laboratoires qui faisaient jusqu'alors partie de la fédération IMAG[2]. Il est placé sous la tutelle du CNRS[3], de l'INPG[4], de l'INRIA[5], de l'UJF[6], et de l'UPMF[7].

FIG. 2.1 – Le Laboratoire d'Informatique de Grenoble

[1] Laboratoire d'Informatique de Grenoble : http ://lig.imag.fr. Dernière consultation : 03-jun-07

[2] Institut d'Informatique et de Mathématiques Appliquées de Grenoble : http ://www.imag.fr. Dernière consultation : 03-jun-07

[3] Centre National de la Recherche Scientifique : http ://www.cnrs.fr. Dernière consultation : 03-jun-07

[4] Institut National Polytechnique de Grenoble : http ://www.inpg.fr. Dernière consultation : 03-jun-07

[5] Institut National de Recherche en Informatique et en Automatique : http ://www.inria.fr. Dernière consultation : 03-jun-07

[6] Université Joseph Fourier : http ://www.ujf-grenoble.fr. Dernière consultation : 03-jun-07

[7] Université Pierre Mendès France : http ://www.upmf-grenoble.fr. Dernière consultation : 03-jun-07

L'existence du LIG est intervenue dans un contexte ou les problématiques et fondements des logiciels ont changé de complexité. Ceux-ci sont devenus de plus en plus dynamiques et portent très souvent sur des systèmes répartis et hétérogènes qui interagissent entre eux mais aussi avec des êtres humains. Ainsi, il devenait nécessaire de faire collaborer des compétences plus larges afin de pouvoir construire de nouvelles modalités.

FIG. 2.2 – Les organismes de tutelle du LIG

Désormais, le LIG regroupe 24 équipes de recherche qui sont structurées autour des quatre thématiques suivantes. La première concerne les infrastructures informatiques et couvre essentiellement les réseaux. La seconde rassemble de multiples compétences autour des interactions au sens large. Ainsi, elle comprend des équipes travaillant sur la perception, la cognition, les interaction homme-machine, l'ergonomie, la réalité virtuelle, la robotique mais encore le dialogue, la parole ou la langue. La troisième, quant-à elle, s'intéresse à la connaissance. Ces dernières années, les technologies de l'information ont permis de produire de très grandes quantités de ressources, rendant l'information complexe et hétérogène. Ainsi, le traitement et l'interprétation de la connaissance nécessite la mise en œuvre de techniques d'extraction ou de représentation de ces connaissances afin de les rendre exploitables. Ce sont ces techniques qu'étudient les équipes de cette thématique. Enfin, la quatrième thématique concerne les logiciels à proprement dit. Les équipes qu'elle rassemble travaillent sur leurs fondements et sur leur ingénierie en utilisant notamment des modèles. C'est sur ce dernier thème que travaille l'équipe VASCO dans laquelle j'ai réalisé mon stage.

2.2 L'équipe VASCO

L'équipe VASCO[8] effectue des recherches qui portent sur la validation des logiciels à l'aide de modèles. Cette validation peut intervenir à divers stades du développement, allant de la validation de code existant jusqu'à la production rigoureuse de logiciels à partir de spécifications. Plusieurs approches sont étudiées. Elles reposent toutes sur des modèles formels ou semi-formels fournissant des descriptions du logiciel proprement dit (fonctionnalités, sémantique) ou de certaines caractéristiques non fonctionnelles de son comportement. Le choix d'utiliser des modèles qui ont un certain niveau de formalité est justifié par la perspective d'automatiser certaines activités de développement ou de validation.

Les activités de l'équipe s'organisent principalement selon les trois axes suivants :

[8] VAlidation, Spécification et COnstruction de logiciels : http ://www-lsr.imag.fr/Les.Groupes/VASCO/. Dernière consultation : 04-jun-07

- La construction prouvée de programmes et de systèmes.
- La validation des logiciels par des techniques de test.
- La construction de modèles pour l'expression et la vérification de propriétés de sécurité.

2.2.1 La construction prouvée de programmes et de systèmes

Dans certains domaines d'application pour lesquels il existe de fortes exigences en terme de sûreté de fonctionnement du logiciel, l'utilisation des techniques formelles dans l'industrie est maintenant effective, comme décrit dans les rapports commandés par le NIST [CGR93] et par l'agence fédérale allemande pour la sécurité de l'information [BCK+00]. Ces utilisations sont motivées par des besoins en vérification mais aussi pour des raisons d'efficacité et de rationalisation du processus de développement. En effet, les méthodes formelles permettent d'assurer la traçabilité des exigences jusqu'au code, et surtout d'utiliser des outils engendrant du code à partir de descriptions de haut niveau. Notons que les besoins ne portent généralement pas sur l'application en son entier, ni sur tous les aspects d'une application. C'est par exemple le cas de la sûreté de fonctionnement qui ne porte que sur le sous-ensemble bien isolé des fonctions critiques.

L'équipe VASCO expérimente des méthodes formelles qui sont essentiellement orientées « modèles » et basées sur la logique du premier ordre avec la théorie des ensembles : Z et B, mais aussi des logiques constructives comme Coq. Ces méthodes offrent l'avantage de pouvoir donner lieu à des études qui conduisent à des outils. Cependant, devant la difficulté d'utiliser les notations formelles comme support d'échange entre communautés différentes (informaticiens, donneurs d'ordre, spécialistes-métier), d'autres représentations plus accessibles sont également étudiées. Celles-ci sont essentiellement à base de diagrammes, comme la notation UML qui est largement répandue et standardisée.

2.2.2 La validation des logiciels par des techniques de test

Dans le cadre de cette activité, les spécifications sont utilisées comme point de départ du processus de génération des tests. Elles sont vues ici comme des modèles comportementaux associés soit au logiciel lui-même (spécification fonctionnelle, spécification de propriétés non fonctionnelles), soit à son environnement (utilisateur, environnement physique). Ces activités concernent principalement l'outil Lutess, qui exploite des modèles synchrones, pour la génération de cas de tests. Les travaux portent sur l'évolution de l'outil, et sur son application à divers domaines (recherche d'interactions, aéronautique, interfaces homme-machine).

Les spécifications sont également utilisées comme oracle pour des tests produits indépendamment de celles-ci. Il s'agit principalement de spécifications en JML (Java Modeling Language) pour des programmes Java. La génération de données de test s'appuie sur une modélisation de l'utilisateur implantée par des stratégies de test (aléatoire, combinatoire, guidé par des propriétés ...). Deux outils (TOBIAS[9] et Jartège[10]) ont été réalisés dans le cadre de

[9] Un outil pour le test combinatoire : http ://www-lsr.imag.fr/Les.Groupes/PFL/Tobias. Dernière consultation : 04-jun-07

[10] Un outil de génération automatique aléatoire de programmes de test pour des classes Java : http ://www-lsr.imag.fr/Les.Personnes/Catherine.Oriat/jartege.html. Dernière consultation : 04-jun-07

ces études.

2.2.3 Construction de modèles pour l'expression et la vérification de propriétés de sécurité

Avec la généralisation des traitements informatiques au cœur de la plupart des systèmes et des activités humaines, la sécurité et la sûreté des systèmes informatiques sont devenus des enjeux de société majeurs. Les deux autres thèmes abordés par l'équipe VASCO concernent la sûreté des systèmes. Le troisième thème est consacré à des aspects de la sécurité, dans le sens de la protection des systèmes informatiques contre des erreurs ou des actions malveillantes.

Au sein de la problématique de la sécurité informatique, l'équipe VASCO s'intéresse à l'application de techniques issues du génie logiciel à base formelle pour la spécification, la validation et la construction de systèmes assurant la sécurité. Plus précisément, elle base son approche sur des modèles de politiques de sécurité, qui permettent de représenter les exigences ou les règles de sécurité appliquées au sein de systèmes ou d'organisations. Bien entendu, les démarches recouvrent celles des autres thèmes de l'équipe.

Les travaux de l'équipe sur ce thème ont débuté au cours de l'année 2003, faisant suite à la collaboration de Marie-Laure Potet avec l'équipe DCS de Vérimag. Des collaborations avec d'autres partenaires académiques et industriels sont venus enrichir l'activité, débouchant sur le montage de plusieurs projets : le projet MODESTE[11] de l'IMAG (2004-2006), le projet GECCOO[12] (2003-2006), le projet POTESTAT[13] (2004-2007), ...

Dans le paragraphe suivant, je vais présenter le projet POSÉ dans lequel j'ai travaillé. Ce projet fait intervenir les deux derniers thèmes que je viens de présenter. Il s'agit d'abord de la construction de modèles pour l'expression et la vérification de propriétés de sécurité, partie qui a été au cœur de mon travail. Ensuite ces modèles sont utilisés pour la génération de tests, ce qui m'a également permis d'aborder ce domaine.

[11] Modélisation pour la sécurité : http ://www-verimag.imag.fr/~lakhnech/MODESTE. Dernière consultation : 05-jun-07

[12] Génération de code certifié pour des applications orientées objet : http ://geccoo.lri.fr. Dernière consultation : 05-jun-07

[13] Politiques de sécurité : Test et Analyse par le test de systèmes en réseau ouvert : http ://www-lsr.imag.fr/POTESTAT. Dernière consultation : 05-jun-07

2.3 Le projet POSÉ

Le projet POSÉ[14] est un projet de recherche proposé dans le cadre du RNTL[15] pour une durée de 24 mois. Il a été financé par l'ANR[16].

Le RNTL encadre 3 types de projets :

- des projets **exploratoires** : ces projets visent à démontrer la faisabilité d'une approche particulièrement innovante en s'attaquant à un verrou technologique bien identifié. Le résultat de ce type de projet peut être un démonstrateur.

- des projets **pré-compétitifs** : ces projets visent à la réalisation d'un produit ou d'un service à visée opérationnelle à court terme, soit en apportant au plan technique fonctionnel ou ergonomique une innovation à un outil existant, soit en réalisant un nouvel outil.

- des projets de type **plate-forme d'expérimentation** : ces projets visent à construire le support nécessaire à l'expérimentation de passage à l'échelle pour des projets de recherche de type exploratoire ou pré-compétitifs.

Le projet POSÉ, pour sa part, est un projet pré-compétitif. Il vise à produire des outils conceptuels, méthodologiques et techniques pour un processus de validation et de vérification de politiques de sécurité sur des applications de types cartes à puce. Les sections suivantes donneront une présentation de ce projet en montrant ses objectifs et son organisation. Elles permettront de comprendre le positionnement de mon travail au sein du projet avant de présenter la partie qui sera à l'étude dans ce mémoire.

2.3.1 Objectifs

L'objectif fondamental du projet POSÉ est l'**automatisation de la génération et de l'exécution de tests permettant la validation de conformité d'un système à la politique de sécurité** qui lui est assignée. A côté des campagnes de test de validation fonctionnelle d'un système, c'est une véritable campagne de validation du respect des politiques de sécurité qui sera réalisée en s'appuyant sur des techniques de modélisation formelle et de génération automatique de tests, accompagnées d'une **traçabilité des exigences de sécurité** de haut-niveau vers les tests mis en œuvre.

L'automatisation du test des politiques de sécurité constitue un défi technologique auquel ne répondent pas les techniques actuelles de génération de tests fonctionnels. En effet, il s'agit de s'assurer que les différents niveaux de propriétés de sécurité (e.g. confidentialité, atomicité, authentification, intégrité, standardisation des exceptions, cycle de vie, ...) font l'objet chacune de tests spécifiques, correspondant à la simulation d'attaques possibles et au test des réponses du système.

[14] http ://www.rntl-pose.info. Dernière consultation : 06-jun-07
[15] Réseau National des Technologies Logicielles
[16] Agence Nationale de la Recherche

Le projet POSÉ s'appuie sur un ensemble de savoir-faire et de connaissances scientifiques des partenaires : d'une part en génération automatique de tests fonctionnels à partir d'un modèle formel et d'autre part en formalisation et vérification de propriétés de sécurité. Il s'agit d'adapter ces techniques et de prendre en compte les besoins effectifs issus du terrain (i.e. les défis de la validation des politiques de sécurité sur les applications carte à micro-processeur ou sur terminaux, développées et livrées aujourd'hui) pour mettre au point un démonstrateur d'environnement dédié au test de conformité des politiques de sécurité d'un système.

L'état de la recherche et l'évolution de pratiques industrielles avec l'émergence de l'utilisation de techniques de génération automatique de tests fonctionnels à partir de modèles, crédibilisent cet objectif. Pour autant, il s'agit là d'un défi scientifique et technologique. Le projet POSÉ permettra de montrer quels types de propriétés peuvent être abordés efficacement dans un processus de génération automatique de tests, quels critères de tests peuvent être adaptés et comment intégrer cette approche au processus actuel de validation des applications. Il s'agit aussi d'aborder les questions liées au passage à l'échelle des techniques étudiées en traitant une application opérationnelle en grandeur réelle.

2.3.2 Partenaires

Le projet POSÉ s'appuie sur une forte complémentarité et synergie de l'ensemble des partenaires.

Le projet intègre 3 partenaires industriels :

FIG. 2.3 – Les partenaires industriels du projet POSÉ

- **Leirios Technologies**, éditeur de solutions de génération automatique de tests, en charge du pilotage du projet.
- **Gemalto**, leader mondial dans le domaine des cartes à micro-processeur, qui apporte ses besoins, ses compétences et un contexte d'expérimentation en situation réelle.
- **Silicomp-AQL**, société de conseil, possédant une forte expertise en sécurité, intégrant un laboratoire d'évaluation de la sécurité des systèmes d'information (CESTI) et possédant des compétences en modélisation, méthodes formelles et génération automatique de tests.

et 2 partenaires académiques :

- **IMAG/LIG**, qui développe une forte activité de recherche dans le domaine des méthodes formelles, du test et de la sécurité des systèmes informatiques.
- **INRIA/Projet CASSIS**, qui développe une activité de recherche dans le domaine des techniques symboliques pour la vérification et le test de systèmes infinis, paramétrés et de grandes tailles.

FIG. 2.4 – Les partenaires académiques du projet POSÉ

2.3.3 Organisation

Le projet POSÉ est structuré en 6 sous-projets. Pour chacun d'entre eux, sa responsabilité est donné au partenaire dont c'est le domaine de compétence et qui a majoritairement en charge sa réalisation. Voici une courte description de chaque sous-projet.

- **Caractérisation** : Ce sous-projet a pour objet de caractériser des types de propriétés de sécurité devant être testées sur le système ainsi que les éléments spécifiques pour le test boîte noire. Il est sous la responsabilité de Silicomp-AQL.

- **Formalisation et Vérification du modèle** : Cette partie concerne la modélisation des propriétés de sécurité, des attaques potentielles, des défenses attendues et la vérification formelle des modèles. Le LIG est responsable de ce sous-projet. C'est donc sur cette partie qu'on retrouvera les différentes implications du LIG et en particulier de mon stage.

- **Génération et exécution de tests** : Ce sous-projet traite de l'élaboration des stratégies de génération de tests de conformité de sécurité à partir de ces modélisations et la résolution de la concrétisation, de l'observation et du contrôle de l'exécution et verdict automatique. Il est sous la responsabilité du projet CASSIS de l'INRIA.

- **Démonstrateurs** : Le but de ce sous-projet sera de produire des démonstrateurs d'outils de génération de tests de conformité de politiques de sécurité. Sa responsabilité est donnée au partenaire Leirios Technologies.

- **Application industrielle** : Cette partie aura pour objectif la mise en œuvre en situation opérationnelle des méthodologies et technologies développées dans les quatre premiers sous-projets sur une application carte à puce, type passeport électronique. C'est la société Gemalto qui fournira cette application et sera par conséquent la responsable de ce sous-projet.

- **Gestion de projet** : La gestion du projet POSÉ est sous la responsabilité de Leirios.

Chapitre 3

La technologie JavaCard

Dans le chapitre 2 qui a présenté le contexte du stage, il a été fait mention que le domaine d'application du projet POSÉ était celui de la carte à puce. Pour être plus précis, nos travaux s'appliquent en particulier à la technologie de carte à puce Java Card. Ce chapitre a d'abord pour objectif de présenter les caractéristiques de la carte à puce. Il se focalisera ensuite plus particulièrement sur la technologie JavaCard et servira de base pour la suite de nos travaux.

3.1 Présentation de la carte à puce

Une carte à puce est une carte plastique qui intègre un circuit électronique capable de manipuler (stocker, calculer, ...) des informations de façon sécurisée. C'est en quelque sorte un ordinateur portable et résistant aux altérations. Il s'agit donc bien d'une carte *intelligente* (*smart card* comme on l'appelle dans les pays anglo-saxons) et non d'une simple carte à piste magnétique puisqu'elle embarque un circuit logique. D'ailleurs, contrairement à cette dernière, la carte à puce n'a pas besoin d'accéder à une base de données distante lors d'une transaction puisqu'elle possède sa propre puissance de calcul et sa propre capacité de stockage d'informations dans un mode sécurisé [Sau04].

La plupart des normalisations applicables à ces cartes sont décrites dans le standard ISO 7816 qui est découpé en 15 parties, couvrant chacune un aspect différent des cartes. Chaque partie est noté ISO 7816-n ou n désigne le numéro de la norme ISO 7816. Par la suite, dès qu'une explication se réfère à une de ces normes, son numéro sera cité.

3.1.1 Historique

Contrairement aux idées reçues, la carte à puce n'est pas seulement une invention française de Roland Moreno. En réalité, c'est en 1967 qu'elle a vu le jour grâce à deux allemands : Jürgen Dethloff et Helmut Grötrupp. Cependant, Roland Moreno, qui s'est assuré des retombées financières des cartes à puce grâce à de très nombreux brevets est aujourd'hui considéré comme l'inventeur officiel de la carte à puce [Wik07].

Voici en quelques dates les principaux évènements qui ont marqué la carte à puce :

- 1967 : Jürgen Dethloff et Helmut Grötrupp sont les premiers à introduire un circuit intégré dans une carte à puce.
- 1970 : Kunitaka Arimura, un japonais dépose un brevet sur la carte à puce
- 1971 : Paul Castrucci, un américain de chez IBM dépose à son tour un brevet sur la carte à puce.
- 1974 à 1978 : Roland Moreno, dépose 47 brevets dans 11 pays et est considéré comme l'inventeur de la carte à puce.
- 1979 : Création par la société Motorola pour Bull de la première carte à base de microprocesseur 6805 et de 1 Ko de mémoire programmable.
- 1983 : Apparition des premières cartes téléphoniques à mémoire.
- 1984 : Adoption de la « carte bleue » par le G.I.E carte bancaire.
- 1984 à 1987 : Définition des normes internationales de l'ISO sur la carte à puce à contact sous la référence 7816.
- 1997 : Apparition des premières Java Cards.

3.1.2 Les principales catégories de cartes à puce

La carte à puce succède à plusieurs autres familles de cartes dont la carte embossée, la carte à code barre ou encore la carte à piste magnétique. La carte à puce elle-même connaît plusieurs déclinaisons. D'une part, elle supporte deux types d'architecture matérielle. La première concerne les cartes à mémoire alors que la seconde, plus récente, correspond aux cartes à microprocesseur. D'autre part, l'interface entre la carte et le lecteur peut se faire soit de manière électrique dans le cas des cartes à contact, soit de manière radio dans le cas des cartes sans contact. Les deux points suivants décrivent brièvement chaque type d'architecture. De même, chaque technologie d'interface sera abordée dans les deux points suivants [Lan06, Sau01].

3.1.2.1 Les cartes à mémoire

Les cartes à mémoire possèdent, comme leur nom l'indique, une puce mémoire qui intègre de la mémoire non volatile permettant de stocker quelques centaines de bits, ainsi qu'un peu de logique permettant d'effectuer quelques opérations. Celles-ci vont de la lecture ou l'écriture jusqu'à l'exécution de quelques algorithmes basiques d'authentification. Ces cartes correspondent typiquement aux télécartes qui contiennent un compteur d'unités, un numéro de série et une donnée secrète permettant d'authentifier la carte. Les avantages de ces cartes sont leur technologie simple et leur faible coût de revient (moins de 1$). Cependant, elles présentent également plusieurs inconvénients comme la dépendance du lecteur de carte pour le calcul mais aussi la possibilité d'être dupliquée « facilement » (bien que plus difficilement que les cartes à piste magnétique).

3.1.2.2 Les cartes à microprocesseur

Contrairement aux cartes à mémoire qui ne peuvent que mémoriser de l'information, car constituées seulement de mémoire, les cartes à microprocesseur peuvent servir à de multiples usages. Selon les mémoires qu'elles contiennent, elles peuvent être reprogrammées. De plus, leur contenu est encrypté, et elles embarquent de multiples mécanismes qui offrent de la sécurité.

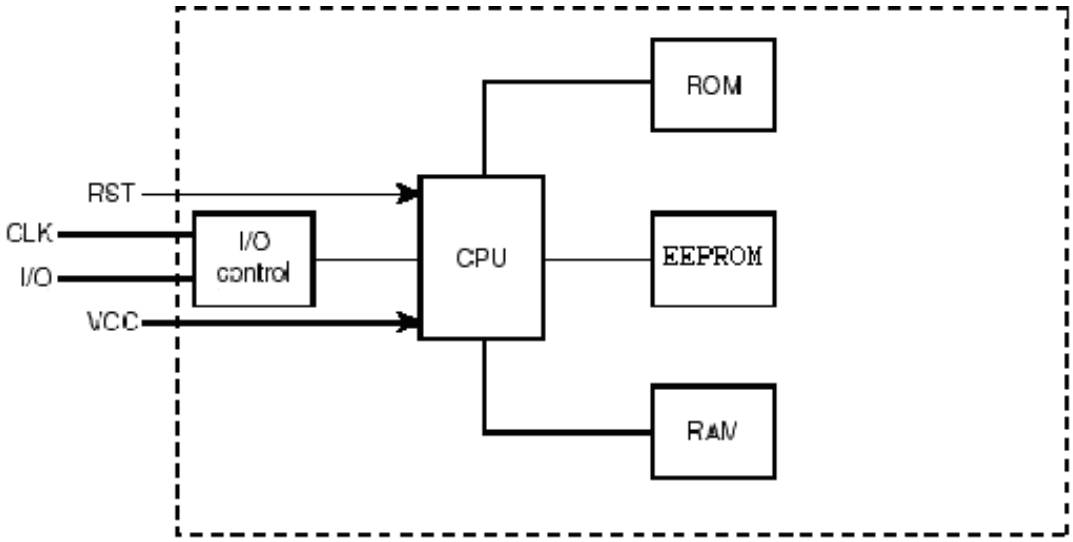

FIG. 3.1 – Architecture d'une carte à microprocesseur

Au niveau matériel, elles s'apparentent quasiment à l'architecture d'un ordinateur portable. Elles contiennent un microcontrôleur qui sur une seule puce rassemble les composants suivants :

- un microprocesseur
- la ROM (Read Only Memory) qui est de la mémoire morte non modifiable. Cette mémoire sert à stocker le système d'exploitation de la carte (COS[1]) qui est programmé en usine.
- l'EEPROM (Electrical Erasable Programmable Read Only Memory) qui est comme la ROM persistante. Cependant, à la différence de la ROM, cette mémoire peut être modifiée par une application. Cette mémoire présente les inconvéniants d'être limitée en nombre de cycles d'écriture (100000 cycles) et en temps (10 ans). De plus, ses accès sont relativement lents comparés à ceux de la RAM (d'un facteur 1000).
- la RAM (Random Access Memory) qui est utilisée comme mémoire de travail grâce à sa rapidité et pour les données non persistantes car ces données sont perdues lorsque l'alimentation est coupée.

La figure 3.1 présente l'architecture classique d'une carte à microprocesseur. Cependant, la particularité des cartes à microprocesseur actuelles, communément appelées *smart card*, est qu'elles contiennent en plus du microprocesseur, un cryptoprocesseur, dont le rôle est d'exécuter différents algorithmes de cryptage de manière optimisée et sécurisée. Cette architecture permet ainsi d'encapsuler les fonctions de cryptage en limitant au maximum les interactions possibles entre une application et ces fonctions, afin d'éviter par exemple les tentatives de violations des algorithmes par une application.

3.1.3 La communication entre la carte et le lecteur

De part son architecture, une carte à puce peut être assimilée à un ordinateur. Cependant, pour des raisons d'extrême compacité, elle n'embarque pas de batterie, ce qui la rend passive.

[1] Card Operating System

De plus, son interface aussi bien matérielle que logicielle ne lui permet pas de dialoguer directement avec d'autres ordinateurs. Pour ce faire, il est nécessaire d'utiliser un adaptateur spécifique appelée *lecteur* (ou lecteur de carte).

Comme présenté sur la figure 3.1, seuls quelques signaux sont nécessaires pour la communication entre la carte et le lecteur. Le premier, noté VCC (associé à un signal de référence noté GND), permet au lecteur de fournir l'alimentation nécessaire à la carte pour son fonctionnement. Le lecteur fournit également deux signaux, un noté RST qui permet de démarrer proprement le logiciel embarqué et un signal d'horloge noté CLK. Enfin, l'échange bi-directionnel des données s'effectue au moyen d'un unique signal noté I/O.

Cette partie présente d'abord les différentes interfaces matérielles normalisées entre une carte et un lecteur. Elle décrira ensuite le protocole qu'utilise la carte pour communiquer avec un lecteur.

3.1.3.1 Les interfaces matérielles

Il existe deux types d'interface matérielle pour la connexion de la carte au lecteur et donc deux types de carte. La première est la *carte avec contact* alors que la seconde est la *carte sans contact*. Il existe également un autre type de carte appelé *carte dual interface* qui associe à la fois les deux types précédent dans une même carte.

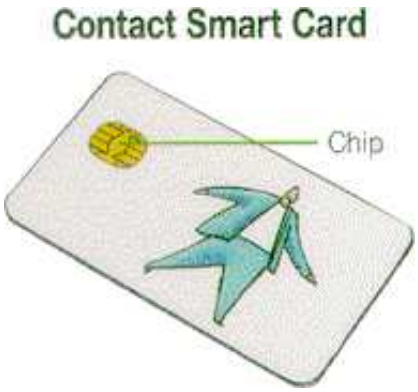

FIG. 3.2 – La carte à contact

La carte à contact est la plus répandue car elle est la moins chère et historiquement la plus ancienne. Elle est présentée sur la figure 3.2. Ce type de carte utilise un *microcontact* qui sert de connecteur avec le lecteur. Ce microcontact est relié à la puce (*microchip*) via des fils d'or. L'assemblage du microcontact avec le microchip s'appelle le *micromodule*.

La figure 3.3 est un example de micromodule. Les huit contacts sont définis dans la norme ISO 7816-2 de la manière suivante :

- C1 - VCC (Tension d'alimentation de la carte)
- C2 - RST (Signal de reset de la carte)
- C3 - CLK (Signal d'horloge)
- C4 - RFU (Reserved for Future Usage) / USB+ (interface USB sur les nouvelles cartes)
- C5 - GND
- C6 - Vpp (Tension d'alimentation utilisée sur les anciennes cartes pour les programmer)

- C7 - I/O (Signal de données bi-directionnel)
- C8 - RFU / USB- (interface USB sur les nouvelles cartes)

La plupart des cartes récentes supportent de plus en plus la possibilité de communiquer au moyen d'une liaison USB. Pour cela, seuls sont utilisés les contacts C1, C4, C5 et C8. L'alimentation est toujours fournie au moyen de C1 et C5 et le transfert de données se fait grâce à un signal différentiel. Ce signal est véhiculé sur les contacts C4 et C8 qui n'étaient pas définis dans la norme ISO 7816-2. La liaison USB présente l'avantage d'être universelle et beaucoup plus rapide. Dans ce cas, le rôle du lecteur n'est plus que d'adapter mécaniquement l'interfaçage matériel en acheminant ces 4 signaux vers le port USB du PC sur lequel il est connecté.

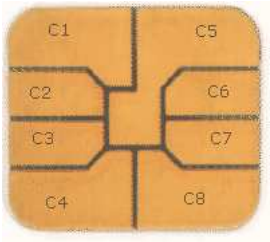

FIG. 3.3 – Le microcontact

Pour communiquer avec un lecteur, les cartes à contact doivent nécessairement être introduites dans un lecteur afin que le microcontact soit relié physiquement au connecteur du lecteur. Il en découle alors deux problèmes qui sont d'une part une usure de la carte et d'autre part une obligation de respecter le sens d'insertion de la carte.

Pour pallier à ces contraintes, il existe un autre type de carte, les cartes sans contact. Celles-ci fonctionnent au moyen d'ondes radio et ne subissent alors pas d'usure comme les cartes à contact. Elles sont alimentées au moyen d'un couplage inductif (couplage distant effectué par l'antenne) ou dans certains cas au moyen d'un couplage capacitif (via une batterie embarquée sur la carte). Les différents éléments de ces cartes sont représentés sur la figure 3.4. De plus, ces cartes fonctionnent à distance (à quelques centimètres du lecteur) en les présentant sans tenir compte de leur orientation. Le standard de ces cartes est défini dans la norme ISO 10536.

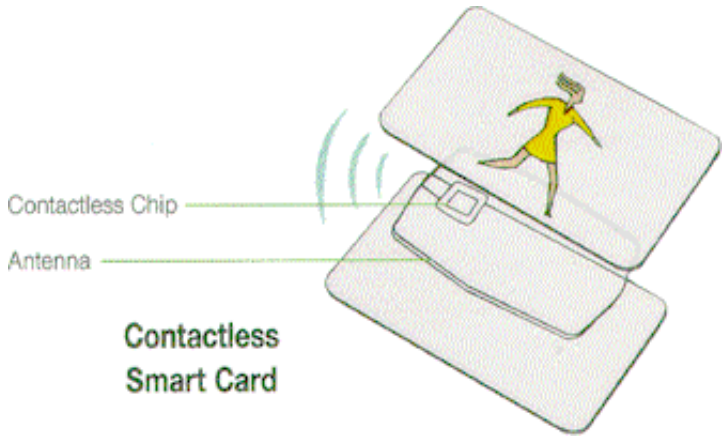

FIG. 3.4 – La carte sans contact

Enfin, la figure 3.5 montre un exemple de carte dual-interface qui est simplement l'addition des deux types précédents au sein d'une même carte. Ces cartes présentent les avantages de chaque type d'interface mais sont peu répandues du fait de leur coût plus élevé.

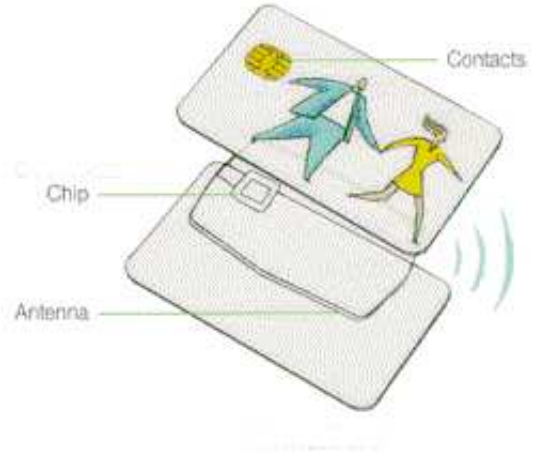

FIG. 3.5 – La carte dual-interface

3.1.3.2 Le protocole de communication

Le protocole de communication se répartit en deux niveaux. Le premier correspond au protocole de transmission appelé TPDU[2]. Le second, quant-à lui, est de niveau application et s'appelle APDU[3].

Le TPDU est défini dans la norme ISO 7816-3, tout comme les signaux électriques. Ce protocole n'est pas particulièrement intéressant vis-à-vis du projet POSÉ. Par conséquent, il ne sera pas développé ici. Voici simplement un bref apperçu des points normalisés dans l'ISO 7816-3 :

- les caractéristiques électriques comme la fréquence d'horloge
- la vitesse de communication (qui peut aller jusqu'à 115200 bauds)
- le mode de transfert des données (protocole orienté octet, paquet ou protocole propriétaire)
- la réponse au reset (ATR : Answer To Reset), c'est-à-dire quelles données sont envoyées par la carte immédiatement après la mise sous tension

L'APDU, qui est spécifié en détail dans la norme ISO 7816-4 [iso] est un élément central en ce qui concerne la sécurité des cartes à puce. En effet, il s'agit du seul point d'entrée qui permette d'interagir avec un programme contenu dans la carte (qu'il s'agisse du système d'exploitation ou d'une application en particulier) à partir du lecteur. L'APDU se présente sous la forme d'un protocole maître-esclave assez basique qui comporte des commandes et des réponses.

[2]Transmission Protocol Data Unit
[3]Application Protocol Data Unit

Une *commande APDU* (C-APDU) est toujours émise par le lecteur (le maître) vers la carte. De même, la *réponse APDU* (R-APDU) est retournée par la carte (l'esclave) au lecteur.

La carte est toujours en attente d'une commande APDU. Dès qu'elle reçoit une commande APDU, elle doit toujours retourner une réponse, même en cas d'erreur.

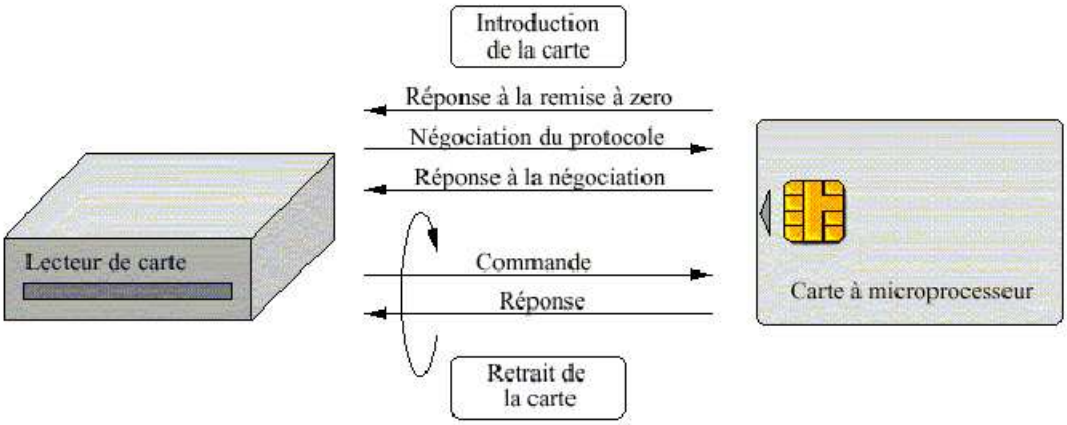

FIG. 3.6 – Le protocole de communication entre la carte et le lecteur

Le tableau 3.1 présente la structure générale d'une commande APDU. Celle-ci contient toujours un entête qui est composé des quatre octets suivants :
- CLA, un octet de classe qui identifie la catégorie de la commande et de la réponse.
- INS, un octet d'instruction qui spécifie l'instruction de commande.
- P1 et P2, deux octets qui correpondent aux paramètres de l'instruction.

Elle peut aussi contenir, de manière optionnelle, les octets suivants :
- Lc, un octet qui spécifie la taille du champ de données.
- Un champ de données qui contient les données à envoyer à la carte.
- Le, un octet qui spécifie le nombre d'octets attendu en retour par le terminal.

Entête obligatoire				Corps optionnel		
CLA	INS	P1	P2	Lc	Champ de données	Le

TAB. 3.1 – Structure d'une commande APDU

Comme celle de la commande APDU, la structure de la réponse APDU, présentée dans le tableau 3.2, est également basique.

Si le champs « Le » est présent dans la commande APDU et est différent de 0, la réponse APDU contient un corps de données de longueur « Le ». D'autre part, elle se termine toujours par les deux octets suivants :
- SW1 et SW2 (Status Word), ces octets indiquent l'état de la carte après exécution de la commande APDU. Ainsi, la valeur « 0x9000 » indique que l'exécution s'est déroulée complètement et avec succès. Dans tous les autres cas, les octets SW1 et SW2 auront des valeurs différentes de « 0x9000 ». Dans la plupart des cas d'erreurs, les valeurs de SW1 et SW2 sont définies dans la norme ISO7816-4 et commencent par 6 (Par exemple

0x6E00 qui indique que l'octet CLA n'est pas supporté). Cependant, il est également possible de définir ses propres valeurs de SW d'erreur.

Corps optionnel	Champs obligatoires	
Champ de données	SW1	SW2

TAB. 3.2 – Structure d'une réponse APDU

La présentation du protocole APDU permet de mettre en évidence une caractéristique essentielle de celui-ci, à savoir qu'il se comporte de manière totale. En effet, grâce au mécanisme des Status Word, il est possible d'exécuter une application carte à puce et de vérifier la comformité de celle-ci par rapport à sa spécification en se basant uniquement sur ses traces. Il s'agit ici uniquement de vérifier si l'enchaînement des valeurs SW1 et SW2 retournées par l'application est conforme à celui qui est prévu. Cette caractéristique sera essentielle pour nos travaux car elle en constitue la base.

3.2 Spécificité de la JavaCard

La technologie JavaCard combine un sous-ensemble du language de programmation Java avec un environnement d'exécution optimisé pour les cartes à puces et des propriétés similaires pour les dispositifs à mémoire réduite. Le but de la technologie JavaCard est d'apporter la plupart des bénéfices de la programmation logicielle Java au monde des cartes à puce, fortement contraint par ses ressources.

L'API[4] JavaCard est compatible avec les standards internationaux, tel que ISO 7816 et des standards spécifiques aux industriels comme Europay, MasterCard ou encore Visa (EMV).

3.2.1 L'architecture de la machine virtuelle JavaCard

Comme pour le langage Java, on retrouve aussi avec JavaCard une Machine Virtuelle Java (JVM). Celle-ci est identique à celle de Java mais est découpée en deux parties : une sur la carte, l'autre hors carte (Java Card Converter). Il est donc nécessaire de pré-compiler les applications avant de les charger sur la carte. La figure 3.7 présente le cycle de développement d'une application JavaCard.

La partie programmation est similaire à Java et s'effectue par conséquent sur n'importe quel environnement de programmation Java. Elle consiste donc à produire des fichiers avec pour extension « `.java` ». Ces fichiers sont d'abord compilés afin d'obtenir le bytecode au moyen de fichiers « `.class` ». Ensuite, ce bytecode passe dans la machine virtuelle « hors carte » qui est découpée en 3 modules ayant chacun un rôle précis :

[4]Application Programming Interface : elle contient un ensemble de librairies de classes fournissant des services de haut niveau à une application

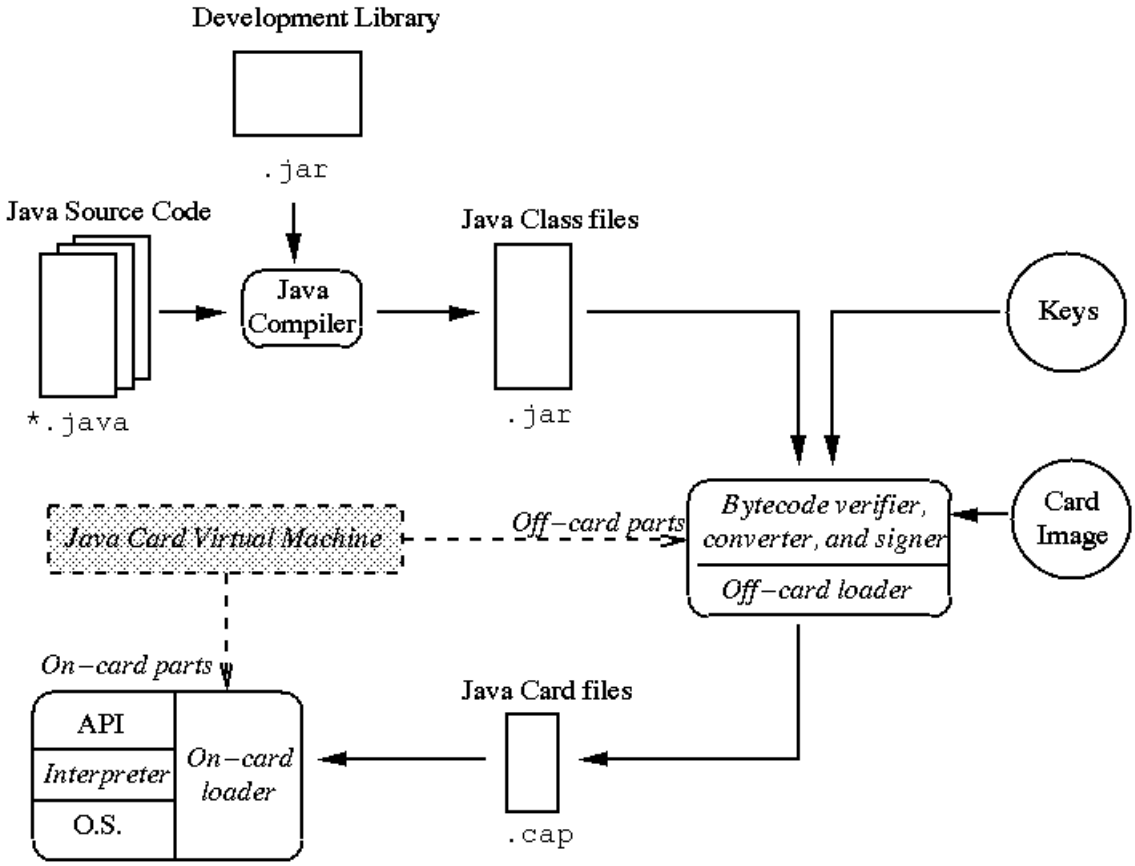

FIG. 3.7 – Architecture de la machine virtuelle JavaCard

- Le **Vérifieur** de ByteCode : Il utilise le vérifieur de Bytecode Java classique. Il contrôle le sous-ensemble JavaCard (langage + API).

- Le **Convertisseur**. Il gère les 3 phases suivantes :
  Préparation : initialise les structures de données de la JCVM.
  Optimisation : ajuste l'espace mémoire, remplace certains InvokeVirtual par des InvokeStatic, etc.
  Edition de liens : résout les références symboliques à des classes déjà présentes dans la carte (via image de la carte).

- Le **Signeur** : Il valide le passage par le vérificateur et le convertisseur par une signature vérifiée par la carte au moment du chargement.

Le programme fournit alors des fichiers « .cap » qui peuvent être chargés dans la carte.

3.2.2 L'applet JavaCard

L'applet JavaCard est un programme serveur de la carte JavaCard. Une fois chargée, elle est sélectionnée par le terminal par un APDU de sélection. Cette sélection est faite en fonction de l'ID de l'applet qui doit être unique (représenté par la classe AID signifiant Applet ID).

Une applet a les propriétés suivantes :
- Une fois installée, elle est toujours disponible.
- L'applet doit hériter de la classe `javacard.framework.Applet`
- Elle doit implémenter les quatre méthodes qui interagissent avec le JCRE[5] : `install()`, `select()`, `deselect()` et `process()`.

La figure 3.8 montre le principe d'exécution d'une applet une fois qu'elle a été chargée et installée sur la carte (appel de la méthode `install()` par le JCRE.

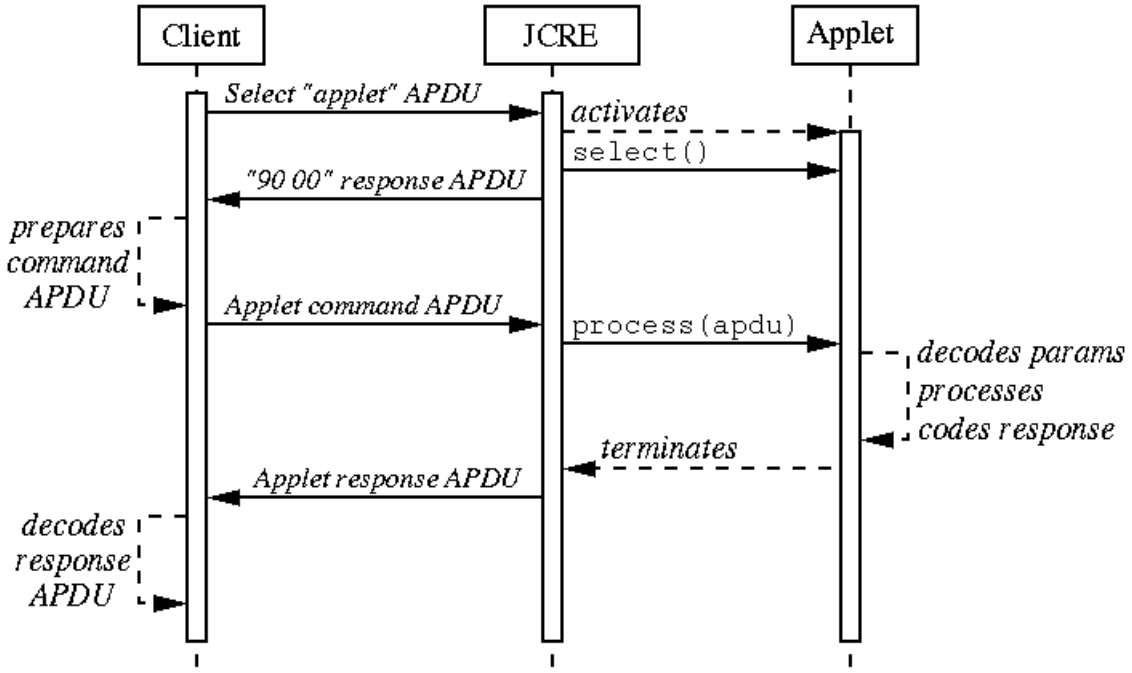

FIG. 3.8 – Exécution d'une applet JavaCard

[5] JavaCard Runtime Environment

Chapitre 4

La norme des Critères Communs

Le chapitre 3 a permis de dresser un état des lieux de la technologie des cartes à puce, et en particulier des JavaCard. Il a pu faire apparaître les particularités du domaine dans lequel seront appliqués nos travaux. Ainsi, comme nous le verrons dans le chapitre 6, les choix retenus quant à la modélisation des politiques de sécurité ont été fortement guidés par les spécificités du monde de la carte à puce (notamment le fait que chaque fonction implémentée est totale, c'est-à-dire retourne toujours une valeur, quelque soit les paramètres d'entrée).

Une autre contrainte forte prévue dans les objectifs du projet POSÉ était de fournir une traçabilité des exigences de sécurité de haut-niveau vis-à-vis des tests mis en œuvre en s'appuyant sur la norme des « Critères Communs » (plus connues sous son anglicisme *Common Criteria*). Ce chapitre a pour but de présenter cette norme et de faire une classification des exigences de sécurité qu'elle peut traiter. D'autre part, une extraction des exigences intéressantes pour le projet POSÉ sera faite à partir de cette classification.

4.1 Présentation des Critères Communs

Dans le domaine de la sécurité des systèmes d'information[1] [Bou97], la première question que l'on se pose généralement est la suivante : **comment être certain de l'absence de failles de sécurité dûes par exemple à l'implémentation ?** De cette question fondamentale découle une seconde question tout aussi importante : **quel degré de confiance peut-on attribuer à tel ou tel produit en ce qui concerne sa sécurité ?**

C'est pour répondre à ces questions que certains gouvernements ont mis en place dans leur pays des critères d'évaluation. Ces critères visent à permettre aux producteurs de logiciels de fournir des garanties sur les parties sensibles de leurs produits. L'application de ces critères débouchent sur l'obtention d'une note qui permet de connaître le soin apporté à leur développement et par conséquent de déduire quel est leur degré de confiance [Sto04].

Lorsque ces critères ont été définis initialement, ils étaient différents selon les pays. Ainsi, les critères définis en Europe depuis 1993 étaient les ITSEC[2] alors que ceux du département de

[1] On appelle **SI** (**S**ystème d'**I**nformation) « l'utilisation pratique d'une information au travers de son traitement par un ordinateur de quelque nature qu'il soit »

[2] Information Technology Security Evaluation Criteria

la défense américaine étaient les TCSEC[3]. Le principal défaut de ces différents critères résidait dans leur manque d'équivalence. Par conséquent, un produit certifié en Europe pouvait ne pas être reconnu aux Etats-Unis à moins qu'il y ait subit une seconde certification spécifique.

Pour résoudre ces problèmes d'incompatibilité entre pays, une harmonisation de ces critères a eu lieu en 1999 avec la création de la norme des « Critères Communs », également référencée par l'ISO[4] sous la norme ISO 15408. Cette norme est issue de la coopération entre trois nations : le Canada, les Etats-Unis et l'Europe. L'apparition de cette norme a offert un standard international qui a permis aux producteurs de logiciels de faire reconnaître la qualité de leur produit dans le monde entier.

Les **CC** (Critères Communs) permettent d'évaluer une large gamme de **SI** (Systèmes d'Information), en s'intéressant aussi bien à leur partie matérielle que logicielle. Pour n'en citer que quelques uns, voici les principaux systèmes concernés par les CC :

- **les systèmes d'exploitation**.
- **les dispositifs dédiés aux communications** : routeurs, commutateurs réseaux, réseaux privés virtuels (VPN), etc.
- **les systèmes consacrés à la sécurité informatique** : les systèmes d'accès (e.g. accès Internet), les systèmes d'authentification, les pare-feu, les logiciels anti-virus, etc.
- **les cartes à puce**.

Historiquement, le type de produit privilégié par les CC et ses ancêtres est le système d'exploitation centralisé. Aujourd'hui (fin des années 2000), le type de produit le plus souvent évalué est la carte à puce.

4.1.1 Niveaux d'évaluation des Critères Communs

Lors de l'évaluation d'un SI, ce dernier n'est pas évalué dans son intégralité. Son évaluation ne porte que sur les parties dites « sensibles ». **La partie du SI évaluée est appelée « cible d'évaluation » ou TOE (Target Of Evaluation)**.

Dans le cadre des CC, le degré de confiance donné à un produit se traduit par l'attribution d'une note sur une échelle allant de 1 à 7. Cette échelle d'assurance s'appelle les EAL[5] et permet d'indiquer quel est le niveau de défense procuré par les composants de sécurité du produit évalué. Pour simplifier, on peut dire que les niveaux EAL1 à EAL4 correspondent à des systèmes courants de bonne qualité et à la mise en oeuvre de bonnes pratiques. Les niveaux EAL5 à EAL7, quant-à eux, correspondent à des systèmes conçus avec une démarche et des méthodes de sécurisation particulièrement poussées. Le niveau EAL7 répond notamment à

[3] Trusted Computer Security Evaluation Criteria
[4] International Standard Organisation
[5] Evaluation Assurance Level

des problématiques de stratégie nationale de sécurité.

Dans le détail, les exigences qui sont requises pour chacun des sept niveaux d'assurance sont décrites ci-dessous :

- **EAL1** : testé fonctionnellement.
- **EAL2** : testé structurellement.
- **EAL3** : testé et vérifié méthodiquement.
- **EAL4** : conçu, testé et vérifié méthodiquement.
- **EAL5** : conçu et testé de façon semi-formelle.
- **EAL6** : vérifié, conçu et testé de façon semi-formelle.
- **EAL7** : vérifié, conçu et testé de façon formelle.

Depuis 2006, la norme des Critères Communs en est à la version 3.1. Aujourd'hui, ce standard est reconnu par 15 pays. Seuls, l'Australie (et la Nouvelle-Zélande), le Canada, la France, l'Allemagne, le Royaume-Uni et les Etats-Unis sont habilités à délivrer un certificat. Pour leur part, la Finlande, la Grèce, l'Italie, Israël, le Japon, la Hollande, la Norvège et l'Espagne prennent en compte les Critères Communs jusqu'au niveau EAL4, sans toutefois pouvoir délivrer, eux-mêmes, des certificats. Entre pays européens, il y a une reconnaissance jusqu'au niveau EAL7.

4.1.2 Organisation des Critères Communs

En France, la définition des CC est disponible sur le site de la DCSSI[6] (Direction Centrale de la Sécurité des Systèmes d'Information) qui reste toujours en service même si cette organisation a été remplacée depuis le 7 juillet 2009 par l'ANSSI[7] (Agence Nationale de la Sécurité de Systèmes d'Information). Cette agence correspond à l'autorité nationale qui est chargée de la régulation de la sécurité des systèmes d'information et dépend du Premier Ministre. D'un point de vue documentaire, les Critères Communs sont structurés en trois publications :

- partie 1 : **Introduction et modèle général** [CC006a]. Ce document présente les motivations générales et décrit le mode d'utilisation conseillé des deux autres documents.
- partie 2 : **Exigences fonctionnelles de sécurité** [CC006b]. Ce document est une liste exhaustive de spécifications de modules de sécurités reconnues par les Critères Communs. Ces exigences sont triées en fonction de leur rôle.

[6] http ://www.ssi.gouv.fr/archive/fr/confiance/methodologie.html
[7] http ://www.ssi.gouv.fr/

– partie 3 : **Exigences d'assurance de sécurité** [CC006c]. Ce document est une liste exhaustive des critères de qualité reconnus par les Critères Communs. La note donnée lors d'une évaluation selon les Critères Communs n'est que le reflet de la qualité du Système d'Information en fonction des éléments de cette liste.

Les exigences fonctionnelles permettent d'exprimer de façon standardisée les règles de sécurité que doit respecter la cible d'évaluation (TOE). Pour cela, elles offrent un catalogue de composants à utiliser en fonction des objectifs de sécurité à garantir. Plus précisément, ces exigences définissent comment la TOE doit gérer les accès à ses ressources et quels contrôles doivent être effectués sur les services offerts par la TOE. Nous verrons dans la partie 4.2 quels sont les composants fournis par les exigences fonctionnelles et comment ils sont hiérarchisés.

Les exigences d'assurances, quant-à elles, définissent les actions à entreprendre pour garantir le respect des objectifs des exigences fonctionnelles. Elles permettent elles aussi d'exprimer de manière standardisée ces actions au moyen d'un catalogue de composants d'assurance. L'évaluation d'une cible de sécurité se fait alors en se basant sur ces exigences afin de permettre d'établir leur véracité. Ces exigences feront l'objet de la partie 4.3

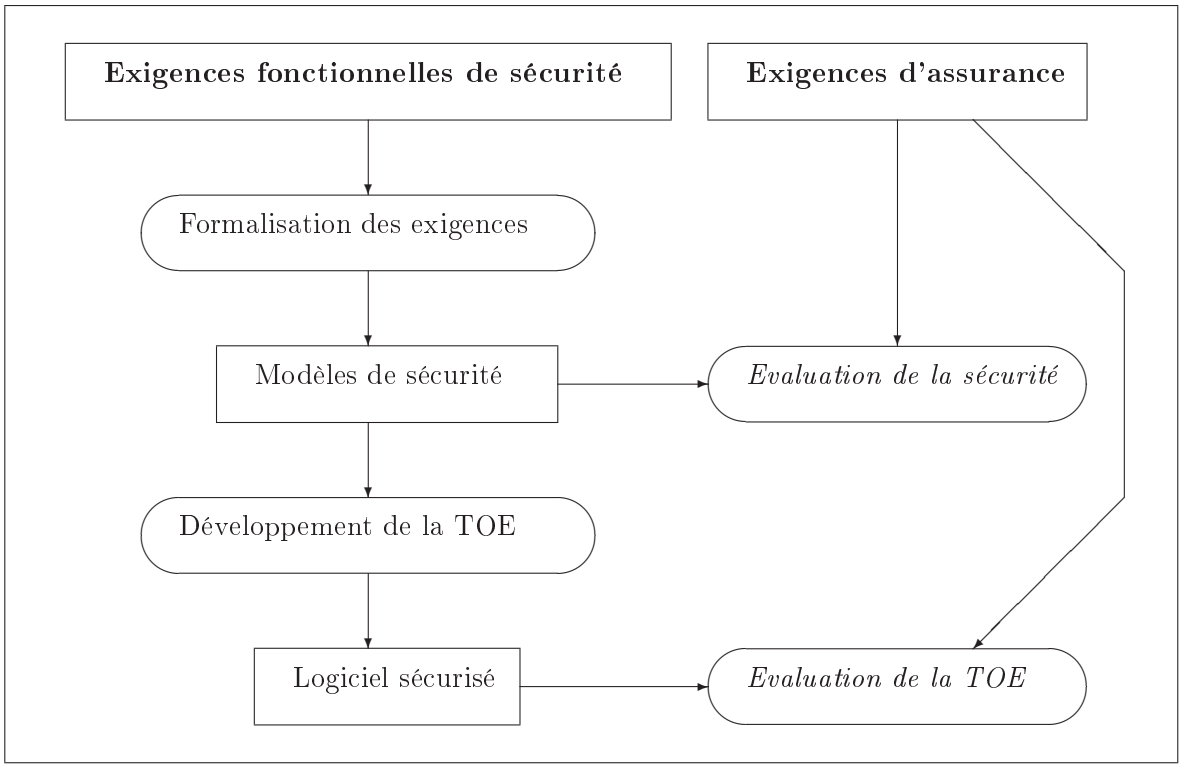

FIG. 4.1 – Principe de l'approche Critères Communs

La figure 4.1 montre le principe de développement d'un produit en appliquant l'approche des Critères Communs dans l'objectif d'être évalué et d'obtenir une note grâce à cette norme.

4.2 Exigences fonctionnelles

Pour exprimer les exigences fonctionnelles de sécurité, les CC offre un catalogue de onze classes qui fournissent des spécifications des fonctions de sécurité. Chaque classe est identifiée par un nom de trois lettres, commençant par la lettre F (pour Functional). Ces classes sont ensuite subdivisées en familles, identifiées également par un nom de trois lettres. Enfin, une famille est un ensemble de composants numérotés et regroupés en fonction d'objectifs communs mais qui peuvent différer dans l'accentuation ou dans la rigueur.

Classe	Objectifs	Descrition
FAU	Audit de sécurité	Auditer la sécurité implique la reconnaissance, l'enregistrement, le stockage et l'analyse d'informations associées à des activités touchant à la sécurité.
FCO	Communication	Cette classe ne contient que deux familles, dédiées à la non répudiation des émissions et des réceptions.
FCS	Support cryptographique	Ensemble des composants permettant la gestion de crypto-systèmes.
FDP	Protection des données de l'utilisateur	Cette classe contient des familles qui spécifient des exigences pour les fonctions de sécurités de la TOE et pour les politiques des fonctions de sécurité portant sur la protection des données de l'utilisateur.
FIA	Identification et authentification	Les familles de cette classe traitent des exigences pour que des fonctions établissent et contrôlent l'identité annoncée d'un utilisateur.
FMT	Administration de la sécurité	Cette classe est destinée à définir l'administration de plusieurs aspects des fonctions de sécurité de la TOE : attributs de sécurité, données et fonctions de sécurité.
FPR	Protection de la vie privée	Cette classe contient des exigences qui fournissent à un utilisateur une protection contre la découverte et le mauvais usage de son identité par d'autres utilisateurs.
FPT	Protection des fonctions de sécurité de la TOE	Cette classe contient des familles qui se rapportent à l'intégrité et à l'administration des mécanismes qu'offrent les fonctions de sécurité.
FRU	Utilisation des ressources	Cette classe inclut trois familles qui concernent la disponibilité des ressources nécessaires telles que la capacité de calcul ou la capacité de stockage.
FTA	Accès à la cible d'évaluation	Cette classe spécifie des exigences fonctionnelles pour contrôler l'établissement d'une session utilisateur.
FTP	Chemins et canaux de confiance	Les familles de cette classe fournissent des exigences pour l'établissement d'un chemin de confiance entre les utilisateurs et les fonctions de sécurité de la TOE.

TAB. 4.1 – Les classes d'exigences fonctionnelles

Au sein d'une même famille, il est possible de trouver plusieurs composants ayant un lien d'héritage entre eux, c'est-à-dire qu'un composant fils contient au moins les spécificités de son composant père. Les composants fonctionnels constituent la base des exigences fonctionnelles car ce sont eux qui expriment les comportements de sécurité qui sont attendus d'une TOE.

Pour définir la liste des exigences fonctionnelles de sécurité d'un produit, il suffit alors de choisir l'ensemble des composants correspondants à ces exigences. Pour identifier chaque composant, les CC utilise la notation suivante `Classe_Famille.NumeroDuComposant`. Par exemple, le composant `FMT_MSA.1` désigne le composant numéro `1` de la famille `MSA` (Gestion des attributs de sécurité) de la classe `FMT` (Gestion de la sécurité).

L'objectif du tableau 4.1 n'est pas de faire la liste exhaustive de tous les composants des CC. Il se limite seulement à présenter les objectifs de chacune des onze classes disponibles. La partie 4.4 détaillera seulement quelques composants intéressants utilisés dans le projet POSÉ.

4.3 Exigences d'assurance

Tout comme pour les exigences fonctionnelles, les CC définissent un second catalogue de dix classes cette fois-ci, permettant d'exprimer les exigences d'assurance. L'organisation de ces classes est quasi similaire à celle des exigences fonctionnelles. Les exigences d'assurance sont elles aussi identifiées par un nom de trois lettres, mais commençant par la lettre A (pour Assurance). On retrouve également le découpage en familles puis composants. La seule différence est qu'il n'existe pas d'héritage multiple, c'est-à-dire qu'un composant père d'assurance ne peut avoir qu'un seul composant fils.

Les exigences d'assurance ont un rôle qui est tout aussi important que celui des exigences fonctionnelles. En effet, c'est sur ces exigences d'assurance que repose l'évaluation du produit ou système d'information, auquel on doit accorder sa confiance. Pour ce faire, elles fournissent un ensemble de vérifications à effectuer pour garantir que les exigences fonctionnelles soient bien respectées sur le produit évalué.

Lorsque les experts CC évaluent un produit dans l'objectif de donner une note sur l'échelle des niveaux EAL, ils mesurent la validité de la documentation et du produit ou système d'information, en mettant l'accent de manière croissante, sur le champs d'application, la profondeur et la rigueur. Un niveau d'assurance plus élevé signifie que des efforts plus importants sont fournis dans l'évaluation, pour chacun de ces trois éléments. Le champs d'application désigne la part du SI évalué. Plus elle est importante, plus les efforts d'évaluation sont importants. La profondeur désigne un effort plus important sur le niveau de détail lors de la conception et de l'implémentation. Enfin, la rigueur signifie qu'un effort plus important a été apporté pour structuration et la formalisation lors du développement du produit ou SI.

Le tableau 4.2 dresse la liste des classes disponibles pour les exigences d'assurance. Comme pour les exigences fonctionnelles, quelques composants d'assurance intéressants pour le projet POSÉ seront détaillés dans la partie 4.4.

Classe	Objectifs	Descrition
ACM	Gestion de configuration	Cette classe aide à garantir que l'intégrité de la TOE est bien préservée, en exigeant une discipline et des contrôles dans le processus de raffinement de la TOE.
ADO	Livraison et exploitation	Cette classe définit les exigences pour les mesures, procédures, normes qui parlent de livraison, d'installation et d'utilisation opérationnelle sûre de la TOE.
ADV	Développement	Cette classe définit des exigences pour le raffinement pas-à-pas des fonctions de sécurité depuis les spécifications globales de la TOE jusqu'à l'implémentation effective.
AGD	Guides	Cette classe définit des exigences destinées à permettre la compréhension, la couverture et la complétude de la documentation d'exploitation fournie par le développeur.
ALC	Support au cycle de vie	Cette classe offre des exigences pour obtenir une assurance au moyen de l'adoption d'un modèle de cycle de vie bien défini qui couvre toutes les étapes du développement.
AMA	Maintenance de l'assurance	Cette classe est destinée à maintenir l'assurance que la TOE continuera à satisfaire à sa cible de sécurité quand des changements sont effectués sur la TOE.
APE	Evaluation d'un profil de protection	Cette classe permet de montrer que le profil de protection est complet, cohérent et techniquement correct.
ASE	Evaluation d'une cible de sécurité	Le but de l'évaluation d'une cible de sécurité est de montrer que cette dernière est complète, cohérente, techniquement correcte et convient pour servir de base à l'évaluation.
ATE	Tests	Cette classe formule des exigences de tests qui démontrent que les fonctions de sécurité satisfont aux exigences fonctionnelles de sécurité de la TOE.
AVA	Estimation des vulnérabilités	Cette classe définit des exigences destinées à l'identification des vulnérabilités exploitables. Elle concerne, de façon spécifique, les vulnérabilités introduites pendant la construction, l'exploitation, l'utilisation impropre ou la configuration incorrecte de la TOE.

TAB. 4.2 – Les classes d'exigences d'assurance

4.4 Familles d'exigences de POSÉ

Dans le cadre du projet POSÉ, une extraction d'un certain nombre d'exigences fonctionnelles mais aussi d'exigences d'assurance a été faite à partir des CC. Cette partie propose de présenter celles qui sont intéressantes pour les parties que nous avons traitées au LIG à Grenoble. D'autres exigences ont pu être traitées par nos partenaires du projet mais ne seront pas abordées ici.

Concernant les exigences fonctionnelles de sécurité, nous nous sommes principalement appuyés sur les familles suivantes dont voici un aperçu des objectifs visés :

- `FDP_ACC` : Cette famille concerne les **politiques de contrôle d'accès**. Elle fournissent deux composants qui exigent que soit défini le domaine d'application de la politique au moyen de trois ensembles : les sujets contrôlés par la politique, les objets contrôlés par la politique et les opérations concernant les sujets et les objets contrôlés par la politique.

- `FDP_ACF` : Cette famille concerne les **fonctions de contrôle d'accès**. Elle présente les règles relatives aux fonctions spécifiques qui peuvent implémenter une politique de contrôle d'accès. Elle traite notamment de l'utilisation des attributs de sécurité et des caractéristiques des politiques.

- `FMT_MSA` : Cette famille se rapporte à l'**administration des attributs de sécurité**. Elle vise à permettre aux utilisateurs autorisés de contrôler l'administration des attributs de sécurité.

- `FMT_MOF` : Cette famille s'intéresse à l'**administration des fonctions dans la TSF** (Fonction de sécurité de la TOE). Elle vise à permettre aux utilisateurs autorisés de contrôler l'administration des fonctions de la TSF.

En ce qui concerne les exigences d'assurance, la principale famille intéressante pour la modélisation est la suivante :

- `ADV_SPM` : Cette famille concerne la **modélisation de la politique de sécurité**. Elle a pour objectif d'apporter une assurance complémentaire afin de garantir que les fonctions de sécurité figurant dans les spécifications fonctionnelles mettent en oeuvre les politiques de sécurité de la TOE.

Dans la seconde partie de ce mémoire relative aux réalisations de POSÉ, lorsque des choix seront guidés par des familles d'exigences des CC, ils seront à la fois expliqués et justifiés en mentionnant à quelle famille des CC ils se rapportent. Ces mentions se retrouveront dans les chapitres 6 et 8

Chapitre 5

La méthode B

Ce chapitre a pour objet d'introduire ce qu'est la méthode B [Abr96]. Il permet de se familiariser avec les différentes notions qu'offre cette méthode qui est également un langage de conception et de spécification de programmes. Ainsi, les bases de ce langage seront également présentées. Le lecteur pourra ainsi comprendre ses mécanismes sur des exemples simples, avant que soient développés dans le chapitre 6 les formats des modèles B des politiques de sécurité qui ont été définies pour l'outil Meca.

5.1 Introduction à B

Le langage B est un langage formel de spécification et de conception de programmes. L'expression « méthode B » est fréquemment employée pour désigner l'utilisation du langage B dans le cycle de développement. Le terme méthode sous-entend que B ne se limite pas à un simple langage de spécification. En réalité, il s'agit d'une méthode complète qui fournit aussi des moyens permettant de réaliser des logiciels ou systèmes qui peuvent être prouvés mathématiquement. Grâce à ceux-ci, il est possible de garantir qu'un système produit avec cette méthode, réponde bien à ses besoins.

Ainsi, l'emploi de B implique l'utilisation de raisonnements mathématiques rigoureux, ce qui se traduit par l'obtention de programmes qui sont corrects par construction.

Cette méthode permet en particulier :

- l'écriture de spécifications à état et la preuve de propriétés invariantes sur ces spécifications.
- le raffinement de spécification. Le raffinement permet d'écrire des spécifications plus précises et de montrer leur correction par rapport à la spécification abstraite.
- la production de code. Le dernier niveau de raffinement, appelé *implémentation*, permet de construire des programmes directement exécutables, et qui peuvent être traduits dans différents langages.
- la construction de spécifications et de développements par assemblage.

5.2 Les étapes d'un développement B

Les différentes étapes qui se retrouvent généralement dans un développement en B sont les suivantes. La première est la spécification abstraite de niveau 1. Cette spécification correspond à une modélisation grossière du système et ne s'intéresse pas aux détails de réalisation. L'objectif de cette spécification est de dessiner les grandes lignes du systèmes et de vérifier qu'aucune erreur n'a été introduite. Ensuite, la deuxième étape qui peut être décomposée en de multiples sous-étape correspond à une phase de raffinement. L'objectif de ce processus est d'introduire petit à petit de nouvelles données afin d'obtenir une modélisation plus précise du systèmes à produire. Enfin, la dernière étape de raffinement est appelée implémentation. Cette étape consiste à obtenir uniquement des instructions exécutables par une machine.

Cette étape sera seulement survolée car elle n'a pas été mise en œuvre dans mes travaux qui se situent plutôt en amont des phases de développement.

5.2.1 La spécification

Les spécifications sont décrites en B à l'aide de machines abstraites. Une machine abstraite ressemble à un composant d'un langage de programmation classique (ou à un objet). Les constituants d'une machine sont :

- un état interne ;
- une initialisation ;
- des opérations ;
- des propriétés invariantes.

La méthode B est basée sur la théorie des ensembles. Ainsi, les ensembles servent à modéliser les données du modèle (constantes et variables). L'initialisation et les opérations sont décrites à l'aide du langage des *substitutions généralisées*, qui peut être vu comme un langage de programmation abstrait. Enfin les propriétés sont exprimées à l'aide de la logique du premier ordre, étendue à la théorie ensembliste considérée. Ces différents langages sont récupitulés dans le tableau 5.1

Modèle	Langage
Données	Ensembles
Initialisation / Opérations	Substitutions généralisées
Propriétés	Prédicats du premier ordre

TAB. 5.1 – Les langages de modélisation utilisés dans B

Pour faciliter la compréhension des exemples de ce chapitre ainsi que celle des modèles que j'ai réalisés dans le chapitre 6, un rappel des principales notations logiques et ensemblistes est fait dans l'annexe A.

A partir d'une machine abstraite B, il est possible de produire des *obligations de preuve*. Elles permettent de vérifier que les propriétés invariantes soient bien respectées par la dynamique, c'est-à-dire l'initialisation et les opérations.

5.2.1.1 La machine abstraite

La machine abstraite est le composant B qui correspond au premier niveau de la spécification. Elle est constituée d'un entête, d'un ensemble de données (la partie statique) et d'un ensemble d'opérations (la partie dynamique). L'entête contient le nom de la machine, suivi éventuellement de paramètres qui peuvent être des ensembles ou des valeurs. Cette structure est présentée sur la figure 5.1.

MACHINE
Partie entête :
nom de la machine
paramètres de la machine
contraintes sur les paramètres
Partie statique :
déclaration d'ensembles
déclaration de constantes
propriétés des constantes
variables (état)
invariant (caractérisation de l'état)
Partie dynamique :
initialisation de l'état
opérations
END

FIG. 5.1 – Structure d'une machine abstraite

Dans les parties statique et dynamique, chaque sous-partie est introduite par un mot clé qui définit une clause. Voici les principales clauses d'une machine B.

Dans la partie statique, les données contenues sont les suivantes :

- la définition des ensembles (clause SETS). Ces ensembles peuvent être caractérisés par leur seul nom. Il est également possible de définir les éléments qui les composent (ensemble énuméré). Sous cette clause, il est interdit de définir des ensembles vides ou des ensembles qui ne soient pas finis.
- les constantes. Leur définition est faite par l'énumération de leur nom. Celles-ci peuvent être *abstraites* (clause ABSTRACT_CONSTANTS) ou *concrètes* (clause CONSTANTS ou CONCRETE_CONSTANTS)
- les propriétés des constantes (clause PROPERTIES). Ces propriétés sont exprimées sous la forme d'un prédicat grâce à des axiomes.

- les variables. Leur définition est similaire à celle des constantes. Ainsi, les variables peuvent logiquement être soit abstraites (clause ABSTRACT_VARIABLES), soit concrètes (clause CONCRETE_VARIABLES). Contrairement aux constantes, la clause VARIABLES désigne les variables abstraites et non concrètes.
- un invariant (clause INVARIANT). C'est un prédicat qui donne le typage des variables et les contraintes qu'elles doivent satisfaire.
- des assertions (clause ASSERTIONS). C'est une liste de prédicats qui sont des conséquences logiques des autres axiomes (et de l'invariant) déclarés dans la machine. Ces assertions peuvent être vues comme des lemmes utiles pour les preuves ou pour la compréhension.

Dans la partie dynamique se trouvent les deux sous-parties suivantes :

- l'initialisation des variables (clause INITIALISATION). Celle-ci est une substitution généralisée.
- les opérations (clause OPERATIONS). Chaque opération est composée d'un entête et d'un corps. L'entête contient le nom de l'opération et des paramètres optionels (des paramètres d'entrée et des paramètres de sortie). Le corps est constitué d'une précondition (qui sert à donner le typage des paramètres d'entrée s'ils sont présents) et d'une substitution généralisée qui spécifie l'effet de l'opération.

```
MACHINE
   M(X,u)
CONSTRAINTS
   C              /* spécification des paramètres */
SETS
   S;             /* ensembles donnés*/
   T = a,b        /* ensembles énumérés */
CONSTANTS
   c              /* liste de constantes (concrètes) */
PROPERTIES
   R              /* spécification des constantes */
VARIABLES
   x              /* liste de variables (abstraites) */
INVARIANT
   I              /* spécification des variables */
ASSERTIONS
   J_1;...;J_n    /* liste de prédicats d'assertion */
INITIALISATION
   U              /*substitution d'initialisation */
OPERATIONS
   r ← nom_op(p)≙ PRE P THEN K END
   ...
END
```

FIG. 5.2 – Forme générale d'une machine B

5.2.1.2 Application sur un exemple

Après avoir présenté ce qu'était une machine, cette partie présente un exemple simple [1] de machine abstraite.

Le système à modéliser est un service de réservation de places ayant les caractéristiques suivantes. Les places sont numérotées de 1 à nb_max, cette valeur étant un paramètre de la spécification. Les opérations offertes par ce service permettent de tester s'il reste des places (opération `place_libre`), de réserver une place (opération `réserver`) et de libérer une place déjà réservée (opération `libérer`).

Pour réaliser la machine abstraite du système, on commence par construire sa partie statique, c'est-à-dire ses données.

- Le paramètre nb_max est le nombre maximum de places. Ce paramètre vérifie certaines contraintes : il est supérieur ou égal à 1 et inférieur ou égal à $MAXINT$. Grâce à cette contrainte, il sera plus aisé d'implanter cette spécification sur une machine réelle qui ne supporte que des entiers.
- L'ensemble $SIEGES$ est défini comme l'intervalle $1..nb\_max$, pour des questions de lisibilité.
- L'état du système est modélisé par un ensemble, $occupes$ qui correspond aux places déjà allouées.
- La variable nb_libre, permet d'accéder directement au nombre de places libres. Elle n'apporte pas d'information supplémentaire mais elle est introduite pour illustrer la notion d'invariant.

A partir des données précédentes, trois propriétés invariantes sont considérées :

$$occupes \subseteq SIEGES \quad (1)$$
$$\wedge\ nb\_libre \in 0\ ..\ nb\_max \quad (2)$$
$$\wedge\ nb\_libre = nb\_max - \texttt{card}(occupes) \quad (3)$$

La propriété (2) est une conséquence de la (1) et (3), puisque nb_max est positif (hypothèse sur le paramètre de la machine RESERVATION). Elle permet de typer la variable nb_max.

Maintenant, intéressons nous à la partie dynamique de cette machine.

Initialement, l'ensemble des sièges occupés est vide et le nombre de sièges libres est nb_max. Les opérations offertes ont les comportement suivant :

- l'opération `place_libre` renvoie la valeur de la variable nb_libre.
- l'opération `réserver` a une précondition (notation PRE) :elle ne peut être appelée que si il reste des sièges libres. Dans ce cas, elle attribue une place parmi celles disponibles. Cette opération est non-déterministe : la politique d'allocation des sièges n'est pas précisée.
- l'opération `libérer` a une précondition qui vérifie que la place à libérer est effectivement occupée.

[1] Cet exemple est inspiré d'un support de cours de Marie-Laure Potet.

La machine abstraite correspondante à cet exemple est modélisée sur la figure 5.3.

```
MACHINE
    RESERVATION(nb_max)

CONSTRAINTS
    nb_max ∈ 1 .. MAXINT

DEFINITIONS
    SIEGES == (1 .. nb_max)

VARIABLES
    occupes, nb_libre

INVARIANT
    occupes ⊆ SIEGES
    ∧ nb_libre ∈ 0 .. nb_max
    ∧ nb_libre = nb_max − card(occupes)

INITIALISATION
    occupes := ∅ ||
    nb_libre := nb_max

OPERATIONS
    nb ← place_libre ≙
        BEGIN
            nb := nb_libre
        END ;

    pp ← reserver ≙
        PRE
            card(occupes) ≠ nb_max
        THEN
            ANY place WHERE place ∈ SIEGES − occupes
            THEN
                pp := place ||
                occupes := occupes ∪ {place} ||
                nb_libre := nb_libre - 1
            END
        END ;

    liberer (place) ≙
        PRE
            place ∈ occupes
        THEN
            occupes := occupes - {place} ||
            nb_libre := nb_libre + 1
        END
END
```

FIG. 5.3 – Machine abstraite de l'exemple de réservation de place

5.2.2 Le raffinement

Le raffinement permet d'ajouter des précisions à une spécification. Il permet ainsi de se rapprocher d'un cahier des charges (en phase de spécification) ou d'aller vers du code (en phase de conception). Dans le deuxième cas, le raffinement aura pour but de réduire le non-déterminisme et de représenter les données abstraites.

En pratique, un raffinement se présente sous la même forme qu'une spécification, c'est-à-dire qu'il contient aussi un état interne, une initialisation, des opérations, des propriétés invariantes. Cependant, il contient également une relation entre *variables abstraites* et *variables concrètes*. Cette relation fait partie de l'invariant du raffinement. De plus, il est introduit par le mot clé REFINEMENT au lieu de MACHINE.

5.2.3 L'implémentation

L'implémentation est un raffinement particulier. Elle vérifie certaines conditions correspondant à celles des langages de programmation classiques. Le mot clé IMPLEMENTATION permet de la différencier des raffinements classiques.

A ce niveau, les variables doivent correspondre à des structures de données du langages de programmation (entiers bornés, booléens, tableaux ...). Seules les variables concrètes (clause CONCRETE_CONSTANTS) sont admises dans une implémentation. Les substitutions généralisées autorisées doivent correspondre à des instructions classiques des langages de programmation. Ainsi, les seules instructions B autorisées sont l'affectation, les instructions conditionnelles, le séquencement, l'itération et la déclaration de variable locale. Ce langage est appelé B0. Il offre aussi un ensemble d'opérateurs prédéfinis sur les types permis (opérateurs arithmétiques, accès aux éléments d'un tableau ...).

Deuxième partie

Réalisations

Chapitre 6

Introduction à Meca

Les chapitres de la première partie ont dressé un état des lieux des technologies et des principes mis en œuvre dans le cadre du projet POSÉ. Dans les chapitres de cette seconde partie, je vais maintenant aborder les réalisations auxquelles j'ai participé.

Je commencerai dans ce chapitre par présenter les besoins ainsi que les objectifs définis dans le cadre du projet POSÉ. Je ferai également apparaître la place de Meca dans POSÉ, ainsi que les résultats qui sont attendus par cet outil.

Suite à cette présentation, les chapitres suivants se focaliseront sur l'outil Meca. Le chapitre 7 s'intéressera à l'étude de quelques politiques classiques de contrôle d'accès qui ont été formalisées pour Meca. Ces travaux ont servi de base avant d'aboutir au format de règles défini dans le cadre de la cible IAS et retenu dans le projet POSÉ. Ce format sera détaillé dans le chapitre 8.

Le chapitre 9, quant-à lui, s'intéressera aux développement à proprement dit de Meca. Pour terminer, le chapitre 10 s'intéressera aux aspects organisationnels du projet. Ce sera l'occasion de présenter la méthode utilisée ainsi que la gestion du planning du projet.

6.1 La démarche POSE

Les principaux objectifs du projet POSÉ étaient de deux natures. La première consistait à définir des méthodes et des techniques permettant de valider la conformité d'un sytème aux politiques de sécurité que celui-ci doit respecter. Ensuite, le deuxième objectif était d'outiller cette démarche par un processus complet permettant d'automatiser cette validation. Pour ce faire, le projet POSÉ s'inscrit dans le cadre d'une expérience importante sur la validation fonctionnelle de systèmes de cartes à puce. Pour ces applications, cette validation s'appuie fortement sur des méthodes de tests basées sur des modèles, également connue sous le nom de méthode MBT[1]. La figure 6.1 illustre le principe de cette méthode.

Cette méthode propose de séparer le processus de développement d'un système en deux phases distinctes. D'un côté, l'implémentation du système est réalisée à partir des exigences fonctionnelles de celui-ci. De l'autre, une modélisation fonctionnelle du système est faite de

[1] Model-Based Testing

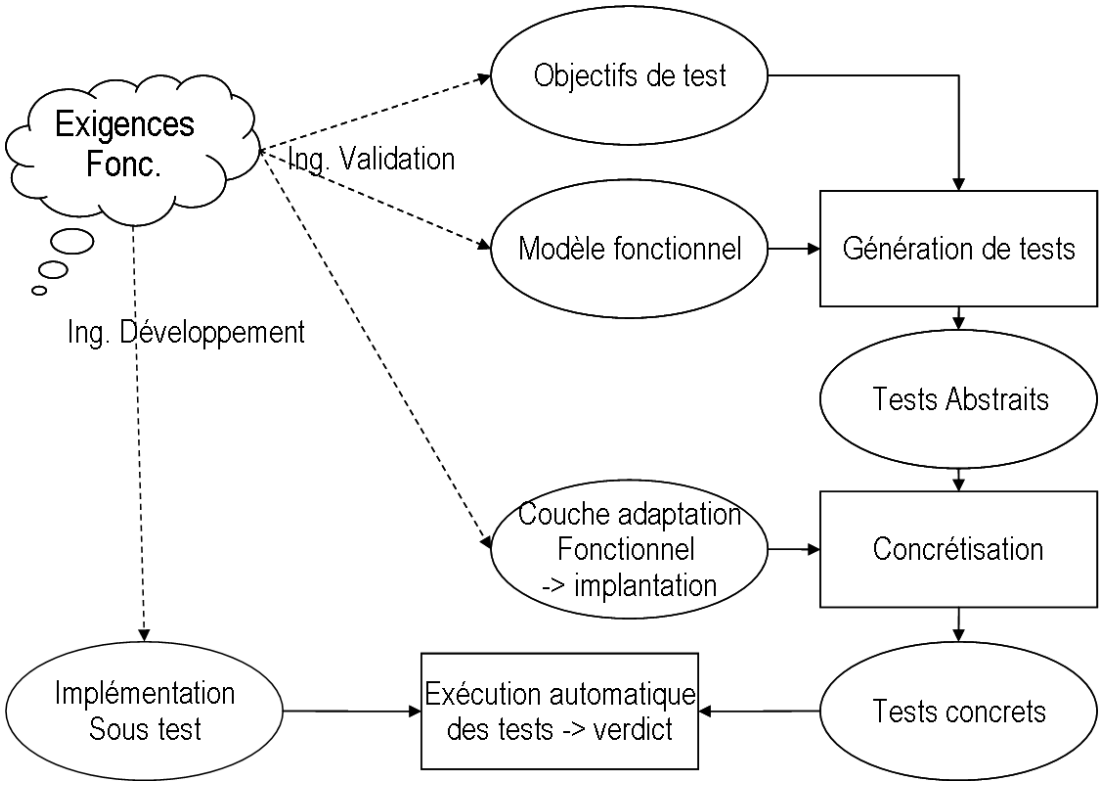

FIG. 6.1 – La méthode MBT

manière indépendante afin qu'elle ne soit pas influencée par les contraintes de réalisation. A partir de ce modèle, il est possible de générer des tests abstraits en fonction d'objectifs de tests définis initialement. En fin de cycle ces objectifs sont validés en exécutant les tests sur l'implémentation. Pour cela, il est souvent nécessaire d'utiliser une couche d'adaptation permettant de rendre compatibles les tests abstraits avec les méthodes réelles de l'implémentation.

La méthode ci-dessus est particulièrement bien adaptée à la validation fonctionnelle de systèmes. Cependant, elle ne permet pas de répondre exactement aux exigences rencontrées dans le cadre de la validation de politiques de sécurité. En effet, afin de permettre la certification de la sécurité des systèmes pour les cartes à puce, il est impératif d'utiliser une méthode permettant d'offrir une traçabilité des exigences de sécurité de haut-niveau. De plus, il est également nécessaire de pouvoir garantir que chacune de ces exigences fasse l'objet de tests dédiés, simulant ainsi des types précis d'attaques comme préconisé par la norme des critères communs. Contrairement à la validation d'exigences fonctionnelles qui se base sur le comportement attendu pour donner le verdict d'un test, la validation d'exigences de sécurité doit veiller à ce que chaque opération ne permette pas d'obtenir des droits qui ne soient autorisés. Ainsi, des comportements fonctionnels anormaux peuvent alors être valides d'un point de vue de la sécurité, tant qu'ils ne violent pas la politique imposée. Par conséquent, le processus de validation de la sécurité ne se substitue pas à la validation fonctionnelle mais ces deux validations sont complémentaires. Enfin, pour valider cette démarche sur des applications industrielles, il était crucial qu'elle soit outillée.

Les applications sur lesquelles est ciblé le projet POSÉ sont développées par le partenaire GEMALTO. Pour effectuer les tests de validation fonctionnelle de celles-ci, GEMALTO dispose déjà de modèles fonctionnels formels écrits avec la notation B. Constituant un effort important, la réutilisation de ces modèles a fait partie des problématiques essentielles de la méthodologie de POSÉ. En effet, mutualiser ces efforts représentait un enjeu majeur pour réduire les coûts lors des phases futures d'industrialisation.

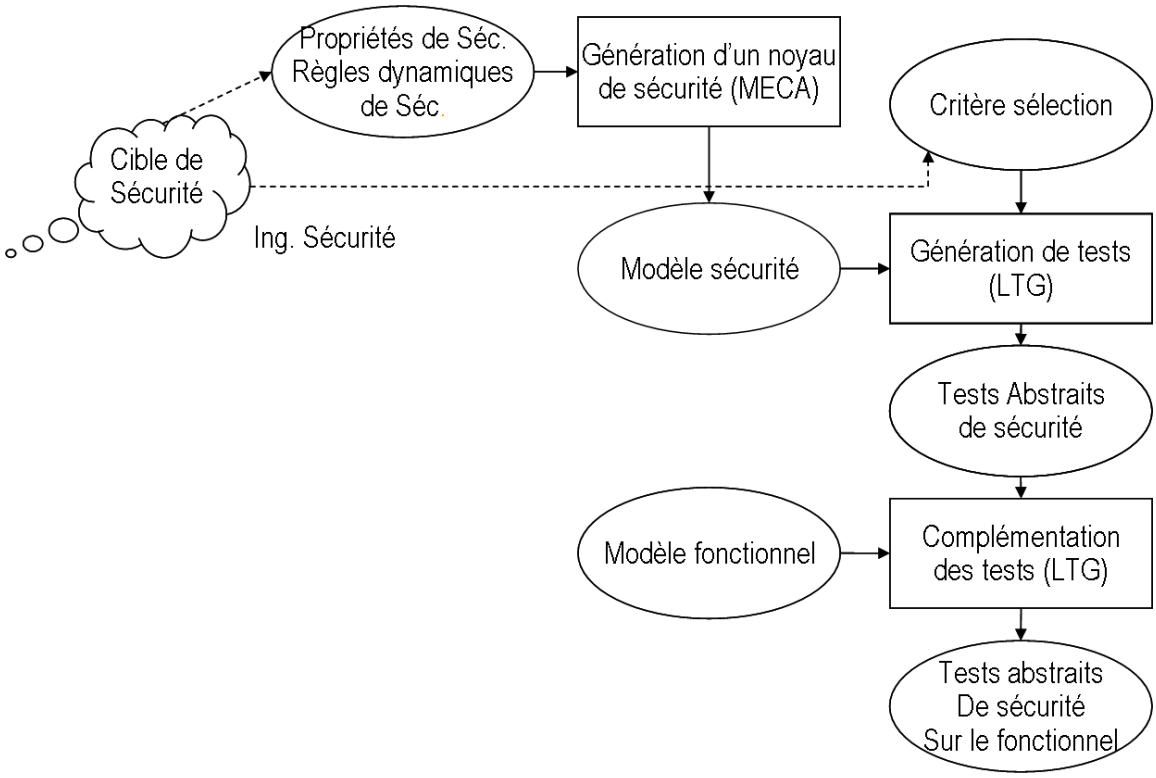

FIG. 6.2 – Processus outillé POSÉ

La figure 6.2 schématise la méthode utilisée dans POSÉ. Cette chaîne de validation de la sécurité inclut notamment les différents outils (nommés entre parenthèses) qui ont été utilisés pour automatiser un maximum de tâches possibles. Le processus proposé ci-dessus, pour aboutir à l'exécution de tests à partir de la spécification de la sécurité, s'appuie sur les étapes suivantes :

- Etape 1 : *Rédaction de la cible de sécurité.* Cette étape consiste à définir les règles de sécurité attendues. Cette cible constitue le référentiel que la vérification devra couvrir.

- Etape 2 : *Formalisation des politiques de sécurité à vérifier.* Le but de cette phase est de traduire la cible de sécurité en modèles formels organisés autour des deux aspects suivants. D'une part, il s'agit de définir de manière statique un certain nombre de règles de sécurité dans un premier modèle. D'autre part, il faut prendre en compte les actions du système pour modéliser la dynamique des attributs de sécurité dans un second modèle.

- Etape 3 : *Production d'un modèle de sécurité appelé TSPM*[2]. Ce modèle est produit à partir des modèles de politiques de sécurité (modèle de règles et modèle dynamique).

- Etape 4 : *Production des tests abstraits de sécurité*. Ils sont générés à partir du TSPM et s'appuient sur des critères de sélection qui correspondent aux objectifs de test.

- Etape 5 : *Complétion des tests abstraits à partir du modèle fonctionnel*. Cette étape est nécessaire pour amener les tests de sécurité au même niveau d'abstraction que les tests fonctionnels. De plus, elle permet de réutiliser la couche d'adaptation réalisée pour les tests fonctionnels.

- Etape 6 : *Exécution après une phase de concrétisation des tests abstraits*. Cette phase permet d'établir le verdict sur le banc de test. N'étant pas propre au tests de sécurité et par conséquent au projet POSÉ, elle n'est pas représentée sur la figure 6.2.

Dans la définition du processus de tests de sécurité POSÉ, les premières phases (étape 1 et 2) doivent obligatoirement être réalisées par des experts (ingénieurs sécurité). Cependant, afin de démontrer la pertinence de cette approche face aux pratiques industrielles, il était impératif de développer des prototypes d'outils permettant de l'automatiser au maximum. C'est ainsi qu'a été développé l'outil Meca pour automatiser l'étape 3 et que l'outil LTG, déjà utilisé pour la génération des tests fonctionnels, a servi de base pour automatiser les étapes 4 et 5.

6.2 Rôle de Meca dans POSE

Après avoir présenté l'approche générale qui a été adoptée dans le projet POSÉ, je vais à présent me focaliser sur l'outil Meca. En effet, cet outil correspond en partie à la contribution du LIG au projet POSÉ car son champ d'application touche aux savoir-faire de l'équipe. De plus, la définition des modèles d'entrée de Meca ainsi que son développement ont été au cœur de mon stage.

La mise en place de l'approche POSÉ a nécessité de définir son cadre théorique. Pour cela, il fallait décrire comment les modèles fonctionnels et de sécurité sont construits et comment on les relie à une application. Le modèle fonctionnel étant déjà exprimé en B, nous avons aussi utilisé cette approche pour exprimer à la fois des politiques de sécurité et la conformité d'une application vis-à-vis de ces politiques. Ceci a été rendu possible par le fait d'une part que la méthode B est bien adaptée au domaine d'application (règles de contrôle d'accès, description abstraite de comportements) et d'autre part supporte un processus de développement allant de la spécification jusqu'à la production de code, basé sur le raffinement. Pour exprimer la conformité, nous avons donc utilisé une forme de raffinement reliant les comportements de l'application à ceux d'une politique de sécurité. Enfin, l'utilisation de la méthode B nous a permis de vérifier des propriétés dès le niveau du modèle de sécurité, ce qui correspond à une demande des Critères Communs.

Nous avons défini des formats de spécification permettant de décrire le modèle de sécurité (composant `ADV_SPM.1` des Critères Communs). Pour rester homogène avec le modèle fonction-

[2]TOE Security Policy Model

nel exprimé en B, nous nous sommes basés sur des formes particulières de machines abstraites B. Enfin, nous nous sommes concentrés sur des politiques de contrôle d'accès, choix qui a été fait dans le projet POSÉ. D'autre part, dans les applications visées, ce contrôle d'accès porte sur l'exécution des commandes (opérations en B). L'expression formelle de la sécurité est donc constituée de deux parties complémentaires :

- Un modèle de règles qui décrit les sujets, les objets, les opérations à contrôler et les attributs de sécurité.
- Un modèle dynamique qui décrit comment les sujets, objets et attributs de sécurité évoluent dans le temps.

Le premier modèle correspond à ce qui est classiquement utilisé pour décrire des politiques de contrôle d'accès : des règles énonçant les conditions sous lesquelles une exécution est permise (les interdictions ne sont pas traitées ici). Le modèle dynamique est moins classique et constitue la partie dédiée à la génération de test, dans le sens où ce modèle décrit les évolutions possibles de l'application à valider.

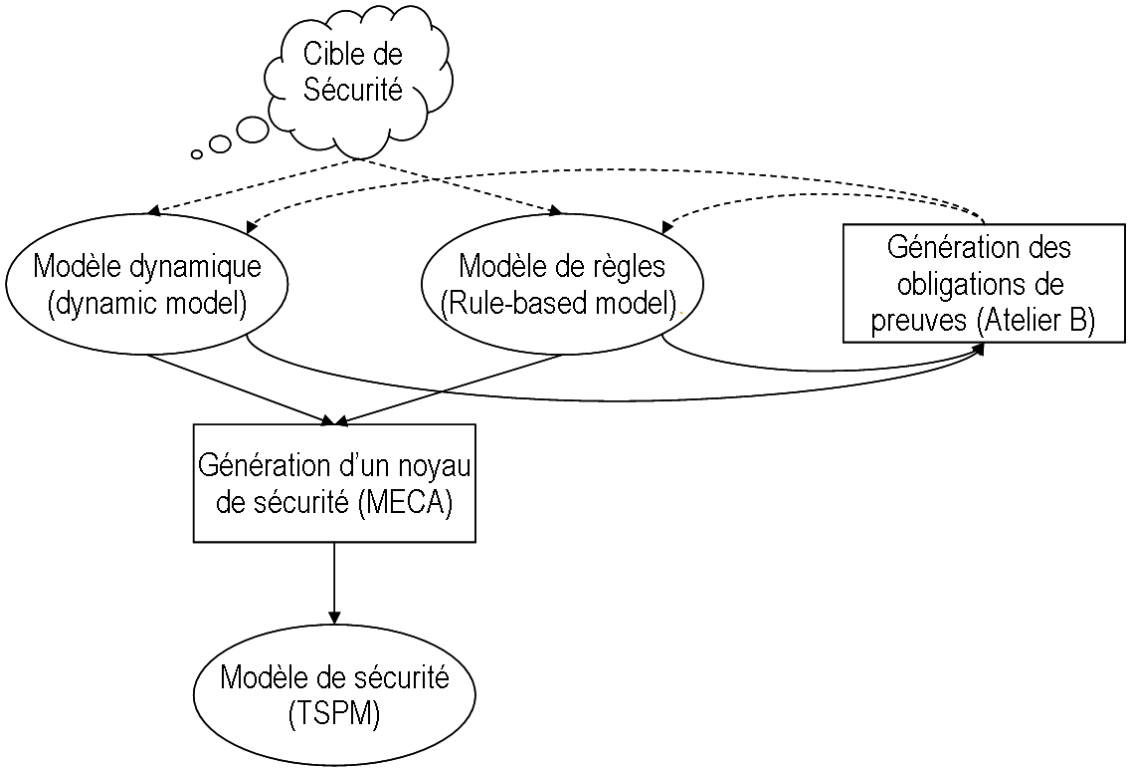

FIG. 6.3 – Principe de l'approche Meca

Ces deux modèles doivent pouvoir être lus et validés indépendamment : en utilisant le principe d'obligations de preuve de la méthode B, on vérifie le bon typage et les propriétés énoncées sur les objets du modèle. Enfin le modèle de règles et le modèle dynamique sont tissés pour produire le modèle de sécurité, entrée du processus de génération de test. Le choix de la séparation en deux modèles (règles et dynamique) repose sur les pratiques : les préconisations Critères Communs pour le contrôle d'accès (famille `FDP_ACC` et `FDP_ACF` des

exigences fonctionnelles de sécurité des Critères Communs) reposent sur la notion de règles. On peut donc assurer une traçabilité entre une cible de sécurité au format Critères Communs et le modèle de règles. De la même manière, le modèle dynamique peut correspondre à une autre famille d'exigences fonctionnelles de sécurité, en lien avec la gestion des attributs de sécurité (famille `FMT_MSA` des exigences fonctionnelles de sécurité des Critères Communs). La figure 6.3 montre le principe de l'approche POSÉ.

La formalisation du modèle de sécurité permet de s'assurer de la consistance de la politique de sécurité, et plus généralement de la valider à l'aide de différents outils (animation, preuve, calcul des comportements autorisés). Il est de plus possible de vérifier des propriétés sur ce modèle, comme des invariants que doivent respecter les attributs de sécurité. Les Critères Communs préconisent de telles vérifications (complétude et cohérence vis-à-vis de la cible de sécurité, consistance du modèle TSPM).

Chapitre 7

Etude de politiques classiques

Dans notre approche de formalisation, nous nous sommes d'abord intéressés aux trois principales familles de politiques de sécurité. Ce sont les politiques discrétionnaires [Lam71], obligatoires [Bib77, BL73] et celles basées sur les rôles [FK92]. Cela nous a d'abord permis de valider notre démarche à partir de politiques connues. Ensuite, cela nous a servi de point de départ pour définir un format plus générique permettant de répondre aux besoins du projet POSE et adapté à la formalisation des politiques de sécurité de la plateforme IAS. Ce chapitre présente les différents formats que nous avons définis pour chaque famille de politiques de sécurité. Elles permettront d'introduire le format spécifique à IAS et qui sera développé dans le chapitre 8.

Pour chaque type de politiques classiques de sécurité, nous avons construit un format particulier d'entrée Meca [Had07]. Ce format est composé d'un modèle déclaratif qui décrit quelles sont les entités qui entrent en jeu dans la sécurité. Il comporte également un second modèle (conformément à la figure 6.3), présentant les fonctionnalités du système en lien avec la politique de sécurité.

A partir de ces modèles, Meca produit un noyau qui dépend de la politique concernée. Dans tous les cas, les opérations qu'il contient recensent quelles actions peuvent être exécutées. Pour chaque action, on retrouve par quel(s) sujet(s) elle peut être exécutée et sur quel(s) objet(s). Lors de la génération du noyau, il est possible de piloter Meca pour qu'il produise au choix une des deux formes suivantes de noyau :

- un noyau *défensif* : Cette forme est orientée pour la génération de tests fonctionnels et permet de garantir qu'une implémentation respecte la politique de sécurité. Cette approche a été présentée dans le cadre d'une publication [DHM09], en la mettant en œuvre sur un exemple.

- un noyau *offensif* : Cette forme est utilisée pour prouver le respect de la spécification fonctionnelle d'une application par rapport à la politique de sécurité. Cette approche a elle aussi été présentée dans le cadre d'une autre publication [HM07].

7.1 Les politiques discrétionnaires

Les politiques discrétionnaires ou DAC[1] [Lam71] accordent au propriétaire de l'information, généralement le créateur, tous les droits d'accès ainsi que la possibilité de les propager aux autres selon sa discrétion. Dans le cadre d'une politique discrétionnaire, les permissions sont vérifiées en fonction des droits possédés par le sujet effectuant l'accès à l'objet. Les permissions attribuées aux sujets se rapportent en général aux droits de lecture, d'écriture ou d'exécution. Par exemple, un sujet s_1 a le droit de lire un objet o_1 s'il possède l'autorisation de lecture sur o_1.

7.1.1 Formats d'entrée Meca pour DAC

Pour formaliser ce modèle dans le cadre de MECA, la première tâche consiste à définir les sujets, les objets et les permissions. Pour ce faire, nous avons proposé deux formats de contrôle d'accès : dans le premier, nous considérons que les sujets sont les opérations du modèle fonctionnel, les objets sont les variables et les permissions sont les accès aux variables en lecture et écriture. Dans le second format, nous considérons que les sujets sont des utilisateurs, les objets sont les opérations du modèle fonctionnel et les permissions sont les actions d'exécution d'opérations.

```
MACHINE
    DAC_Dynamic
VARIABLES
    v1, v2, ...
    us1, us2, ...
INVARIANT
    /* Invariant pour chaque variable */
    ...
OPERATIONS
    list_return ⟵ op1 (list_param) ≙
    PRE
        /* Typage de list_param */
    THEN
        /* Accès en lecture ou en écriture aux variables */
    END;
    ...
END
```

FIG. 7.1 – Schéma du modèle dynamique DAC

La figure 7.1 présente le schéma général du modèle dynamique qui décrit le comportement de l'application. Le schéma de ce modèle est similaire quel que soit le type de politique que nous avons modélisé. Pour cette raison, sa forme ne sera pas fournie à nouveau pour les autres types de politiques classiques abordées.

[1] Discretionary Access Control

MACHINE
DAC_Rules
SETS
SUBJECT = {*op1, op2, ...*} ;
OBJECT = {*v1, v2, ...*}
CONSTANTS
read_permission, write_permission
PROPERTIES
read_permission ∈ SUBJECT ↔ OBJECT ∧
write_permission ∈ SUBJECT ↔ OBJECT ∧
(*op1* ↦ *v1*) ∈ read_permission ∧
...
(*op2* ↦ *v1*) ∈ write_permission ∧
...
END

FIG. 7.2 – Schéma du modèle de règles DAC

La figure 7.2 présente le schéma général du modèle de règle du premier format DAC. Sur celui-ci, on retrouve les deux constantes read_permission et write_permission qui décrivent les permissions en lecture et en écriture que possèdent les sujets sur objets. Ces deux permissions sont des relations de la forme SUBJECT ↔ OBJECT, où A ↔ B dénote une relation binaire entre les deux ensembles A et B. Les éléments de read_permission et write_permission sont des couples notés (*op1* ↦ *v1*), où a ↦ b fait référence à un couple. Signalons que pour tout couple (*op1* ↦ *v1*) qui appartient à read_permission (respectivement write_permission), cela signifie que l'opération *op1* peut lire (respectivement écrire) la variable *v1*.

MACHINE
UAC_Rules
SETS
SUBJECT = {*us1, us2, ...*} ;
OBJECT = {*op1, op2, ...*}
CONSTANTS
execute_permission
PROPERTIES
execute_permission ∈ SUBJECT ↔ OBJECT ∧
(*us1* ↦ *op1*) ∈ execute_permission ∧
...
END

FIG. 7.3 – Schéma du modèle de règles UAC

La figure 7.3 présente le second format discrétionnaire. Sur celui-ci, la constante execute_permission décrit les droits d'exécution que possèdent les utilisateurs sur les opérations. Dans le cadre de Meca, cette variante du format DAC est appelé UAC[2] pour le différencier du

[2]User Access Control

premier format relatif aux contrôle d'accès à la mémoire.

7.1.2 Formes du noyau de sécurité DAC produit par Meca

Après avoir vérifié que les fichiers donnés en entrée sont conformes au format annoncé, MECA génère le noyau de sécurité. Pour chaque format discrétionnaire (DAC ou UAC), Meca peut générer une des deux formes de noyau, telles que présentées en début de ce chapitre.

```
MACHINE
    DAC_Monitor
...
OPERATIONS
    /* accès en lecture */
    val, rs ⟵ read_v1 (su) ≙
    PRE
        su ∈ SUBJECT
    THEN
        IF      su = op1      /* construit à partir du modèle de règles */
        THEN    val := v1 || rs := btrue
        ELSE    rs := bfalse
        END
    END;
    ...
    /* accès en écriture */
    rs ⟵ write_v1 (su, val) ≙
    PRE
        su ∈ SUBJECT ∧
        val ∈ TYPE_val        /* issu du modèle dynamique */
    THEN
        IF      su = op2      /* construit à partir du modèle de règles */
        THEN    v1 := val || rs := btrue
        ELSE    rs := bfalse
        END
    END;
    ...
END
```

FIG. 7.4 – Schéma du modèle du moniteur DAC

La figure 7.4 montre la forme défensive du noyau de sécurité DAC qui est utilisée pour le test. Dans ce noyau, les opérations read_*v1* et write_*v1* sont générées pour chaque variable à protéger en fonction des permissions. Le typage de la variable est récupéré dans le modèle dynamique. Ajoutons que le paramètre val de l'opération write_*v1* référence la nouvelle valeur de l'objet. Pour effectuer la traçabilité lors du test, une valeur de retour, notée rs, a été introduite et permet de savoir si la condition de sécurité a été satisfaite.

Lorsqu'on demande à Meca de produire la forme offensive du noyau de sécurité, il intègre la condition de sécurité (située ici dans le IF) directement dans la précondition. Dans ce cas,

il n'y a plus de valeur de retour rs car l'opération ne sera exécutée que si la précondition est satisfaite. Par la suite, seule la forme défensive des modèles de noyaux de sécurité est présentée.

Pour le second type de politique discrétionnaire UAC, le format du noyau de sécurité concerne la sécurisation de l'exécution des opérations par les utilisateurs. Ainsi, une opération execute_*op1* est générée pour chaque opération *op1* du modèle dynamique. Pour valider le respect de ce type de politique, il suffit de remplacer les appels directs aux opérations du modèle dynamique par un appel à l'opération sécurisée du noyau de sécurité. Ce format apparait sur la figure 7.5.

```
MACHINE
    UAC_Monitor
...
OPERATIONS
    /* accès en exécution */
    list_result, rs ⟵ execute_op1 (su,list_param) ≙
    PRE
        pre_op1 ∧              /* précondition de l'opération op1 */
        su ∈ SUBJECT
    THEN
        IF      su = us1       /* construit à partir du modèle de règles */
        THEN    sub_op1 || rs := btrue
        ELSE    rs := bfalse
        END
    END;
    ...
END
```

FIG. 7.5 – Schéma du modèle du moniteur UAC

Pour chaque opération execute_*op1*, son exécution n'est possible que si l'utilisateur possède ce droit. Cela est traduit par la condition de la clause IF. Si elle est satisfaite, la substitution *sub_op1* associée à l'opération *op1* est réalisée. Les paramètres d'entrée *list_param* et de sortie *list_result* sont ceux qui sont passés à l'opération *op1* du modèle dynamique.

Un exemple est présenté en annexe pour chacune des politiques discrétionnaires (DAC et UAC) dans le cadre de la sécurité des accès sur une application de type porte-monnaie électronique.

7.2 Les politiques obligatoires

Les politiques obligatoires ou MAC[3] ont été proposées à l'origine pour des systèmes militaires. Elles contrôlent le flot d'information afin de contrer le transfert illégal d'informations. Dans le cadre de ces politiques, des niveaux hiérarchiques sont attribués aux sujets et aux objets.

Les permissions d'accès sont calculées à partir de ces niveaux. Cette hiérarchie se matérialise par des classes d'accès et une relation de dominance dont les caractéristiques sont les suivantes :

- Une classe d'accès est affectée à chaque sujet et objet. Elle contient deux composants : un niveau de sécurité et un ensemble de catégories. L'ensemble des niveaux de sécurité est muni d'une relation d'ordre total symbolisé par $\geq$. Dans cette relation, un niveau de sécurité n_1 domine un niveau n_2 ($n_1 \geq n_2$) si n_1 est plus restrictif en terme de sécurité que n_2. L'ensemble des catégories décrit divers domaines du système, par exemple les catégories « nucléaire » et « armée » pour les systèmes militaires.

- La relation de dominance $\geq$ sur les classes d'accès est définie comme suit : Soit L, l'ensemble des niveaux de sécurité munis de la relation $\geq$. Soit C, un ensemble de catégories muni de la relation d'inclusion $\supseteq$. Soient l_1, l_2 deux niveaux de sécurité appartenant à L. Soient c_1, c_2 deux ensembles de catégories inclus dans C. Soient deux classes d'accès ac_1 et ac_2 tel que ac_1 est définie par l_1 et c_1, ac_2 par l_2 et c_2. ac_1 domine ac_2 ($ac_1 \geq ac_2$) si et seulement si : $l_1 \geq l_2 \wedge c_1 \supseteq c_2$.

En fonction de cette classification, des règles contrôlant les accès en lecture et en écriture sont définies dans les modèles traitant ces politiques. Ces modèles visent à préserver la confidentialité (le modèle Bell et Lapadula [BL73]) ou l'intégrité (le modèle Biba [Bib77]) des données.

Pour valider la démarche sur la première version de Meca et afin de ne pas passer trop de temps sur l'étude préliminaire des politiques classiques, les catégories n'ont pas été prises en compte dans les formats proposés. Ainsi, les règles sont vérifiées en fonction des niveaux de sécurité seulement.

7.2.1 Les politiques obligatoires basées sur la confidentialité

Pour préserver la confidentialité, les restrictions sur les opérations de lecture et d'écriture se traduisent par les deux principes suivants :

- No read up : un sujet ne peut lire un objet que si sa classe d'accès domine celle de l'objet (un sujet ne doit pas apprendre des informations qui ne lui sont pas autorisées).

- No write down : un sujet ne peut écrire dans un objet que si la classe d'accès de l'objet domine celle du sujet (un sujet ne doit pas révéler des secrets).

[3] Mandatory Access Control

Il découle de ces deux principes que le flot d'information autorisé est celui qui va du niveau le plus bas vers le niveau le plus haut.

7.2.2 Les politiques obligatoires basées sur l'intégrité

Pour préserver l'intégrité, un sujet ne doit pas écrire dans des objets appartenant à des niveaux plus bas que le sien car il peut y ajouter des informations non appropriées. Il ne doit pas non plus lire des objets appartenant à des niveaux plus bas car il risquerait de diminuer son niveau d'intégrité en se basant sur les informations que ces objets contiennent. Ces deux règles énoncées dans le modèle Biba sont l'inverse des règles concernant la confidentialité, soit :

- No read down : un sujet ne peut lire un objet que si la classe d'accès de l'objet domine celle du sujet.

- No write up : un sujet ne peut écrire dans un objet que si la classe d'accès du sujet domine celle de l'objet.

Il découle de ces deux principes que le flot d'information autorisé est celui qui va du niveau le plus haut vers le niveau le plus bas.

7.2.3 Formats d'entrée Meca pour MAC

Le modèle dynamique utilisé dans le cadre de la formalisation des politiques MAC est similaire à celui des politiques DAC (contrôle d'accès en lecture et écriture à la mémoire). Cependant, le modèle de règle est un peu plus évolué pour prendre en compte les spécificités de ces politiques. Le format de ce modèle est présenté sur la figure 7.6.

MACHINE
MAC_Rules
SETS
SUBJECT = {*op1, op2, ...*} ;
OBJECT = {*v1, v2, ...*}
LEVEL = {*l1, l2, ...*}
CONSTANTS
order, clearance, classification
PROPERTIES
order $\in$ LEVEL $\rightarrow$ NAT_1 $\wedge$
clearance $\in$ SUBJECT $\rightarrow$ LEVEL $\wedge$
classification $\in$ OBJECT $\rightarrow$ LEVEL $\wedge$
order = {(*l1* $\mapsto$ *1*), (*l2* $\mapsto$ *2*), ...} $\wedge$
clearance = {(*op1* $\mapsto$ *l1*), (*op2* $\mapsto$ *l2*), ...} $\wedge$
classification = {(*v1* $\mapsto$ *l1*), (*v2* $\mapsto$ *l2*), ...}
END

FIG. 7.6 – Schéma du modèle de règles MAC

Dans ce format, nous nous intéressons à contrôler les accès en lecture et en écriture effectués par les opérations sur les variables. Par conséquence, les ensembles SUBJECT et OBJECT font

référence respectivement aux opérations et aux variables du modèle dynamique. L'ensemble LEVEL spécifie les différents niveaux hiérarchiques existants.

L'assignation des niveaux aux sujets et aux objets se fait par l'intermédiaire des deux constantes clearance et classification. Chaque sujet et chaque objet est associé à un seul niveau, ce qui explique le choix de la fonction totale symbolisée par $\rightarrow$ pour clearance et classification. La hiérarchie entre les niveaux est matérialisée par la fonction totale order. Un niveau n_1 plus haut qu'un autre n_2 est affecté à un entier plus grand.

7.2.4 Formes du noyau de sécurité MAC produit par Meca

Le format du noyau de sécurité MAC est assez proche de celui du noyau de sécurité DAC. Cependant, la condition de sécurité est un peu plus évoluée que celle de DAC. En effet, elle s'appuie sur les classe d'accès définies en début de partie 7.2 et non plus sur des attributions de permissions directement des sujets aux objets.

La figure 7.7 illustre ce format sur le modèle de confidentialité Bell et Lapadula.

```
MACHINE
    MAC_Monitor
...
OPERATIONS
    /* accès en lecture */
    val, rs ⟵ read_v1 (su) ≙
    PRE
        su ∈ SUBJECT ∧
    THEN
        IF      clearance(su) ≥ classification(v1)
        THEN    val := v1 || rs := btrue
        ELSE    rs := bfalse
        END
    END;
    ...
    /* accès en écriture */
    rs ⟵ write_v1 (su, val) ≙
    PRE
        su ∈ SUBJECT ∧
        val ∈ TYPE_val        /* issu du modèle dynamique */
    THEN
        IF      clearance(su) ≤ classification(v1)
        THEN    v1 := val || rs := btrue
        ELSE    rs := bfalse
        END
    END;
    ...
END
```

FIG. 7.7 – Schéma du modèle de sécurité MAC Bell et Lapadula

Pour chaque variable *v1* appartenant à l'ensemble OBJECT, l'opération read_*v1* est permise par l'opération su (appartenant à l'ensemble SUBJECT) si l'habilitation de celle-ci est supérieure ou égale à la classification de *v1*. L'opération write_*v1*, quant-à elle, est autorisée sur la variable *v1* que si l'habilitation de su est inférieure au niveau de classification de *v1*. Dans le cas de cette dernière opération, le typage du paramètre val est récupéré du modèle dynamique.

Dans le cas du modèle d'intégrité Biba, le noyau de sécurité est similaire à celui généré pour le modèle Bell et Lapadula. La seule différence concerne les règles de sécurité qui sont inversées.

Ainsi, la règle pour l'opération read_*v1* est :

$$\text{clearance(su)} \leq \text{classification}(v1)$$

De même, la règle pour l'opération write_*v1* est :

$$\text{clearance(su)} \geq \text{classification}(v1)$$

7.3 Les politiques basées sur les rôles

Les politiques basées sur les rôles ou RBAC[4] [FK92] ont été principalement conçues afin de prendre en compte des applications déployées sur des vastes organisations ou des applications inter-organisations (Extranet par exemple) [Had95]. L'originalité des politiques basées sur les rôles réside dans le fait d'intercaler des rôles entre les utilisateurs et les permissions. Les permissions sont accordées aux rôles, les utilisateurs se voient affecter un ou plusieurs rôles. Ils obtiennent les permissions affectées aux rôles qu'ils détiennent. Les concepts utilisés dans le modèle RBAC sont les suivants :

- *Les utilisateurs* : ce terme fait en général référence à un être humain mais il peut aussi désigner des machines, des réseaux ou des agents intelligents.

- *Les rôles* : un rôle est une fonction d'un emploi dans une organisation faisant allusion aux responsabilités accordés aux utilisateurs. Les utilisateurs sont assignés à des rôles via une relation utilisateur-rôle.

- *Les permissions* : une permission est une opération effectuée sur un objet. Une relation relie les rôles aux permissions.

[4] Role Based Access Control

Les rôles peuvent être hiérarchisés [San96] : un rôle possède par héritage les permissions qui sont attribués au rôle supérieur. La notion de séparation de tâches [FSG+01] a aussi été ajoutée dans le modèle RBAC. L'idée de base est qu'au sein des organisations, certains rôles dits conflictuels ne doivent pas être attribués au même utilisateur (c'est la séparation statique des tâches). La séparation dynamique des tâches, quant-à elle, interdit à un utilisateur d'exercer des rôles conflictuels au sein d'une même session. La figure 7.8 illustre le modèle RBAC tel qu'il est proposé par le NIST [FSG+01].

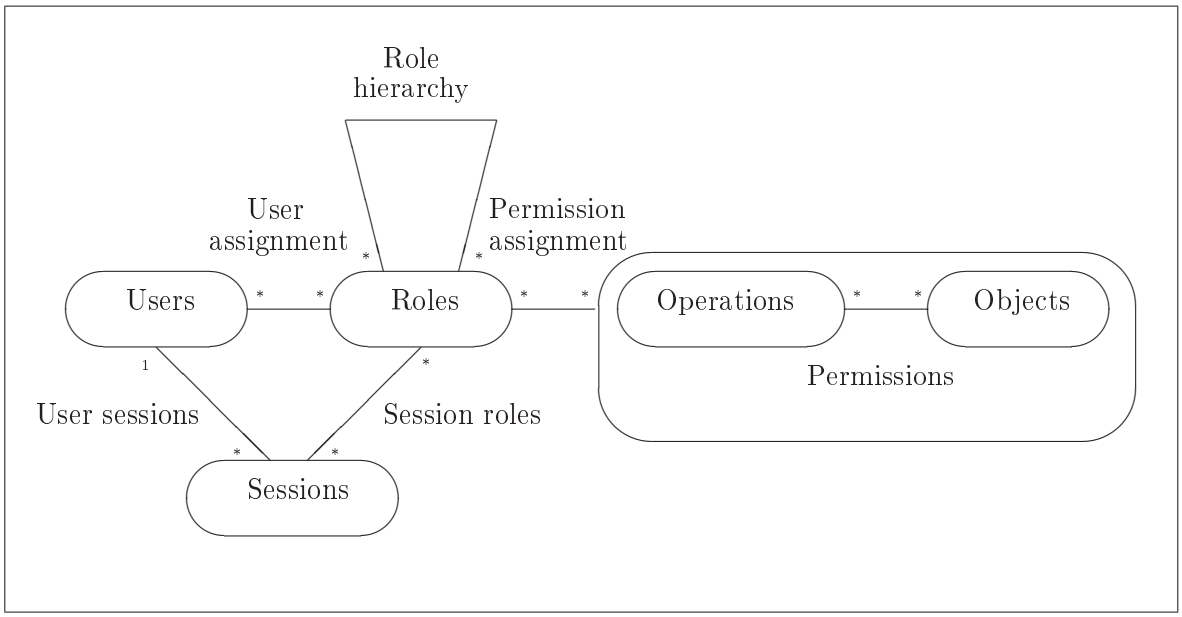

FIG. 7.8 – Le modèle RBAC

7.3.1 Formats d'entrée Meca pour RBAC

Le format du modèle de règles RBAC est présenté sur la figure 7.9. Sur celui-ci, r^{-1} désigne la relation inverse, *r1 ; r2* la composition séquentielle de deux relations et r^{+} la fermeture transitive de la relation r.

Dans ce format, l'ensemble SUBJECT fait référence aux utilisateurs, ROLE fait référence aux rôles affectés aux utilisateurs et PERMISSION décrit les opérations qui peuvent être exécutées par les sujets sur les objets. La constante subject_role est une relation binaire qui décrit l'affectation des rôles aux sujets. Supposons, par exemple, que le couple ($su1 \mapsto ro1$) appartient à subject_role. Cela signifie que le sujet *su1* possède le rôle *ro1*. La constante role_permission décrit l'affectation des permissions aux rôles. La hiérarchie entre les rôles est décrite par la constante inherit_of : si le rôle *ro1* hérite d'un rôle *ro2*, le couple ($ro1 \mapsto ro2$) appartient à inherit_of. La constante conflict est une relation binaire qui contient les couples de rôles conflictuels ne pouvant pas être affectés à un même utilisateur.

En plus de la définition des ensembles et des relations, certaines propriétés devant être vérifiées sur ces relations sont également décrites dans le modèle. Il s'agit des propriétés sui-

MACHINE
RBAC_Rules
SETS
SUBJECT = {*su1, su2, ...*} ;
ROLE = {*ro1, ro2, ...*}
PERMISSION = {*pe1, pe2, ...*}
CONSTANTS
subject_role, role_permission, inherit_of, conflict
PROPERTIES
subject_role $\in$ SUBJECT $\leftrightarrow$ ROLE $\wedge$
role_permission $\in$ ROLE $\leftrightarrow$ PERMISSION $\wedge$
inherit_of $\in$ ROLE $\leftrightarrow$ ROLE $\wedge$
conflict $\in$ ROLE $\leftrightarrow$ ROLE $\wedge$
conflict = conflict^{-1} $\wedge$
conflict $\cap$ id(ROLE) = $\emptyset$
inherit_of^{+} $\cap$ conflict = $\emptyset$
(subject_role ; inherit_of^{+} ; conflict) $\cap$ (subject_role ; inherit_of^{+}) = $\emptyset$
subject_role = {*(su1* $\mapsto$ *ro1), (su2* $\mapsto$ *ro2), ...*}
role_permission = {*(ro1* $\mapsto$ *pe1), (ro2* $\mapsto$ *pe2), ...*}
inherit_of = {*(ro1* $\mapsto$ *ro2), ...*}
conflict = {*(ro1* $\mapsto$ *ro3), ...*}
END

FIG. 7.9 – Schéma du modèle de règles RBAC

vantes :

- La relation conflict est une relation symétrique (conflict = conflict^{-1}), ainsi si le rôle *ro1* est en conflit avec le rôle *ro2*, le rôle *ro2* l'est aussi avec le rôle *ro1*.

- La propriété conflict $\cap$ id(ROLE) = $\emptyset$ décrit l'antiréflexivité de la relation conflict, afin d'éviter qu'un rôle ne soit en conflit avec lui-même.

- La propriété inherit_of^{+} $\cap$ conflict = $\emptyset$ stipule que les deux relations inherit_of et conflict sont exclusives. Cela signifie qu'un rôle *ro1* ne peut pas être en conflit avec un rôle *ro2* dont il hérite.

- La propriété (subject_role ; inherit_of^{+} ; conflict) $\cap$ (subject_role ; inherit_of^{+}) = $\emptyset$ stipule qu'un sujet ne peut pas être affecté à deux rôles conflictuels d'une manière directe ou par héritage.

7.3.2 Formes du noyau de sécurité RBAC produit par Meca

Dans le noyau de sécurité RBAC, la condition de sécurité stipule que le sujet exécutant l'opération doit posséder cette permission, via un rôle qu'il détient directement ou par héritage. Dans le cas où cette condition est remplie, l'opération *op1* est appelée. Pour la génération de ce noyau de sécurité, la précondition *pre_ op1*, les paramètres d'entrée *list_ param* et de sortie

list_ result ainsi que la substitution décrivant le corps de l'opération *sub_ op1* sont extraits du modèle dynamique. Ce noyau est modélisé sur la figure 7.10.

```
MACHINE
    RBAC_ Monitor
...
OPERATIONS
    /* accès en exécution */
    list_result, rs ⟵ execute_op1 (su,list_param) ≙
    PRE
        pre_ op1 ∧              /* précondition de l'opération op1 */
        su ∈ SUBJECT
    THEN
        IF      (subject_role(su) ↦ op1) ∈ (inherit_of+ ; role_permission)
        THEN    sub_ op1 || rs := btrue
        ELSE    rs := bfalse
        END
    END;
    ...
END
```

FIG. 7.10 – Schéma du modèle du moniteur RBAC

Chapitre 8

Le modèle du contrôle d'accès de POSÉ

Les politiques classiques, étudiées dans le chapitre précédent, ont d'abord permis de valider l'approche de Meca. Dans celles-ci, le contrôle d'accès d'une application y était modélisé grâce à la définition de règles statiques, indiquant quelles étaient les actions autorisées. Pour ce faire, chacun des formats proposés exigeait que soient définis la liste des sujets, acteurs du contrôle d'accès, et des objets sur lesquels il s'appliquait. La modélisation du contrôle d'accès, pour ces politiques, consistait alors simplement à définir la liste des couples de sujets autorisés à accéder aux objets.

Cependant, ces politiques présentaient de multiples limitations faces aux besoins du projet POSÉ. En effet, en terme de modélisation des exigences de sécurité, un objectif de POSÉ était d'être aussi proche que possible des pratiques des spécialistes Sécurité, et en particulier des exigences fonctionnelles de sécurité des Critères Communs. Par conséquent, il n'est pas toujours possible, au niveau de la validation du contrôle d'accès de connaître les valeurs précises de tous les sujets et de tous les objets intervenant dans la sécurité. Au contraire, les règles de sécurité qui doivent être modélisées sont souvent plus générales et font plutôt intervenir **des ensembles de sujets et d'objets génériques**. Ce n'est que lors de l'exécution que ces entités seront instanciées sur des entités réelles de l'application.

De même, plusieurs critères essentiels devaient être pris en compte pour que le modèle proposé puisse être utilisable dans une approche de génération de tests basée sur des modèles. Sur les applications ciblées par POSÉ, le contrôle d'accès ne se traduit pas seulement par des permissions simples de sujets accédant à des objets. Ces permissions dépendent parfois d'autres informations qui peuvent être associées aux sujets ou aux objets et dont la valeur dépend de l'exécution. Ce sont ces informations qui sont appelées **attributs de sécurité**.

Pour mieux comprendre la notion d'attributs de sécurité, voici un exemple simple de contrôle d'accès relatif à une application de type porte-monnaie électronique. Cet exemple s'interesse au contrôle de l'exécution de deux opérations, l'une visant à s'authentifier au moyen d'un code PIN (`checkpin`), l'autre permettant de créditer la carte (`credit`). Les sujets correspondent aux différents types de terminaux (terminal bancaire, terminal administratif ou pda). Le contrôle d'accès porte sur l'autorisation des sujets à exécuter les opérations du porte-

```
MACHINE
    Security_Attribute_Rule_Example
SETS
    SUBJECT = {admin, bank, pda} ;
    OBJECT = {checkPin, ...} ;
    MODE = {perso, use, invalid}
CONSTANTS
    /* Les attributs de sécurité */
    mode, isHoldAuth,
    /* La relation de permission */
    permission ∧
PROPERTIES
    permission ∈ SUBJECT ↔ OBJECT ∧
    (mode = use) ⇒ (bank ↦ checkPin) ∈ permission ∧
    (mode = use ∧ isHoldAuth = TRUE ) ⇒ (bank ↦ credit) ∈ permission ∧
    ...
END
```

FIG. 8.1 – Exemple de règles faisant intervenir des attributs de sécurité

monnaie. Les attributs de sécurité, quant-à eux, sont relatifs au cycle de vie de la carte (`mode`) et à l'état des authentifications (`isHoldAuth`). En terme de modélisation, la figure 8.1 montre ce que pourraient être les règles correspondant aux permissions d'exécution de ces opérations. Ici, `mode` et `isHoldAuth` sont les attributs de sécurité dont la valeur peut évoluer pendant l'exécution.

En parallèle à la description des attributs de sécurité qui interviennent dans les règles de permissions, il convenait de formaliser, conjointement au modèle les décrivant, **un second modèle informant sur des aspects dynamiques du système restreint à l'évolution des attributs de sécurité**. Ce modèle a pour vocation de définir toutes les modifications autorisées des sujets, des objets et des attributs de sécurité. Même s'il est moins habituel dans la description de contrôles d'accès, sa présence est justifiée par l'objectif du projet POSÉ, de faire de la génération de tests à partir de modèles. Le modèle dynamique décrira donc toutes les modifications possibles, ceci en donnant une spécification abstraite des opérations de l'application modifiant ces entités.

Le modèle donné dans la figure 8.2 reprend l'exemple précédent. Il décrit la dynamique de l'opération `checkpin`. La substitution `subject :∈ SUBJECT` décrit l'affectation non déterministe du sujet par un élément quelconque de l'ensemble `SUBJECT`. Le processus d'authentification n'étant pas décrit par le modèle considéré, l'initialisation décrit ce processus d'authentification.

Ici l'opération `checkPin` est décrite en terme de son profil, un argument représentant le code PIN présenté en entrée. Le résultat de l'opération est indéterministe : soit l'authentification a réussi (donnée d'un code PIN correct), soit l'authentification a échoué et la carte se bloque. Le modèle dynamique pourrait bien sûr être plus précis, en donnant une spécification déterministe de l'opération `checkPin`. Dans ce cas, des données fonctionnelles devraient être introduites

```
MACHINE
    e_Purse_Dynamic_Example
SETS
    SUBJECT={admin, bank, pda};
    MODE={perso, use, invalid}
VARIABLES
    subject, mode, isHoldAuth
INVARIANT
    subject ∈ SUBJECT ∧ mode ∈ MODE ∧ isHoldAuth ∈ BOOL
INITIALISATION
    subject :∈ SUBJECT  || mode := perso || isHoldAuth := FALSE
OPERATIONS
    res ⟵ checkPin(p) ≙
      PRE p ∈ 0..9999
      THEN
         CHOICE isHoldAuth := TRUE || res := success
         OR isHoldAuth := FALSE || res := failure
         OR isHoldAuth := FALSE || mode := invalid || res := blocked
         END
     END ;
 ...
END
```

FIG. 8.2 – Exemple de spécification pour le test du porte-monnaie électronique

comme la valeur du code PIN et le nombre d'essais. Nous avons choisi ici de construire un modèle dynamique le plus abstrait possible. Néanmoins le paramètre `res` permet de distinguer les comportements possibles de l'opération `checkPin` : `success` si l'authentification réussit, `failure` si l'authentification a échoué et `blocked` si l'authentification échoue et provoque le blocage de la carte. Remarquons aussi que l'opération `checkPin` ne traite pas des aspects fonctionnels, comme le cas où le paramètre d'entrée n'est pas conforme ($p \notin 0..9999$).

Enfin, pour coller au mieux avec les préconisations des Critères Communs, il était nécessaire de permettre la définition de permissions faisant intervenir **des règles ternaires**. L'objectif était ici de généraliser le format des règles présentées au chapitre précédent pour traduire des autorisations exprimant quels sujets peuvent exécuter quelles opérations sur quels objets. Dans les applications du projet POSÉ, les sujets concernés correspondent aux différents types de terminaux, les opérations aux commandes offertes par l'application et les objets aux objets sécurisés de la carte (code PIN ou encore fichier).

8.1 Le modèle de sécurité POSÉ

Après avoir présenté les exigences du projet POSÉ, je vais désormais me focaliser sur la solution qui a été adoptée pour répondre à ces exigences. Dans la mesure du possible, j'en profiterai pour justifier les choix effectués et je préciserai quels critères sont pris en compte.

Les travaux abordés dans le chapitre 7 avaient déjà permis d'étudier certaines extensions imposées par le projet POSÉ. Par exemple, le format de règles appelé UAC (format sur lequel j'ai travaillé lors de la première partie de mon stage), avait déjà servi à la modélisation de règles conditionnées par des attributs de sécurité. Il était donc logique de repartir de ce format en commençant par le généraliser puis en le complétant.

Ainsi, pour satisfaire les exigences listées au début de ce chapitre, un modèle de sécurité POSÉ[1] [DLMP08] est constitué de deux points de vue :

- un *modèle de règles* décrivant les paramètres du contrôle d'accès (sujet, objet, opération et attributs de sécurité) et les règles du contrôle d'accès.

- un *modèle dynamique* décrivant toutes les modifications possibles des attributs de sécurité, des sujets et des objets ainsi que l'état initial.

Le modèle de règles est composé de deux parties. Dans la première, on retrouve la définition des entités impliquées par le contrôle d'accès. La seconde partie décrit les règles du contrôle d'accès. Dans l'approche Critères Communs, lorsqu'on formalise une politique de sécurité, on peut trouver des règles qui sont de deux natures différentes : ces règles peuvent représenter soit des règles d'autorisation, soit des règles d'interdiction. Cependant, la validation de tels contrôles d'accès est assez complexe. Par exemple, il suffit de définir deux règles dans lesquelles on autorise et on interdit à la fois le même appel d'opération pour aboutir à une contradiction. De plus, cette solution n'offre pas la possibilité d'ajouter de nouvelles règles sans remettre en cause les propriétés qui ont pu être validées auparavant sur la politique de sécurité. Pour ces raisons, nous avons fait le choix dans POSÉ de ne formaliser que des règles décrivant des permissions. Pour connaître ce qui est interdit, on se base sur une règle par défaut qui stipule que tout ce qui n'est pas explicitement défini est interdit.

Le modèle dynamique, pour sa part, permet de répondre aux besoins mentionnés en introduction de ce chapitre afin de cibler les évolutions du système qui sont admises par l'application. Son contenu sera détaillé par la suite.

Après avoir présenté les principales évolutions du modèle de sécurité POSÉ par rapport aux modèles de sécurité du chapitre précédent, je vais décrire plus précisément chacun des modèles qui le compose. Je ferai également le lien avec les exigences fonctionnelles de sécurité des Critères Communs. Pour terminer, j'illustrerai la démarche globale du projet POSÉ sur un exemple, en reprenant des parties de l'étude de cas du projet POSÉ.

[1] Livrable 2.5 : Etudes des relations de raffinement : Application de la méthodologie POSÉ à l'étude de cas.

8.1.1 Le modèle de règles POSÉ

Le modèle de règles contient la définition de la politique de sécurité. Pour faciliter l'automatisation des traitements effectués par Meca sur ces modèles, un certain nombre de mots clés, propres à Meca, ont été introduits. Certains sont obligatoires, d'autres optionnels. C'est ainsi que les opérations cibles du contrôle d'accès doivent toujours être définies sous la forme d'un ensemble énuméré (clause SETS en B), grâce à l'identificateur de nom OPERATION. Pour offrir plus de souplesse sur la définition du modèle de règles, deux squelettes sont proposés :

- **un format de règles instancié** : Celui-ci impose que les ensembles de sujets et d'objets du contrôle d'accès soient des ensembles énumérés. On impose alors que la clause SETS du modèle de règles définisse les deux ensembles de noms SUBJECT et OBJECT. Les éléments des permissions correspondent alors à des triplets de la forme :

  $$(subj \mapsto (oper \mapsto obj))$$

  où *subj*, *oper* et *obj* ont respectivement pour valeur un sujet, une opération et un objet qui sont définis dans chacune des listes d'ensembles énumérés.

  Cependant, un des ensembles SUBJECT ou OBJECT peut être omis. Dans ce cas, chacune des permissions ne correspond plus qu'à des couples d'une des formes suivantes :

  $$(subj \mapsto oper)$$
  $$(oper \mapsto obj)$$

  La figure 8.3 présente le schéma de ce format. La notation *C1* désigne la condition pour laquelle la règle 1 est autorisée. Cette condition dépend des attributs de sécurité, notés *(list_attr)*.

```
MACHINE
    POSE_Rules
SETS
    SUBJECT = {s1, s2, ...} ;
    OPERATION = {op1, op2, ...} ;
    OBJECT = {ob1, ob2, ...}
CONSTANTS
    permission, list_attr
PROPERTIES
    ...        /* Déclaration des attributs de sécurité */
    permission ∈ SUBJECT ↔ OPERATION ↔ OBJECT ∧
     C1(list_attr) ⇒ (s1 ↦ (op1 ↦ ob1)) ∈ permission ∧
    ...
END
```

FIG. 8.3 – Schéma du modèle de règles POSÉ

- **un format de règles générique** : Dans celui-ci, chaque règle de permission est relative à un sujet et/ou à un objet courant. Ces deux entités sont définies par l'utilisateur comme

constantes (clause CONSTANTS du modèle B) et il est possible, comme pour le format précédent, d'en omettre une.

Dès lors qu'elles sont définies, Meca impose que chacune de ces entités soit typée de la manière suivante :

$$curr\_subj \in SUBJECT\_SET$$
$$curr\_obj \in OBJECT\_SET$$

où $SUBJECT\_SET$ et $OBJECT\_SET$ représentent des ensembles définis dans la clause SETS.

En généralisant ce type de politique de sécurité, un autre intérêt par rapport au format énuméré est d'offrir plus de souplesse sur les conditions des règles. Notamment, il est possible de les paramétrer pour qu'elles dépendent du sujet ou de l'objet sur lequel porte la règle considérée (voir le schéma sur la figure 8.4).

```
MACHINE
    POSE_Rules_Generic
SETS
    SUBJECT_SET ;
    OPERATION = {op1, op2, ...} ;
    OBJECT_SET
CONSTANTS
     permission, list_attr,
    curr_subj, curr_obj
PROPERTIES
    ...        /* Déclaration des attributs de sécurité */
    permission ∈ SUBJECT_SET ↔ OPERATION ↔ OBJECT_SET ∧
     C1(list_attr, curr_subj, curr_obj) ⇒ (curr_subj ↦ (op1 ↦ curr_obj)) ∈ permission ∧
    ...
END
```

FIG. 8.4 – Schéma du modèle de règles générique POSÉ

En ce qui concerne les exigences fonctionnelles de sécurité des Critères Communs, les deux formats ainsi définis prennent en compte les classes suivantes :

- la définition des *sujets, objets et opérations* concernés par le contrôle d'accès sont relatifs aux composants de la classe `FDP_ACC`.

- la définition des *attributs de sécurité*, c'est-à-dire les informations dont dépendent les autorisations/interdictions est relative au composant `FDP_ACF.1.1` .

- la définition des *règles du contrôle d'accès* sont relatives, quant-à elles, à trois composants : le composant `FDP_ACF.1.2` concerne les *règles d'autorisations* alors que les composants `FDP_ACF.1.3 et 1.4` sont relatifs aux *règles d'interdictions*, et ne sont pas traités dans nos modèles.

8.1.2 Le modèle dynamique POSÉ

En complément du modèle de règles, le modèle dynamique sert à décrire les évolutions autorisées des sujets, des objets et des attributs de sécurité. Ainsi, dans le cadre des applications cartes à puces que nous considérons, le modèle dynamique spécifie :

- les attributs de sécurité.
- la liste des sujets qui apparaissent dans le modèle de règles (ensemble SUBJECT) ou le sujet courant dans le cas du format générique.
- la liste des objets du modèle de règles (ensemble OBJECT) ou l'objet courant.
- les opérations de la cible de sécurité décrites en terme des modifications qu'elles effectuent sur les attributs de sécurité. Chaque opération du contrôle d'accès (valeur de l'ensemble OPERATION du modèle de règles) doit être spécifiée.
- les opérations qui ne font pas parties de la cible mais qui font elles aussi évoluer les attributs de sécurité.
- l'initialisation des attributs de sécurité.

Le schéma 8.5 montre le squelette du modèle dynamique relatif au format énuméré alors que le 8.6 présente le cas du format générique.

```
MACHINE
    POSE_Dynamic
SETS
    SUBJECT = {list_subj} ;
    OBJECT = {list_obj}
CONSTANTS list_cst PROPERTIES ...
VARIABLES list_var INVARIANT ...
INITIALISATION ...
OPERATIONS
    list_return ⟵ op1 (list_param) ≙
    PRE pre_op1
    THEN body_op1
    END;
    ...
END
```

FIG. 8.5 – Schéma du modèle dynamique de POSÉ

En résumé, pour chaque nom d'opération qui se trouvent dans l'ensemble OPERATION du modèle de règles, lorsque le modèle de sécurité contient des règles génériques portant sur un sujet courant et un objet courant, la correspondance entre les entités du modèles de règles avec celles du modèle dynamique doit respecter la relation :

$$\{curr\_subj, curr\_obj\} \cup list\_attr \subseteq list\_cst \cup list\_var \cup list\_param$$

```
MACHINE
    POSE_Dynamic_Generic
CONSTANTS list_cst PROPERTIES ...
VARIABLES list_var INVARIANT ...
INITIALISATION ...
OPERATIONS
    list_return ⟵ op1 (list_param) ≙
    PRE pre_op1
    THEN body_op1
    END;
    ...
END
```

FIG. 8.6 – Schéma du modèle dynamique générique de POSÉ

Comme pour le modèle de règles, on peut établir des correspondances entre le modèle dynamique et les exigences de sécurité des Critères Communs. En particulier, les exigences de sécurité relatives à la gestion des attributs de sécurité sont regroupées dans la famille `FMT_MSA` des critères communs et sont prises en compte par les classes suivantes :

- les *sujets autorisés à accèder ou modifier les attributs de sécurité* sont préciser dans le composant `FMT_MSA.1`.
- la *validité des valeurs des attributs de sécurité* vis-à-vis de certaines propriétés de sécurité est garantie par le composant `FMT_MSA.2` .
- l'initialisation statique des attributs de sécurité se réfère au composant `FMT_MSA.3`.

D'autres composants de la classe d'exigence fonctionnelle FMT peuvent être concernés, en particulier ceux relatifs au management des fonctions de sécurité et des rôles (`FMT_MOF`).

8.1.3 Le mécanisme de correspondance des entités des modèles de règles et dynamique

Pour permettre la définition de chacun des modèles précédents de manière indépendante, il était intéressant d'offrir un mécanisme de nommage des entités. Ce mécanisme avait d'abord pour objectif que les noms de ces entités ne soient pas forcément identiques dans le modèle de règle et le modèle dynamique, mais également que Meca puisse retrouver leur correspondance afin de permettre l'automatisation du traitement de ces modèles. Dans cette optique, nous avons associé à chaque modèle un fichier de nommage, dont le rôle est d'indiquer quelles entités sont présentes dans le modèle, et également de donner leur nom, tel qu'il apparait.

Lorsque nous nous sommes posés la question de la syntaxe à adopter pour ces fichiers, il nous a semblé logique de choisir la même notation que les modèles du contrôle d'accès. Il y avait plusieurs raisons à cela. Premièrement, l'utilisation d'une seule méthode, et pour les modèles, et pour les fichiers de nommage, allait dans le sens de la simplification de l'écriture d'un modèle de sécurité pour Meca. Deuxièmement, cela permettait de mutualiser les efforts de développements en réutilisant les analyseurs que j'avait réalisé sur les versions précédentes de Meca.

MACHINE
POSE_Rules_Naming
SETS
ENTITIES
CONSTANTS
SUBJECT, current_subject,
OBJECT, current_object,
null,
/* Définition du nom du sujet courant utilisé dans le modèle de règles */
SUBJ_SET, subj_name ,
/* Définition du nom de l'objet courant utilisé dans le modèle de règles */
OBJ_SET , obj_name
PROPERTIES
SUBJECT $\subset$ ENTITIES $\wedge$
current_subject $\in$ SUBJECT $\wedge$
OBJECT $\subset$ ENTITIES $\wedge$
current_object $\in$ OBJECT $\wedge$
null $\in$ ENTITIES $\wedge$

/* Typage par l'utilisateur des entités du modèle de règles */
SUBJECT = *SUBJ_SET* $\wedge$ *subj_name* $\in$ SUBJECT $\wedge$
OBJECT = *OBJ_SET* $\wedge$ *obj_name* $\in$ OBJECT $\wedge$
/* Définition du nom du sujet courant (null si non défini) */
current_subject = *subj_name* $\wedge$
/* Définition du nom de l'objet courant (null si non défini) */
current_object = *obj_name* $\wedge$
END

FIG. 8.7 – Schéma du modèle de nommage du modèle de règles

Une fois que le modèle de règles a été formalisé, l'écriture du fichier de nommage qui lui est associé doit respecter le schéma de la figure 8.7. Par convention, certains identificateurs sont obligatoires et doivent toujours avoir le même nom. Ils permettent à Meca de faire la correspondance entre chaque modèle et servent de critères de recherche. C'est le cas de l'ensemble ENTITIES, qui sert au typage, et des constantes SUBJECT, OBJECT, current_subject et current_object qui servent à référencer les noms utilisés pour chaque entité correspondante dans le modèle de règle. l'identification null permet d'indiquer à Meca qu'une entité optionnelle est omise du modèle.

```
MACHINE
    POSE_Dynamic_Naming
SETS
    ENTITIES ;
    DEFINITION = {operation_parameter, variable, constant, undef}
CONSTANTS
    SUBJECT, current_subject, current_subject_definition ,
    OBJECT, current_object, current_object_definition ,
    null,

    /* Définition du nom de l'objet courant utilisé dans le modèle dynamique */
    obj_name ,
    /* Définition du nom du sujet courant utilisé dans le modèle dynamique */
    subj_name
PROPERTIES
    SUBJECT ⊂ ENTITIES ∧
    current_subject ∈ SUBJECT ∧
    current_subject_definition ∈ DEFINITION ∧
    OBJECT ⊂ ENTITIES ∧
    current_object ∈ OBJECT ∧
    current_object_definition ∈ DEFINITION ∧
    null ∈ ENTITIES ∧

    /* Typage par l'utilisateur de l'objet et du sujet du modèle dynamique */
    subj_name ∈ SUBJECT ∧
    obj_name ∈ OBJECT ∧

    /* Définition du nom du sujet courant et de l'endroit ou il est défini (null si non défini) */
    current_subject = subj_name ∧
    current_subject_definition = variable ∧

    /* Définition du nom de l'objet courant et de l'endroit ou il est défini (null si non défini) */
    current_object = obj_name ∧
    current_subject_definition = constant ∧
END
```

FIG. 8.8 – Schéma du modèle de nommage du modèle dynamique

Concernant le modèle dynamique, son fichier de nommage est identique à celui de règles (voir schéma 8.8). Cependant, du fait que le sujet et l'objet courant des règles peuvent se

retrouver parmi plusieurs clauses du modèle dynamique (variables ou constantes), voire même sous la forme de paramètres d'opérations, deux identificateurs supplémentaires ont été ajoutés. Il s'agit de current_subject_definition et current_object_definition qui indiquent où sont définies les entités correspondantes.

8.1.4 Le noyau de sécurité POSÉ

Le modèle de règle et le modèle dynamique peuvent être tissés pour produire un noyau de sécurité, dont le rôle est de mettre en oeuvre le contrôle d'accès. Chaque demande d'exécution d'une opération cible du contrôle d'accès sera traitée par le noyau de sécurité : si les droits sont accordés l'appel aura effectivement lieu, sinon le noyau de sécurité empêchera l'exécution. Nous décrivons ci-après comment ce noyau est produit à partir du modèle de règles et du modèle dynamique.

Le noyau de sécurité contient toutes les déclarations du modèle dynamique. Pour chaque opération du modèle dynamique il contient l'opération contrôlée associée. Le modèle dynamique doit aussi contenir le sujet et l'objet courant sous la forme soit d'une constante, soit d'une variable, soit d'un paramètre des opérations.

La première forme de ce noyau est offensive et se construit de la manière suivante :

```
MACHINE
    POSE_Offensive_Monitor
CONSTANTS list_cst PROPERTIES ...
VARIABLES list_var INVARIANT ...
INITIALISATION ...
OPERATIONS
    /* accès en exécution */
    list_result ⟵ execute_op1 (list_param) ≙
    PRE
        subject = s1 ∧ object = o1 ∧  C1(list_attr)  ∧
        pre_op1 ∧                /* précondition de l'opération op1 */
    THEN
        body_op1
    END;
    ...
END
```

FIG. 8.9 – Schéma du modèle offensif du moniteur de POSÉ

Dans ce noyau l'exécution de l'opération est préconditionnée. L'appel ne sera effectif que si on a pu établir la précondition. Ce type de noyau est plutôt utilisé pour la preuve, lorsqu'on veut montrer qu'une application met correctement en oeuvre une politique de contrôle d'accès.

Une autre solution est de produire un noyau défensif : cf. figure 8.10.

```
MACHINE
    POSE_Defensive_Monitor
CONSTANTS list_cst PROPERTIES ...
VARIABLES list_var INVARIANT ...
INITIALISATION ...
OPERATIONS
    /* accès en exécution */
    rs, list_result ⟵ execute_op1 (list_param) ≙
    PRE
        pre_typ ∧                /* précondition de typage de l'opération op1 */
    THEN
        IF
            subject = s1 ∧ object = o1 ∧  C1(list_attr)  ∧
            pre_op1 ∧                /* précondition de l'opération op1 */
        THEN
            body_op1 || rs := btrue
        ELSE
            rs := bfalse
        END
    END;
    ...
END
```

FIG. 8.10 – Schéma du modèle défensif du moniteur de POSÉ

Dans ce noyau les opérations sont défensives : lorsque les conditions de sécurité ou la précondition ne sont pas valides alors rien ne se passe, c'est-à-dire que ni les sujets, ni les objets et ni les attributs de sécurité ne sont modifiés. On réalise la substitution skip sur les attributs de sécurité. Le résultat `rs` permet de savoir si l'appel a été autorisé ou non (résultat OK/KO). Ce noyau est plus proche des implémentations et peut être utilisé par exemple pour le test. La précondition `pre_typ` est une précondition de typage des paramètres. Elle peut être extraite syntaxiquement de la précondition des opérations.

Chapitre 9

Développement de Meca

Les premiers chapitres de la seconde partie ont consisté à présenter la démarche de modélisation adoptée dans le projet POSÉ. Ils ont également servi à détailler les formats des modèles qui ont été définis, tout en justifiant ces choix. Ces chapitres traitaient des aspects aussi bien théoriques que méthodologiques du projet POSÉ. Je vais à présent aborder dans ce chapitre les aspects relatifs aux réalisations logicielles que j'ai développées dans le cadre du développement de Meca.

9.1 Objectifs de Meca

Le logiciel Meca intervient le plus en amont de la démarche outillée de POSÉ. A partir de deux modèles écrits en B (modèle de règles et modèle dynamique), il produit le noyau de sécurité correspondant, également en B (Cf chapitres 7 et 8). Son principe de fonctionnement est schématisé sur la figure 9.1.

Les principaux rôles de l'outil Meca sont les suivants :

- guider l'utilisateur dans la phase de modélisation de la cible de sécurité en proposant des squelettes de formats prédéfinis, en fonction des besoins de sécurité à couvrir. Ces squelettes ont été présentés dans les deux chapitres précédents.

- effectuer des vérifications sur les modèles fournis en entrée de Meca et indiquer à l'utilisateur les erreurs qui peuvent être décelées.

- vérifier la cohérence entre le modèle de règles et le modèle dynamique, grâce notamment aux fichiers de nommage dans les cas des formats des modèles de règles POSÉ.

- automatiser la phase de tissage du modèle de règles avec le modèle dynamique afin de produire, de manière systématique, le noyau de sécurité.

- offrir le choix dans la génération du noyau de sécurité, entre une forme défensive utilisable pour de la génération de test et une forme offensive plutôt destinée à faire de la preuve sur les modèles.

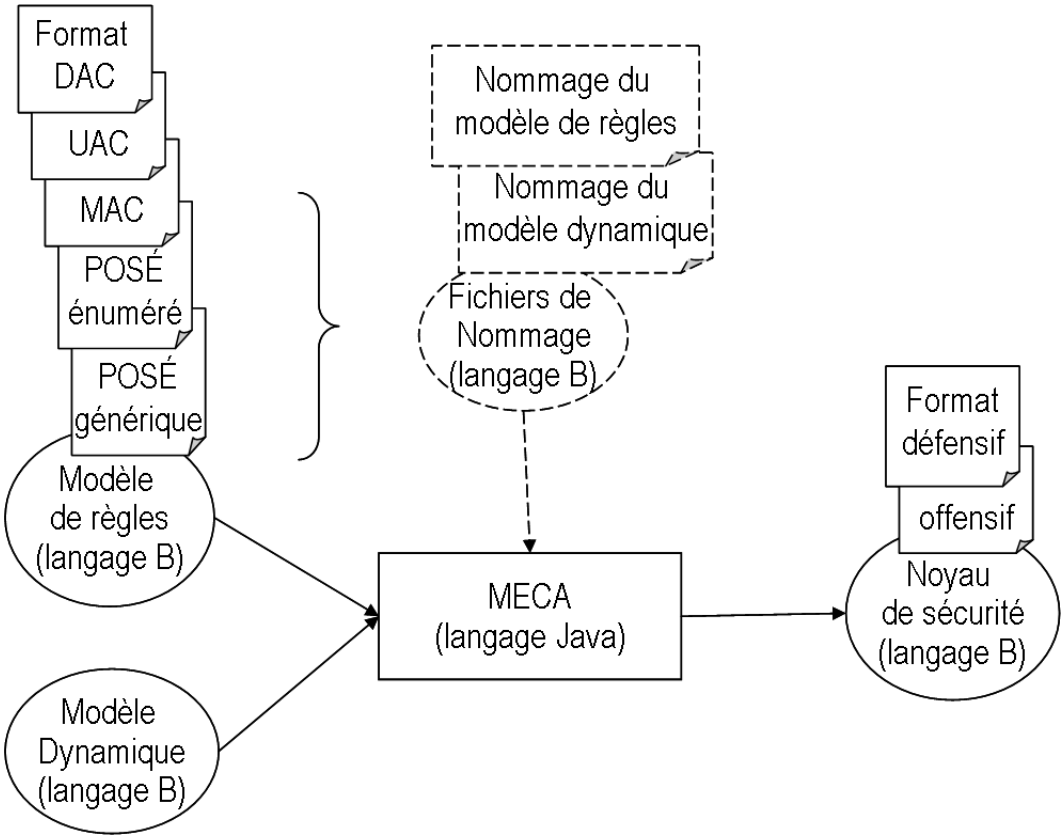

FIG. 9.1 – Les entrées et les sorties de Meca

Quand j'ai débuté mon stage, un prototype de Meca avait déjà été réalisé par une personne au cours de son année de DEA [Had05]. Il contenait une classe unique pour quelques formats (une classe de 1300 lignes pour le format DAC, une de 1000 lignes pour MAC et une de 450 lignes pour RBAC). Ces classes permettaient de produire le noyau de sécurité sur un exemple de modèle de règles et de modèle dynamique. Cependant, ces classes ne faisaient que peu ou pas de vérifications sur les modèles traités et n'étaient pas suffisamment robustes pour élargir leur champ d'application à d'autres modèles que ceux des exemples proposés. L'objectif était donc de développer un outil intégré qui soit relativement fiable et puisse accepter n'importe quels modèles pour chacun des formats proposés.

Lors de la définition du cahier des charges de Meca, plusieurs contraintes étaient imposées.

La première concernait l'utilisation du **langage Java**. D'une part, l'équipe VASCO utilise couramment ce langage pour ses développements. Elle dispose ainsi de multiples compétences dans sa programmation. De plus, comme le prototype avait été écrit en Java, il était logique de repartir de celui-ci. D'autre part, l'équipe travaille sur des modèles en langage B depuis de nombreuses années. Pour faciliter les traitements sur ce type de modèles, il était naturel de s'appuyer sur une bibliothèque Java (appelée « **Boîte à outils** B » ou « **BoB** »), que l'équipe a développé et enrichi au cours de ses différents projets. Le cœur de cette bibliothèque est

un analyseur qui permet de construire un arbre syntaxique correspondant à une machine B. La BoB fournit également un ensemble de classes Java permettant de parcourir cet arbre et de manipuler ses éléments. Enfin, elle fournit quelques fonctions pour la réalisation de calculs usuels sur des machines B (calcul de la plus faible précondition, etc).

Une seconde contrainte portait sur la prise en compte dès la conception de Meca de certains principes du génie logiciel :

- **la lisibilité du code** : le code devait être structuré et commenté de manière suffisamment claire pour qu'il puisse être rapidement repris à la suite de mon stage pour le faire évoluer.

- **la modularité et l'extensibilité** : une partie des traitements effectués par Meca est identique quelques soient les formats de modèles utilisés, qu'il s'agisse des algorithmes de recherche, des algorithmes de vérification des modèles d'entrée ou encore de ceux de production de code B. Il était donc important de découper le logiciel en modules et fonctions spécialisés afin de capitaliser les efforts de validation et de gagner du temps pour l'extention des formats supportés, voire l'ajout de nouveaux formats.

- **la réutilisabilité** : dans un soucis de réutilisabilité du code sur d'autres projets de l'équipe, les différentes briques logicielles développées pour Meca devaient être organisées en paquetages Java qui puissent être intégrés indépendamment les uns des autres.

- **la documentation du code** : il convenait d'intégrer au fur et à mesure du développement les commentaires spécifiques qui permettrait de générer automatiquement la documentation des classes au moyen de l'outil Javadoc, intégré au kit de développement Java.

Dans la suite de ce chapitre, je vais aborder les différentes étapes que j'ai suivies lors du développement de Meca. Ainsi, j'introduirai dans un premier temps la BoB en décrivant son architecture et les classes que j'ai utilisées. Je présenterai ensuite la librairie que j'ai réalisée pour Meca. Le but de cette librairie est d'encapsuler toutes les fonctions qui sont partagées par les différents générateurs de noyau de sécurité de Meca. Je terminerai enfin par le cœur de Meca, en décrivant ce que font les différents générateurs de noyau de sécurité en fonction des formats traités.

9.2 La Boîte à outils B (BoB)

La BoB est une librairie qui a été développée au sein de l'équipe VASCO du LIG et qui permet de manipuler des constructions B. Elle est contenue dans l'archive Java **bob.jar**. Cependant, avant de pouvoir travailler sur une machine B avec la BoB, il est nécessaire que celle-ci soit chargée en mémoire. Cette opération est effectuée grâce à l'outil jbtools[1], développé au LIFC[2]. Cette outil est contenu dans l'archive Java **B.jar**.

[1] http ://lifc.univ-fcomte.fr/ tatibouet/JBTOOLS/
[2] Laboratoire d'Informatique de l'Université de Franche-Comté

Les deux archives bob.jar et B.jar sont distribuées sous la forme d'un fichier compressé **bobUtil.tar.gz** [3] et sont diffusées sous contrat de licence CeCILL [4].

9.2.1 Chargement en mémoire d'une machine B

Pour charger une machine B contenue dans un fichier, il est nécessaire d'utiliser le parseur de l'outil jbtools au moyen de la méthode `analyse` fournie par la classe `BParser`. Ce parseur effectue l'analyse syntaxique du fichier et retourne sa représentation sous la forme d'une liste chaînée. Si une erreur de syntaxe est détectée dans le fichier, l'exception `ParseException` est levée. De même, si une erreur se produit à l'issue de l'analyse, l'exception `AfterParseException` peut également être levée.

Si tout se passe bien lors de cette phase d'analyse, la BoB peut alors utiliser la liste chainée obtenue pour manipuler la représentation de la machine B contenue dans le fichier. Pour cela, il faut utiliser la méthode `traduire` de la classe `Traduire` du paquetage `bob.traduire`. Cette méthode prend en paramètre le premier élément de la liste chaînée (objet de la classe `Noeud`) et construit un arbre abstrait correspondant à la machine B. Cet arbre est retourné sous la forme d'une instance de la classe `TComposant`. La figure 9.2 montre les classes de l'outil jbtools utilisées (paquetage `tatibouet`[5]) et la relation qui existe avec la bibliothèque BoB (paquetage `bob`).

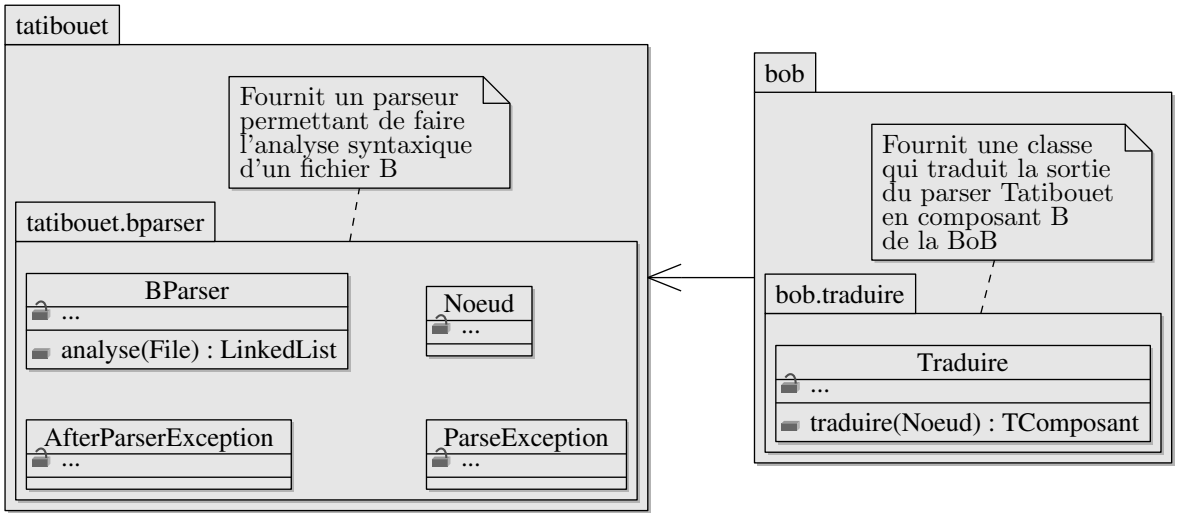

FIG. 9.2 – Diagramme UML de paquetages du parseur Tatibouet de l'outil jbtools

[3] http ://www.lri.fr/ stoulsn/index.php ?ZoomSur=Logiciels#Logiciels

[4] CeCILL (**Ce**a ; **C**nrs ; **I**nria ; **L**ogiciels **L**ibres).
Il s'agit d'un contrat de licence de diffusion de logiciels libres. Le contrat est disponible sur : http ://www-lsr.imag.fr/Les.Personnes/Thierry.Moutet/licence/Licence_CeCILL_V2-fr.php

[5] parseur développé par Bruno Tatibouet

9.2.2 Contenu de la boîte à outils B

Une fois que l'on a récupéré une instance de type `TComposant` représentant un fichier B, il est possible d'effectuer toutes sortes d'opérations sur ce composant. Pour ce faire, la BoB fournit 9 paquetages dont les rôles sont décrits sur la figure 9.3.

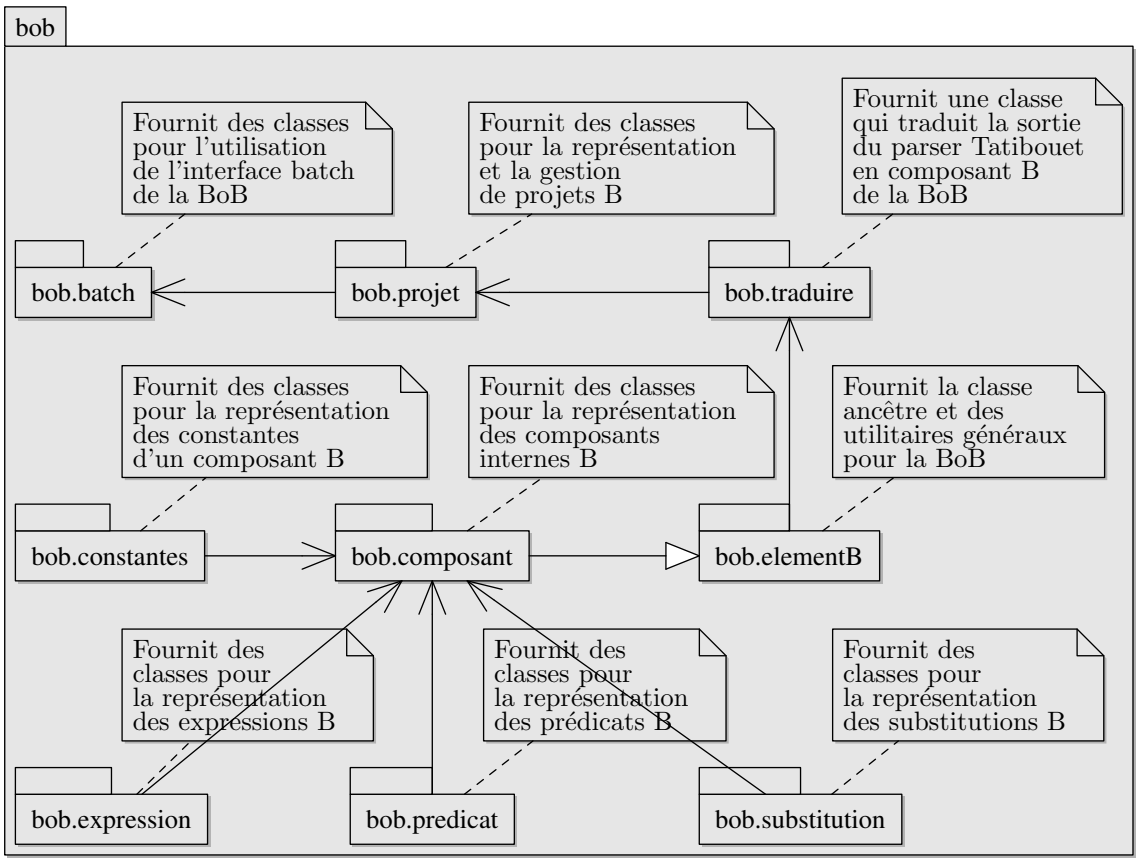

FIG. 9.3 – Diagramme UML de paquetages de la BoB

Le paquetage `bob.batch` fournit une interface batch en ligne de commande et est plutôt destiné à manipuler des machines B en mode « standalone », c'est-à-dire en exécutant directement la méthode `main` de la BoB.

Le paquetage `bob.projet`, quant-à lui, permet de gérer plusieurs machines B au sein d'un même projet. Cela est intéressant lorsqu'il existe une relation entre celles-ci. Ces deux paquetages n'ont pas été utilisés dans la conception de Meca. En effet, comme les machines B traitées étaient indépendantes, leur gestion a simplement été faite en instanciant plusieurs composants de type `TComposant`.

En ce qui concerne les autres paquetages, `bob.traduire` n'a été utilisé que pour obtenir une instance de `TComposant` (comme présenté dans la section 9.2.1). Toutes les manipulations réalisées sur ces instances ont été faites au moyen des paquetages bob.composant, bob.constantes, bob.predicat, bob.expression et bob.substitution. Ces 5 paquetages seront détaillés dans les sections suivantes.

9.2.3 Classes fournies dans le paquetage bob.composant

Le paquetage `bob.composant` fournit un ensemble de classes dédiées à la manipulation d'un composant B (voir figure 9.4).

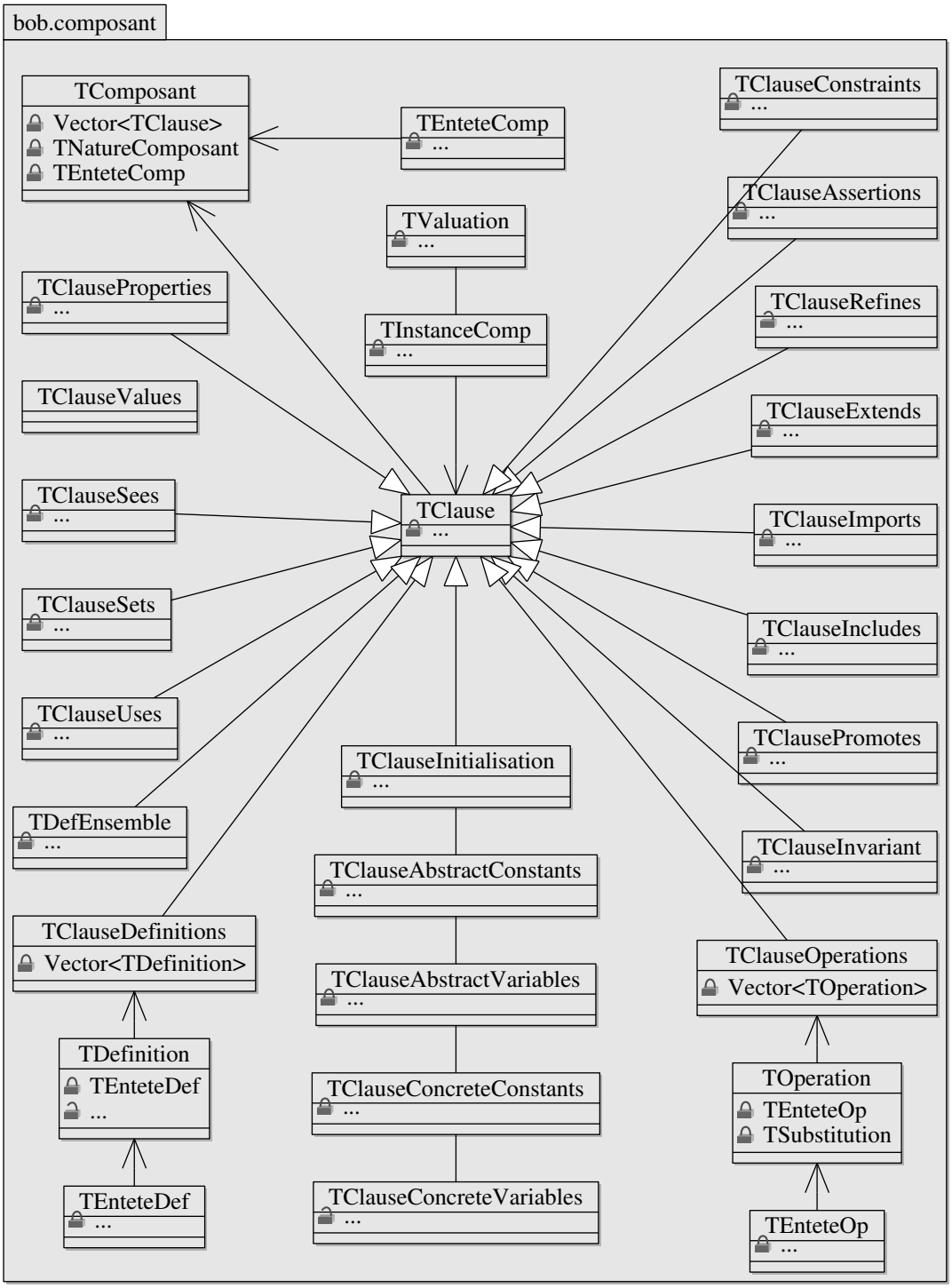

FIG. 9.4 – Diagramme UML de classes du paquetage bob.composant de la BoB

Ainsi, il est possible à partir d'une instance de la classe `TComposant`, d'accéder aux éléments suivants du composant B grâce à ses méthodes : sa nature (machine, raffinement ou implémentation B), son entête (nom de la machine, paramètres d'entrée et de sortie) et la liste des clauses qu'elle contient.

Pour chacune des clauses du composant, une classe spécifique, héritant de la classe `TClause`, permet d'accéder à son contenu. Selon le type de la clause, celle-ci peut être constituée de constantes, d'expressions, de prédicats ou encore de substitutions.

La classe `TComposant` fournit également une méthode (`Afficher`) qui permet d'afficher le composant vers un flux de sortie de type `writer`. Grâce à elle, on peut obtenir la traduction inverse du parseur tatibouet, c'est-à-dire passer d'un composant B en mémoire à une sortie imprimable d'une machine B. Lorsqu'on souhaitera produire des fichiers de sortie B, on utilisera donc cette méthode en lui passant une instance de type `FileWriter` (`writer` d'écriture vers un fichier).

9.2.4 Classes fournies dans le paquetage bob.constantes

Le paquetage `bob.constantes` fournit un jeu de classes qui permettent de représenter les symboles constants de la syntaxe B ($\in$, $\subset$, ...). Ces classes héritent toutes de la classe `TOperateur`. Parmi celles-ci, on trouve la nature du composant (classe `TNatureComposant`), ou encore tous les types d'opérateurs qui peuvent être utilisés. Ces classes sont modélisées dans la figure 9.5.

Ces classes sont utilisées par Meca pour de multiples traitements. Par exemple, pour extraire le prédicat de typage dans la précondition des opérations du modèle dynamique, Meca ne conserve que les prédicats ensemblistes pour lesquels l'opérateur est une instance de type `TOpEnsemble` qui correspond aux constantes `APPARTIENT` ou `INCLUS` et où l'identificateur situé à gauche de l'opérateur est un paramètre de l'opération.

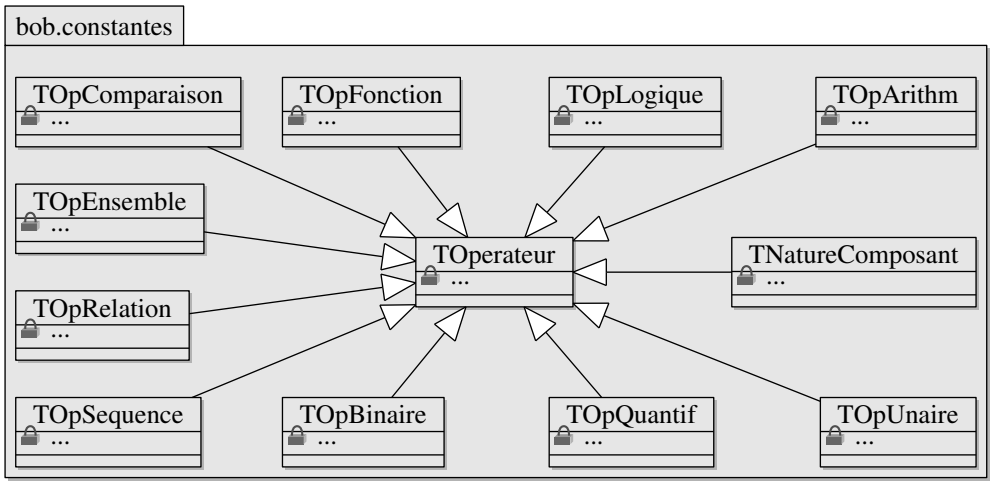

FIG. 9.5 – Diagramme UML de classes du paquetage bob.constantes de la BoB

9.2.5 Classes fournies dans le paquetage bob.predicat

Le paquetage `bob.predicat` fournit toutes les classes qui permettent de représenter un prédicat du langage B. Il est modélisé dans la figure 9.6. Pour manipuler ces prédicats, six classes (héritant de la classe `TPredicat`), sont disponibles :

- `TPredComparaison` représente les prédicats de la forme :
  `TExpression TopComparaison TExpression`

- `TPredConst` représente les prédicats `VRAI` et `FAUX`

- `TPredEnsemble` représente les prédicats de la forme :
  `Vector<TExpression> TopEnsemble TExpression`

- `TPredLogique` représente les prédicats de la forme :
  `TPredicat TopLogique TPredicat`

- `TPredParenthese` représente les prédicats de la forme :
  `(TPredicat)`

- `TPredQuantif` représente les prédicats de la forme :
  `TopQuantif(Vector<TExprIdentificateur>).(TPredicat)`

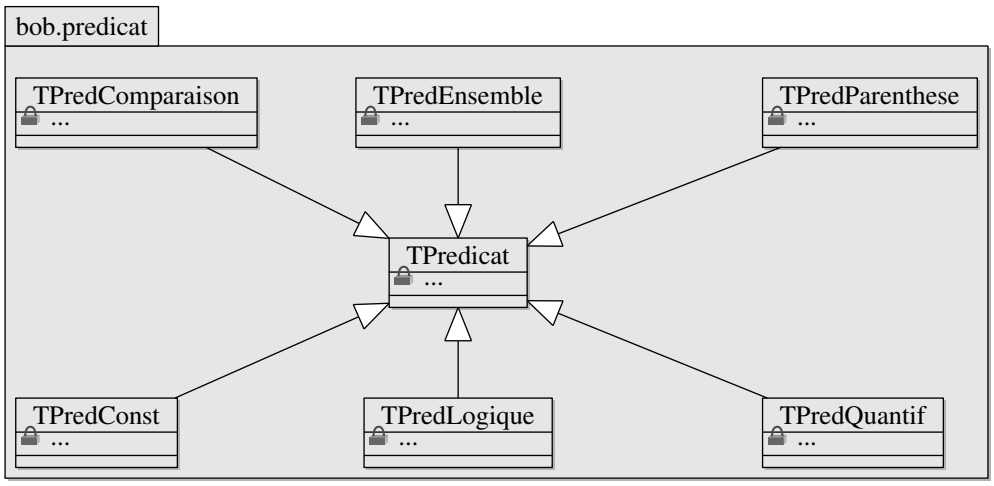

FIG. 9.6 – Diagramme UML de classes du paquetage bob.predicat de la BoB

On peut remarquer qu'un prédicat logique (instance de la classe `TPredLogique`), est composé de sous-prédicats. La recherche d'une forme particulière de prédicat dans un prédicat logique nécessite donc de faire appel à un algorithme récursif. Cette algorithme sera par exemple nécessaire pour extraire toutes les règles de permission du modèle de règles, les règles étant reliées entre elles par des « et logique ».

9.2.6 Classes fournies dans le paquetage bob.expression

Toutes les classes qui sont fournies dans le paquetage `bob.expression` servent à manipuler les expressions B. La modélisation UML de ce paquetage est présentée sur la figure 9.7. Tout comme pour les prédicats, il existe des expressions B qui sont elles-mêmes constituées de sous-expressions. C'est notamment le cas pour les cinq types d'expressions suivantes :

- `TExprOpBinaire` représente les expressions de la forme :
  `TExpression TOpBinaire TExpression`

- `TExprOpUnaire` représente les expressions de la forme :
  `TExpression TOpUnaire`

- `TExprParenthese` représente les expressions de la forme :
  `(TExpression)`

- `TExprSequence` représente les expressions de la forme :
  `[TExpression [,TExpression [,...]]]`

- `TExprEnsExt` représente les expressions de la forme :
  `{TExpression [,TExpression [,...]]}`

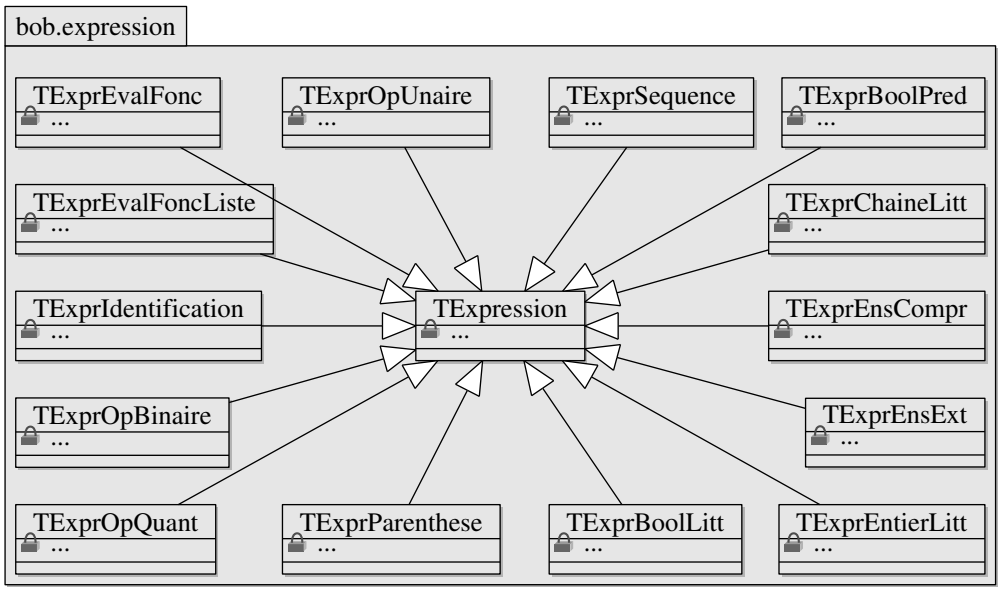

FIG. 9.7 – Diagramme UML de classes du paquetage bob.expression de la BoB

On retrouve donc, ici aussi, la nécessité d'employer des algorithmes récursifs dés lors qu'il s'agit de manipuler ou de rechercher des expressions particulières dans un composant B.

Pour illustrer la façon dont procèdent les deux paquetages précédents pour manipuler les prédicats et les expressions, l'exemple suivant donne sur la figure 9.8 l'arbre abstrait produit par la BoB pour le prédicat suivant :

(**subject** $\mapsto$ **object**) $\in$ **permission**

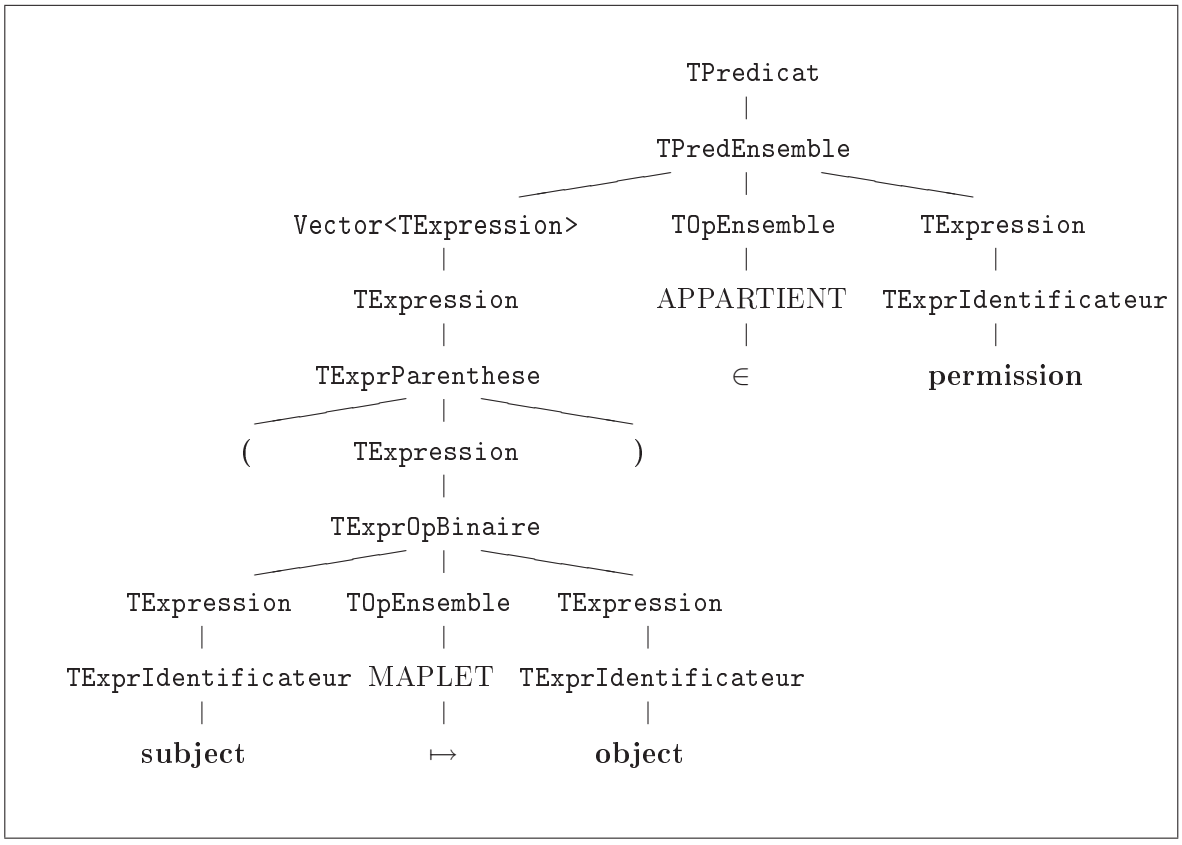

FIG. 9.8 – Représentation d'un prédicat avec la BoB

9.2.7 Classes fournies dans le paquetage bob.substitution

Le paquetage `bob.substitution` fournit des classes permettant de manipuler toutes les substitutions d'un composant B. La figure 9.9 présente sa modélisation.

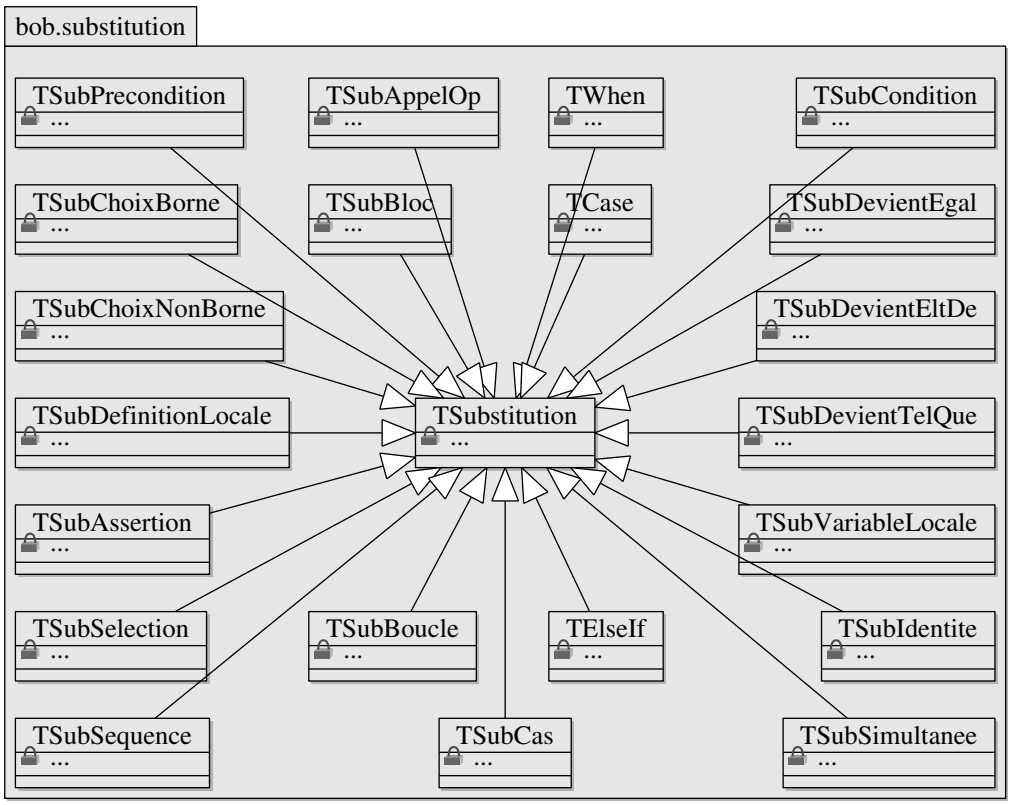

FIG. 9.9 – Diagramme UML de classes du paquetage bob.substitution de la BoB

Dans Meca, les classes de ce paquetage seront essentiellement utilisées pour deux sortes de manipulations. La première correspond à la récupération des opérations du modèle dynamique et à leurs traitements. Parmi ceux-ci, on peut citer l'extraction de la précondition de chaque opération et dans celle-ci, la récupération du prédicat de typage. Le second usage de ces classes correspond à la génération des opérations du noyau de sécurité.

9.3 Les classes de Meca

Maintenant que l'architecture de la BoB et les interactions entre ses paquetages ont été présentées, je vais montrer l'organisation de l'outil Meca et détailler son développement.

Préalablement à la conception de Meca, j'ai commencé par analyser la possibilité de réutiliser le prototype qui avait été réalisé. C'est ainsi que j'ai commencé par chercher à comprendre son fonctionnement et essayer d'étendre son code. Mais face à sa complexité d'écriture (une classe unique contenant une seule méthode, écrite de manière linéaire) et aux contraintes qui en découlaient (difficulté pour la rendre modulaire et extensible), je me suis rapidement orienté vers la décision de redévelopper complètement Meca. L'objectif était de profiter de cette occasion pour bien définir le fonctionnement de Meca afin qu'il réponde au mieux aux attentes exprimées dans son cahier des charges.

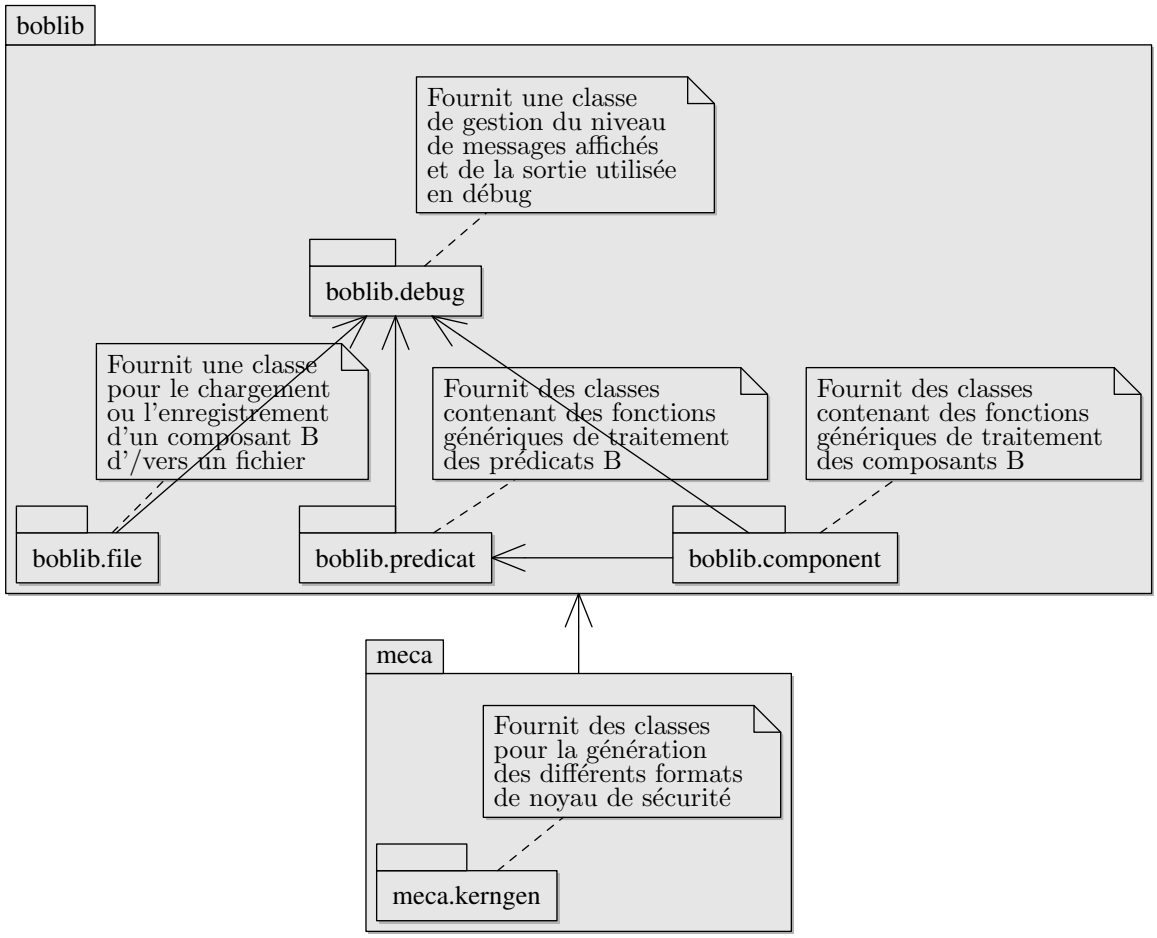

FIG. 9.10 – Diagramme UML de paquetages de Meca

Lors de la conception de Meca, j'ai choisi de structurer cet outil en deux modules :

- Le premier, appelé `boblib`, consiste en une sur-couche sur la BoB. L'idée était de fournir à Meca une librairie plus évoluée que la BoB et mettant à sa disposition un ensemble de services et de méthodes pour les usages qui sont fréquents et relativement génériques.

- Le second module, appelé `meca`, correspond au cœur même de Meca. Pour commencer, il regroupe tous les générateurs de noyau de sécurité Meca, pour chacun des formats d'entrée admis. Ensuite, il intègre une interface utilisateur servant au paramétrage de l'outil et permettant de lancer la génération d'un noyau. Cette interface fournit également une aide en ligne spécifiant l'usage de l'outil.

La figure 9.10 montre le diagramme UML des paquetages qui composent Meca. Le module boblib regroupe les quatre paquetages suivants :

- le paquetage `boblib.debug` : L'objectif de ce paquetage est de fournir des interfaces destinées à l'affichage ou l'enregistrement des messages renseignant sur l'état de la génération en cours. Il permet de centraliser la gestion des messages avec pour objectif de simplifier l'évolution ou la migration de l'interface utilisateur (par exemple, passage ultérieur à une interface graphique).

- le paquetage `boblib.file` : Le rôle de ce paquetage est d'encapsuler les fonctions de mise en mémoire d'un composant B à partir d'un fichier ou la génération d'un fichier à partir d'un composant déjà chargé en mémoire.

- le paquetage `boblib.predicat` : Ce paquetage fournit un ensemble de classes pour la manipulation des prédicats B (algorithmes de recherche dans un prédicat, génération de prédicats particuliers, ...).

- le paquetage `boblib.component` : Ce paquetage fournit un ensemble de classes pour la manipulation de composants B (recherche de prédicat dans un composant, application de filtres sur un composant, ...).

Le module meca, quant-à lui, est composé comme suit :

- le paquetage `meca.kerngen` : Ce paquetage fournit une classe pour chacun des formats de modèles de sécurité. Chacune de ces classes fournit une implémentation d'un générateur de noyau de sécurité.

- la classe `Meca` : Cette classe correspond à l'interface utilisateur de Meca.

Après avoir donné le modèle conceptuel de Meca, je vais à présent me focaliser sur chacun des paquetages qui le composent.

9.3.1 Classes du paquetage boblib.debug

Le paquetage `boblib.debug` fournit la classe `Debug`. Le rôle de cette classe est de gérer tous les messages qui sont affichés pour le dialogue avec l'utilisateur. Les méthodes de cette classe sont modélisées sur la figure 9.11.

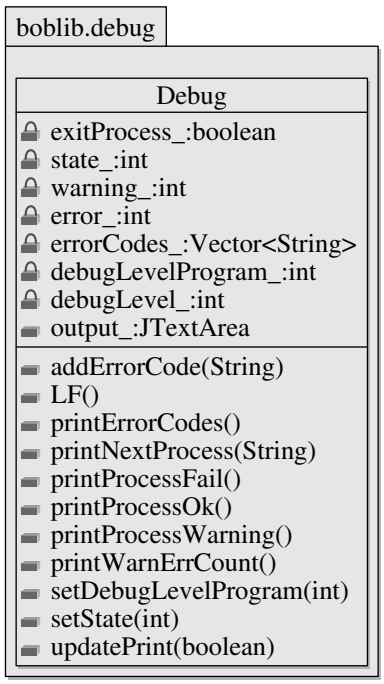

FIG. 9.11 – Diagramme UML de classes du paquetage boblib.debug

La classe `Debug` offre de multiples services. Premièrement, elle permet de définir le canal de sortie qui est utilisé pour l'affichage des messages. Dans la version que j'ai développée, la canal de sortie utilisé est celui de la sortie du terminal à partir duquel est lancé le programme. La possibilité d'exécuter Meca en ligne de commande présente l'avantage de pouvoir facilement automatiser cette tâche au moyen de scripts batch ou en appelant directement Meca à partir d'un autre programme. Cependant, j'ai aussi eu l'occasion au cours de mon stage d'encadrer un stagiaire de DUT d'informatique. Le sujet de son stage consistait à développer une interface graphique à Meca. Pour cette réalisation, je lui avais donné quelques points d'entrées de Meca à appeler. Concernant les messages de dialogue avec l'opérateur, il lui a suffit de rediriger la sortie de la classe `Debug` vers un objet de type `JTextArea` pour récupérer les messages dans une boîte de dialogue.

Un second intérêt de la classe `Debug` réside dans la possibilité d'associer un niveau à chaque message. Ce niveau indique dans quel mode le message sera affiché. Grâce à ce niveau, il est possible de paramétrer le détail souhaité des messages lors de la génération du noyau de sécurité. En mode standard, seul l'état des étapes principales est affiché. En cas de problème, il est possible de connaître la cause en choisissant un niveau de debug plus avancé.

En plus du niveau de debug, il est possible de spécifier pour un groupe de messages s'ils ont un caractère informatif, un caractère impliquant de faire attention (warning) ou bien s'il s'agit

de messages d'erreur. Les deux derniers types de messages sont associés chacun à un compteur qui donne un indicateur sur la bon déroulement de la génération. Si plusieurs problèmes ont lieu, leur nombre peut être affiché grâce à une méthode spécifique.

9.3.2 Classes du paquetage boblib.file

Le paquetage `boblib.file` fournit une classe relative à la manipulation des fichiers B (voir la figure 9.12 pour son diagramme UML).

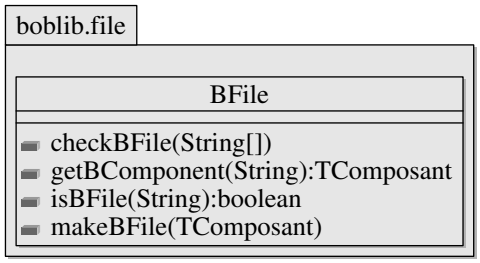

FIG. 9.12 – Diagramme UML de classes du paquetage boblib.file

La classe BFile met à disposition trois méthodes publiques. La méthode `checkBFile` permet de vérifier sur une liste de fichiers passés en paramètres que leur contenu correspond bien à une machine B qui est syntaxiquement correcte. Les deux autres méthodes `getBComponent` et `makeBFile` servent respectivement à charger en mémoire le composant d'un fichier B et à sauvegarder dans un fichier la machine B d'un composant contenu en mémoire.

```
/**
 * Load in memory the component of a B machine from a file
 * @param filename name of the source file
 * @return instance of the loaded B component
 */
public static TComposant getBComponent (String filename) {
  try{
    File f = new File(filename);
    LinkedList list_syntaxB = BParser.analyse(f);          // tatibouet service
    Noeud prog_syntax = (Noeud)list_syntaxB.getFirst();
    return (bob.traduire.Traduire.traduire(prog_syntax)); // bob service
  }catch (IOException io){                                 // input file exception
  }catch (ParseException p){                               // B machine syntax exc.
  }catch (AfterParserException ap){}                       // abstract tree exception
  return null;
}
```

FIG. 9.13 – Code simplifié de l'algorithme de chargement d'un fichier B

```
/**
 * Save in a file a memory instance of a B component
 * @param bComponent B component instance
 * @param filename name of destination file
 */
public static void makeBFile (TComposant bComponent, String filename) {
  try{
    FileWriter bFile = new FileWriter(filename);
    bComponent.Afficher.(BFile);                    // bob service
    bFile.Flush()
  }catch (IOException io){                          // file creation exception
  }finally{
    if (bFile != null)
      try{ bFile.close (); }                        // close the created file
      catch (IOException io){}                      // close file exception
  }
}
```

FIG. 9.14 – Code simplifié de l'algorithme de sauvegarde d'un composant B

Les codes de ces deux dernières méthodes sont repris sur les figures 9.13 et 9.14. Ils permettent de voir le rôle de chacun des paquetages `tatibouet` et `bob` (déjà présentés en début de chapitre), lors des phases de chargement et d'enregistrement de fichier B. Pour simplifier le code, la vérification des paramètres et le traitement des exceptions n'apparaît pas ici. Pour la vérification, celle-ci consiste simplement à s'assurer que le nom de fichier passé en paramètre correspond à un fichier existant et qu'il a également une extension valide pour une machine B (extension `.mch`). Quant aux exceptions, leur gestion repose sur la remontée d'informations à l'utilisateur par l'intermédiaire des interfaces de la classe Debug. Selon le type d'exception, le traitement associé peut aussi aboutir en une terminaison de l'exécution de Meca (notamment si la syntaxe de la machine B est incorrecte).

9.3.3 Classes du paquetage boblib.predicat

Dans la construction du noyau de sécurité, certains traitements sont communs quel que soit le format du modèle de sécurité (DAC, UAC, ...) fourni en entrée de Meca. Parmi ceux-ci, une partie a trait à des manipulation sur des prédicats. Dans le souci de mutualiser les algorithmes relatifs à ces traitements, j'ai réalisé deux classes dédiées, `PredicatSearch` et `PredicatTransform`, regroupées dans le paquetage `boblib.predicat`. Le diagramme UML de ce paquetage est donné sur la figure 9.15. Le but de cette section n'est pas de décrire toutes les méthodes fournies par ces classes mais plutôt d'en présenter les principales.

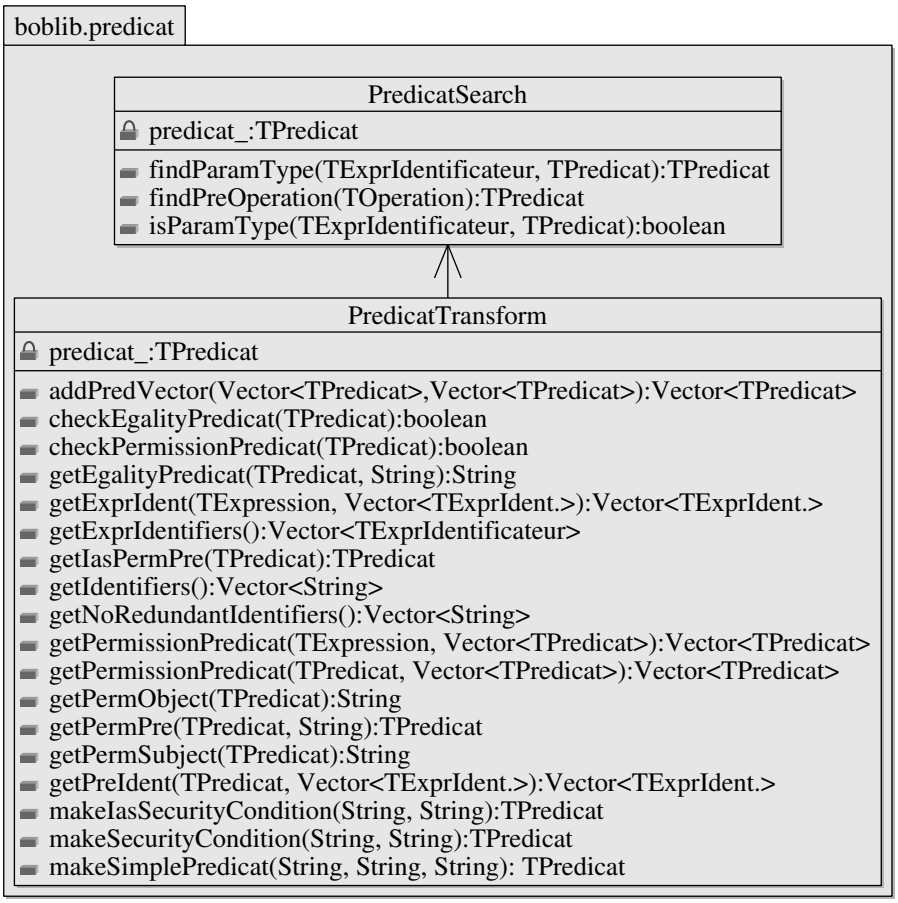

FIG. 9.15 – Diagramme UML de classes du paquetage boblib.predicat

Pour commencer, chaque opération produite dans le noyau de sécurité (sur laquelle on souhaite contrôler l'accès), doit comporter une précondition portant sur le typage des paramètres de l'opération. Ce typage existe forcément dans l'opération d'origine issue du modèle dynamique, du moins pour les paramètres hérités de cette opération.

Par exemple, si on souhaite produire l'opération sécurisée du noyau de sécurité pour l'opération `credit` du modèle dynamique ci-après, le typage du paramètre `amount` correspond au prédicat : `amount` $\in$ `INT`.

```
MACHINE
    epurse_Dynamic
VARIABLES mode, isOpenSess, ... INVARIANT ...
OPERATIONS
    credit (amount) ≙
    PRE amount ∈ INT ∧ isOpenSess = TRUE ∧ mode = ...
    THEN ...
    END;
END
```

Plus généralement, pour connaître le typage d'un paramètre d'une opération, il faut extraire de la précondition de cette opération un prédicat ensembliste composé de l'opérateur $\in$, $\subset$ ou $\subseteq$ et de l'identificateur du paramètre en partie gauche. Ces manipulations ont été regroupées au sein des méthodes de la classe `PredicatSearch`. En particulier, `findPreOperation` retourne la précondition d'une opération si elle existe, et `findParamType` extrait le prédicat de typage d'un paramètre à partir de la précondition de l'opération.

Une seconde caractéristique qui est récurrente dans la génération du noyau de sécurité réside dans la synthèse de la condition de sécurité relative à l'exécution d'une opération. Pour obtenir cette condition, Meca procède en trois étapes :

- la première consiste en une décomposition du prédicat de la clause PROPERTIES du modèle de règles en une liste de sous-prédicats. Comme chaque sous-prédicat peut se décomposer à son tour en un ou plusieurs sous-prédicats, cette étape nécessite de faire appel à un algorithme récursif.

- la seconde étape vise à ne conserver que les sous-prédicats qui peuvent être identifiés comme des règles de permission.

- enfin, la troisième étape est la recherche du bon prédicat (celui qui contient la bonne opération) et la récupération de sa condition de sécurité.

L'algorithme récursif mentionné précédemment est implémenté au moyen des deux méthodes `getPermissionPredicat`, avec deux signatures différentes (voir le tableau 9.1). La première, prend en paramètre le prédicat à décomposer (noté `p`) et un vecteur contenant une liste de prédicat de permission (noté `vp`). Initialement, cette liste est initialisée avec un vecteur vide. A chaque fois que cette méthode est appelée par récursivité, elle fait appel à la méthode `checkPermissionPredicat` qui vérifie si le prédicat `p` a la forme d'une règle de permission. Si tel est le cas, `p` est ajouté au vecteur `vp` qui est alors retourné. Autrement, `p` est décomposé en sous-prédicats et expressions. La méthode `getPermissionPredicat` est alors de nouveau appelée sur chacun des sous-prédicats ou expressions de `p`. La seconde méthode getPermissionPredicat est similaire à la première mais effectue la décomposition sur une expression et non un prédicat.

Méthode	paramètres d'entrée	paramètre de retour
getPermissionPredicat	`p` :TPredicat, `vp` :Vector<TPredicat>	Vector<TPredicat>
getPermissionPredicat	`e` :TExpression, `vp` :Vector<TPredicat>	Vector<TPredicat>
checkPermissionPredicat	`p` :TPredicat	boolean

TAB. 9.1 – Méthodes utilisées pour l'extraction des règles de permissions

Pour mieux comprendre le principe de l'algorithme d'extraction précédent, une partie du code de la méthode `getPermissionPredicat` est présentée sur la figure 9.16.

```
/**
 * Transform a predicate in a vector of sub-predicate representing permission rules
 * @param   p          predicate to compute
 * @param   vp         list of permission predicate parsed at anytime
 * @return             returned list of permission predicate
 */
public Vector<TPredicat> getPermissionPredicat(TPredicat p, Vector<TPredicat> v){
  if (checkPermissionPredicat(p)){
    v.add(p);
  else{
    if (p instanceof TPredComparaison){
      TExpression ex1 = ((TPredComparaison) p).DonnerMembreGauche();
      v = getPermissionPredicat(ex1, v);
      TExpression ex2 = ((TPredComparaison) p).DonnerMembreDroite();
      v = getPermissionPredicat(ex2, v);
    }else if (p instanceof TPredLogique){
      TPredicat pr1 = ((TPredLogique) p).DonnerMembreGauche();
      v = getPermissionPredicat(pr1, v);
      TPredicat pr2 = ((TPredLogique) p).DonnerMembreDroite();
      v = getPermissionPredicat(pr2, v);
    }else{
       ...
    }
  }
}
```

FIG. 9.16 – Code général de l'algorithme d'extraction des permissions

Pour identifier un prédicat comme une règle de permission, `checkPermissionPredicat` effectue des dérivations successives du prédicat `p` jusqu'à l'obtention exacte d'une des formes identifiées comme permission. Une suite de dérivations admissibles est synthétisée dans le tableau 9.2. Pour alléger l'écriture, les règles de dérivation ont été simplifiées au maximum. De même, la gestion des parenthèses (supportée par l'implémentation) n'est pas présentée ici.

règles de dérivation	règles avec les classes sur lesquelles les instances sont testées
p → p1	TPredicat → TPredLogique
p → p3	TPredicat → TPredEnsemble
p1 → p2 ⇒ p3	TPredLogique → TPredicat IMPLIQUE TPredEnsemble
p3 → e1 ∈ e2	TPredEnsemble → TExprOpBinaire APPARTIENT TExprIdent.
e2 → permission	TExprIdent. → permission
e1 → e3 ↦ e4	TExprOpBinaire → TExprIdent. MAPLET TExprIdent.

TAB. 9.2 – Règles de dérivation vérifiées pour identifier une règle de permission

9.3.4 Classes du paquetage boblib.component

La section précédente a montré l'intérêt de développer un paquetage particulier pour regrouper tous les calculs spécifiques aux prédicats. Suite à ce paquetage, je vais m'intéresser au dernier paquetage de la librairie boblib, soit `boblib.component` (Voir figure 9.17). Ce paquetage s'appuie en partie sur le paquetage `boblib.predicat`.

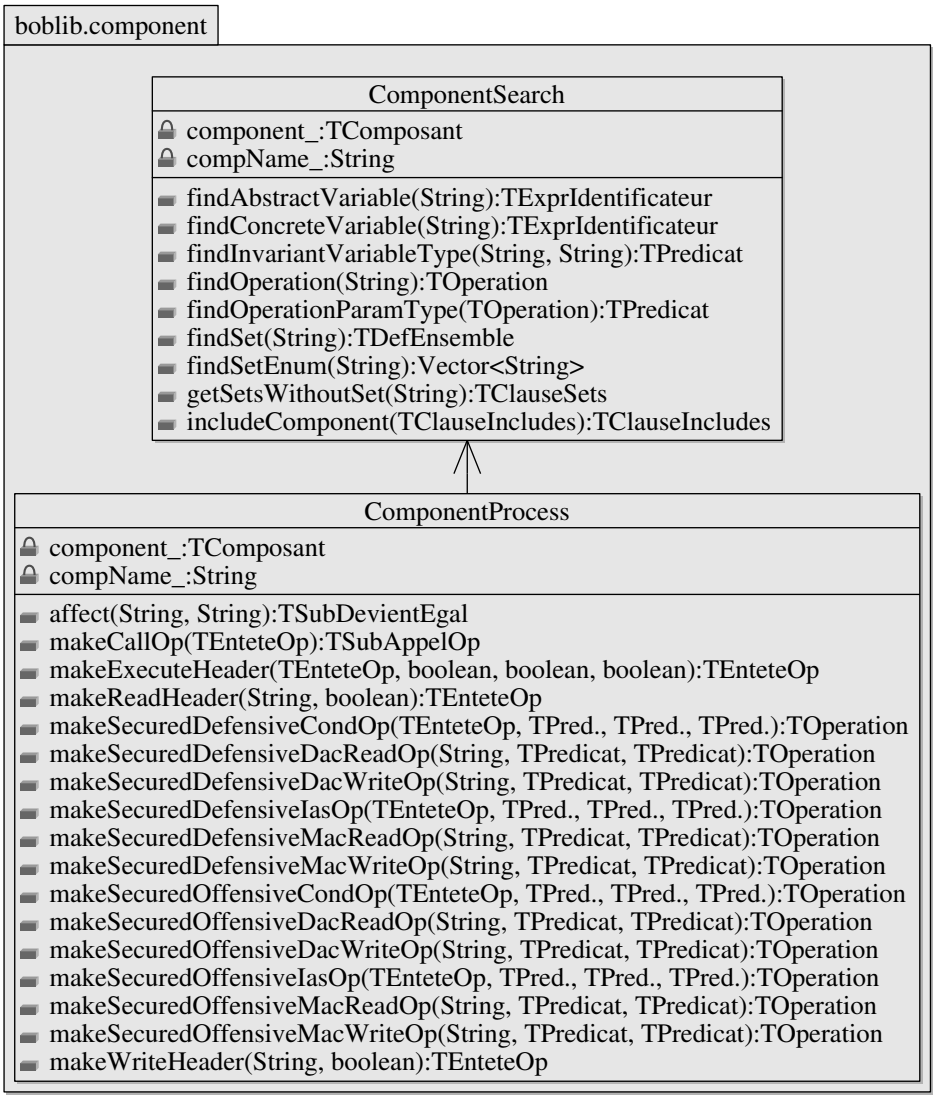

FIG. 9.17 – Diagramme UML de classes du paquetage boblib.component

Les services fournis par le paquetage `boblib.component` sont scindés en deux parties.

La classe `ComponentSearch` a pour objet d'implémenter tous les algorithmes utilisés par Meca pour rechercher des informations dans des composants. Meca s'en sert notamment pour vérifier la cohérence entre ses modèles d'entrées (existence dans le modèle dynamique des opérations dont les noms sont listées sous formes de SETS dans le modèle de règles, etc.). Il

sert également à récupérer toutes les informations qui lui serviront à construire le noyau de sécurité.

La seconde classe du paquetage `boblib.component`, `ComponentProcess`, encapsule toutes les fonctions de génération des opérations du noyau de sécurité. Grâce à ces fonctions, il est possible de produire tous les types d'opérations sécurisées qui sont gérées par Meca, c'est-à-dire pour chacun des formats supportés et quelle que soit la forme du noyau demandée (défensive ou offensive).

9.3.5 Classes du paquetage meca

Après avoir passé en revue dans les sections 9.3.1 à 9.3.4 les différents modules que j'ai développés au sein de la librairie boblib, je vais maintenant terminer par le cœur de Meca. Celui-ci regroupe six classes au sein du paquetage `meca`. Ce paquetage contient un sous-paquetage, `meca.kerngen` (voir figure 9.18) et la classe `Meca`, qui correspond au point d'entrée de l'outil Meca.

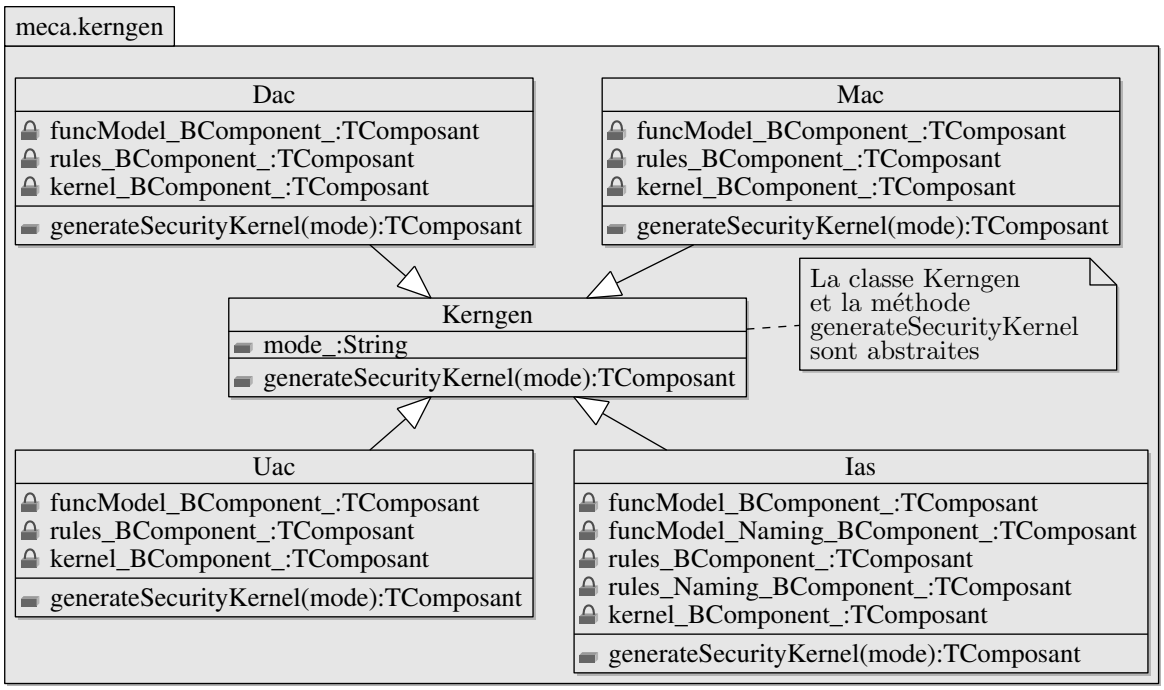

FIG. 9.18 – Diagramme UML de classes du paquetage meca.kerngen

Le paquetage `meca.kerngen` dispose de la classe abstraite `Kerngen` qui fournit le squelette général d'un générateur de noyau de sécurité. Cette classe contient une méthode unique appelée `generateSecurityKernel` et l'attribut `mode_` qui sert à spécifier si le noyau est de forme défensive ou offensive. En plus de cette classe, `meca.kerngen` fournit une implémentation de la méthode `generateSecurityKernel` pour chacun des formats de noyau supportés par Meca. Ces implémentations sont faites au sein de quatre classes `Dac`, `Mac`, `Uac` et `Ias`, et héritant toutes de la classe `Kerngen`. Pour chacune de ces classes, les composants d'entrée de Meca sont passés au moyen du constructeur. Ces composants sont au nombre de deux (modèle de règles

et modèle dynamique) sauf pour la classe `Ias` où deux composants supplémentaires sont requis pour chacun des fichiers de nommage.

9.3.6 Classe principale de Meca

La classe principale de l'outil Meca est la classe `Meca`, située dans le paquetage `meca`. Implémentant la méthode principale `main` de l'outil, cette classe assure les rôles suivants :

- elle contient un décodeur qui parse les arguments passés en paramètres à Meca avant de vérifier leur validité.
- si aucun argument n'est fourni, la méthode `main` affiche l'aide en ligne de l'outil (voir figure 9.19).

```
_____________________________
| Meca version 1.0 Usage |
_____________________________

> Meca -f FUNCTIONAL -r RULES [-k KERNEL] [-d DEBUGLEVEL] [-m MODE]
[-t TYPE] | -c FILE

FUNCTIONAL : this is the input functional model file written in the
             B language
RULES      : this is the input security policy rules file written
             in the B language
KERNEL     : this is the ouput file name of the security kernel
             that Meca will generate
             this argument is optional
DEBUGLEVEL : this argument determines the detail of the output
             messages
             the possible values are : LOW, MED, STD, HIG add FUL
             this argument is optional
MODE       : this argument determines the mode of the output
             operations in the security kernel
             the possible values are : OFF, DEF
             this argument is optional and the default value is DEF
TYPE       : this argument determines the type of security policy
             rules to be applied
             the possible values are : DAC, MAC, UAC, IAS
             this argument is optional and the default value is IAS
FILE       : this argument corresponds to the file to convert
             the option -c must be use alone without any other
             options
```

FIG. 9.19 – Aide en ligne de Meca

– si les arguments sont valides, la classe effectue d'abord le chargement en mémoire des composants B à partir des fichiers spécifiés en paramètres. Elle appelle ensuite la méthode `generateSecurityKernel` de la classe correspondant au format spécifié. Enfin, si le noyau est généré avec succès, elle l'enregistre dans le fichier spécifié en paramètre.

9.4 Synthèse du développement de Meca

Pour conclure ce chapitre, je terminerai par faire une synthèse des résultats obtenus lors du développement de Meca. Au terme du développement de Meca, plusieurs constats ont été faits.

D'abord, la nouvelle conception de Meca a permis de tirer plusieurs avantages par rapport aux précédents prototypes. D'une part, le code obtenu est beaucoup plus compact - moins de 1000 lignes utilisées rien que pour les formats DAC et MAC contre 2300 lignes environ pour les deux prototypes correspondants - alors que les vérifications ont été multipliées et l'outil fiabilisé. D'autre part, le découpage modulaire et la séparation des traitements communs dans une librairie dédiée ont permis de gagner du temps pour l'ajout successif de chacun des nouveaux formats (Uac puis Ias). Il n'a pas été fait le choix d'intégrer à l'outil Meca le support du format Rbac (dont un prototype avait été étudié), car il présentait moins d'intérêt vis-à-vis du projet POSÉ. Cependant, le développement d'un générateur de noyau pour ce format ne serait sans doute pas très long car de nombreuses fonctions pourraient être réutilisées.

Ensuite, la conformité de l'outil par rapport aux besoins du projet POSÉ a pu être établie. Dans un premier temps, Meca a été validé en générant le noyau de sécurité à partir de quelques exemples de modèles d'entrée. Puis, en fin de projet, c'est une partie de l'application de POSÉ qui a été soumise à Meca, après avoir opéré quelques légères adaptations pour se conformer aux spécificités de l'outil (voir chapitre 8). Le noyau produit par Meca a alors été utilisé par nos partenaires pour produire de nouveaux tests permettant de déceler des failles de sécurité jusqu'alors non détectées par les méthodes de tests antérieures.

Au final, le code que j'ai développé porte sur environ 4000 lignes (incluant les commentaires spécifiques à la javadoc), réparties dans 12 classes. Sur ces 4000 lignes, 2500 constituent la librairie boblib et 1500 le cœur de Meca. Cette taille de code fait clairement apparaître que Meca n'est pas un outil très conséquent. Cela est permis grâce à l'utilisation de la BoB qui fournit un service relativement évolué (grâce à ses 95 classes et ses 24000 lignes de code).

Chapitre 10

La gestion du projet

Après avoir traité des aspects techniques de mon stage, je vais terminer ce mémoire par les aspects plus organisationnels. Ce sera ainsi l'occasion dans un premier temps de faire le point sur la manière dont s'est réellement passé le stage par rapport à ce qui avait été défini initialement. Pour ce faire, je commencerai par rappeler le planning qui était prévu et je le confronterai avec le planning qui a été réellement suivi. J'en profiterai alors pour expliquer les raisons des écarts qui ont pu se produire. Enfin, je terminerai en abordant les expériences annexes auxquelles j'ai participées durant ce stage et ce qu'elles m'ont apportées.

10.1 Planning prévisionnel du stage

Le stage d'ingénieur CNAM se déroule normalement sur une durée de 12 mois à temps complet. Initialement, il avait été prévu qu'il soit effectué de septembre 2006 à août 2007.

SEPTEMBRE 2006			***OCTOBRE 2006***			***NOV. - DEC. 2006***		
sem	*nb*	*tâches*	*sem*	*nb*	*tâches*	*sem*	*nb*	*tâches*
S36 à S39	4	Démarrage du projet - Formation à la méthode B	S40 à S44	5	Reprise de l'outil Meca - Définition du format condionnel	S45 à S52	8	Développement d'un nouveau module Meca pour le format conditionnel

JAN - MARS 2007			***AVR. - JUIL. 2007***			***AOUT 2007***		
sem	*nb*	*tâches*	*sem*	*nb*	*tâches*	*sem*	*nb*	*tâches*
S1 à S13	13	Etude d'IAS - Définition de modèles de sécurité pour IAS	S14 à S31	18	Développement d'un module Meca traitant les modèles pour IAS	S32 à S35	4	Finalisation du mémoire

TAB. 10.1 – Planning prévisionnel du stage

Le tableau 10.1 reprend ce planning avec le plan de travail prévisionnel. Il liste pour chacune des tâches, les numéros (colonne *sem*) et le nombre (colonne *nb*) de semaines affectées à celles-ci. Hormis les phases de démarrage au début et celle de rédaction du mémoire à la fin, la majeure partie du stage devait être constituée de deux étapes principales, chacune découpée en deux sous-étapes. Ces étapes sont indiquées ci-dessous :

- **Etape 1 : Etude de modèles de politiques de sécurité conditionnées par des attributs de sécurité** (13 semaines) : L'objectif de cette étude était de s'assurer de la faisabilité en terme de modélisation de ce type de politique de sécurité sur des exemples simples avant de s'atteler à la modélisation des politiques plus complexes intervenant dans le projet POSÉ. Cette étude se décompose en deux phases distinctes :

  • Définition et modélisation d'un format de politique de sécurité conditionnée.

  • Développement d'une couche Meca permettant de traiter ce format.

- **Etape 2 : Etude de modèles de politiques de sécurité de la plateforme IAS pour le projet POSÉ** (31 semaines) : L'objectif de cette étape était de repartir de l'étude menée dans l'étape 1 pour modéliser la politique de sécurité de la plateforme IAS. Comme pour l'étape 1, on retrouve également deux phases différentes pour cette étape :

  • Définition et modélisation d'un format de politique de sécurité pour la plateforme IAS.

  • Développement d'une couche Meca permettant de traiter ce format.

Après avoir vu le planning prévisionnel du stage, la section suivante propose de dresser un bilan du calendrier qui a réellement été appliqué. Il permettra d'observer et d'expliquer les écarts qui ont lieu et de faire apparaître les diverses implications annexes que j'ai pu avoir.

10.2 Déroulement réel du stage

Lorsque j'ai débuté mon stage d'ingénieur CNAM, je travaillais en tant qu'analyste programmeur sur systèmes embarqués au sein d'une TPE[1] de 8 salariés (dont 2 analystes programmeurs), et ce depuis déjà 6 années. J'ai ainsi pu bénéficier d'un congé formation par le biais du FONGECIF[2] pour pouvoir m'absenter de mon entreprise pendant 12 mois tout en conservant une partie de ma précédente rémunération. Cependant, devant la difficulté pour trouver une personne immédiatement opérationnelle pour me remplacer sur les projets en cours, j'ai préféré, à la demande de mon employeur, étaler les 12 mois de stage effectif sur une durée de 16 mois (en laissant 4 mois de mise à disposition pour effectuer ponctuellement des missions pour mon employeur). Cette organisation de mon stage a fait l'objet d'une validation au préalable, aussi bien par mon employeur que par mon maître de stage.

[1] Toute Petite Entreprise
[2] FONd de GEstion du Congé Individuel de Formation

Le tableau 10.2 permet de mieux se rendre compte de l'organisation qui a été adoptée pour ce stage. Il fait notamment apparaître les 11 semaines de missions effectuées pour mon employeur (notées MISSION dans le tableau) ainsi que les 6 semaines de congés prises sur les congés payés de mon entreprise (notées CONGÉS). Si on cumule les semaines de mission avec les semaines de congés, on trouve 17 semaines, ce qui correspond au nombre de semaines de septembre à décembre 2007, qui ne faisaient pas parties initialement du stage (cf tableau 10.1).

SEPT. 2006		***OCT. 2006***		***NOV. 2006***		***DEC. 2006***	
sem	*tâche*	*sem*	*tâche*	*sem*	*tâche*	*sem*	*tâche*
S36	T1 :	S40	T1	S45	existants	S49	aux cartes
S37	Initiation	S41	MISSION	S46	de Meca	S50	MISSION
S38	à la	S42	T2 :	S47	T3 : Initiation	S51	à puces
S39	méthode B	S43	Récupération	S48	ATAC	S52	CONGÉS
		S44	des modules				

JANVIER 2007		***FEVRIER 2007***		***MARS 2007***		***AVRIL 2007***	
sem	*tâche*	*sem*	*tâche*	*sem*	*tâche*	*sem*	*tâche*
S1	T4 : Préparation	S6	Modélisation	S10	T6 : Re-	S14	MISSION
S2	Cours Master	S7	du format	S11	développement	S15	CONGÉS
S3	CONF B	S8	de politiques	S12	de Meca	S16	
S4	MASTER	S9	UAC	S13	SEM Meudon	S17	T6 :
S5	T5 :						

MAI 2007		***JUIN 2007***		***JUILLET 2007***		***AOUT 2007***	
sem	*tâche*	*sem*	*tâche*	*sem*	*tâche*	*sem*	*tâche*
S18	CONGÉS	S23	T7	S27	SEM Nancy	S31	MISSION
S19	T7 : Etude	S24	CONF Afadl	S28	de politiques	S32	CONGÉS
S20	des politiques	S25	T8 :	S29	MISSION	S33	CONGÉS
S21	de sécurité	S26	Modélisation	S30	Rules based	S34	
S22	du projet POSÉ		du format			S35	T8

SEPT. 2007		***OCT. 2007***		***NOV. 2007***		***DEC. 2007***	
sem	*tâche*	*sem*	*tâche*	*sem*	*tâche*	*sem*	*tâche*
S36	MISSION	S40	du format	S45	MISSION	S49	MISSION
S37	T9 : Ajout	S41	rules based	S46	du	S50	
S38	à Meca	S42	MISSION	S47	rapport	S51	T10
S39	MISSION	S43	T10 :	S48	MISSION	S52	CONGÉS
		S44	Rédaction				

TAB. 10.2 – Planning réel du stage

Les tâches notées T1 à T10 représentent les tâches effectuées durant le stage avec leur durée réelle. On peut constater qu'à l'exception des tâche T3 et T4, on retrouve bien toutes les tâches qui avaient été définies dans le planning prévisionnel (tableau 10.1). Ainsi, on peut noter que les tâches T5 et T6 correspondent à l'étape 1 (étude de politiques conditionnées)

de la section précédente alors que les tâches T7, T8 et T9 correspondent à l'étape 2 (étude de la plateforme IAS).

La comparaison entre les deux plannings fait clairement apparaître que j'ai passé plus de temps que prévu sur les aspects formalisation et modélisation (5 semaines au lieu de 3 pour la modélisation du format UAC et 12 au lieu de 13 sur le format Rules Based). A l'inverse, le développement en Java des modules Meca a été beaucoup plus rapide que les estimations initiales (5 semaines au lieu de 8 pour UAC et 4 au lieu de 18 pour Rules Based). Cela est essentiellement dû à l'apprentissage qui a été nécessaire sur la méthode B pour la modélisation alors que j'avais déjà une expérience en programmation qui m'a fortement aidé pour le développement en Java. De plus, la conception de routines modulaires sur UAC a facilité leurs réutilisations pour le format rules based.

Les tâches notés SEM Meudon et SEM Nancy correspondent aux séminaires techniques auxquels j'ai participé avec les partenaires du projet POSÉ (séminaires de deux jours à chaque fois).

10.3 Implications annexes

Dans cette dernière section, je vais présenter les implications périphériques aux projets POSÉ, sur lesquelles j'ai eu la chance de pouvoir travailler. Tout d'abord, j'ai eu l'opportunité de pouvoir suivre le projet ATAC[3], dispensé aux étudiants en troisième année d'école d'ingénieur à l'INP. Ce projet est encadré par Jean-Louis Lanet et Claude Barral, tout deux consultants de la société Gemalto (J.L. Lanet qui m'a apporté par la suite une aide précieuse sur mes travaux relatifs à la carte à puce et que je remercie). Ce projet, qui s'est déroulé sur 3 jours, était constitué d'un tiers de cours et de deux tiers de travaux pratiques sur le développement d'applications JavaCard.

Suite à l'intérêt que j'ai développé pour les JavaCard à l'issue du projet ATAC, Marie-Laure POTET m'a permis de monter avec elle un projet similaire pour des étudiants en Master Crypto (noté MASTER sur le planning), où j'ai pu assurer le cours sur la partie JavaCard et encadrer deux demi-journées de travaux pratiques. Etant donné l'engouement des étudiants sur ce projet, nous l'avons reconduit l'année suivante (mars 2008) en dehors de mon stage.

Une seconde implication, fort intéressante et enrichissante a été la participation à plusieurs publications, relatives à nos travaux, avec des thésards, dont une avec Marie-Laure POTET. La première [DHM09] a été publiée lors de la conférence francophone AFADL[4], à laquelle j'ai pu participer (cf CONF Afadl sur le planning). La seconde [HM07] a été publiée lors de la conférence internationale CRiSIS[5], conférence à laquelle je n'ai pu assister car elle s'est déroulée en même temps que le séminaire à Nancy.

[3] Applications sécurisées pour Technologies javACard
[4] Approche Formelle pour l'Aide au Développement Logiciel : http ://www.info.fundp.ac.be/~pys/AFADL07
[5] International Conference on Risks and Security of Internet and Systems : http ://crisis.enseeiht.fr/crisis07/

Chapitre 11

Conclusion

La carte à puce fait partie des domaines où la confiance des utilisateurs et la qualité des produits sont d'une importance capitale. Tout simplement parce qu'elle est présente au cœur des activités les plus importantes de notre quotidien. Elle impacte aussi bien nos finances (avec la carte bancaire), notre santé (avec la carte vitale), notre sphère privée (avec la carte SIM[1]) ou encore notre identité (avec le passeport électronique).

Depuis son origine, la sécurité offerte par la carte à puce a été principalement basée sur le fait que les applications qu'elle embarquait ont été développées de manière monolitique, intégrant aussi bien les fonctions du système d'exploitation que celles de l'application. La plupart du temps, ces applications ont subi de lourdes batteries de tests, au coût très élevé, avant d'être mises sur le marché. Un second aspect de la sécurité résidait dans le secret, c'est-à-dire que les applications étaient fermées. Cela permettait au fabricant de maîtriser l'environnement de ses cartes à puce.

Avec l'arrivée des cartes à puce dites ouvertes, cette problématique a changé. En effet, ces nouvelles générations de cartes (comme les JavaCard) intègrent maintenant un système d'exploitation générique et permettent de développer ses applications de manière indépendante. Ces cartes, basées sur des technologies issues de l'informatique classique comme Java, présentent des atouts majeurs comme l'amélioration de l'interopérabilité, la possibilité d'intégrer plusieurs applications, ou encore la possibilité de faire évoluer facilement les applications. Cependant, elles posent le problème de la séparation entre la sécurité du système et celle des applications. Par exemple, comment garantir que mon application est sécurisée quand elle est utilisée sur une carte à puce qui héberge aussi d'autres applications que je ne maîtrise pas ?

[1] Subscriber Identity Module : Il s'agit des cartes à puce qui équipent les téléphones mobiles pour identifier l'abonné et lui permettre d'utiliser différents services offerts par les opérateurs mobiles dont la téléphonie mobile.

Pertinence de l'utilisation des méthodes formelles pour la modélisation de politiques de sécurité et pour le test et perspectives de POSÉ

Le test de politiques de sécurité constitue un enjeu économique et technologique pour les systèmes de sécurité tels que les cartes à puce. Dans ce contexte, chaque nouvelle application va donner lieu a des efforts importants pour tester la sécurité du système. Le projet POSÉ a pu montrer la pertinence de baser notre approche sur la modélisation des politiques de sécurité au moyen de la méthode B. En effet, cette méthode est bien adaptée car elle couvre le cycle de développement des applications cartes à puce et possède des outils industriels facilitant la preuve automatique des lemmes générés.

D'autre part, POSÉ a permis de produire un démonstrateur de processus outillé pour la validation de conformité d'un système aux politiques de sécurité qui lui sont assignées. Grâce à ce démonstrateur, POSÉ a ainsi pu montrer la pertinence d'une approche de test de politiques de sécurité à partir de modèles. Il amène une approche de test systématique des propriétés de sécurité et a ouvert la voie au déploiement industriel de techniques de Model-Based Security Testing qui permettront à la fois de diminuer les coûts des phases de test de la politique de sécurité et de garantir une couverture systématique, traçable et automatisée.

L'industrialisation des résultats du projet POSÉ pour passer de démonstrateurs issus de la recherche à des solutions industrialisées et déployables à grande échelle est maintenant du ressort des industriels qui ont participé au projet.

Bilan personnel

Sur une note plus personnelle, le bilan du projet POSÉ est pour moi extrèmement positif aussi bien par les connaissances que les compétences que j'ai pu en retirer. Pour commencer, il m'a permis de découvrir de nouveaux domaines technologiques et scientifiques que je ne connaissais pas comme la méthode B, la carte à puce et en particulier la JavaCard ou encore la norme des Critères Communs.

Ensuite, il m'a permis d'élargir considérablement mes méthodes et outils de travail grâce à une équipe de personnes très dynamiques qui m'ont permis d'adopter assez facilement le système d'exploitation Linux pour les besoins du projet, comme pour mes travaux annexes comme la rédaction au moyen de l'environnemnet LaTeX.

Enfin, ce stage a été l'occasion pour moi de vivre plusieurs expériences très enrichissantes, qui ont suscitées chez moi l'envie de donner suite à certaines d'entre elles. Je pense en particulier aux trois publications auxquelles j'ai participées, et dont la dernière a été acceptée en avril 2009, soit bien après la fin de mon stage. Mais je pense aussi à mes implications dans la prise en charge de cours et de travaux pratiques pour un projet carte à puce dispensés à des étudiants de Master Crypto.

Annexes

Annexe A

Les notations logiques et ensemblistes

Cette annexe liste les principales notations qui sont utilisées dans un modèle B et donne la définition de chacune d'entre elles. Ces notations sont utilisées dans le langage B pour spécifier les états, les invariants, les conditions, donner les propriétés des constantes etc.

A.1 Les notations logiques

Les expressions logiques dénotent des prédicats qui ont une interprétation dans le domaine de valeurs de vérité : *btrue* et *bfalse*. Ces prédicats peuvent être combinés entre eux au moyen des connecteurs usuels de la logique qui sont présentés dans le tableau A.1.

Symbole	Signification	Définition
$\wedge$	et logique	
$\neg$	négation	
$\vee$	ou logique	$a \vee b \mathrel{\hat{=}} \neg (\neg a \wedge \neg b)$
$\Rightarrow$	implication	$a \Rightarrow b \mathrel{\hat{=}} \neg a \vee b$
$\Leftrightarrow$	équivalence	$a \Leftrightarrow b \mathrel{\hat{=}} (a \Rightarrow b) \wedge (b \Rightarrow a)$

TAB. A.1 – Les notations logiques

Les prédicats peuvent également être quantifiés en utilisant les quantificateurs universels du tableau A.2. Le non-terminal *Id_liste* représente une liste d'identificateurs séparés par des virgules. Le non-terminal *Prédicat* désigne un prédicat quelconque. Dans le cas du « $\forall$ », le prédicat quantifié est toujours de la forme : $P \Rightarrow Q$, où P définit le typage ensembliste des

variables liées par le quantificateur. Dans le deuxième cas, le prédicat est de la forme $P \wedge Q$.

Symbole	Signification	Syntaxe	Définition
$\forall$	pour tout	$\forall$ *Id_ liste* $\bullet$ (*Prédicat*)	
$\exists$	il existe	$\exists$ *Id_ liste* $\bullet$ (*Prédicat*)	$\exists x \bullet (P) \mathrel{\hat{=}} \neg\forall x \bullet (\neg P)$

TAB. A.2 – Les quantificateurs universel et existentiel

Enfin, il est aussi possible de trouver en B des prédicats d'égalité tels que définis dans le tableau A.3.

Symbole	Signification	Définition
$=$	est égal à	
$\neq$	n'est pas égal à	$x \neq y \mathrel{\hat{=}} \neg(x = y)$

TAB. A.3 – Le prédicat d'égalité

A.2 Les notations ensemblistes

Cette partie n'a pas vocation de donner une liste exhaustive de toutes les notations ensemblistes. Elle rappelle seulement les principales notations qui ont pu servir aux modèles du projet.

A.2.1 La construction des ensembles

Les ensembles peuvent être des ensembles d'éléments sans structure, ou bien des produits cartésiens d'ensembles, ou encore des parties d'un ensemble. Les notations de construction d'ensembles sont présentées dans le tableau A.4.

Le non-terminal *Ensemble* désigne une expression construisant un ensemble. Le non-terminal *Expression_liste* représente un liste d'expressions quelconques séparées par des virgules. Les éléments d'un produit cartésien sont des paires d'éléments.

Symbole	Signification	Syntaxe
$\emptyset$	ensemble vide	
$\times$	produit cartésien	$Ensemble \times Ensemble$
$\mathbb{P}$	ensemble des sous-ensembles	$\mathbb{P}(Ensemble)$
$\mathbb{P}_1$	ensemble des sous-ensembles non vides	$\mathbb{P}_1(Ensemble)$
$\mathbb{F}$	ensemble des sous-ensembles finis	$\mathbb{F}(Ensemble)$
$\mathbb{F}_1$	ensemble des sous-ensembles finis non vides	$\mathbb{F}_1(Ensemble)$
{ , , ...}	ensembles définis en extension	$\{Expression\_liste\}$
{ | }	ensembles définis en compréhension	$\{Id\_liste \mid Prédicat\}$

TAB. A.4 – Notations des constructions d'ensemble

A.2.2 Les prédicats sur les ensembles

Le tableau A.5 donne la liste des différents types d'ensemble qui constituent généralement les prédicats.

Symbole	Signification	Définition
$\in$	appartient à	
$\notin$	n'appartient pas à	$x \notin s \mathrel{\hat{=}} \neg(x \in s)$
$\subseteq$	est inclus dans	$s \subseteq t \mathrel{\hat{=}} s \in \mathbb{P}(t)$
$\not\subseteq$	n'est pas inclus dans	$s \not\subseteq t \mathrel{\hat{=}} \neg(s \subseteq t)$
$\subset$	est strictement inclus dans	$s \subset t \mathrel{\hat{=}} (s \subseteq t \wedge s \neq t)$
$\not\subset$	n'est pas strictement inclus dans	$s \not\subset t \mathrel{\hat{=}} \neg(s \subset t)$

TAB. A.5 – Les notations ensemblistes

A.2.3 Les expressions d'ensembles

Les opérateurs qui sont présentés dans le tableau A.6 permettent d'effectuer des opérations simples entre deux ensembles. La définition qui est donnée pour chaque opérateur suppose comme hypothèses que $s_1 \subseteq t$ et que $s_2 \subseteq t$.

Symbole	Signification	Définition
$\cup$	union	$s_1 \cup s_2 \mathrel{\hat{=}} \{x \mid x \in t \wedge (x \in s_1 \vee x \in s_2)\}$
$\cap$	intersection	$s_1 \cap s_2 \mathrel{\hat{=}} \{x \mid x \in t \wedge (x \in s_1 \wedge x \in s_2)\}$
$-$	différence d'ensembles	$s_1 - s_2 \mathrel{\hat{=}} \{x \mid x \in t \wedge (x \in s_1 \wedge x \notin s_2)\}$

TAB. A.6 – Les expressions d'ensembles

A.2.4 Les relations

Les relations sont un cas particulier de construction d'ensembles. Elles sont très utilisées dans les spécifications (invariants, propriétés, etc.). Il s'agit simplement de couples d'éléments. La définition d'une relation est donnée dans le tableau A.7.

Symbole	Signification	Définition
$\leftrightarrow$	relation entre deux ensembles	$E_1 \leftrightarrow E_2 \mathrel{\hat{=}} \mathbb{P}(E_1 \times E_2)$

TAB. A.7 – Les relations

Le domaine d'une relation est défini comme les éléments du premier ensemble E_1 qui sont effectivement en relation avec des éléments du second ensemble E_2. Le codomaine est l'ensemble des points du second ensemble E_2 qui sont en relation avec des élements du premier ensemble E_1. L'image d'un ensemble par une relation, noté r[F] est l'ensemble des éléments de E2 qui sont en relation avec les éléments de F par la relation r. Ces définitions se retrouvent formellement dans le tableau A.8

Expression	Condition	Définition
$\mathsf{dom}(r)$	$r \in E_1 \leftrightarrow E_2$	$\{x \mid x \in E_1 \land \exists y \bullet (y \in E_2 \land (x \mapsto y) \in r)\}$
$\mathsf{ran}(r)$	$r \in E_1 \leftrightarrow E_2$	$\{y \mid y \in E_2 \land \exists x \bullet (x \in E_1 \land (x \mapsto y) \in r)\}$
$r[F]$	$r \in E_1 \leftrightarrow E_2$ et $F \subseteq E_1$	$\{y \mid y \in E_2 \land \exists x \bullet (x \in F \land (x \mapsto y) \in r)\}$

TAB. A.8 – Domaine, codomaine et image d'une relation

A.2.5 Les fonctions

Les fonctions sont des relations dont chaque élément du domaine n'est associé qu'à un seul élément du codomaine. Les fonctions les plus générales sont les fonctions partielles. Les fonctions totales, injectives, surjectives et bijectives peuvent ensuite être définies à partir des fonctions partielles. Le tableau A.9 rassemble les définitions de toutes ces fonctions.

Expression	Signification	Définition
$s \pfun t$	fonction partielle	$\{r \mid r \in s \leftrightarrow t \land \forall x,y,z \bullet (x,y \in r \land x,z \in r \Rightarrow y = z)\}$
$s \fun t$	fonction totale	$\{f \mid f \in s \pfun t \land \mathsf{dom}(f) = s\}$
$s \pinj t$	injective partielle	$\{f \mid f \in s \pfun t \land f^{-1} \in t \pfun s\}$
$s \inj t$	injective totale	$\{s \pinj t \cap s \fun t\}$
$s \psur t$	surjective partielle	$\{f \mid f \in s \pfun t \land \mathsf{ran}(f) = t\}$
$s \surj t$	surjective totale	$\{s \psur t \cap s \fun t\}$
$s \pbij t$	bijective partielle	$\{s \pinj t \cap s \psur t\}$
$s \bij t$	bijective totale	$\{s \inj t \cap s \surj t\}$

TAB. A.9 – Les fonctions

Bibliographie

[Abr96] J.-R. Abrial : *The B-book : assigning programs to meanings*. Cambridge University Press, 1996.

[BCK+00] Robin E. Bloomfield, Dan Craigen, Frank Koob, Markus Ullmann et Stefan Wittmann : Formal methods diffusion : Past lessons and future prospects. *In* Floor Koornneef et Meine van der Meulen, éditeurs : *SAFECOMP*, volume 1943 de *Lecture Notes in Computer Science*, pages 211–226. Springer, 2000.

[Bib77] K. J. Biba : Integrity considerations for secure computer systems. Technical report tr-3153, The Mitre Corporation, Bedford, 1977.

[BL73] D. E. Bell et L. J. Lapadula : Secure computer systems : A mathematical model. Technical report ESD-TR-278, vol. 2, The Mitre Corporation, Bedford, 1973.

[Bou97] Gérard Bouget : Les critères communs pour l'évaluation de la sécurité des systèmes d'information. Lettre numéro 23, Institut européen de cindynique, aout 1997.

[CC006a] Common criteria for information technology security evaluation, part 1 : Introduction and general model. Rapport technique CCMB-2006-09-001, sept 2006.

[CC006b] Common criteria for information technology security evaluation, part 2 : Security functional components. Rapport technique CCMB-2006-09-002, sept 2006.

[CC006c] Common criteria for information technology security evaluation, part 3 : Security assurance components. Rapport technique CCMB-2006-09-003, sept 2006.

[CGR93] D. Craigen, S. Gerhart et T.J. Ralston : An international survey of industrial applications of formal methods (volume 1 : Purpose, approach, analysis and conclusions, volume 2 : Case studies). Rapport technique NIST GCR 93/626-V1 & NIST GCR 93-626-V2 (Order numbers : PB93-178556/AS & PB93-178564/AS), U.S. National Institute of Standards and Technology, National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161, USA, 1993.

[DHM09] F. Dadeau, A. Haddad et T. Moutet : Test fonctionnel de conformité vis-à-vis d'une politique de contrôle d'accès. *TSI - Technique et Science Informatiques*, 28(4):533–563, apr 2009.

[DLMP08] F. Dadeau, J. Lamboley, T. Moutet et M.-L. Potet : A verifiable conformance relationship between smart card applets and B models. *In ABZ'2008, International Conference on ASM, B and Z*, volume 5238 de *LNCS*, pages 237–250, London, UK, septembre 2008. Springer.

[FK92] D. F. Ferraiolo et D. R. Kuhn : Role based access control. *In 15th National Computer Security Conference*, 1992.

[FSG+01] D.F. Ferraiolo, R.S. Sandhu, S. Gavrila, D. R. Khun et R. Chandramouli : Proposed NIST standard for role-based access control. *ACM Transactions on Information and System security, volume 4, n. 3*, aug 2001.

[Had95] Stéphan Hadinger : Le contrôle d'accès au SI. *In projet IS@ France Télécom, www.rd.francetelecom.fr/fr/conseil/mento*, 1995.

[Had05] Amal Haddad : Modélisation et vérification de politiques de sécurité. Mémoire de D.E.A., Université Joseph Fourier, Grenoble, sep 2005.

[Had07] Amal Haddad : Meca : A tool for access control models. *In B'2007, the 7th Int. B Conference*, volume 4355 de *LNCS*, pages 281–284, Besançon, France, jan 2007. Springer.

[HM07] A. Haddad et T. Moutet : Modélisation et vérification de contrôle d'accès selon une démarche critères communs. *In CRiSIS2007, en cours de publication, Marrakech, Maroc*, jul 2007.

[iso] Smart card standard : Part 4 : Interindustry commands for interchange. Rapport technique 7816-4, ISO/IEC.

[Lam71] Butler W. Lampson : Protection. *In 5th Princeton Symposium on Information Science and Systems*, pages 437–443, 1971.

[Lan06] J.L. Lanet : Les cartes à puce. Support de cours, 2006.

[San96] R.S. Sandhu : Role hierarchies and constraints for lattice-bases access controls. *Lecture Notes in Computer Science 1146 : 65-79*, 1996.

[Sau01] D. Sauveron : La technologie java card. Rapport technique RR-1259-01, LaBRI, may 2001.

[Sau04] D. Sauveron : *Etude et réalisation d'un environnement d'expérimentation et de modélisation pour la technologie JavaCard. Application à la sécurité.* Thèse de doctorat, Université de Bordeaux I, dec 2004.

[Sto04] Nicolas Stouls : Présentation des critères communs et application au projet eden. Projet RNTL EDEN, jul 2004.

[Wik07] Carte à puce, jun 2007.

Mémoire d'Ingénieur C.N.A.M. en Informatique

Description de propriétés de sécurité pour un processus de validation/vérification

Thierry Moutet

Grenoble, le 30 juin 2010

Résumé

Les travaux présentés dans ce rapport s'articulent autour de la validation du contrôle d'accès défini par des politiques de sécurité. Nous nous intéressons à la validation par génération de tests à partir d'un modèle de sécurité écrit en B et décrivant les fonctionnalités de l'application. Nous utilisons l'outil Meca, qui prend en entrée un modèle fonctionnel et une description d'une politique de sécurité sous la forme de machines abstraites B, et qui génère un noyau de sécurité pour le modèle fonctionnel. Ce noyau de sécurité est en charge d'intercepter tous les accès des sujets aux objets, et restreint les comportements à ceux satisfaisant les exigences de sécurité. Nous présentons une étude de cas complète, sur une application de type porte-monnaie électronique, agrémentée d'une politique de sécurité discrétionnaire.

Mots clés : Contrôle d'accès - Politique de sécurité - Noyau de sécurité - Test de conformité - Critère Communs - Méthode B - Meca

---

Abstract

The work presented in this report deals with the validation of access control defined through security policies. We are interested in the validation by means of test generation based on a security model, written in B. We use the Meca tool, which takes as input a functional model and a description of the security policy, both expressed as B models. Meca generates a security kernel, that is in charge of intercepting the access of the subjects to the objects, and restricts the behaviors to those that satisfy the security requirements. We present a complete case study, led on an electronic purse case study, using a discretionary access control policy.

Keywords : Access control - Security policy - Security kernel - Conformance testing - Common Criteria - B Method - Meca

---