Renforcement formel et automatique de politiques de sécurité dans des applications Android par réécriture

Thèse

Marwa Ziadia

Doctorat en informatique
Philosophiæ doctor (Ph. D.)

Québec, Canada

Résumé

Autant les applications Android ont réussi à positionner Android parmi les systèmes d'exploitation les plus utilisés, autant elles ont facilité aux créateurs de maliciels de s'introduire et de compromettre ses appareils. Une longue liste de menaces causées par les applications téléchargées vise l'intégrité du système et la vie privée de ses utilisateurs. Malgré l'évolution incessante du système Android pour améliorer son mécanisme de sécurité, le niveau de sophistication des logiciels malveillants a augmenté et s'adapte continuellement avec les nouvelles mesures.

L'une des principales faiblesses menaçant la sécurité de ce système est le manque abyssal d'outils et d'environnements permettant la spécification et la vérification formelle des comportements des applications avant que les dommages ne soient causés. À cet égard, les méthodes formelles semblent être le moyen le plus naturel et le plus sûr pour une spécification et une vérification rigoureuses et non ambiguës de telles applications.

Notre objectif principal est de développer un cadre formel pour le renforcement de politiques de sécurité dans les applications Android. L'idée est d'établir une synergie entre le paradigme orienté aspect et les méthodes formelles. L'approche consiste à réécrire le programme de l'application en ajoutant des tests de sécurité à certains points soigneusement sélectionnés pour garantir le respect de la politique de sécurité. La version réécrite du programme préserve tous les bons comportements de la version originale qui sont conformes à la politique de sécurité et agit contre les mauvais.

Abstract

As much as they have positioned Android among the most widely used operating systems, Android applications have helped malware creators to break in and infect its devices. A long list of threats caused by downloaded applications targets the integrity of the system and the privacy of its users. While the Android system is constantly evolving to improve its security mechanism, the malware's sophistication level is skyrocketing and continuously adapting with the new measures.

One of the main weaknesses threatening smartphone security is the abysmal lack of tools and environments that allow formal specification and verification of application behaviors before damage is done. In this regard, formal methods seem to be the most natural and secure way for rigorous and unambiguous specification and verification of such applications.

Our ultimate goal is to formally enforce security policies on Android applications. The main idea is to establish a synergy between the aspect-oriented paradigm and formal methods such as the program rewriting technique. The approach consists of rewriting the application program by adding security tests at certain carefully selected points to ensure that the security policy is respected. The rewritten version of the program preserves all the good behaviors of the original one that comply with the security policy and acts against the bad ones.

Table des matières

Liste des tableaux

Liste des figures

Listings

À la mémoire de mon père,
Habib Ziadia.

Remerciements

Le parcours doctoral a été pour moi une expérience enrichissante sur le plan personnel et professionnel. Une épreuve qui a constitué l'un de mes plus grands défis personnel. Pour rendre ce travail possible, il y a ceux et celles qui ont contribué avec leurs appuis multiformes.

J'exprime ma profonde reconnaissance à Pr. Mohamed Mejri pour la direction de mon travail. Je le remercie vivement pour le support multiple qu'il m'a apportée tout au long de cette recherche. J'ai eu la chance d'être dirigée par une personne dotée d'une très grande capacité d'écoute et de nombreuses qualités. Ses conseils lumineux et ses généreux secours au cours de certains de mes moments difficiles ont été d'un immense réconfort. Sa capacité à balancer entre encouragements et critiques constructives a contribué à la concrétisation de mes idées de recherches et à la construction de ce travail.

J'adresse également mes sincères remerciements à mon codirecteur Dr. Jaouhar Fattahi, qui a su me transmettre sa passion, son souci de la perfection et son dévouement pour la recherche. Ses qualités scientifiques m'ont permis de profiter de ses connaissances et m'ont donnée l'envie d'aller plus loin. Je lui suis extrêmement reconnaissante pour sa grande disponibilité, sa bonne humeur et ses conseils sur tous les plans.

J'aimerais également remercier Mme Samia Loukil pour son aide et sa gentillesse.

J'exprime ma gratitude aux membres de Jury pour avoir accepté l'évaluation de ma thèse.

Je dédie ce travail à l'âme de mon père. Bien qu'il ne soit pas avec nous pour récolter le fruit de ses sacrifices et les principes qu'il m'avait inculquée, il reste toujours le plus présent.

Un remerciement spécial à ma tendre mère, qui a toujours cru en moi. Son support illimité et son amour inconditionnel ont fait de moi ce que je suis. Qu'elle trouve en l'aboutissement de cette œuvre, la récompense de ses sacrifices et des moments passés loin d'elle.

Une reconnaissance spéciale à mon époux Najd qui m'a toujours épaulée et accompagnée tout au long de ce parcours. Son soutien moral inestimable, sa compréhension et son affection ont énormément contribué à mener à terme ce travail. Je dédie cette thèse également à ma source d'inspiration et de bonheur, mon adorable fille Line.

Je voudrais remercier ma sœur Rania, qui ne cesse de soutenir et de m'encourager à progresser. Je

remercie par la même occasion mon frère Rami et mes nièces Hala, Jana et Rana.

Enfin, un grand merci à mes beaux parents, mes belles-sœurs et mes amis en Tunisie et au Canada, qui m'ont soutenue et encouragée le long de la réalisation de ce travail.

Introduction

Problématique et motivations

À mesure que les utilisateurs se tournent aujourd'hui vers l'utilisation des téléphones intelligents plus que les ordinateurs, les données qui y résident (messages instantanés, photos personnelles, contacts, messagerie vocale, localisation géographique, etc.) ont tendance à être plus personnelles, récentes, sensibles et précieuses. Ces petits appareils permettent à leurs propriétaires d'effectuer une multitude de tâches telles que l'envoi de courriers électroniques, l'achat des biens en ligne, l'accès à leurs comptes bancaires, etc. En 2021, avec la pandémie de la Covid-19, la consommation d'applications mobiles a connu une forte croissance, en particulier les applications de travail et d'enseignement [2]. Les consommateurs sont contraints de rester chez eux, et de tourner vers le mobile afin de rester connectés, informés et divertis.

Parmi les systèmes d'exploitation mobiles se démarque Android comme étant la plateforme mobile la plus répandue, qui équipe aujourd'hui, à un rythme effréné, des centaines de millions des téléphones intelligents. En 2020, plus que 84% du marché mondial sont des appareils équipés d'un système d'exploitation Android [3]. Une popularité qui a fait de ce système une cible idéale pour des applications malveillantes qui exploitent le point le plus critique d'un tel système, à savoir, sa sécurité.

Aujourd'hui, la sécurité en Android ne suit pas la même courbe de sa croissance, laquelle est devenue sujet à une augmentation exponentielle des attaques menées à l'encontre de ses appareils. En réalité, tout le succès qu'a connu Android revient à la panoplie d'applications disponibles offertes. Un large éventail de choix destiné à tous les goûts et toutes les catégories d'âge. Toutefois, ces applications représentent une arme à double tranchant. Leurs variétés ont contribué à étendre les fonctionnalités et les services de la plateforme, mais en même temps, elles en ont fait une cible privilégiée pour les exploits malveillants. En effet, elles représentent pour les créateurs de maliciels et ceux qui cherchent l'argent facile un moyen de s'introduire et de mettre la main sur les données personnelles qui résident dans ces appareils. De surcroît, la possibilité de télécharger ces applications à partir des sources inconnues et non officielles, telles que *AppsApk.com* et *pandaapp*, et ce sans être soumises à un processus d'examen rigoureux, a aggravé davantage la situation.

Effectivement, les utilisateurs sont de plus en plus exposés à des attaques ciblant l'environnement Android via des applications non fiables. Le rapport McAfee 2020 sur les menaces [4] confirme que

les applications illégitimes constituent la catégorie de menaces mobiles la plus active, générant près de la moitié de toutes les télémesures malveillantes, avec une augmentation de 30% par rapport à 2018 [4].

Ce type de fausses applications met en péril les informations relatives à la vie privée. Une autorisation comme "*accéder au stockage externe*" peut être exploitée pour divulguer des données personnelles comme les photos, le contenu des messages privés ou pour collecter des informations bancaires sensibles, surtout avec l'utilisation croissante des applications de gestion bancaire. Le cheval de Troie mobile *Anubis* [5] est un exemple de maliciels dissimulé dans des applications, en apparence inoffensives, qui ciblent les comptes bancaires. De surcroît, l'installation d'une application malveillante en apparence légitime peut mener à l'exploit des services payants pour causer des pertes financières à l'utilisateur au profit de l'attaquant. Ce comportement a été aussi observé avec les chevaux de Troie SMS tels que *AsiaHitGroup* [6] et *GGTracker* [7]. Les deux ont manifesté au niveau des applications téléchargées comme l'application *Fake Player* [8]. Une autre catégorie d'applications malveillantes tirent profit des composants d'un téléphone intelligent et exploitent ses ressources de calcul pour, par exemple, miner des crypto-monnaies. Dans ce sens, récemment, huit applications mobiles frauduleuses ont été détectées (p. ex. *AndroidOS_FakeMinerPay* et *AndroidOS_FakeMinerA*) [9]. En plus d'inciter les victimes à regarder des publicités et à payer pour des services d'abonnement, ces applications sont capables d'affecter l'autonomie et la performance de l'équipement et dans certains cas l'appareil risque de surchauffer jusqu'à devenir inutilisable.

Ces menaces à la sécurité mobile sont de plus en plus imposantes et sollicitent beaucoup de préoccupations dont l'objectif commun est de renforcer la sécurité des applications téléchargées et par la suite protéger l'utilisateur. Toutefois, l'une des principales faiblesses des solutions proposées est l'absence abyssale d'outils et d'environnements permettant la vérification formelle du comportement des applications, et en particulier la détection précoce de tout comportement malveillant, avant que des dommages irréversibles ne soient causés. À cet égard, plusieurs techniques de renforcement de politiques de sécurité dans les applications Android ont été mises en place durant les dernières années. Cependant, malgré quelques résultats prometteurs, le problème est loin d'être résolu et sans une base formelle, rien n'est garanti.

Le recours aux méthodes formelles avec leur fondement sur la logique mathématique, permettra d'atteindre un certain raisonnement rigoureux et non ambigu dans la spécification d'un système et de ses propriétés tout en garantissant la correction du résultat. En effet, contrairement aux approches basées sur les tests, qui ne peuvent aboutir qu'à des systèmes qui satisfont aux exigences de scénarios de test, les méthodes formelles utilisent des preuves mathématiques irréfutables pour produire des systèmes de haute qualité, aux propriétés éprouvées.

Par ailleurs, la nature d'Android comme étant un système complexe, combinant plusieurs aspects tels que la concurrence, la réflexion et la sécurité, complique davantage la tâche de spécification et de vérification. De ce fait, l'utilisation des méthodes formelles uniquement paraît insuffisante.

Face à ces problématiques, le paradigme orienté aspects combiné à l'utilisation de méthodes formelles nous semble une orientation prometteuse. Cette synergie entre les deux permettra à la fois d'incorporer automatiquement des aspects dans la spécification du système, tout en produisant des systèmes qui respectent leurs spécifications. C'est dans cet axe de recherche que se situe notre travail. La réécriture de programmes est un excellent exemple d'approche qui réunit ces deux principes. Elle vise à réécrire le programme de manière statique en fonction d'une propriété de sécurité donnée afin de générer une nouvelle version exécutable qui la satisfait.

La version renforcée doit satisfaire deux propriétés fondamentales, à savoir : la correction et la complétude. La première stipule que la version réécrite satisfait la politique de sécurité qui lui est associée. La deuxième assure que la version renforcée préserve tout comportement de la version originale du programme qui satisfait la politique et ne rajoute aucun nouveau comportement.

Objectifs

Notre objectif principal consiste à renforcer [1] automatiquement une politique de sécurité dans une application Android. L'approche permettra, à partir d'une spécification formelle d'une application Android et d'une politique de sécurité de générer une nouvelle version de l'application qui se comporte conformément à la politique.

Afin d'atteindre cet objectif, il fallait passer par des sous-objectifs dans l'ordre suivant :

1. Formalisation d'une application Android (syntaxe et sémantique) ;
2. Spécification formelle de la politique de sécurité ;
3. Définition d'une technique formelle pour le renforcement de la politique de sécurité qui résulte de l'étape précédente au sein du programme obtenu en 1 de façon que le nouveau programme généré respecte cette politique ;
4. Preuve de correction et de complétude de l'approche de renforcement proposée ;
5. Automatisation de l'approche au complet.

Méthodologie

Notre contribution consiste en l'élaboration d'une approche formelle pour le renforcement automatique de politiques de sécurité au sein des applications Android. Pour y parvenir, nous prévoyons une exploitation des méthodes formelles dans chaque étape.

Figure 0.1 résume les grandes étapes de notre approche. L'entrée consiste en une application Android au format *APK* (pour « Android Package Kit »). Ce fichier est converti en premier lieu en un format lisible via un outil d'aide à la rétro-ingénierie. Le résultat est un fichier *Smali* contenant le code

1. le mot "renforcer" est utilisé dans tout ce document dans le sens de contraindre ou obliger le système à respecter la politique de sécurité.

de l'application accompagné d'un fichier *AndroidManifest* dans une représentation intelligible. La formalisation de ces deux fichiers représente le code de bas-niveau de l'application, ainsi que toutes les informations nécessaires extraites du fichier *AndroidManifest*. À ce niveau, une politique de sécurité exprimant un comportement particulier rentre dans la configuration. La politique est décrite avec une logique LTL. Ensuite, un opérateur de renforcement $\sqcap$ prend en entrée la politique de sécurité et le programme et génère une nouvelle version de ce dernier qui respecte cette politique.

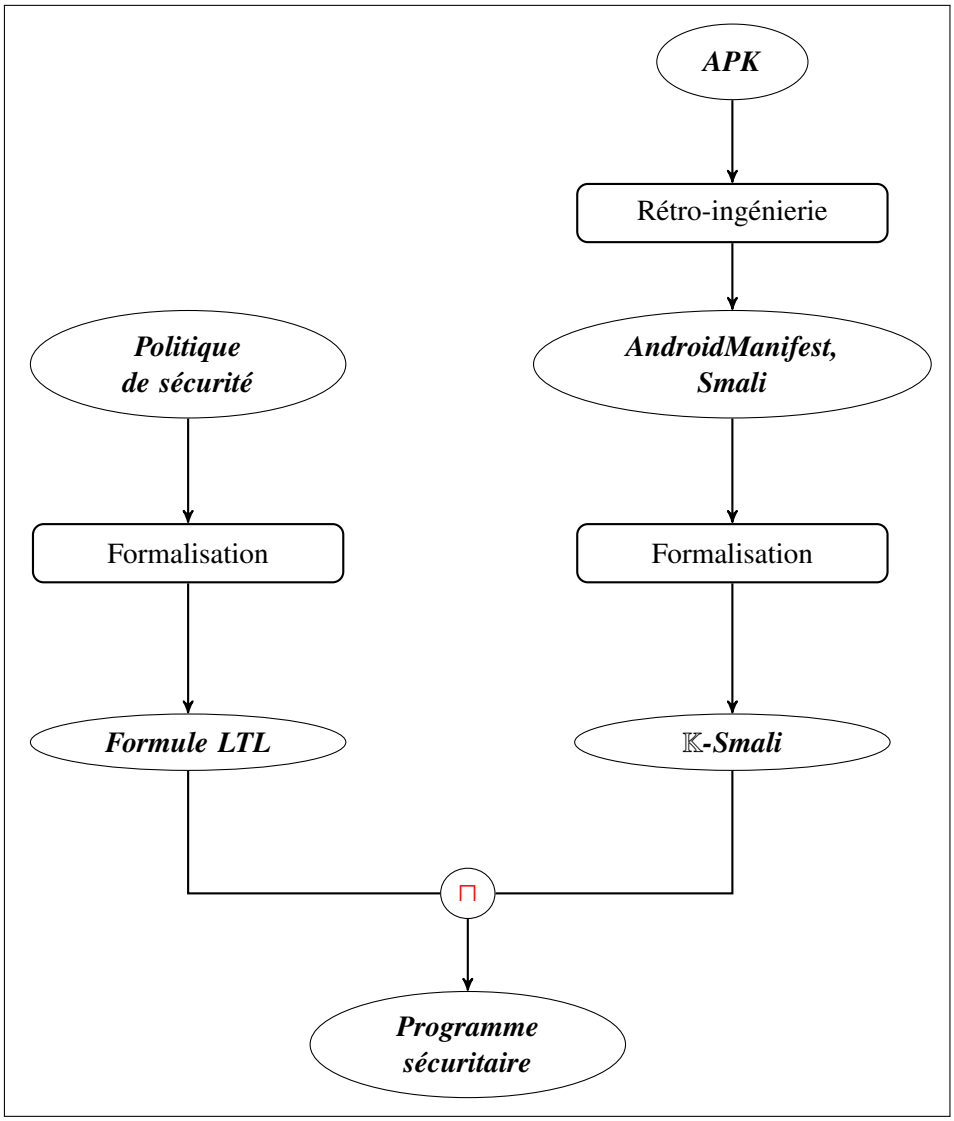


FIGURE 0.1 – Étapes de renforcement.

Contributions

Les principales contributions de cette thèse sont les suivantes :

1. *Définition d'une sémantique opérationnelle pour une application Android,* Smali^+ [10]. Dans cette contribution, nous avons défini une sémantique opérationnelle dédiée du code Smali, obtenu suite à la rétro-ingénierie d'une application Android. Les règles sémantiques conçues modélisent le comportement séquentiel et parallèle du langage.

2. *Définition d'une sémantique exécutable pour une application Android,* $\mathbb{K}$-*Smali* [11]. L'objectif de cette contribution est d'améliorer plusieurs points qui n'ont pas été pris en compte lors de la première formalisation : en plus d'être sujette à l'erreur, une sémantique dédiée nécessite la conception des outils personnalisés, pour l'interprétation, le débogage et la vérification du langage, etc., souvent coûteux. Afin de contourner ces inconvénients, nous avons utilisé un cadre pour la définition formelle des langages appelé $\mathbb{K}$ *Framewrok*. Cet environnement est doté d'un ensemble d'outils, certains sont utiles pendant le processus de formalisation, d'autres durant la vérification et l'analyse syntaxique. La sémantique générée est ainsi exécutable. Elle a été compilée, et par la suite testée, contre un nombre potentiellement important de programmes afin de gagner en confiance quant à sa correction. Dans cette contribution, nous avons également vérifier des propriétés liées à certaines vulnérabilités relatives à Android, telles que l'espionnage et les chevaux de Troie SMS en utilisant l'évaluateur de modèle offert par $\mathbb{K}$.

3. *Renforcement formel de politiques de sécurité dans des applications Android par réécriture* [12]. Dans cette contribution, nous avons défini une technique formelle pour renforcer des politiques de sécurité dans une application Android, de façon à générer une nouvelle version sûre de l'application. L'idée a été initiée en [13], où nous avons défini un opérateur de renforcement qui prend en entrée un programme parallèle spécifié par une version étendue de l'algèbre de processus *ACP* (Algebra of Communicating Process), une politique de sécurité présentée comme une formule logique, et un seuil de risque fixé par l'utilisateur. Le résultat est un nouveau processus qui respecte la politique de sécurité sans dépasser la valeur de risque spécifiée. Cette idée a été affinée par la suite pour s'adapter au système cible de renforcement, qui est l'application Android. Dans cette contribution, nous avons spécifié dans un premier temps la politique de sécurité à l'aide d'une variante proche de la logique LTL. En plus d'avoir une syntaxe simple et facile à assimiler, nous avons trouvé qu'elle se marie bien avec la syntaxe du langage formel cible. Afin d'intégrer cette politique dans le programme, nous avons ajusté sa forme syntaxique afin qu'elle s'intègre avec les constructions du langage du programme, et par la suite, nous l'avons transformée en un moniteur qui génère les mêmes traces. Le programme $\mathbb{K}$-*Smali* de son côté a subi une réécriture de façon d'être en harmonie avec la politique. Ensuite, nous avons défini un opérateur de renforcement $\sqcap$. Ce dernier permet à partir du programme réécrit et de la politique transformée de générer une nouvelle version du programme qui se comporte conformément à la politique. L'avantage majeur de cette technique est qu'elle inclut intrinsèquement dans sa spécification les deux propriétés de complétude et de correction. En effet, les systèmes de réécriture fournissent une approche naturelle et avantageuse pour le problème de renforcement sur différents points. Son côté intuitif vient du fait que le système renforcé cor-

respond vraiment à une forme réécrite de la version originale, en insérant, si besoin, des tests de contrôle à des points bien précis. Toutefois, le grand avantage que cette technique peut nous offrir est au niveau de la simplification des preuves des résultats visés. En effet, au lieu de faire un seul bloc de transformation et de prouver qu'il aboutit aux résultats visés, il sera divisé en des petites transformations (règles de réécriture) qui préservent des propriétés aboutissant aux résultats escomptés et dont les preuves sont relativement courtes et moins compliquées. Grâce à ceci, nous avons prouvé d'une façon simplifiée que la version générée satisfait la politique de sécurité, tout en étant conforme à sa spécification d'origine.

4. *Définition d'une sémantique $\mathbb{K}$ pour le renforcement automatique de politiques de sécurité dans les applications Android* [14]. Dans cette contribution, nous nous sommes focalisés sur l'automatisation du processus complet de renforcement. Pour ce faire, nous avons utilisé du nouveau $\mathbb{K}$ *Framework* pour la définition d'une sémantique $\mathbb{K}$ (syntaxe, configuration et règles de réécriture) pour le renforcement de politiques de sécurité. Toutes les étapes nécessaires au renforcement, y compris la transformation de la politique, la réécriture du programme et la définition de l'opérateur de renforcement sont établies par $\mathbb{K}$. Cet environnement agit comme un prototype de l'approche qui prend la spécification $\mathbb{K}$ de l'application Android et la politique de sécurité, applique une séquence de modifications et d'ajustements, et génère une nouvelle application qui atteint les objectifs de la politique. La sémantique identifie automatiquement les points pertinents du code dans lesquels les contrôles de sécurité devraient être appliqués. La particularité de cette approche est que le résultat de renforcement peut être vérifié directement grâce à l'interprète offert par $\mathbb{K}$. Il suffit de simuler le programme renforcé et voir concrètement si les traces d'exécutions générées sont conformes à la politique de sécurité introduite.

Le tableau suivant présente les chapitres correspondants à chaque contribution.

Contribution 1	Chapitre 4
Contribution 2	Chapitre 5
Contribution 3	Chapitre 6
Contribution 4	Chapitre 7

Publications

La liste des publications qui contribuent à cette thèse est la suivante :

1. M. Ziadia and M. Mejri, *Formal Enforcement of Security Policies on Parallel Systems with Risk Integration*, in Codes, Cryptology, and Information Security - First International Confe- rence, C2SI 2015, Rabat, Morocco, May 26-28, 2015, Proceedings - In Honor of Thierry Berger (S.

E. Hajji, A. Nitaj, C. Carlet, and E. M. Souidi, eds.), vol. 9084 of Lecture Notes in Computer Science, pp. 133–148, Springer, 2015.

2. J. Fattahi, M. Mejri, M. Ziadia, E. Pricop, and O. Samoud, *Introduction to SinJAR (a New Tool for Reverse Engineering Java Applications) and Tracing Its Malicious Actions Using Hidden Markov Models*, in New Trends in Intelligent Software Methodologies, Tools and Techniques - Proceedings of the 16th International Conference, SoMeT_17, Kitakyushu City, Japan, September 26-28, 2017 (H. Fujita, A. Selamat, and S. Omatu, eds.), vol. 297 of Frontiers in Artificial Intelligence and Applications, pp. 441–453, IOS Press, 2017.

3. M. Ziadia, J. Fattahi, M. Mejri, and E. Pricop, Smali+ : *An Operational Semantics for Low-level Code Generated from Reverse Engineering Android Applications*, Information, vol. 11, no. 3, 2020. (Journal)

4. M. Ziadia, M. Mejri, and J. Fattahi, *Formal and Automatic Security Policy Enforcement on Android Applications by Rewriting*, in New Trends in Intelligent Software Methodologies, Tools and Techniques - Proceedings of the 20th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques, SoMeT 202, Cancun, Mexico, 21-23 September, 2021 (H. Fujita and H. Pérez-Meana, eds.), vol. 337 of Frontiers in Artificial Intelligence and Applications, pp. 85–98, IOS Press, 2021.

5. M. Ziadia, M. Mejri, and J. Fattahi, *K-smali : An Executable Semantics for Program Verification of Reversed Android Applications*, The 14th International Symposium on Foundations & Practice of Security. Paris, France, FPS, pp. 1–17, 2021.

6. M. Ziadia, M. Mejri, and J. Fattahi, K Semantics for Security Policy Enforcement on Android Applications with Practical Cases, in the 2nd EAI International Conference on Computational Intelligence and Communications November 18-19, 2021 Versailles, France (E. CICom 2021, ed.), pp. 1–17, EAI CICom 2021, 2021.

Plan de la thèse

Le contexte de notre thèse peut être réparti en deux volets principaux. Le premier volet est lié au système cible de renforcement de la sécurité, qui est le système Android en général et ses applications en particulier. Ainsi tous les détails liés à ces deux cibles, incluant l'architecture d'Android, le mécanisme de sécurité qu'il adopte, le cycle de vie d'une application Android depuis le développement, la compilation jusqu'à son installation et exécution sur un appareil Android, les différentes vulnérabilités découvertes et le langage généré suite à sa rétro-ingénierie, sont présentés dans le premier chapitre. Le deuxième chapitre est consacré au deuxième volet lié au renforcement de politiques de sécurité. Ce chapitre recense l'état de l'art de différents mécanismes de renforcement sur un système en général.

Le troisième chapitre connecte les deux chapitres précédents et présente l'application de différentes approches discutées dans le chapitre 2 sur le système Android détaillé dans le chapitre 1.

Dans le chapitre 4, nous proposons une formalisation d'une application Android, notamment le code obtenu suite à sa rétro-ingénierie Smali. Le langage formel résultant est appelé *Smali*$^+$. Dans le chapitre 5, nous proposons une nouvelle sémantique exécutable définie à l'aide d'un environnement de définition formelle des langages, $\mathbb{K}$ *Framework*. La sémantique obtenue s'appelle $\mathbb{K}$-*Smali*. Ce chapitre termine par une comparaison entre les deux sémantiques *Smali*$^+$ et $\mathbb{K}$-*Smali*.

Dans le chapitre 6, nous proposons une approche formelle pour le renforcement de politiques de sécurité dans les applications Android par réécriture. Cette approche a été automatisée dans le chapitre 7 à l'aide d'une sémantique $\mathbb{K}$.

Le chapitre 8 présente une comparaison avec des travaux connexes. Finalement, une conclusion reprend les principales contributions des présents travaux et propose des pistes de recherche futures.

Première partie

État de l’art

Chapitre 1

Le système Android

1.1 Introduction

Android est le système d'exploitation axé sur l'utilisateur final le plus déployé. Parmi les différents supports (téléphones, tablettes, TV, automobiles et autres catégories à usage spécifique), il existe une gamme vaste et toujours croissante de cas d'utilisation allant de la communication, la consommation de médias, le divertissement à la santé et l'accès à des capteurs, des actionneurs, des caméras ou des microphones. Avec toutes ces fonctionnalités, son modèle de sécurité doit trouver un équilibre difficile entre la sécurité, la convivialité pour les utilisateurs finaux, les garanties pour les développeurs d'applications et les performances du système dans le cadre de contraintes matérielles strictes.

Ce chapitre documente les principales composantes de l'écosystème d'Android, incluant son architecture et ses mesures de sécurité. Il détaille également les différentes caractéristiques liées aux applications Android. Ces détails couvrent les étapes de compilation, le contenu, les différents composants, la programmation concurrentes, les APIs sensibles et le processus de rétro-ingénierie. Dans la deuxième partie, nous mettons l'accent sur le code Smali obtenu suite à la rétro-ingénierie des applications Android. Sa structure, son architecture, le comportement itératif et concurrent seront tous étudiés. Ce code sera notre point de départ pour renforcer la sécurité des applications Android.

1.2 Étude de l'architecture

Comme l'illustre Figure 1.1, l'architecture d'Android est un empilement de quatre couches fondamentales. Chaque couche représente un ensemble spécifique de tâches et fournit des services aux couches au-dessus. Le sens de lecture se fait de bas en haut, allant des services du système d'exploitation (SE) de bas niveau qui dirigent le périphérique mobile jusqu'aux applications.

<table>
<tr><td colspan="2">Couche applicative
SMS/MMS, Contacts, Téléphone, Calculatrice, Alarme, Navigateur, Photo, etc.</td></tr>
<tr><td colspan="2">Cadriciel Android
Gestionnaires, Système de vues, Fournisseurs de contenu, etc.</td></tr>
<tr><td>Bibliothèques
SQLite, SSL, Gestionnaire de surface, WebKit, Gestionnaire audio, Freetype, Libc, OpenGL/ES, etc.</td><td>Engin d'exécution
MVD/ART</td></tr>
<tr><td colspan="2">Noyau Linux
Gestion de la mémoire et d'alimentation, Pilotes de périphériques, etc.</td></tr>
</table>

FIGURE 1.1 – Architecture d'Android

Noyau Linux : Le SE Android est construit sur la base d'un noyau Linux avec quelques modifications architecturales apportées par Google. Cette version est conçue pour faire correspondre à l'environnement mobile et assurer une gestion particulière de la mémoire et de l'alimentation. *Binder wakelocks* et *Ashmemet* sont des exemples des modifications apportées respectivement sur les modules de gestion de mémoire, de gestion d'alimentation, et de gestion d'exécution.

Le noyau Linux représente le pont entre le matériel et le logiciel et agit comme une couche d'abstraction entre les deux. Comme il s'agit du système d'exploitation principal, il est responsable de la gestion des fonctionnalités de base d'Android, telles que la gestion des processus, de la mémoire, du réseau, de la sécurité et des périphériques (p. ex. appareil photo, clavier, écran, etc.). Cette couche contient également de nombreux pilotes de périphériques matériels importants, qui contrôlent le matériel sous-jacent. Par exemple, lors d'un clic sur le bouton de l'appareil photo, l'instruction est transmise au pilote correspondant dans le noyau, qui envoie à son tour les commandes nécessaires au matériel de l'appareil photo. La couche noyau Linux est constituée d'autres pilotes comme les pilotes de clavier, audio et Wi-Fi.

Android utilise un mécanisme de communication inter-processus appelé Binder, largement utilisé pour les interactions entre les applications ainsi que pour les interfaces entre les applications et le SE. Binder est établi par un pilote du noyau et exposé comme un nœud de périphérique spécial sur lequel les applications individuelles opèrent.

Bibliothèques et engin d'exécution : Au-dessus de la couche noyau Linux se trouve la couche des bibliothèques natives d'Android. Ces bibliothèques sont écrites en langage C/C++ et exposées aux développeurs via la couche au-dessus (Cadriciel Android). Parmi les bibliothèques de base, le moteur de base de données SQL, les graphiques 2D et 3D, le moteur de navigation Web Kit et la bibliothèque d'exécution C, libc.

Situé sur la même couche que les « bibliothèques natives », l'engin d'exécution d'Android se compose de la machine virtuelle Dalvik (MVD) ou de son successeur Android Runtime (ART). Une fois installée sur un appareil mobile, une application Android s'exécute dans sa propre instance de machines virtuelle appelée Dalvik [1]. Ce type de machine est destinée pour les équipements mobiles à faible puissance de traitement et à faible mémoire. La MVD tire parti des fonctionnalités de base de Linux, permettant d'obtenir une meilleure isolation entre les différentes applications qui s'exécutent sur l'équipement, réduire la consommation mémoire et assurer la sécurité et le support du threading.

À partir de la version Android 5.0, la MVD a été remplacée par le nouvel engin d'exécution, Android Runtime (ART). Il est important de noter que la MVD ou l'ART est un composant très important de cette couche car il agit comme un traducteur entre les applications et le SE, où il traduit les programmes écrits par les utilisateurs dans le langage de programmation Java en un programme exécutable sur le SE.

Ces deux environnements d'exécution utilisent le même bytecode, appelé Dalvik. Le changement le plus significatif entre les deux se manifeste lors de la conversion de ce bytecode en instructions-machines. La MVD adopte un compilateur à la volée (en anglais «*Just In Time*, JIT»), qui procède par traduire dynamiquement une partie du bytecode Dalvik en code machine optimisé ou ODEX (Optimized DEX) chaque fois que l'application est exécutée. À mesure que l'exécution progresse, d'autres codes sont compilés et mis en cache. Tandis que l'ART utilise l'approche de compilation anticipée (en anglais «*Ahead Of Time*», AOT) et compile le fichier contenant l'exécutable Dalvik (DEX) de manière statique en un fichier d'extension OAT (Optimized Ahead of Time) durant la phase d'installation. Ce changement, comme le décrit Jeehong et al. [15], a amélioré les performances au démarrage de l'application vu l'absence de compilation au moment de l'exécution, mais demande plus de temps d'installation et d'espace mémoire. Ceci revient à l'étape de compilation statique de l'OAT, qui affecte l'efficacité de l'installation de l'APK et entraîne un allongement de sa vitesse d'installation.

Cadriciel Android : Cette couche fournit aux applications les APIs (*Application Programming Interface*) majeures sous la forme de classes Java et les services applicatifs nécessaires pour accomplir leurs tâches. Les principales composantes de cette couche sont :

- *Resource manager* pour la gestion de différents types de ressources utilisées dans l'application ;
- *Telephone manager* pour la gestion des appels téléphoniques ;
- *Location manager* pour la gestion de la localisation en se basant sur des services tels que les cartes et l'unité GPS ;
- *Activity manager* pour la gestion de cycle de vie des applications ;
- *Content provider* (fournisseur du contenu) pour la gestion du partage des données entre les applications.

1. Le nom Dalvik est tiré de nom d'une ville d'Islande, d'où le créateur Dan Bornstein est originaire.

Couche applicative : Au sommet de la pile se trouve la couche applicative. Certaines applications sont préinstallées, notamment le téléphone, le client SMS, le client de messagerie, le calendrier, le navigateur Web et les contacts. Ces applications fournissent des fonctionnalités clés de l'appareil auxquelles d'autres applications peuvent accéder. Cette couche comprend également les applications installées par l'utilisateur. Un développeur a la possibilité de créer sa propre application ou la remplacer par une application existante via le kit de développement logiciel (Software Development Kit, SDK) offert par le SE Android. Les applications créées peuvent interagir avec le système par le moyen de l'API d'Android.

1.3 Mécanismes de sécurité

Le téléchargement d'applications malveillantes sur *Google Play* est facile, car *Google* n'effectue aucune inspection/test de code avant de mettre une application à la disposition des utilisateurs. Des attaques par élévation de privilèges [16] ont été utilisées par des applications malveillantes pour déclencher des téléchargements non autorisés.

En contre partie, Android se base sur un ensemble de mesures de sécurité utilisé comme défense pour assurer la sécurité des données utilisateurs, des applications, de l'appareil et du réseau. Certaines les hérite du noyau Linux, d'autres sont spécifiquement définies et font partie de son modèle de sécurité. Android a connu plusieurs adaptations visant à améliorer ces mesures. L'équipe d'Android Open Source Project (AOSP) publie continuellement des correctifs de sécurité visant à corriger les vulnérabilités identifiées dans le SE et à limiter tout risque causé par des actions malicieuses. Si cette approche est efficace pour corriger les menaces signalées, elle ne l'est pas contre celles qui ne sont pas signalées ou pour lesquelles un correctif n'est pas encore disponible.

Dans ce qui suit, nous présentons les principales mesures de protection déployées dans Android [2].

1.3.1 Contrôle d'accès

Android est un système multi-processus où chaque application (et certaines parties du système) exécute son propre processus. Les installations standard de Linux renforcent la sécurité entre les applications et le système au niveau du processus.

Linux attribue un identifiant d'utilisateur unique (UID) à chaque fois qu'un nouvel utilisateur est créé. Ces utilisateurs peuvent être ajoutés à un groupe qui a un identifiant unique de groupe (GID), qui est utilisé pour faire la distinction par rapport à d'autres groupes. Chaque fichier sur Linux a l'UID de l'utilisateur particulier qui lui est attribué. Seulement cet utilisateur a la responsabilité principale pour le fichier et il peut également modifier ses autorisations. Le même concept s'applique sur Android. Un UID est attribué dès qu'une application Android est installée, et toutes les données (fichiers/base de données) stockées par cette application auront également le même UID. La sécurité de Linux

2. Android améliore continuellement ses offres de sécurité, le lecteur est invité à se référer à [17] pour une liste complète de ces mécanismes de sécurité par version.

empêche les applications qui ont des UID différents d'accéder aux données, processus ou mémoire d'autres applications. Seul un utilisateur privilégié (c.-à-d. ayant un accès root) peut outrepasser ces contraintes.

Android utilise trois mécanismes de permission distincts pour effectuer le contrôle d'accès :

1. *Le contrôle d'accès discrétionnaire (DAC) :* ce type de contrôle d'accès accorde ou restreint l'accès à un objet par le biais d'une politique d'accès déterminée par le groupe propriétaire et/ou les sujets de l'objet. Les DAC sont discrétionnaires car le sujet (propriétaire) peut transférer l'accès aux objets ou aux informations authentifiées à d'autres utilisateurs. En d'autres termes, le propriétaire détermine les privilèges d'accès aux objets. La sécurité d'Android dépend fortement de la protection du DAC pour le système de fichiers Linux et de la vérification des permissions des APIs Java. Toutefois, comme le démontre Smalley et al. [18], le DAC peut être facilement compromis par des logiciels malveillants.

2. *Le contrôle d'accès obligatoire (MAC) :* c'est une politique de niveau système permettant de déclarer les opérations qu'un processus peut exécuter sur une ressource. Seules les actions explicitement autorisées par la politique sont permises. La politique est définie et gérée de manière centralisée par un administrateur système, elle est mise en œuvre en utilisant SELinux et renforcée par le noyau. Comparé au DAC classique, le MAC est plus strict et offre des garanties plus fortes contre les comportements indésirables [19].

3. *Les permissions Android* : Android détient un mécanisme de gestion de permissions permettant de contrôler les accès de l'application aux différentes ressources et services, tels que le repérage de la position par l'unité GPS, l'accès à la messagerie ou à l'Internet. Cette mesure est utilisée dans le but de limiter l'accès à certaines fonctionnalités sensibles, qui peuvent porter préjudice à l'utilisateur si elles sont octroyées à une application non fiable.

   Les permissions font partie d'une politique de sécurité utilisée pour restreindre l'accès à chaque composant d'une application. Toutefois, ces politiques de sécurité présentent des lacunes dans la mesure où elles ne peuvent pas spécifier à quelle application les droits ou les permissions sont accordés, car elles comptent sur les utilisateurs et le SE pour le faire. Quant à Android, il ne dispose pas de mesures de sécurité permettant de déterminer et de renforcer comment, quand, où et à qui les permissions sont accordées [20].

   Les permissions sont définies de manière statique dans le fichier *AndroidManifest.xml* (*manifest*) d'une application. À partir de la version Android 6.0, un changement majeur a été apporté en n'exigeant plus que toutes les permissions demandées soient accordées lors de l'installation d'une application. C'est le résultat direct de la prise de conscience que les utilisateurs ne sont pas suffisamment informés pour prendre une telle décision au moment de l'installation [21]. Le deuxième changement majeur dans les permissions Android a été introduit avec Android 10 sous la forme de permissions non binaires, dépendantes du contexte. En plus d'autoriser et de refuser, certaines permissions (en particulier la localisation, l'appareil photo et le micro-

phone) peuvent maintenant être définies comme autorisées uniquement pendant l'utilisation de l'application.

Malgré les évolutions qu'a subi le système de gestion de permission avec chaque nouvelle version, l'utilisateur reste toujours la clé de voûte. En effet, la décision finale d'accorder les permissions demandées lui revient toujours, et ce, quelque soit le moment, à l'installation ou à l'exécution. Toutefois, la plupart des utilisateurs acceptent de les accorder sans regarder, un peu comme la plupart des gens qui installent des applications sans lire le Contrat de licence d'utilisateur final (CLUF). De plus, une fois accordées, les permissions ne peuvent pas être modifiées ou restreintes.

Les permissions Android sont classées en cinq catégories par ordre croissant de gravité :

- Permissions de type *Normal* : elles ne posent pas de grands risques en matière de confidentialité ou de sécurité. Elles sont accordées automatiquement au moment de l'installation et ne nécessitent pas l'approbation de l'utilisateur.
- Permissions de type *Dangereux* : ce type de permissions devrait à la fois être déclarées dans le fichier *manifest* et accordé pendant l'utilisation. Il protègent les données sensibles couramment utilisées par les utilisateurs.
- Permissions de type *Signature* : ces permissions ne sont disponibles que pour les composants signés avec la même clé que le composant (plateforme ou application) qui déclare la permission. Elles sont destinées à protéger des actions internes ou hautement privilégiées, comme la configuration des interfaces réseau. Elles sont accordées au moment de l'installation si l'application est autorisée à les utiliser.
- Permissions de type *signatureOrSystem* ou permissions privilégiées : elles sont réservées aux applications pré-installées et permettent des actions privilégiées. Les applications faisant appels à des permissions de ce type doivent être signées par l'application qui a déclaré la permission ou être un UID système qui est accessible seulement par les fabricants d'équipements originaux (OEM) disposant d'un accès root.

1.3.2 Principe du bac à sable

Le principe de base adopté pour contrer les menaces introduites par les applications est la conception d'un bac à sable. Il s'agit d'un environnement restreint pour l'exécution de l'application où les actions ne peuvent pas accéder aux ressources au-delà de ce qui a été autorisé au moment de son installation. Cela isole les applications les unes des autres et contrôle toute interférence avec les autres applications et avec les ressources de dispositif en suivant les mesures de contrôle d'accès et les restrictions de privilèges. À titre d'exemple, une application de jeu qui tente d'accéder à l'historique des appels téléphoniques sans autorisation sera inhibée si elle ne dispose pas des privilèges de l'utilisateur par défaut. Rossi et al. [19] compare l'application du bac à sable au principe de sécurité du "moindre privilège", qui stipule que chaque programme doit fonctionner en utilisant le minimum de privilèges nécessaires pour accomplir sa fonction. Ce modèle de sécurité s'applique sur le code interprété comme

sur le code natif. De même, tous les logiciels situés au-dessus du noyau (bibliothèques de SE, engin d'exécution, etc.) s'exécutent dans le bac à sable [22]. Le bac à sable reste le principal mécanisme de sécurité d'Android et une base solide sur laquelle des restrictions supplémentaires peuvent être ajoutées. Ses lacunes ont été atténuées de plusieurs façons au fil des versions ultérieures, notamment par l'ajout de politiques MAC avec SELinux en mode d'application, et de nombreux autres mécanismes tels que les permissions Android et le DAC. Chacun d'entre eux s'alignant globalement sur la manière dont les utilisateurs, les développeurs et la plateforme accordent respectivement leur consentement.

1.3.3 SELinux

Faisant partie de son modèle de sécurité, Android utilise le système SELinux (Security-Enhanced Linux) pour renforcer sa sécurité. Un usage intensif qui a pour but de protéger les composants du système et affirmer les exigences du modèle de sécurité pendant les tests de compatibilité. SELinux met en œuvre le mécanisme de contrôle d'accès obligatoire (MAC). Il a été introduit dans Android 4.3 en 2013. Il utilisait un ensemble limité de domaines système et visait principalement à séparer les ressources système des applications utilisateurs. Dans les versions suivantes, la configuration de SELinux est devenue progressivement plus complexe [23], avec un ensemble croissant de domaines isolant différents services et ressources, afin qu'un bogue ou une vulnérabilité dans un composant du système ne conduise pas à une compromission directe de l'ensemble du système. Shabtai et al. [24] démontrent l'utilité de SELinux dans Android à l'aide de plusieurs scénarios. Dans ces scénarios, ils supposent une vulnérabilité existante dans Android et montrent comment réduire les dommages qu'elle pourrait causer en prolongeant le temps disponible afin de la corriger convenablement.

Étant donnée que le noyau Android est légèrement différent d'un noyau Linux standard, le SEAndroid (Security Enhancement for Android) [25] a été utilisé par la suite afin d'intégrer et adapter SELinux dans Android de manière complète et cohérente. Des composants supplémentaires spécifiques à Android ont été ajoutés. En effet, le SEAndroid apporte toutes les fonctionnalités de SELinux à Android et rend l'infrastructure de sécurité beaucoup plus robuste. Il permet de confiner les démons système privilégiés s'ils sont compromis et de limiter les dommages qu'ils peuvent causer. Il permet également de créer un bac à sable applicatif plus solide, en plus des mécanismes existants basés sur le DAC. Il renforce l'isolation des données entre les applications en contrôlant la communication entre les applications et entre les applications et les services système.

1.3.4 Signature des applications

Android exige que chaque application soit signée avec une clé privée incluse dans un certificat généré par le développeur afin de renforcer les permissions de signature. Toutefois, l'application de la signature des applications ne se fait pas de manière centralisée, c.-à-d. que les développeurs doivent eux-mêmes signer le code de l'application avec la clé auto-certifiée. Ainsi, la signature des applications ne fournit pas de protection contre les logiciels malveillants, mais permet d'établir des relations de confiance entre les applications provenant du même développeur [26]. Les applications signées

avec la même clé peuvent demander à partager le même UID, c.-à-d. qu'elles seront placées dans le même bac à sable. En effet, la vérification des signatures des certificats nécessite souvent l'utilisation des protocoles cryptographiques hautement sécurisés dont la vérification de leur sécurité est parfois incertaine [27–29].

1.3.5 Typage

Les applications Android natives sont développées en Java. Ce langage est fortement typé, ce qui minimise le risque de commettre des erreurs d'incohérence de type à la programmation, susceptibles d'être exploitées pour l'exécution de codes arbitraires ou pour des attaques de débordement de tampon (*buffer overflow*) [30].

1.3.6 Unité de gestion mémoire

À l'instar des autres systèmes d'exploitation, la plateforme Android implémente une unité de gestion de mémoire qui a pour rôle de séparer et protéger les plages mémoires. Ceci permet de dissuader une application d'accéder à l'espace mémoire d'une autre.

1.4 Applications Android

1.4.1 Étapes de compilation

Les applications Android sont principalement écrites en Java. Android fournit également un kit de développement natif (NDK) qui permet le développement du code natif en C/C++. Il est possible alors que l'application adopte un mode d'exécution hybride permettant un basculement transparent entre le code Java et le code natif.

L'interaction entre les composants Java et natifs est assurée avec le support de l'interface Java de programmation native (JNI). D'ailleurs, dans un mode d'exécution hybride, le code natif est compilé sous forme de fichiers Linux autonomes à objets partagés (.so) et emballé avec l'application. Tandis que le code source Java est compilé en premier en un bytecode [3] Java au moyen d'un compilateur Java classique appelé *javac*. Suite à cette étape, les fichiers sources *.java* sont convertis en des fichiers *.class* (contenant le bytecode Java). Ensuite, les fichiers *.class* sont convertis et consolidés en un seul fichier d'extension *dex*, contenant le bytecode optimisé Dalvik. Cette conversion est effectuée par le moyen de l'outil *dx* inclus dans le SDK Android. Le fichier résultant correspond au fichier "*classes.dex*", appelé aussi DEX (Dalvik Exécutable). L'outil "dx" élimine toutes les informations redondantes qui sont présentes dans les classes. Comme l'illustre Figure 1.2, un seul fichier DEX est créé et tous les fichiers de classe sont inclus dans ce fichier. Ainsi, tous les pools de constantes séparés sont regroupés en un seul pool de constantes partagé. De cette façon, non seulement les informations redondantes sont éliminées, mais l'espace de stockage pour stocker le pool de constantes partagé est également

3. Langage de haut niveau intermédiaire entre code source et code machine fourni par l'environnement d'exécution.

conservé. Les fichiers Dex sont des fichiers exécutables Dalvik pour les environnements d'exécution ART et Dalvik.

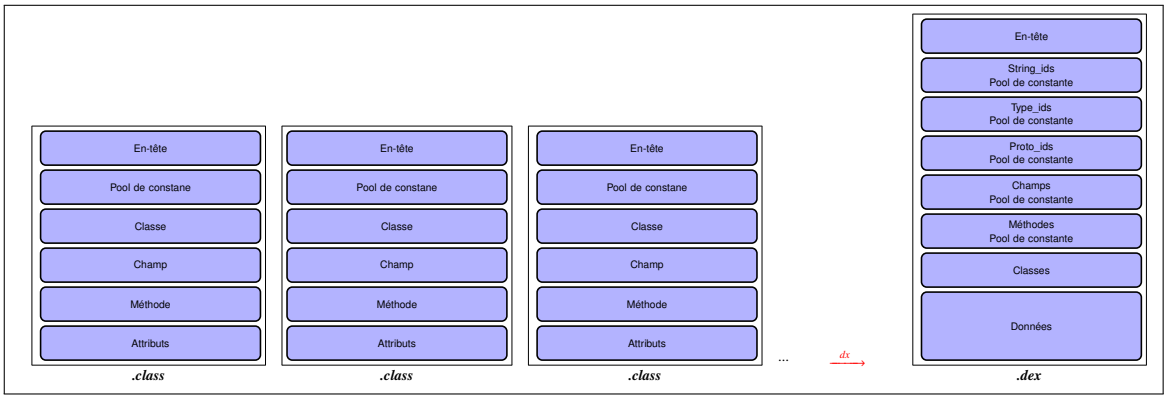


FIGURE 1.2 – Conversion des fichiers *.class* en un fichier *.dex*

Une application Android est construite principalement à partir du fichier DEX accompagné d'un ensemble des ressources, d'éventuelles librairies partagées, un fichier *AndroidManifest.xml* et un répertoire META-INF contient des informations sur la signature. Ce dernier inclut un fichier MANIFEST.MF, qui contient la signature cryptographique et permet de valider l'ensemble du contenu du paquet de distribution. Tous consolidés dans une archive compressée au format Zip nommée Android Package (APK). À cette étape, une fois signée, l'application est prête à être publiée et par la suite installée sur un périphérique. La source officielle de téléchargement d'une application Android est *Google Play*. Figure 1.3 décrit les étapes de compilation d'une application Android.

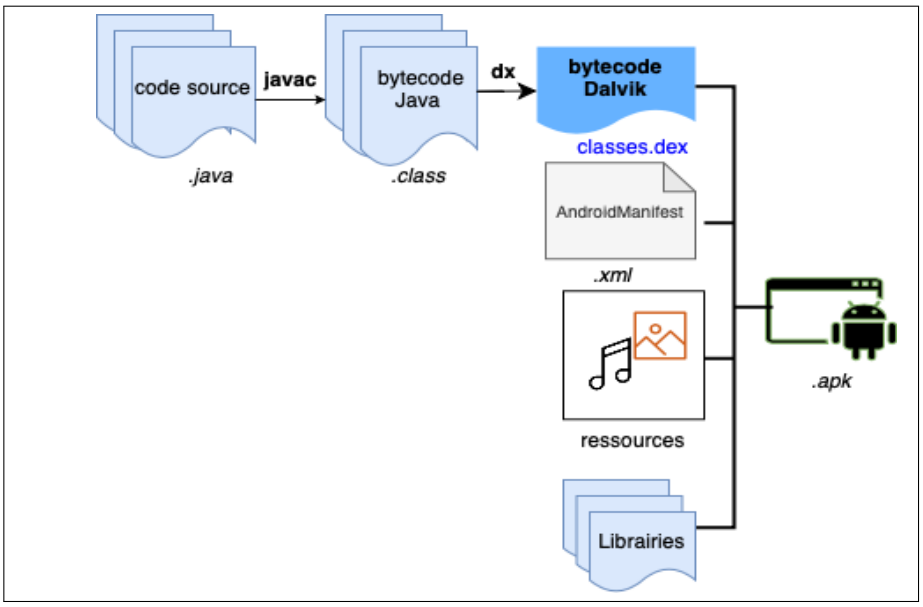


FIGURE 1.3 – Étapes de compilation d'une application Android

Chaque application Android devrait contenir un fichier *manifest* dans son répertoire racine. Ce fichier comprend des informations architecturales de haut niveau sur l'application, qui sont des paramètres d'exécution qui devraient être reconnus et initiés par le système afin de faire fonctionner l'application. Figure 1.1 montre la structure basique d'un fichier *manifest*.

- <manifest> est l'élément racine du fichier manifest. Il comprend essentiellement l'attribut « package » portant le nom du package de l'application ;
- <uses-sdk> est un sous-élément de <manifest>. L'attribut « minSKVersion » indique la version minimum de l'API requis pour faire fonctionner l'application. L'attribut « targetSKVersion » propose une autre version d'API ciblée par l'application. Si cette version n'est pas définie, la valeur par défaut est égale à celle donnée par « minSdkVersion » ;
- <application> comprend des informations sur l'application comme le nom, l'icône et le thème. Cet élément contient également plusieurs sous-éléments qui déclarent le composant de l'application ;
- <component> est le sous-élément de <application> et représente un composant (activité, service, etc.) ainsi que sa description (nom, thème, label, etc.) ;
- <intent-filter> est le sous-élément de <component>. Il spécifie le type d'*intent* (message) auquel le composant concerné peut répondre ;
- <action> indique l'action acceptée par l'intent. <intent-filter> doit avoir au moins un élément <action> ;
- <category> déclare la catégorie de l'*intent* acceptée. Il s'agit d'une chaîne contenant des informations supplémentaires sur le type de composant qui doit gérer l'*intent*.

```
1  <manifest xmlns:android=""
2    package=""
3    android:versionCode=""
4    android:versionName="" >
5    <uses-sdk
6    android:minSdkVersion=""
7    android:targetSdkVersion="" />
8    <application
9    android:icon=""
10   android:label=""
11   android:theme="" >
12   <component
13   android:name=""
14   android:label="" >
15   <intent-filter>
16     <action android:name="" />
17     <category android:name="" />
18    </intent-filter>
19   </component>
20     </application>
21 </manifest>
```

Listing 1.1 – Structure d'un fichier manifest

Le fichier *manifest* peut inclure d'autres informations comme le point d'entrée de l'application, les différentes permissions demandées, des bibliothèques logicielles externes, etc.

Quant aux ressources, ce sont des fichiers utilisés par l'application comme des fichiers multimédia, textes, musique, images, thèmes et les dispositions de l'interface graphique.

1.4.2 Composants

Les applications Android sont structurées en composants de quatre types différents : les activités, les services, les récepteurs de diffusion (en anglais «*Broadcast receivers*») et les fournisseurs de contenu (en anglais «*Content providers*»). La configuration et les interactions de ces composants déterminent le comportement de l'application. En effet, chaque entité a une fonctionnalité et une responsabilité différente, ainsi qu'un cycle de vie bien déterminé. Il suffit de lancer un composant, ce dernier peut déclencher un autre et ainsi de suite tout au long du cycle de vie de l'application.

Une activité est une tâche orientée utilisateur. Elle contient les composants de l'interface utilisateur (p. ex. boutons, textes, images, etc.) affichés à l'écran. Lorsque l'utilisateur navigue entre les écrans, les instances d'activité forment une pile. La navigation vers un nouvel écran empile une activité vers la pile, tandis que la navigation en arrière provoque son dépilement. Par conséquent, l'activité en cours d'exécution se trouve toujours au sommet de la pile. Figure 1.4 illustre le cycle de vie d'une activité. Une activité passe principalement par quatre états : (1) « active » si elle apparaît en premier plan ; (2) « en pause » quand elle est partiellement visible, c.-à-d. qu'elle reste attachée au gestionnaire de fenêtres. Dans ce cas, le système peut la détruire en cas de manque de mémoire ; (3) « arrêtée » quand elle n'est plus invisible à l'utilisateur, mais existe toujours en mémoire. Elle peut toujours être détruite par le système pour libérer les ressources pour d'autres activités actives ; (4) « détruite » et supprimée de la pile des activités. Il fallait la redémarrer afin d'être affichée à nouveau.

Le service est par contre une tâche de fond (sans interface graphique). Un exemple typique de celui-ci est le lecteur musique qui continue à s'exécuter en arrière-plan même s'il n'est pas affiché en premier plan. Un fournisseur de contenu encapsule les données et assure leurs partages entre les différentes applications autorisées. Tandis qu'un récepteur de diffusion répond aux diffusions du système Android, auxquels l'application s'est enregistrée.

Les activités, les récepteurs de diffusion et les services peuvent communiquer par le biais des messages asynchrones, baptisés *intents* utilisés également pour la communication inter-applications. Il s'agit d'une description abstraite d'une opération à effectuer, utilisée pour lancer une activité, communiquer avec des services en arrière-plan, ou pour envoyer des données à tout composant récepteur de diffusion intéressé.

Comme l'illustre Figure 1.5, l'objet *intent* contient plusieurs champs. La façon dont ces champs sont décrits détermine sa nature et sont objectif. Un *intent explicite* précise explicitement le composant destinataire ou l'application qui peut lui répondre, en fournissant soit le nom du package de l'applica-

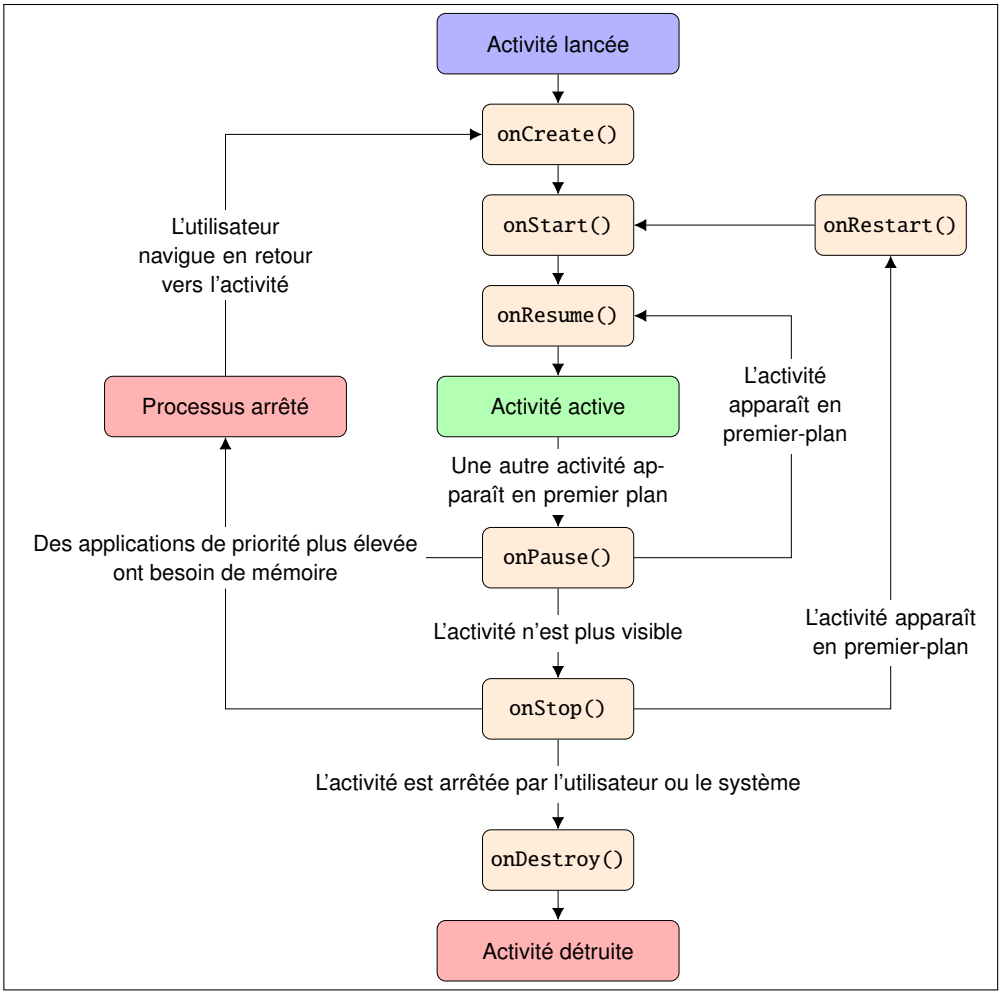


FIGURE 1.4 – Cycle de vie d'une activité

tion cible, soit le nom entièrement qualifié de la classe de composant. Il peut être utilisé, par exemple, pour lancer un service permettant de télécharger un fichier en arrière-plan. Un *intent implicite* ne désigne pas un composant récepteur. À titre d'exemple, lorsque l'utilisateur clique sur un lien avec un *intent* implicite, le choix est laissé aux paramètres du système et/ou l'utilisateur pour déterminer l'application navigateur assurant l'ouverture de lien. Pour ce faire, le système Android compare les différents champs et les données sérialisées qui décrivent l'objet *intent* (affichés dans Figure 1.5) aux *intents filters* déclarés dans le fichier *manifest* de l'application. En cas de correspondance, le système démarre ce composant et lui fournit l'objet *intent*. Si plusieurs *intents filters* sont compatibles, le choix de l'application est laissé à l'utilisateur.

FIGURE 1.5 – Architecture d'un objet intent

1.4.3 Points d'entrée

Contrairement aux paradigmes de programmation où l'application se lance à partir de sa méthode *main*(), une application Android ne dispose pas d'une fonction principale ou d'un point d'entrée unique. Si on la compare à une application Java standard qui est une fois lancée, la machine virtuelle Java (JVM) exécutante procède en localisant sa méthode *main*() et commence l'exécution à partir de ce code. Dans le cas d'une application Android, chaque composant représente un point d'entrée potentiel et différent. Ces composants sont identifiés dans le fichier *manifest* de l'application et appelés au moment de l'exécution. Par exemple, quand l'utilisateur clique sur l'icône de l'application pour la lancer, le point d'entrée dans ce cas et d'ailleurs avec toute interaction utilisateur est une activité. La MVD localise la première activité qui se lance au démarrage de l'application. Il s'agit de l'activité ayant la valeur «Main» dans la sous-balise <action> et la valeur «LAUNCHER» dans le sous-élément «category». Ces informations sont récupérées à partir du fichier *manifest* de l'application comme illustré dans Listing 1.2. L'exécution commence à partir de la méthode *OnCreate()* de cette activité. Cette méthode est surchargée automatiquement lors de lancement d'une application afin de mettre en place l'interface graphique, initialiser les variables, etc. Certaines applications n'ont pas d'activité "LAUNCHER", en particulier les applications sans interface utilisateur. Des exemples de telles applications sont les applications pré-installées qui exécutent des services en arrière-plan, comme la messagerie vocale.

```
1  <activity android:name=".MainActivity">
2  <intent-filter>
3  <action android:name="android.intent.action.MAIN"/>
4  <category android:name="android.intent.category.LAUNCHER"/>
5  </intent-filter>
6  </activity>
```

Listing 1.2 – Point d'entrée d'une application Android : Activité

Un récepteur de diffusion peut également former un point d'entrée pour une application Android. En effet, les applications peuvent s'inscrire pour recevoir des diffusions spécifiques. Lorsqu'une diffusion est envoyée, le système l'achemine automatiquement vers les applications qui y sont abonnées. Cet évènement peut être un évènement système tel que le démarrage du système, comme l'illustre Listing1.3, où l'application s'inscrit avec un intent-filter ayant comme action, l'action «BOOT_COMPELTED». Un autre exemple d'événement est une diffusion annonçant que la batterie est faible, ou bien tout autre événement provenant d'une autre application.

```
<receiver android:name="BootReceiver"
  android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
  <intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED">
  </intent-filter>
</receiver>
```

Listing 1.3 – Point d'entrée d'une application Android : Récepteur de diffusion

1.4.4 Programmation concurrente

Android est un système multi-utilisateur et multi-tâche qui peut exécuter plusieurs applications en même temps et permettre à l'utilisateur de passer d'une application à l'autre sans remarquer de délai important. Pour ce faire, il applique un système de classement des processus qui hiérarchise l'importance de chaque application en cours d'exécution afin de s'assurer que seules les applications les moins prioritaires sont arrêtées. Pour augmenter les performances, une application doit répartir les opérations entre plusieurs threads afin que le code soit exécuté simultanément. Le multithreading permet d'améliorer les performances et la réactivité nécessaires pour une bonne expérience utilisateur.

Chaque application est lancée avec de nombreux threads qui sont intégrés au processus Linux et à la MVD pour gérer leurs exécutions internes. Tous ces threads sont basés sur les *pthreads* natifs de Linux avec une représentation des threads en Java, mais la plateforme attribue toujours des propriétés spéciales aux threads qui les rendent différents [31]. Du point de vue application, les types de threads sont les suivants : *UI* (*User Interface*), *binder* et *arrière-plan*.

Chaque processus Linux contient un thread spécifique qui est responsable de la mise à jour de l'interface utilisateur. Toutes les opérations longues doivent être maintenues en dehors du UI thread et exécutées sur d'autres threads. Le *UI* thread est le thread principal de l'application. Les threads *binder* sont utilisés pour communiquer entre les threads de différents processus. Chaque processus maintient un ensemble de threads, appelé *pool de threads*, qui n'est jamais terminé ou recréé, mais qui peut exécuter des tâches à la demande d'un autre thread du processus. Les demandes entrantes peuvent être sous la forme des services système, des *intents*, des fournisseurs de contenu et des services. Tous les threads qu'une application créée explicitement sont des threads d'*arrière-plan*. Par défaut, un processus nouvellement créé ne contient aucun thread d'*arrière-plan*. Il appartient toujours à l'application de les créer si nécessaire.

Chaque application Android doit adhérer au modèle de programmation multithread intégré au langage Java. Un thread dans une application Android est représenté et instancié de la classe *java.lang.Thread*. Il s'agit de l'environnement d'exécution le plus basique d'Android. Un thread prend en charge l'exécution de tâches qui sont des implémentations de l'interface *java.lang.Runnable*. Une implémentation définit la tâche dans la méthode *run*(). Toutes les variables locales dans les appels à l'intérieur d'une méthode *run*() seront stockées sur la pile mémoire locale du thread.

Les différents threads exécutants sont gérés par un ordonnanceur. L'ordonnancement consiste à sélectionner un thread à exécuter, lui allouer un temps d'exécution avant de sélectionner un nouveau et changer de contexte. Le prochain thread à exécuter est choisi en fonction de certaines propriétés du thread, qui sont différentes pour chaque type d'ordonnanceur. Ainsi, comme Android se base sur l'environnement d'exécution Linux, les threads sont planifiés à l'aide de l'ordonnanceur standard du noyau Linux, connu sous le nom de *Completely Fair Scheduler* (*CFS*) ou ordonnanceur complètement équitable. Ce dernier se base sur la priorité du thread comme propriété de sélection. Sous Linux, la priorité du thread est une valeur baptisée "*nice*" ou "*niceness*". Une faible valeur correspond à une priorité élevée et vice-versa. Sous Android, un thread a une valeur "*nice*" comprise entre -20 (le plus prioritaire) et 19 (le moins prioritaire), avec une valeur "*nice*" 0 par défaut. Le *CFS* est "équitable" dans le sens où il essaie d'équilibrer l'exécution des tâches non seulement en fonction de la priorité du thread mais aussi en suivant le temps d'exécution qui lui a été accordé. Si un thread a déjà eu un faible accès au processeur, il sera autorisé à s'exécuter avant les threads les plus prioritaires. Si un thread n'utilise pas le temps alloué pour s'exécuter, le *CFS* veillera à ce que sa priorité soit abaissée afin qu'il obtienne moins du temps d'exécution à l'avenir [31].

La durée de vie d'un thread est déterminée par la longueur de la tâche qu'il exécute. Plus clairement, il démarre lors de l'exécution de la tâche et se termine lorsque la tâche est terminée (c.-à-d. la méthode *run*() retourne) ou qu'il n'y a plus de tâches à exécuter ou qu'il est arrêté par l'ordonnanceur.

1.5 Vulnérabilités et APIs sensibles

Malgré l'évolution continuelle du système Android pour améliorer son mécanisme de sécurité, les concepteurs de maliciels s'adaptent également aux nouvelles mesures.

Les applications Android interagissent avec le SE via des APIs (Application Programming Interface). Tracer les appels aux méthodes de l'API permet de surveiller l'application étant donnée qu'ils présentent tout accès aux : ressources disponibles (liste de contacts, messages, photos, etc.) ; services systèmes (téléphonie, messagerie, caméra, GPS, etc.) ; communication inter-applications et inter-composants (lancer une activité ou un service, etc.) ; exécution d'un programme externe ; gestion des événements ; concurrence ; etc. Pour cette raison, la plupart des approches existantes (p. ex. [1, 32–36]) reposent sur l'analyse de l'API Android pour la détection comportementale de maliciels.

Certaines des ressources partagées par le biais des APIs sont considérées plus critiques que d'autres, par conséquent, les APIs qui y sont liés sont considérés plus sensibles. Par exemple, envoyer un message texte est un API plus sensible qu'un autre permettant de changer la sonnerie du téléphone. Dans cette section, nous examinons de plus près quelques APIs sensibles qui peuvent être exploités pour appliquer des maliciels dans les applications Android.

1.5.1 Services systèmes

Les appareils mobiles stockent une pléthore d'informations sur notre vie personnelle à travers différents services offerts par le système, tels que les capteurs, le GPS, l'appareil photo, le microphone, qui offrent la possibilité de nous tracer à tout moment.

Il existe des logiciels espions qui visent ce type de services. Leurs buts est de surveiller les activités des utilisateurs, recueillir les frappes au clavier ou récolter des informations sensibles. La liste des contacts, à titre d'exemple, s'avère très utile pour envoyer des *spams* et des liens de *phising* [37]. D'autres écoutent et enregistrent clandestinement des conversations téléphoniques [38] et d'autres sont capables de tracer la localisation géographique de l'appareil, de prendre des photos et d'enregistrer des vidéos à l'insu de l'utilisateur. *KidsGuard Pro*, *mSpy*, *Spyic* sont des exemples des applications d'espionnage Android.

L'API permettant d'envoyer un SMS est parmi les APIs les plus sensibles. Une telle fonctionnalité peut être exploitée par les concepteurs de maliciels pour envoyer des SMS à des numéros surtaxés, sans le consentement de l'utilisateur. Les chevaux de Troie SMS sont moins facilement détectables par l'utilisateur. Généralement, après l'envoi d'un SMS à un numéro payant, les attaquants interceptent, surveillent les SMS entrants et bloquent ceux venant du même numéro, pour que l'utilisateur ne soit pas averti. En outre, la permission *sendTextMessage* les autorise à envoyer les messages en arrière-plan sans notifier l'utilisateur. Une fois accordée, aucun contrôle n'empêche l'application d'envoyer des SMS à des numéros surtaxés sans le consentement de l'utilisateur. Selon [39], 82 % des applications malveillantes nécessitent des permissions d'accès aux SMS. Ce comportement a été observé dans le cheval de Troie *GGTracker* [7] par exemple. Le premier cheval de Troie par SMS découvert par Kaspersky Lab nommé "*Trojan-SMS.AndroidOS.FakePlayer.a*" [40]. Ce cheval de Troie invite l'utilisateur à installer un lecteur multimédia. En parallèle, un service d'envoi de SMS vers des numéros payants coûteux est mis en place sur l'appareil. Ensuite et en arrière-plan, des SMS sont expédiés à ces numéros. Cet API à été également exploité pour envoyer des SMS à des numéros dans le répertoire d'un téléphone cible, impliquant ainsi des pertes financières. DogWars [41] est un exemple d'application Android qui envoie des SMS à tous les contacts enregistrés dans l'appareil sans l'accord de l'utilisateur. De même que l'envoie des SMS, le service de téléphonie peut être un moyen pour appeler des numéros surtaxés sans que l'utilisateur ne s'en rende compte.

Parmi les services systèmes, Android offre des APIs permettant de verrouiller l'appareil et chiffrer des données. Les attaquants ont tendance à appliquer une de ces fonctions sur l'appareil d'une victime et demander ensuite des rançons pour restaurer la fonctionnalité compromise. Les rançongiciels («*ransomware*» en anglais) sont des attaques particulièrement dévastatrices, car elles peuvent détruire les données sensibles des utilisateurs privés et des entreprises. Scalas et al. [42] se basent sur l'extraction statique des APIs système pour la détection des rançongiciels, notamment les méthodes *lockNow()* et *resetPassword()*. Ces appels de l'API système permettent le verrouillage de l'écran et la réinitialisation du mot de passe. Ils appartiennent à la classe *DevicePolicyManager* et au package *android/app/admin*.

Le cryptage des données peut être effectué à l'aide de la méthode API *getInstance()* à partir de la classe *javax/crypto*.

1.5.2 Code natif

Certaines applications légitimes utilisent le code natif principalement pour des raisons de performance. Bien que la littérature confirme qu'environ 4.52% seulement de toutes les applications sur le marché *Google* utilisent du code natif [43], l'exécution de ce code constitue un risque potentiel pour la sécurité des applications Android. En effet, ce code peut être utilisé pour dissimuler du code malveillant. Spreitzenbarth et al. [44] affirment que 13% des applications malveillantes utilisent du code natif où elles cachent souvent leurs fonctionnalités malveillantes. D'autres utilisent le code natif pour lancer des exploits du noyau. Felder et al. [45] affirment que toutes les exploitations locales actuelles du root (les privilèges de l'utilisateur root), permettant de contourner les mesures de sécurité du SE Android, sont exclusivement implémentées en code natif, qui peut être téléchargé dynamiquement et exécuté par n'importe quelle application.

Au niveau de l'API Android, les applications Android peuvent charger des bibliothèques natives à l'aide des méthodes *Systemload()* et *System.loadLibrary()*. De la même manière, les méthodes *Builderstart()* et *Runtimeexec()* peuvent être utilisées pour exécuter du code natif.

1.5.3 Chargement dynamique du code

La conception du système Android permet aux applications de charger du code supplémentaire provenant de sources externes au moment de l'exécution. Comme les fichiers DEX sont limités à une taille de référence de 64K, les développeurs utilisent généralement cette option pour surmonter cette limitation [46].

Cependant, cette caractéristique a été transformée en une vulnérabilité pour Android qui a été exploitée par les attaquants pour ajouter des fonctionnalités malveillantes à l'application. Elle leur permet d'échapper à l'inspection de l'application par le *store* d'applications ou par un moteur antivirus au moment de l'installation. D'autre part, les applications bénignes risquent de subir une injection de code. En effet, le développeur peut involontairement charger un fichier qui a été remplacé par l'attaquant par un autre malveillant, ce qui est probable vu que le SE n'impose pas de contrôles de sécurité sur le code chargé [47]. Une analyse des permissions sur-privilégiées conduite par Aysan et al. [46] montre que certaines permissions dangereuses ne sont demandées que pour être utilisées dans le code téléchargé par des applications malveillantes.

Les méthodes API *DexClassLoader()* et *ClassLoader()* sont utilisées pour effectuer un chargement dynamique de classes.

1.5.4 Réflexion

Tout comme le chargement dynamique des classes, la réflexion est parmi les caractéristiques utilisées pour améliorer les fonctionnalités des applications Android. Le mécanisme de réflexion de Java est largement utilisé dans les applications Android pour inspecter les classes, les interfaces, obtenir leur structure, ou obtenir des informations sur les méthodes et les champs au moment de l'exécution ou de la compilation. Sur les 900 applications analysées par Felt et al. [48], 545 (61 %) utilisent la réflexion Java pour effectuer des appels d'API. En outre, la réflexion peut être utilisée pour instancier des objets, appeler des méthodes ou modifier les valeurs des champs sans même connaître les noms des classes, méthodes, champs, etc. Fattahi et al. [49, 50] affirment que malgré les règles de visibilité communes, l'utilisation du mécanisme de réflexion Java permet d'accéder aux données privées et méthodes privées ou à tout autre objet privé dans une classe Java donnée, et l'utilisent pour inspecter et détecter les vulnérabilités des applications Java. Alhanahnah et al. [34] démontrent que des attaques qui se forment avec deux ou plusieurs applications, dissimulées par réflexion, provoquent des attaques furtives et collusoires, contournant tous les mécanismes de détection existants.

L'API qui offre les services de réflexion est implémenté dans le package *java.lang.reflect*. Les méthodes *ClassforName()* et *getMethod()* sont parmi les principales méthodes utilisées.

1.5.5 Communication inter-composants (CIC)

Android fournit un modèle flexible de communication entre composants via les *intent*. Dans certaines circonstances, le composant d'une application peut envoyer des messages *intents* aux composants d'une autre application pour effectuer des actions (p. ex. prendre une photo, envoyer un message texte, appeler un numéro, etc.). La communication entre différents composants (inter et intra application(s)) se représente, par exemple, sous forme d'un lancement d'une activité avec la méthode API *startActivity()* et *startservice()* pour lancer un service. Dans certains cas, la méthode de l'appelant peut déclencher une communication bidirectionnelle entre les composants, comme par exemple la méthode *startActivityForResult()*. Un composant peut utiliser cette méthode API pour démarrer un autre composant, qui lui-même appelle implicitement le premier composant avec un nouvel *intent* incluant les résultats une fois l'exécution terminée.

Les APIs liés à la CIC sont considérés comme des APIs sensibles vu qu'ils peuvent engendrer des fuites d'information à travers d'un escalade de privilèges. Listing 1.4 et 1.5 illustrent un exemple de l'attaque par escalade de privilèges. Le premier listage présente une application émettrice qui obtient la position de l'appareil (données GPS) et l'envoie par la suite à un autre composant de l'application réceptrice via la messagerie *intent* avec l'API *startservice()* (ligne 8). Le deuxième listage représente une application réceptrice qui récupère ces données GPS et le numéro du téléphone reçus à partir de l'*intent* (lignes 3 et 4) et envoie le message par SMS (ligne 9), avec la méthode API *sendTextMessage* à ce numéro, et ce sans vérifier si la première application ait cette permission. De cette façon, la première application a réussi à envoyer un message contenant des données sensibles par SMS sans

avoir la permission pour effectuer cette action, mais en utilisant une autre application qui en dispose.

```
public class Geolocalisation extends Service {
public void onStartCommand(Intent intent, int flags, int startId){
LocationManager loc = getSystemService(Context.LOCATION_SERVICE);
Location derniere_loc_connue=loc.getLastKnownLocation(LocationManager.GPS_PROVIDER);
Intent intent = new Intent();
intent.setAction("affichier localisation");
intent.putExtra("localisation", derniere_loc_connue.toString());
startService(intent);
}
```

Listing 1.4 – Application émettrice

```
public class EnvoieMsg extends Service {
public void onStartCommand(Intent intent, int flags, int startId) {
String numerotel = intent.getStringExtra("PHONE_NUM");
String message = intent.getStringExtra("TEXT_MSG");
sendTextMessage(number, message);
...}
void sendTextMessage (String num, String msg) {
SmsManager m = SmsManager.getDefault();
m.sendTextMessage(num,null,msg,null,null); }
...}
```

Listing 1.5 – Application réceptrice

1.6 Rétro-ingénierie

Le bytecode Dalvik est le seul format que la plateforme comprend. Cependant, ce code binaire est inintelligible et aride à interpréter. Ce qui fait qu'il est pratiquement impossible de le modifier ou de l'analyser sans le convertir vers un format plus compréhensible.

La rétro-ingénierie rend possible de convertir un fichier binaire lisible par machine en un fichier lisible par l'homme, comme le cas du fichier DEX. Apktool [51] est un outil d'aide à la rétro-ingénierie qui simplifie le processus complet d'assemblage et de désassemblage des applications Android. Il inclut «*smali*» et «*backsmali*» [4], qui permettent le passage depuis et vers le format DEX (voir Figure 1.6). Apktool utilise *backsmali* pour générer un répertoire de fichiers smali, appelé "*smali*", qui sont organisés hiérarchiquement selon leur nom de package. Les fichiers .smali présentent chaque classe créée dans l'application. Ils contiennent un code assembleur appelé «Smali [5]». Ce code n'est autre qu'une traduction du code machine généré par la MVD. C'est une représentation intermédiaire et plus conviviale du bytecode Davlik, qui est proche de la forme originale et qui rend possible l'édition du code d'origine de l'application.

4. Deux termes Islandais qui sont équivalents respectivement à « assembleur » et « désassembleur ».
5. Smali est à la fois le nom d'un langage mnémonique du bytecode Dalvik et sa version assembleur.

De manière caricaturale, considérons l'instruction Java suivante :

$$\text{int } i = 43 ;$$

En supposant que i est la première variable, alors le bytecode Dalvik contiendra probablement une séquence hexadécimale comme :

```
13 00 2B 00
```

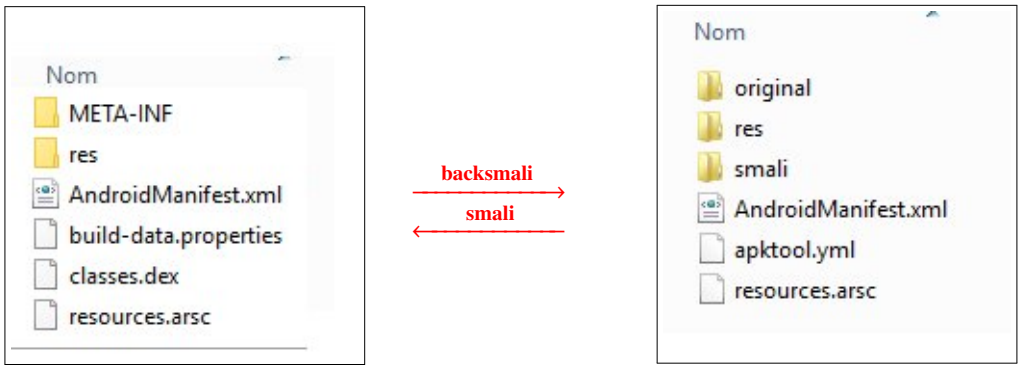
Après la conversion en Smali, on obtient l'instruction suivante :

```
const/16 v0, 43
```

Contrairement à Dalvik, en Smali, on peut comprendre à partir de la même instruction qu'il s'agit d'une constante dont la valeur est 43.

En plus de répertoire *smali*, Apktool génère les ressources de l'application décodées et le fichier *manifest* en version lisible. Figure 1.6 présente le contenu d'une application Android (APK) assemblé et désassemblé via l'outil Apktool. Le fichier "original" contient la signature originale de l'application, ainsi que des données utilisées pour assurer l'intégrité du paquet APK et la sécurité du système.

Le but derrière le recours à la rétro-ingénierie est généralement l'inspection et l'amélioration de la sécurité en recherchant : les principales vulnérabilités, les mauvaises pratiques de codage, les failles de sécurité, les actions malicieuses dissimulées dans des applications [49], etc. Les outils de rétro-ingénierie tels que *dex2jar* [52] et *DED* [53], permettent de décompiler Dalvik en Java ou en bytecode Java. L'un des points les plus importants de la rétro-ingénierie d'une application Android est de savoir où commencer l'analyse et les points d'entrée pour l'exécution du code en sont une partie importante.



Contenu d'un Apk **Contenu d'un Apk désassemblé**

FIGURE 1.6 – Rétro-ingénierie d'une application Android

1.7 Langage Smali

Figure 1.7 représente un exemple de contenu du répertoire *smali* généré par Akptool. *Smali* contient les deux fichiers "android" et" androidx" contenant le code de la bibliothèque de support Android décompilé en smali, et un fichier nommé "com", qui contient du code spécifique à l'application, décompilé en smali aussi. Ce code est réparti dans plusieurs fichiers d'extension smali. En effet, suite au processus de désassemblage, les classes Java internes sont séparées de la classe qui les inclue, chacune dans un nouveau fichier d'extension smali à part, renommé avec un signe $ pour chaque niveau d'imbrication. Autrement dit, *backsmali* crée un fichier .smali pour chaque classe Java de l'application. Par exemple, dans la classe *MainActivity* de l'application originale Java, nous avons défini plusieurs threads. Figure 1.7 représente le résultat suite au désassemblage de cette application. Comme illustrée, Apktool génère plusieurs versions de la classe *MainActivity*. Chacune dans un fichier .smali, renommée avec le signe $ pour chaque niveau d'imbrication (MainActivity$1.smali, MainActivity$2.smali, etc.). Chaque fichier correspond à une sous-classe java/lang/Thread, pour chaque thread défini.

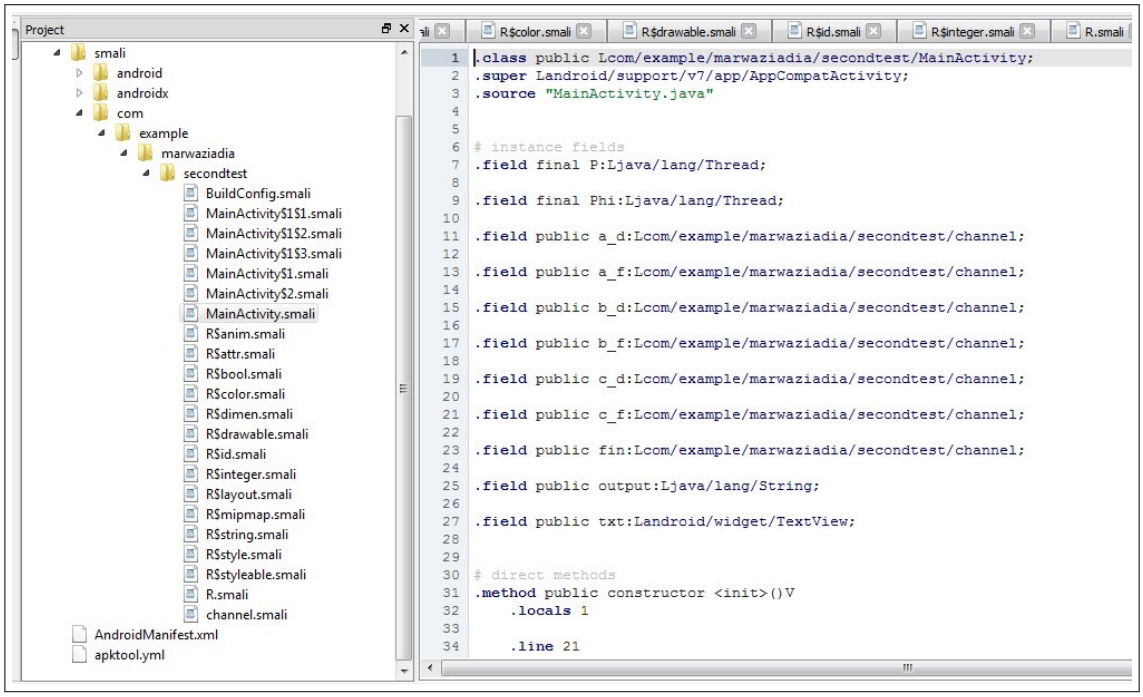


FIGURE 1.7 – Exemple de contenu du fichier *smali*

1.7.1 Structure d'un fichier Smali

La structure d'un fichier .smali est présentée dans Table 1.1. Un fichier .smali débute par le mot clé "*.class*" suivi par les modificateurs d'accès ou la visibilité de la classe, ensuite le nom entièrement qualifié de la classe. Ce nom commence par la lettre "L" suivi par le nom complet de la classe et se termine par ";". Le mot clé "*.super*" spécifie le nom complet de la classe mère. Les chemins complets

des interfaces implémentées sont spécifiées à l'aide de la directive "*.implements*" (si existe). Un fichier Smali indique toujours le nom de classe source Java correspondante identifiée avec la directive "*.source*". Les informations sur la classe sont suivies par les définitions des champs et des méthodes. Les champs peuvent être statiques ou d'instances. Dans le premier cas, le nom de champ est suivi par son type primitif et une valeur (si existe). Dans le deuxième cas, le type de champ correspond au nom complet de la classe à partir de laquelle il est instancié.

.class *modificateurs Lsome/package/Someclass ;*
.super *L*some/package/Someclass ;
.implements *Lsome/package/Someinterface ;*
.source "*someclass.java*"
.field *modificateurs* nom_ champ : type ;
.method *modificateurs* nom_méthode (type_argument,...)type_retour
.registers ... $\#\{p0, p1, \ldots, pn\} \cup \{v0, v1, \ldots, vn\}$
.locals ... $\#\{v0, v1, \ldots, vn\}$
instruction ...
instruction ...
instruction ...
...
.end method

TABLE 1.1 – Structure d'un fichier Smali

1.7.2 Méthodes

Chaque méthode est délimitée par le mot clé "*.method*" et "*.end method*". Le type d'une méthode peut être *direct* pour les méthodes non surchargées, c.-à-d. les constructeurs et les méthodes privées ou finales, *static* pour les méthodes statiques (qui ne sont pas *direct*) et *virtual* pour les méthodes normales qui peuvent être surchargées dans les classes descendantes, y compris celles spécifiées dans les interfaces. Les *modificateurs d'accès* indiquent l'accessibilité et diverses propriétés permettant de gérer ses droits d'accès. Ils comprennent *public*, *private*, *protected*, *final* et *abstract*.

La définition d'une méthode en Smali comprend également la signature de la méthode, composée par le nom de la méthode, les types de ses arguments, et le type de retour. Chaque méthode en Smali détient un ensemble de registres. Ces registres incluent des registres locaux qui stockent ses variables locales et des registres paramètres qui contiennent ses arguments, avec un registre spécial pour sa valeur de retour. Ce dernier permet de transmettre des valeurs de retour de la méthode appelée vers la méthode appelante (peu importe le type de retour *void* ou non).

La MVD se conforme à la convention d'appel de l'Advanced RISC Machines (ARM). Les processeurs ARM sont utilisés en raison de leur architecture simple, ce qui les rend adaptés aux appareils de faible puissance tels que les téléphones portables. Il s'agit d'un schéma de bas niveau, qui dicte la manière dont les sous-programmes reçoivent les paramètres de leur appelant et comment ils renvoient un résultat. En fait, l'appelant et l'appelé partagent une partie de leur tableau de registres de sorte que la méthode appelante transmet des arguments à la méthode appelée en plaçant ses registres paramètres, dans le bon ordre. Dans le cas de l'invocation d'une méthode statique ou de classe, le tableau de registres est défini de sorte que le premier argument de la méthode appelante aboutit au premier registre de paramètres $p1$ de la méthode appelée et ainsi de suite jusqu'au dernier argument. Plus précisément, les n registres arguments mènent aux n registres paramètres, qui sont en effet les n derniers registres du tableau. Lors de l'invocation dynamique d'une méthode d'instance, les registres comprennent un registre supplémentaire qui détient la référence de l'objet sur lequel la méthode est appelée, nommé $p0$ en Smali, et qui représente le premier registre paramètres. Ainsi, le nombre réel de registres paramètres est $p+1$. Dans les deux cas, la syntaxe de l'invocation d'une méthode est le suivant :

$$\textbf{\textit{invoke-type}}\ v1,\ldots,vn\quad \textbf{\textit{nom\_methode}}(type\_argument_1,\ldots,type\_argument_n)\ \textbf{\textit{type\_retour}}$$

La méthode est invoquée avec l'instruction *invoke* suivi par le type de la méthode (*static*, *virtual*, *super*, *direct*). Les registres $v1,\ldots,vn$ correspondent aux arguments de la méthode. Si la méthode est sans paramètres, dans ce cas s'il s'agit de l'invocation d'une méthode statique ce nombre est égale à zéro, mais s'il s'agit d'une méthode non statique (*virtual*), le premier argument correspond toujours à la référence de l'objet sur lequel la méthode est invoquée. Dans le cas d'invocation d'une méthode *super*, le premier argument contiendra la référence de la super-classe. Donc, il faut avoir au moins un registre qui détient la référence de cet objet. Le reste de la syntaxe correspond à la signature de la méthode invoquée.

1.7.3 Registres

Conçue afin de s'exécuter sur des périphériques dont les ressources (mémoire vive, mémoire flash interne et batterie) et la vitesse du processeur sont limitées, l'architecture de la MVD est basée sur les registres. Cette caractéristique la rend plus rapide et efficace pour l'exécution du code d'application. En outre, la MVD utilise les registres pour conserver son état et généralement pour garder une trace d'exécution du bytecode [54]. Les instructions Dalvik agissent sur le contenu des registres plutôt que directement sur des valeurs à l'instar des machines à pile telles que la JVM. Généralement, la machine vituelle basée sur les registres présente des avantages potentiels par rapport à celle basée sur la pile,

L'un des attraits d'une machine à pile est que le code est quasi compact en raison de l'absence d'opérandes explicites. Ainsi, les instructions à registres sont plus longues et généralement plus complexes que les instructions à piles. En contrepartie, pour la même tâche, les machines à base de piles nécessitent plus d'instructions puisqu'elles stipulent des instructions additionnelles pour charger des

données sur la pile et les manipuler. Par exemple, pour la même opération d'addition $a = b + c$, Figure 1.8 montre les instructions correspondantes avec une machine à pile et une machine à registre. Dû à cet écart, la taille du fichier *.dex* est typiquement la moitié de la taille des fichiers *.class* contenant le bytecode Java duquel il est issu. Une comparaison de la taille du code des bibliothèques système, des applications de navigateur Web et d'une application à usage général (application réveil) a été effectuée dans [55]. Dans tous les cas, l'outil dx a réduit la taille du code de plus de 50 %.

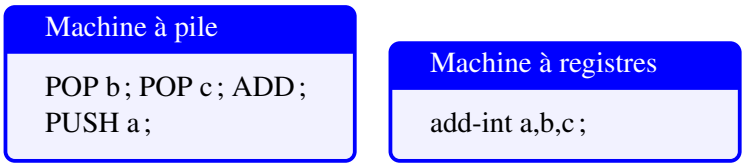


FIGURE 1.8 – Machine à pile vs machine à registres

Comme illustré dans Table 1.1, le corps d'une méthode dans Smali commence toujours par une directive qui indique le nombre de registre alloué par la méthode. Dans Smali, chaque ensemble de registre est désigné différemment, ce qui permet de les distinguer visuellement. La directive *.locals* suivi par un nombre entier indique le nombre de registres locaux utilisés par la méthode. Les registres locaux dans Smali sont désignés par $v0, v1, v2, \ldots, vn$ ($v0$ est le premier registre local, $v1$ le second, jusqu'au dernier registre vn). Les registres paramètres sont dénotés par $p0, p1, p2, \ldots, pn$ et sont stockés dans les derniers registres de la méthode. La directive *.registers* représente le nombre total de registres dans la méthode (c.-à-d. les registres locaux et paramètres).

Les registres sont des mémoires virtuelles. La MVD peut utiliser jusqu'à 256 registres. Chaque registre est de 32 bits. Il peut contenir n'importe quelle valeur, à l'exception des valeurs *double* et *long* qui acquièrent chacune deux registres (64 bits). Les registres dans Dalvik ne sont pas typés et le contenu des registres locaux est initialement indéfini, mais leur nombre est connu statiquement.

1.7.4 Types

Les types dans Smali sont codés d'une manière distinct. Il existe deux classes principales de types, les types primitifs et les types références. La notation des primitifs est particulière, une seule lettre spécifiant chaque type, par exemple, la lettre V désigne le type *void*. Les types par références sont des adresses d'objets ou de tableaux. La représentation de l'objet commence par "L". Le format est *Lnompackage/nomClasse;* où *nompackage* est le nom du package auquel appartient la classe *nomClasse*, tandis que *nomClasse* fait référence au nom de la classe à partir de laquelle l'objet est instanciée. Notez qu'il doit y avoir un point-virgule à la fin. Par exemple, un objet de type thread dans Smali a la syntaxe suivante : *Ljava/lang/Thread;*. Quant au tableau, il est représenté en ajoutant un crochet "[" avant le type de base. Par exemple, un tableau d'entier *int* est représenté par [I. Table 1.2 résume les différents types adoptés par Smali.

Types primitifs	
B	byte
C	char
F	float
I	int
J	long
S	short
V	void
Z	boolean
Types références	
Lnompackage/nomClasse ;	Objet
[... Objets ou Primitifs	Tableau

TABLE 1.2 – Types dans Smali

1.7.5 Instructions

Smali est une syntaxe libre Jasmin/Dedexer [56], qui est un langage similaire au langage d'assemblage dans la syntaxe. Il supporte toutes les fonctionnalités du format dex (annotations, informations de débogage, informations de ligne, etc.). La directive ".line", à titre d'exemple, indique pour chaque instruction Smali la ligne correspondante dans le fichier source Java. Smali contient plus de 200 instructions mnémoniques. Il s'agit d'une forme textuelle qui simplifie la lecture du code. Elle représente une opération, telle que l'addition, le stockage, le chargement. Chaque mnémonique correspond à un nombre entre 0 et 255. Ce nombre est appelé le code d'opération (opcode).

Les paramètres d'une instruction Smali sont pris de la destination vers la source. Par exemple, dans l'instruction d'addition *add-type* $v1,\ v2,\ v3$, les valeurs contenues des registres sources $v1$ et $v2$ sont additionnées et le résultat est stocké dans le registre destination $v1$.

Le nombre de bits dans la description de l'instruction indique la plage de la valeur du registre. Selon la taille et le type du bytecode, certains bytecodes ont un suffixe du nom ajouté pour éliminer toute ambiguïté. Par exemple, le bytecode du type régulier 64 bits ajoute le suffixe $-wide$. Les types spéciaux de bytecodes sont suffixés en fonction du type spécifique, par exemple $add-int$. Ces types peuvent être *boolean*, *byte*, *char*, *short*, *int*, *long*, *float*, *double*, *object*, *string*, *class*, *void*. D'autres suffixes peuvent être ajoutées aux instructions, ils sont séparés du nom principal de l'instruction par la barre oblique "/". Par exemple, dans l'instruction *move-object/from16 vAA, vBBBB*.

- *move* est l'opération ou l'opcode de base ;
- *object* est le type de l'opérande, un registre source portant un objet ;
- *from*16 est le suffixe de l'opcode. Il représente la source d'identification qui est une variable de référence d'un registre de 16 bits ;
- *vAA* est le registre destination ;
- *vBBBB* est le registre source.

La plupart des instructions Smali utilisent des registres comme opérandes de destination ou opérandes source, où :

— $A/B/C/D/E/F/G/H$ représente une valeur de 4 bits qui peut être utilisée pour représenter une valeur de 0 à 15 ou un registre de $v0$ à $v15$;

- $AA/BB/CC/DD/EE/FF/GG/HH$ représente une valeur de 8 bits qui peut être utilisée pour représenter une valeur de 0 à 255 ou un registre de $v0$ à $v255$;

- $AAAA/BBBB/CCCC/DDDD/EEEE/FFFF/GGGG/HHHH$ représente une valeur de 16 bits qui peut être utilisée pour représenter une valeur de 0 à 65535 ou un registre de $v0$ à $v65535$.

Table 1.3 regroupe les instructions Smali en des familles d'instructions selon leurs fonctionnalités. En plus de ces instructions, on trouve le mnémonique de l'instruction d'opération nulle, *nop*. Sa valeur est 00, utilisée généralement pour aligner le code.

Les instructions Smali qui expriment la synchronisation entre les threads sont :

- *monitor-enter* v : pour acquérir le moniteur associé à l'objet référencé dans v ;

- *monitor-exit* v : pour libérer le moniteur de l'objet référencé dans v.

Le reste d'instructions liées à la concurrence sont des appels (c.-à-d. avec les instructions *invoke-type*) aux méthodes API à partir de la classe *java/lang/Thread*. À titre d'exemple, l'instruction Smali permettant de lancer un thread invoque la méthode API *start()* de la classe thread comme suit :

$$\textit{invoke-virtual}\ v0, Ljava/lang/thread; \mapsto start()V$$

1.7.6 Comportement itératif

Le comportement itératif est supporté par Smali avec l'utilisation des étiquettes (ou labels), des sauts et des tests avec les instructions *goto* et *if*. L'idée consiste à marquer le début du corps de la boucle à l'aide d'un label, puis tester avec l'instruction *if* pour juger s'il faut sauter hors de la boucle ou boucler en branchant vers ce label. Par exemple, Listing 1.6 représente un exemple d'un programme Java contenant une boucle *while* et Listing 1.7 représente l'équivalent de ce code en Smali. Pour chaque ligne Java dans Listing 1.6, Listing 1.7 contient les instructions équivalentes en Smali précédées par la directive *.line* suivi du numéro de la ligne Java correspondante. Par exemple, l'équivalent en Smali de l'incrémentation du compteur en Java à la ligne 3 est marqué par la directive *.line 3*. Il s'agit de l'instruction *add-int* qui ajoute 1 tant que la condition (a<=3) est vérifiée. Cette condition est testée avec l'instruction *if-lt*. Le même raisonnement s'applique avec les autres boucles, comme le *for* et le *do...while*.

```
int a = 0;
while(a<=3){
  a++;}
```

Listing 1.6 – Code source Java : *while*

```
.line 1
 const/4 v1, 0x0   //v1=0
.line 2
:label
const/4 v2, 0x3   //v2=3
.line 3
add-int/lit8 v1, v1, 0x1   // v1=v1+1
if-lt v1, v2, :label //si v1<v2, saute vers label
```

Listing 1.7 – Code Smali : *while*

1.8 Conclusion

Ce chapitre représente le premier volet de notre thèse. Il s'agit des applications Android, qui sont notre cible pour le renforcement de la sécurité. Nous jugeons que les détails les plus pertinents à la suite ont été abordés. Étudier le lien entre l'application et le système où elle s'exécute est une étape cruciale pour l'analyse des applications Android. De même, il était essentiel de comprendre l'aspect concurrentiel et itératif afin de pouvoir les modéliser formellement. D'un autre part, l'étude des mécanismes de sécurité adoptés et ses points faibles nous motivera davantage à chercher des solutions permettront de renforcer sa sécurité et combler les lacunes. Par ailleurs, nous nous sommes focalisés sur le langage Smali. Ce code représente le point clé de notre approche, c'est le niveau du code où nous avons choisi d'intervenir pour appliquer les politiques de sécurité.

Le chapitre suivant représentera le deuxième volet lié aux techniques de renforcement de politiques de sécurité.

Description	Famille d'instructions
Déplacement entre registres	*move, move/from16, move-wide, move-wide/from16, move-wide/16, move/16, move-object, move-object/from16, move-object/16*
Retour et obtention du résultat	*move-result, move-result-wide, move-result-object, return-void, return, return-wide, return-object*
Opérations binaires et unaires	*add, sub, mul, div, rem, and, or, xor, shl, shr, ushr, neg-(int, long, float, double), not-(int,long)*
Branchement	*goto, goto/16, packed-switch, sparse-switch, if-eq, if-ne, if-lt, if-ge, if-gt, if-le, if-eqz, if-nez, if-ltz, if-gez, if-gtz, if-lez*
Création d'objet	*new-instance, new-instance/jumbo*
Définition de données	*const/4, const/16, const, const-wide, const/high16, const-wide/16, const-wide/32, const-wide/high16, const-string, const-class*
Manipulation de tableaux	*new-array*, *array-length*, *filled-new-array*, *filled-new- array/-range*, *fill-array-data*
Lecture/écriture d'éléments de tableaux	*aget, aget-wide, aget-object, aget-boolean, aget- byte, aget-char, aget-short, aput, aput-wide, aput-object, aput-boolean, aput- byte, aput-char, aput-short*
Lecture/écriture de champs d'instances	*iget, iget-wide, iget-object, iget-boolean, iget-byte, iget-char, iget-short, iput, iput-wide, iput-object, iput-boolean, iput-byte, iput-char, iput-short*
Lecture/écriture de champs statiques	*sget, sget-wide, sget-object, sget-boolean, sget- byte, sget-char, sget-short, sput, sput-wide, sput-object, sput-boolean, sput- byte, sput-char, sput-short*
Invocation de méthodes	*invoke-virtual, invoke-super, invoke-direct, invoke-static, invoke-interface, invoke-virtual/range, invoke-super/range, invoke- direct/range, invoke-static/range, invoke-interface/range*
Vérification de type	*check-cast, instance-of C, check-cast/jumbo ,instance-of/jumbo*
Gestion des exceptions	*throw, move-exception*
Synchronisation	*monitor-enter, monitor-exit*
Conversion de données	*neg-int, neg-long, not-long, neg-float, neg-double, int-to-float, int-to-double, long-to-int, long-to-float, long-to-double, float-to-int, float-to-long, float-to-double, double-to-int, double-to-long, double-to-float, int-to-byte, int-to-char, int-to-short*
Comparaison	*cmpl-float, cmpg-float, cmpl-double, cmpg-double, cmp-long*

TABLE 1.3 – Instructions Smali

Chapitre 2

Renforcement formel de politiques de sécurité : Notions de base

2.1 Introduction

« Idéalement, les mécanismes de renforcement de la sécurité devraient renforcer la politique qui autorise tous les comportements nécessaires pour accomplir ce que l'utilisateur et les administrateurs du système désirent, et aucun autre comportement », Hamlen [57]. Notre approche est basée sur cette vision. Notre objectif vise à modéliser ce qui a décrit Hamlen avec "désir de l'utilisateur et administrateurs du système" par des politiques de sécurité, qui seront renforcées dans les applications Android, et ce en gardant tout le comportement qui se concorde avec la politique. En d'autres termes, le but est de "pouvoir réaliser ces désirs sans compromettre l'intégrité du système".

Compte tenu de leur diversité et compte tenu de leur omniprésence dans la régulation des processus et de l'utilisation des données dans les systèmes informatiques, il est important de bien comprendre les types de politiques de sécurité, la différence entre politique et propriété de sécurité, la manière dont ces politiques peuvent contraindre un système à respecter ses contraintes, et les mécanismes permettant de les mettre en œuvre.

Le présent chapitre contient les réponses à toutes ces questions. Nous commençons par donner les définitions formelles et informelles d'une politique de sécurité et d'une propriété de sécurité. Basés sur différents travaux, nous décomposons une propriété de sécurité en deux parties : vivacité et sûreté. Puis, nous étudions les différentes techniques de vérification permettant de renforcer une politique de sécurité. Les points forts et faibles de chaque technique seront également discutés. En se basant sur les principales contributions qui ont marqué le champ de recherche en terme de renforcement de politiques de sécurité, nous présentons les principales caractérisations formelles des politiques de sécurité renforçables.

2.2 Notions préliminaires

Cette section rappelle quelques notions de bases de théorie des langages ainsi que les notations adoptées dans la suite de ce document.

- Un système est spécifié par un ensemble d'actions de programme $\mathcal{A}$ et un ensemble d'exécutions possibles Σ ;
- Une exécution σ est une séquence finie d'actions $a_1, a_2, \ldots, a_n$;
- L'ensemble des séquences finies de symboles est dénoté par Σ^* ;
- L'ensemble des séquences infinies de symboles est dénoté par Σ^ω ;
- La concaténation de deux séquences σ et σ' est dénotée par $\sigma\sigma'$;
- La séquence vide est désignée par ϵ ;
- Une séquence σ' est le préfixe d'une autre séquence σ, s'il existe σ'', tel que $\sigma = \sigma'\sigma''$;
- Une séquence σ' est le suffixe de σ, s'il existe σ'', tel que $\sigma = \sigma''\sigma'$;
- $\mathit{Prefx}(\sigma)$ désigne l'ensemble de tous les préfixes de la séquence σ ;
- $\mathit{Sufx}(\sigma)$ désigne l'ensemble de tous les suffixes possibles de la séquence σ.

2.3 Politique de sécurité

Dire que le comportement d'un tel système est acceptable ou non, nécessite de l'évaluer par rapport à une politique de sécurité rigoureusement définie. Cette politique dicte un ensemble de contraintes que le système doit respecter. Ainsi, tout comportement acceptable est considéré comme satisfaisant la politique de sécurité, tandis que tout comportement jugé comme inacceptable par la politique est considéré violateur de cette politique.

Par ailleurs, un comportement peut être acceptable par une politique de sécurité donnée mais pas par une autre. Ceci dit que la définition minutieuse des politiques de sécurité est une notion fondamentale permettant de contraindre un système à "bien" se comporter.

Formellement, comme la définit Bauer et al. [58], une politique de sécurité est définie par un prédicat P sur des ensembles d'exécutions Σ. Nous écrivons $P(\Sigma)$ pour indiquer que Σ satisfait à la politique et $\neg P(\Sigma)$ pour modéliser le contraire. Ainsi, l'ensemble d'exécutions Σ satisfait P si et seulement si $P(\Sigma)$ est vrai. Nous notons $\sigma \in P$ pour désigner une exécution $\sigma \in \Sigma$ qui satisfait P .

Bien qu'elles diffèrent sémantiquement entre plusieurs classes de politiques, les politiques de sécurité visent toutes à garantir des propriétés de sécurité. Les politiques les plus communes peuvent être classées dans les catégories suivantes :

- Contrôle d'accès, visant à limiter les opérations qu'un tiers peut effectuer afin de garantir que seules les entités autorisées peuvent accéder aux ressources d'un système donné ;

- Flot d'informations, permettant de restreindre le flot d'informations de façon à limiter tout ce qui peut être déduit en observant le comportement du système ;
- Disponibilité, garantissant qu'un système est accessible par les personnes autorisées au temps voulu.

Les politiques de sécurité comme celles citées ci-haut sont des politiques à portée générale, c.-à-d. qui peuvent être appliquées à tout système informatique. Ces politiques, comme le confirme Schneider [59], ont fait l'objet de la plus grande attention. Elles contrôlent l'évaluation, l'octroi et la révocation des privilèges d'accès [60], le traçage, la classification des flux potentiels d'informations [61], ou les domaines isolants des utilisateurs et des ressources [62]. Toutefois, afin de protéger des ressources et des fonctionnalités particulières, des politiques de sécurité à granularité fines, spécifiques et adaptées à leur propre domaine d'application sont plus demandées. Un système comme Android, par exemple, requiert des politiques spéciales à plus haut niveau afin de protéger des fonctionnalités typiques et des services sensibles à la sécurité, modifier la configuration de la sécurité et même restreindre des exécutions ultérieures. Ces politiques peuvent viser la plateforme en soi ou ses applications, ou bien les deux ensemble, en régissant l'accès aux méthodes de l'API sensibles de la plateforme par les applications. Ces méthodes comprennent, par exemple, des méthodes de lecture de données personnelles, de création de sockets réseau ou d'accès au matériel d'un appareil comme le GPS ou l'appareil photo, ou tout autre APIs sensibles, comme ceux cités dans section 1.5 du chapitre précédent.

2.4 Propriété de sécurité

En la comparant à une politique de sécurité, une propriété de sécurité est définie par un prédicat qui ne porte pas sur un ensemble d'exécutions, mais sur une exécution individuelle ou trace d'exécution. Les politiques de flots d'informations, par exemple, ne sont pas de propriétés. Alpern et Schneider [63] distinguent les propriétés et les politiques plus générales comme suit. Une politique de sécurité P est considérée comme une propriété lorsque elle a la forme suivante :

$$P(\Sigma) = \forall \sigma \in \Sigma . \hat{P}(\sigma)$$

tel que Σ est l'ensemble d'exécutions et $\hat{P}$ un prédicat sur une exécution individuelle σ.

Par ailleurs, les auteurs démontrent que toute propriété est la conjonction entre des propriétés de sûreté et celles de vivacité.

2.4.1 Propriété de sûreté

Informellement, une propriété de sûreté exprime "*qu'aucun événement indésirable n'aura lieu durant l'exécution du programme*". Ceci signifie qu'une fois une séquence finie viole une propriété de sécurité, nous ne pouvons rien ajouter à cette séquence pour corriger la situation. Tout comme lorsqu'une politique de sécurité exige un mot de passe pour accéder à un compte utilisateur, si le mot de passe

introduit est incorrect (c.-à-d. quelque chose de mauvais se produit), il n'y a rien qu'on peut ajouter à la trace du programme pour satisfaire à nouveau cette propriété. Généralement, le contrôle d'accès définit des propriétés de sûreté. Formellement :

$$\forall \sigma \in \Sigma : \ \sigma \notin P \Rightarrow \ \exists \sigma' \in Prefx(\sigma) : \forall \sigma'' \in \Sigma : \sigma'\sigma'' \notin P$$

2.4.2 Propriété de vivacité

Informellement, une propriété de vivacité exprime le fait que "*quelque chose de bien arrivera certainement durant l'exécution du programme*". En d'autres termes, à partir de n'importe quel état du programme, nous pouvons toujours trouver une solution, de sorte que la propriété soit respectée. La disponibilité, à titre d'exemple, est une propriété de vivacité [58]. Par exemple, si une politique de sécurité exige au programme qu'à chaque fois qu'il réserve une partie de la mémoire de la libérer par la suite. Alors, si sur une séquence d'exécution finie, une partie de la mémoire a été réservée et qu'elle n'a pas été libérée, nous pouvons toujours ajouter une action "libérer mémoire" pour satisfaire la politique.

Une propriété P est une propriété de vivacité si et seulement si pour toute exécution σ, il existe une séquence σ' de σ tel que la séquence $\sigma\sigma'$ satisfait P [64]. En d'autres termes, un préfixe invalide peut toujours être étendue vers une trace satisfaisant la propriété. Formellement, ceci est équivalent à :

$$\forall \sigma \in \Sigma^* : \exists \sigma' \in \Sigma : \sigma\sigma' \in P$$

où $\sigma\sigma'$ désigne la séquence résultant de la concaténation de deux séquences σ et σ' .

2.5 Techniques de vérification

Afin de contraindre un système donné à se conformer à une politique de sécurité, il est primordial d'avoir une technique de vérification permettant de renforcer cette politique dans le système suspect de sorte que tout comportement classé comme acceptable par la politique sera autorisé par la technique de vérification et tout comportement classé comme inacceptable sera inhibé.

La capacité de renforcement d'une politique de sécurité est principalement affectée par deux paramètres : la technique utilisée et les informations disponibles. Une technique peut devenir impossible à appliquer si les données sur lesquelles elle opère sont indisponibles, tout comme elle peut échouer un renforcement même si elle se dispose de toute l'information nécessaire pour accomplir la tâche. Les cibles de renforcement peuvent être des objets, des modules, des processus, des sous-systèmes ou des systèmes entiers [59]. La littérature distingue trois classes de techniques de vérification selon le moment et la manière de leur intervention : l'analyse statique (AS), l'analyse dynamique (AD) et la réécriture de programmes (RW). Choisir la plus appropriée face à certaines exigences de sécurité nécessite la compréhension des capacités et des limites de chacune. Figure 2.1 présente graphiquement ces trois techniques.

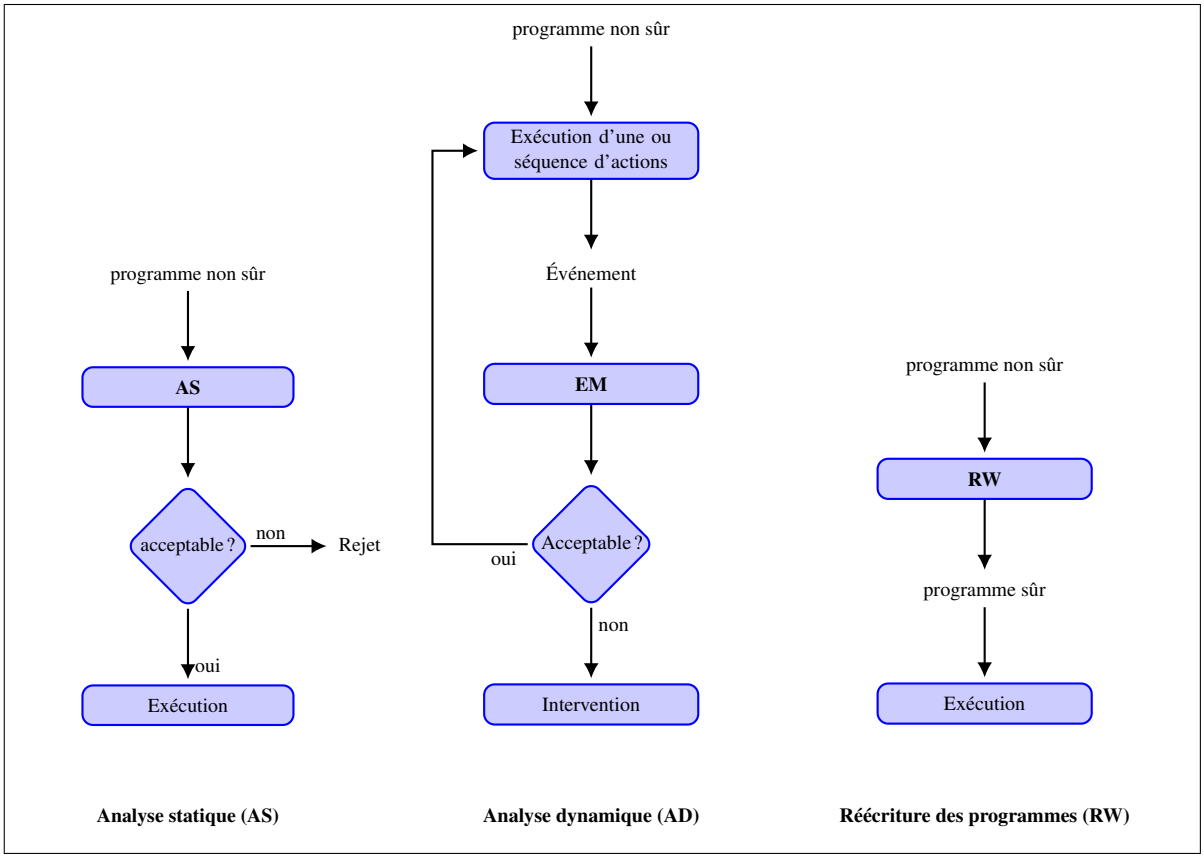


FIGURE 2.1 – Techniques de vérification

2.5.1 Analyse statique

Le mécanisme de renforcement qui intervient avant que le système suspect ne soit exécuté s'appelle analyse statique (AS). Cette technique devrait déterminer, dans un délai fini, que toutes les exécutions possibles du programme en question satisfont aux exigences de la politique de sécurité. Le résultat est comme l'indique Figure 2.1 est binaire, soit une acceptation, et par la suite l'autorisation à l'exécution sans entrave, soit le rejet, dans ce cas le programme ne sera jamais exécuté. D'ailleurs, une seule exécution qui viole la politique de sécurité est suffisante pour rejeter le programme en entier. Cette technique opère généralement sur un modèle qui reflète une abstraction du comportement du système en question. La technique de l'interprétation abstraite [65] est souvent utilisée pour approximer le système par un modèle approprié. Ce modèle sera par la suite interrogé via des outils automatiques ou semi-automatiques afin de comprendre et prédire le comportement du système au cours de son exécution. Les techniques de typage (ou système de type) [66], la vérification par évaluation de modèle (*Model Checking* en anglais) [67] et les analyses orientées flots [68] sont aussi des exemples des méthodes d'analyses statiques les plus utilisées.

La vérification par évaluation de modèle, à titre d'exemple, vise généralement à vérifier si un modèle

d'un système donné satisfait à sa spécification. Généralement, elle implique trois étapes majeures :

1. Construire un modèle fini pour le système à vérifier ;
2. Spécifier la propriétaire de sécurité à renforcer en utilisant un langage formel (souvent la logique temporelle) ;
3. Vérifier si la propriété de sécurité est vraie dans le modèle. La vérification est généralement faite de manière automatique en explorant tous les chemins d'exécutions possibles du modèle.

Le résultat de cette analyse est vrai ou faux, c.-à-d. soit que la propriété est vérifiée par le modèle ou non.

La disposition d'informations sur l'exécution future, les exécutions alternatives ou sur toutes les exécutions possibles du cible fait de l'AS une technique de vérification concluante [59]. Ceci est dû au fait qu'elle repère les problèmes plus tôt sans influencer l'exécution du système. Par la suite, la crainte du dommage est écartée. Ceci est très utile pour certains systèmes critiques (centrales nucléaires, systèmes de contrôle des avions, etc.), qui devraient être analysés préalablement afin d'éviter des conséquences désastreuses qui peuvent être causées suite à leurs exécutions.

Par ailleurs, ce type d'analyse assure généralement une bonne couverture du code et peut rassembler toutes les informations dont le mécanisme d'exécution ne dispose pas initialement, qui incluent même les entrées des utilisateurs et les sorties de choix non déterministes, et ce malgré la possibilité d'explosion de nombre d'états présentant toutes les exécutions possibles. Toutefois, ces informations ne peuvent être tirées et déduites qu'avant l'exécution, ce qui fait que les politiques de sécurité qui stipulent l'observation des étapes de l'exécution d'une cible ne peuvent pas être renforcées par cette approche. Même si l'AS anticipe et essaie d'inférer la présence d'un comportement malicieux, ces approximations conservatrices mènent généralement à un certain nombre de faux positifs. Par exemple, la réflexion constitue un défi majeur pour l'AS. Malgré la possibilité de supposer qu'un appel par réflexion pourrait invoquer n'importe quelle méthode arbitraire, une telle hypothèse augmente la probabilité de faux positifs [69].

Cependant, la disponibilité de ce type de données ne représente pas un obstacle pour d'autres techniques telles que les analyses dynamiques.

Les politiques de sécurité qui peuvent être statiquement renforcées sont appelées *statiquement renforçables*.

2.5.2 Analyse dynamique

L'analyse dynamique (AD), appelée aussi monitorage permet d'observer en temps réel le comportement d'un système. Elle implique ainsi l'exécution d'un échantillon dans un environnement contrôlé et isolé afin d'analyser ses traces d'exécution. Les mécanismes d'exécution qui opèrent à côté d'un programme non fiable sont appelés moniteurs d'exécution (EM). Schneider [59] a commencé à étudier les types de politiques de sécurité qui peuvent être renforcées de cette manière. Il définit le moniteur

d'exécution (EM) comme un programme qui s'exécute en parallèle avec le programme cible et qui observe les actions juste avant qu'elles ne soient exécutées. Ce moniteur intervient dès qu'il constate un événement critique qui entraînerait une violation de la politique à instaurer. La procédure d'intervention dépend du moniteur d'exécution. Elle peut consister en l'arrêt prématuré de l'exécution de la cible à chaque fois qu'elle tente de violer la propriété à assurer. Les EMs qui procèdent de cette façon s'appellent les moniteurs d'exécution conventionnels (CEM). Ils existent des moniteurs d'exécution qui sont plus puissants que les moniteurs conventionnels, qui ne se limite pas à tronquer l'exécution du programme, mais qui sont capables d'ajouter des actions correctives qui font en sorte que la propriété à renforcer soit respectée ou supprimer les exécutions qui violent les exigences dictées par la propriété de sécurité. Les EMS qui effectuent ce type de procédure s'appellent des moniteurs d'exécution basés sur la réécriture (RWEM).

Toutes les informations relatives à l'historique de l'exécutions ou celles permettront de prédire les futures étapes que la cible pourrait suivre sont exclues du pouvoir de EM. Ce dernier ne peut surveiller que les informations disponibles en observant les étapes de l'exécution d'une cible [59].

Contrairement à leurs homologues statiques, les approches dynamiques sont plus précises, étant donné qu'elles connaissent l'état courant du programme. Selon Hamlen et al. [70], le monitorage permet de renforcer plus de propriétés de sécurité que les approches statiques.

En contrepartie, les analyses dynamiques ne peuvent pas assurer la sûreté d'un programme. Comme expliqué, les informations qui y sont fournies sont insuffisantes pour couvrir toutes les exécutions possibles de la cible. Plus clairement, une telle analyse ne peut considérer qu'une seule exécution ou un sous-ensemble d'exécution (échantillon d'exécution). Ce qui fait qu'il est possible d'observer un comportement malveillant seulement s'il se produit sur le chemin d'exécution actuel, sinon il serait impossible de le capturer. En outre, l'AD requiert plus de ressources système que l'AS. Comme le montre Figure 2.1, le moniteur doit capturer chaque événement critique, l'analyser et réagir en réponse. Ceci peut entrainer des ralentissements importants et parfois intolérables sur le système surveillé.

Les politiques de sécurité qui peuvent être renforcées par monitorage d'exécution sont dites *EM-renforçables*.

2.5.3 Réécriture de programmes

Une approche alternative aux deux approches précédentes est la réécriture de programmes (RW). Cette technique vise à réécrire le programme statiquement, en un temps fini, en fonction d'une propriété de sécurité donnée, afin d'en générer une nouvelle version exécutable qui la satisfait.

Les modifications ou les tests de sécurité sont ajoutés à des endroits bien calculés dans le programme afin de l'empêcher de violer la politique de sécurité une fois exécuté. Par conséquent, le code non fiable est transformé en un code auto-surveillé. De cette façon, à la place d'avoir un moniteur qui surveille

le système dynamiquement à l'instar de EM, ce moniteur est éclaté en un nombre de tests insérés aux bonnes positions dans le programme surveillé. Ce qui requiert moins de ressources système et permet de gagner les coûts relatifs aux tests de moniteur, incluant ceux liés à la capture des événements critiques en terme de sécurité, l'analyse de trace d'exécution du programme et la mise en œuvre de la procédure d'intervention. La version réécrite devrait être équivalente mais plus restrictive que l'originale, de sorte qu'elle sera en mesure d'éviter toutes opérations potentiellement dangereuses avant qu'elles ne se produisent.

Afin de répondre à la définition de renforcement de politique de sécurité, une réécriture de programme doit satisfaire deux exigences :

1. *La correction* : une technique de réécriture de programme est correcte par rapport à une politique de sécurité si elle produit toujours un code qui satisfait cette politique ;
2. *La complétude :* une technique de réécriture de programmes est complète par rapport à une politique de sécurité si elle conserve le comportement du code qui satisfait déjà cette politique.

La satisfaction de ces deux propriétés par un mécanisme de sécurité basé sur la réécriture est conditionnée obligatoirement par le fait de garder le programme dans un état cohérent vis-à-vis de sa sémantique de départ.

Erlingsson and Schneider [71] définissent un mécanisme de réécriture de programmes qui consiste à intégrer un moniteur d'exécution au sein d'un programme. L'approche est appelée "monitorage d'exécution en ligne" (IRM, en anglais « *Inlined Reference Monitor* »). Les systèmes IRM intègrent un composant appelé *rewriter* ou *inliner*, qui prend en entrée l'application et une politique de sécurité et injecte des contrôles de sécurité supplémentaires à des points critiques du programme. C'est la spécification de la politique de sécurité qui guide cette réécriture, identifie ces points et définit quel état de sécurité est ajouté à l'application. Le code de renforcement de l'IRM englobe à la fois l'état de sécurité du moniteur et les mises à jour de sécurité nécessaires. La sortie est un programme fiable, prêt à être exécuté plusieurs fois sans être réécrit, tout en garantissant que l'exécution ne violera pas la politique de sécurité.

Hamlen [57] considère l'IRM comme un EM embarqué dans le corps du programme à surveiller et qui injecte ses contrôles de sécurité dynamiques dans des endroits bien déterminés. Afin de mettre en évidence la différence entre les deux, l'auteur suppose qu'un EM et un IRM renforcent la même politique de sécurité, qui interdit aux programmes d'effectuer des opérations qui accèdent à des emplacements mémoire en dehors de son espace d'adressage. L'EM renforce cette politique en surveillant chaque accès à la mémoire et en signalant une interruption si l'emplacement accédé se trouve en dehors des limites de l'espace d'adressage du processus. En revanche, un IRM renforce cette politique en insérant des contrôles dans le code non fiable juste avant chaque instruction du programme qui accède à la mémoire. Le résultat est un arrêt avant tout accès illégal à la mémoire.

Les implémentations IRMs présentent souvent plusieurs avantages par rapport aux EMs. Elles peuvent souvent éviter d'effectuer des contrôles de sécurité inutiles. En plus, le code d'un IRM se trouve dans

l'espace d'adressage du code non fiable, par conséquent il peut souvent tirer profit des informations qui ne sont pas facilement atteignables par l'EM, puisqu'il se trouve séparé du code non fiable. En plus, le monitorage d'exécution entraîne des surcoûts considérables que ce soit en temps machine ou en espace mémoire. La réécriture de programmes, en revanche, permet d'éviter tout le temps qu'un EM met pour décider si l'exécution risque de violer la politique ou non vu qu'elle la transforme directement en une exécution valide.

Le concept d'IRM a reçu une attention accrue dans la littérature. Il a été formalisé pour la première fois dans le développement des systèmes SASI/PoET/PSLang [71, 72]. Les auteurs mettent en œuvre des IRMs pour le code d'assemblage x86 et le bytecode Java. Plusieurs autres implémentations IRM pour Java ont suivi.

En somme, la combinaison de deux mécanismes statiques et dynamiques a fait de cette approche une solution assez pratique et puissante, héritant les avantages de deux approches : une durée et un coût d'application minimisés grâce à l'approche statique associée à la permissivité des mécanismes dynamiques. Hamlen et al. [70] ont démontré que les techniques de réécriture sont susceptibles de renforcer certaines politiques de sécurité dont les approches statiques et dynamiques s'en trouvent incapables.

Les politiques de sécurité qui peuvent être renforcées par réécriture de programmes sont appelées *RW-renforçables*.

2.6 Caractérisation des politiques de sécurité renforçables

2.6.1 Automate de sécurité

Une méthode majeure pour étudier les propriétés de sécurité consiste à les représenter sous forme d'automates de sécurité. Schneider [59] s'est approfondi dans l'étude de la caractérisation formelle des politiques de sécurité renforçables par monitorage d'exécution. Il a défini un nouveau modèle pour la spécification des politiques de sécurité *EM-renforçables* qui l'a appelé *automate de sécurité*. Formellement, un automate de sécurité est défini comme suit :

Définition 2.6.1. *(Automate de sécurité) [59]*
Un automate de sécurité est un quadruplet $\langle A, Q, Q_0, \delta \rangle$*, où*

- A *: un ensemble dénombrables d'actions (ou symboles d'entrée) ;*
- Q *: un ensemble dénombrable d'états ;*
- Q_0 *: un ensemble dénombrable d'états initiaux, tel que* $Q_0 \subseteq Q$ *;*
- $\delta : Q \times A \mapsto Q$ *: une fonction de transition.*

Pour traiter une séquence $a_1 a_2, \ldots$ de symboles d'entrée (ou actions), à partir d'un état actuel Q, l'automate de sécurité commence par un état initial égale à Q_0 en lisant la séquence une action d'entrée à la fois. L'état de l'automate change en fonction de chaque transition prise.

Pour chaque symbole d'entrée a_i lu, l'automate de sécurité change Q en :

$$\bigcup_{q \in Q} \delta(q, a_i)$$

Si Q est égal à l'ensemble vide, ceci signifie que l'entrée a été acceptée, sinon l'entrée sera rejetée. Cette définition de l'acceptation est suffisamment large pour couvrir les séquences infinies et finies.

Intuitivement, afin de contraindre un programme donné à respecter une politique de sécurité, un automate de sécurité procède en vérifiant que l'exécution de chaque action ne mène pas vers un état *erreur* (c.-à.d. ne viole pas la politique de sécurité), et ce, avant son exécution. Si c'est le cas, l'automate met fin à l'exécution du programme. Dans le cas contraire, l'exécution de l'action sera autorisée en effectuant une transition vers un nouvel état. La transition $\delta(q, a)$ n'est définie que lorsque le EM est supposé accepter l'action a étant dans l'état q.

Ligatti et al. [73] trouvent que ce mécanisme est à la fois général et limité. En effet, les automates de sécurité sont généralement des applicateurs de propriétés plutôt que de politiques générales. Cependant, même pour les propriétés, ils se trouvent incapables à exprimer des propriétés de vivacité telles que la disponibilité. Cela revient d'une part à l'incapacité des EMs, qui ne sont en mesure de voir que des séquences d'actions individuelles. Par conséquent, ils ne peuvent pas baser leur décision sur d'autres exécutions de l'application cible et par la suite faire respecter certaines propriétés qui dépendent de l'historique complet de l'exécution du programme. Il peut y avoir certaines données comme les mots de passe, les clés secrètes que le moniteur ne peut pas synthétiser efficacement. D'une autre part, même les violations qu'il peut détecter, le EM n'est pas doté d'un grand pouvoir qui le permet de les manipuler efficacement (insérer, supprimer, etc.).

2.6.2 Automate d'édition

Les résultats de Schneider sur l'applicabilité des politiques de sécurité ont stimulé diverses recherches, tant pratiques que théoriques, sur le développement et l'analyse des mécanismes d'exécution.

Jay Ligatti, Lujo Bauer, and David Walker [73–75] introduisent une hiérarchie d'automates pour la modélisation des moniteurs de programmes baptisés automates d'édition (EA). Ce sont des machines abstraites qui examinent la séquence des actions du programme et transforment celles qui dévient d'une politique spécifiée. Les EAs ont été créés essentiellement dans le but de surmonter les limites des EMs représentés par les automates de sécurité, qui ne font qu'arrêter le système en cas de problèmes. Les auteurs considèrent que les moniteurs peuvent agir comme des "transformateurs" qui modifient le flux d'actions produit par une application non fiable plutôt que des "reconnaissants" de séquences à l'instar des automates de Schneider.

L'utilisation des EAs donne ainsi la possibilité au moniteur d'éditer le flux d'exécution (les actions de systèmes) et par la suite renforcer plus des propriétés de sécurité. Ils disposent d'un riche ensemble de pouvoirs de transformation : ils peuvent mettre fin à l'application, tronquant ainsi le flux d'actions

du programme ; ils peuvent supprimer des actions indésirables ou dangereuses sans nécessairement mettre fin au programme ; et ils peuvent également insérer des actions supplémentaires dans le flux d'événements. En plus de pouvoir renforcer toutes les propriétés captant des séquences finies, les EAs sont en mesure d'aller chercher des propriétés captant des séquences infinies.

Afin d'appliquer ce modèle de EMs, Ligatti et al. définissent quatre types d'automates :

- Automate de troncation : ce type d'automate a les mêmes capacités que les automates de sécurité originaux de Schneider, c.-à-d. reconnaître les "mauvaises" séquences d'actions et tronquer l'exécution du programme avant que la politique de sécurité ne soit violée ;
- Automate de suppression : en plus de pouvoir arrêter l'exécution du programme, cet automate est en mesure de supprimer des actions individuelles du programme ;
- Automate d'insertion : cet automate est capable d'insérer une séquence d'actions dans le flux d'actions du programme ou de mettre fin au programme ;
- Automate d'édition : ce type d'automate combine les pouvoirs des automates cités. Il est capable ainsi de tronquer des séquences d'actions et d'insérer ou de supprimer des actions pertinentes pour la sécurité.

Formellement, un automate d'édition est défini comme suit :

Définition 2.6.2. *(Automate d'édition)*
Un automate d'édition est un quadruplet $\langle A, Q, Q_0, \delta \rangle$, tel que :

— *A : un ensemble dénombrable d'actions (ou symboles d'entrée) ;*

— *Q : un ensemble dénombrable d'états ;*

— *Q_0 : un ensemble dénombrable d'états initiaux, tel que $Q_0 \subseteq Q$;*

— *$\delta : Q \times A \mapsto Q$: une fonction de transition.*

À partir d'un état courant q et une action d'entrée a, la fonction de transition δ est définie comme suit :

$$\delta(q,a) = \begin{cases} (q', a) & (acceptation) \\ (q', \epsilon) & (suppression) \\ (q', \sigma) \quad où \quad \sigma \neq a & (insertion) \\ Indéfini, sinon & (interruption) \end{cases}$$

La fonction de transition de l'automate d'édition spécifie un nouvel état à atteindre et une séquence d'actions à exécuter. Ces deux informations définissent le type d'intervention à effectuer afin de renforcer une propriété. Changer d'état et conserver la même action d'entrée signifie une acceptation de cette action. Une nouvelle action différente de celle d'entrée représente une insertion. Une action vide représente la suppression de l'action d'entrée. Le résultat de la transition est indéfini lorsqu'il s'agit de l'arrêt de l'exécution.

Malgré sa puissance, cette technique entraine des ralentissements importants et parfois intolérables sur le système surveillé. En outre, Ould-Slimane et al. [76] soulignent le pourvoir de la caractérisation de politiques de sécurité renforçables par réécriture de programmes à surmonter les problèmes qui peuvent apparaître en utilisant les EMs défini par Ligatti et al. En effet, l'efficacité des automates d'édition dans le renforcement de la sécurité est due à leur capacité à simuler des actions potentiellement dangereuses. Cependant, ceci entraîne généralement un comportement du programme en désaccord avec sa spécification puisque les effets des actions simulées ne sont pas reflétés dans les états du programme. Contrairement aux automates d'édition, la réécriture de programmes modifie les programmes plutôt que les exécutions. Étant donné un programme non fiable et un automate d'édition appliquant une propriété, les auteurs définissent un opérateur d'injection qui génère une nouvelle version qui satisfait la propriété. Les programmes et les automates d'édition renforçant les propriétés de sécurité sont spécifiés par des machines à états finis étendus (EFSM).

Dans le même axe que les travaux de recherche conduits par Schneider, Ligatti et.al, Talhi et al, [77] proposent une caractérisation des politiques de sécurité qui peuvent être renforcées par des EMs limités par la mémoire. Pour représenter les moniteurs d'exécution contraints par des limitations de mémoire, les auteurs introduisent une nouvelle classe d'automates appelés automates bornés par l'historique (en anglais « *Bounded History Automata* »). Ces automates ont été défini en appliquant une instance de l'abstraction de Fong [78], pour gérer les limitations de mémoire aux automates de sécurité et aux automates d'édition. Les automates de Büchi, présentés pour la première fois par Eilenberg [79], ont été également utilisés pour spécifier des propriétés de sécurité. Alpern et Schneider [63] les utilisent pour séparer les propriétés, qui ne sont ni de vivacité ni de sûreté, en une partie vivacité et une partie sûreté.

2.7 Modes de renforcement

Selon Ligatti et al. [73], tout mécanisme de sécurité qui permet d'observer des exécutions qui ne satisfont pas une politique donnée est un mécanisme non fondé et inadéquat. Pour cette raison, ils considèrent que ces mécanismes ne peuvent accomplir leur tâche efficacement que lorsqu'ils se conforment aux deux principes suivants :

- *Correction* assurant que toute exécution observable doit obéir à la propriété renforcée ;
- *Transparence* garantissant la préservation de la sémantique de toute exécution satisfaisant la propriété.

L'EA assure la correction en transformant les mauvaises exécutions en exécutions satisfaisant la politique, et garantit la transparence en transformant les exécutions valides en exécutions valides équivalentes.

Renforcement conservateur Si une technique de vérification peut satisfaire l'exigence de correction, mais pas nécessairement celui de transparence, alors on dit qu'elle renforce la propriété de manière conservatrice. Le renforcement conservateur donne à l'automate une grande flexibilité et liberté

lors du renforcement. Le plus important pour ce mode de renforcement est de générer une exécution qui satisfait la politique, peu importe le coût sur le comportement du programme, en particulier les exécutions des actions satisfaisant la politique. Ce mode de renforcement est capable de faire respecter n'importe quelle propriété, mais souvent de manière peu efficace.

Renforcement précis Afin d'être précis sur ce que signifie préserver la sémantique de l'exécution d'un programme, il fallait respecter le critère de transparence. Un automate renforce d'une façon précise une propriété si et seulement s'il renforce de manière conservatrice la propriété et produit des actions cohérentes avec le programme cible. L'automate dans ce cas ne doit en aucun cas interférer avec l'exécution du programme lorsque l'entrée satisfait la propriété. Plus clairement, chaque action d'une séquence d'entrée valide doit être acceptée sans interruption ou modification, avant que l'action suivante ne soit considérée.

Renforcement effectif Réunir les deux critères de correction et transparence permettra de renforcer des propriétés de manière effective. Ce mode de renforcement est très puissant comparé aux autres modes sus-cités. Il est en mesure de distinguer les actions qui sont syntaxiquement différentes mais sémantiquement équivalentes, ce qui n'est pas le cas pour le mode de renforcement précis. Par exemple, fermer un fichier deux fois de suite équivaut à le fermer une seule fois. De même, il existe des macro-instructions qui regroupent plusieurs instructions mais qui ont la même fonctionnalité, surtout avec les langages de type assembleur. Par exemple, une action qui permet d'ouvrir un socket suivie d'une action qui envoie des données sur le socket est sémantiquement équivalente à une macro-instruction qui effectue les deux.

2.8 Conclusion

Ce chapitre a présenté les principaux enjeux ayant trait au renforcement de politiques de sécurité. Les techniques de vérification présentées apportent des pistes intéressantes afin de renforcer la sécurité des applications Android. Ce chapitre a démontré qu'il serait pertinent d'explorer un mécanisme de renforcement qui permet de bénéficier des avantages de deux méthodes statiques et dynamiques et éviter leurs inconvénients. La technique de réécriture des programmes semble être la meilleure solution pour le renforcement de politiques de sécurité dans des applications Android. Cette approche a été relativement peu étudiée sur ce SE mobile en général malgré son importance et efficacité lors de son application sur des systèmes séquentiels ([76, 80–82]) et concurrents ([13, 83–85]), en général.

Le prochain chapitre focalisera sur Android et recense l'état de l'art des différentes techniques de vérification appliquées sur ce système ainsi que leurs cibles d'intérêt.

Chapitre 3

Différentes approches de renforcement de politiques de sécurité dans le système Android

3.1 Introduction

La littérature renferme plusieurs travaux prometteurs utilisant des méthodes formelles pour renforcer automatiquement des politiques de sécurité dans des systèmes séquentiels et concurrents. Parallèlement, plusieurs études ont montré que le renforcement formel de politique de sécurité est hautement nécessaire pour le système Android afin de contrer les limites de sécurité de ce système d'exploitation.

Le présent chapitre connecte les deux premiers chapitres. Il représente l'application de trois mécanismes de renforcement de politique de sécurité, discutés dans le deuxième chapitre, sur le système Android détaillé dans le premier chapitre. Pour chaque technique de vérification, on s'appuie sur les principales contributions recensées par l'état de l'art pour discuter ses cibles d'intérêt, les types de données sur lesquelles elle peut raisonner, les propriétés qui peuvent être renforcées sur ces données, ainsi que les points forts et les limites de chacune, le tout vis à vis d'Android.

3.2 Analyse statique

3.2.1 Acquisition

Les analyses statiques appliquées à Android portent principalement sur les éléments qui sont atteignables sans l'exécution de l'application. Ce type de données se trouve dans l'ensemble du contenu d'une application (le contenu de l'APK).

L'inspection du code source récupéré en désassemblant l'application est parmi les ressources utilisées par l'approche statique. D'ailleurs, le point fort de l'AS est que l'entièreté du code est analysé, autre-

ment dit la couverture du code. Cela diffère de l'analyse dynamique où des parties du code ne peuvent être exécutées que dans des conditions spécifiques, qui ne peuvent jamais être connues pendant la phase d'analyse. Cela permet d'assurer non seulement une certaine modularité lorsqu'il s'agit de cibler des milliers d'applications, mais aussi de garantir une exploration de tous les chemins d'exécution possibles. La couverture du code englobe le code à n'importe quel niveau (haut niveau, niveau intermédiaire et bas niveau) et même le code natif comme proposé par Lantz et al. [86]. Quelque soit le niveau, le code peut être obtenu par le moyen des outils de rétro-ingénierie spécialisés. Ainsi, le code analysé peut être le code source Java [87–89] ou le byte code Java ou le byte code Dalvik. Julia [90] est un analyseur statique du bytecode Java. Il a été étendu dans [91] et adapté pour inclure le support du bytecode Dalvik et traiter des caractéristiques spécifiques d'Android telles que la nature événementielle, les points d'entrée potentiellement concurrents. Il applique plusieurs approches statiques pour l'analyse des classes, de la nullité, du code mort et de la terminaison des applications Android. Wognsen et al. [92] proposent une analyse de flux de contrôle au niveau du bytecode Dalvik. Le but est de détecter les applications malveillantes ou d'identifier les appels API sensibles invoqués pendant l'exécution. SCANDAL [93] est un analyseur statique qui analyse le code Smali des applications et détecte les fuites d'informations.

Les fichiers qui accompagnent le code de l'application peuvent également former des candidats potentiels pour l'AS. Ils incluent le fichier *manifest*, notamment les différentes informations qu'il détient, telles que les permissions de l'application, les points d'entrée et la communication inter-composants (CIC) [94–96]. De même, les images, les fichiers XML de configuration et les APIs liés au cycle de vie de l'application sont des exemples d'informations qui se trouvent dans les ressources et qui peuvent servir pour une AS [97, 98].

Analyser ces informations statiquement permet de rechercher et déceler des modèles de comportements malveillants ou des erreurs de programmation typiques dans les applications (erreurs syntaxiques et bogues sémantiques) ; de vérifier les séquences d'instructions et la façon dont les valeurs des variables sont traitées au cours des différents appels de fonctions ; de suivre les emplacements dans l'application où les données sensibles peuvent se propager afin de déterminer si l'application risque de divulguer ces données ; de détecter une utilisation d'une permission non autorisée ou des privilèges excessifs. Par conséquent, des politiques de sécurité qui régissent ce genre de comportement peuvent être renforcées statiquement. L'outil ComDroid est décrit comme un outil qui effectue une "analyse statique intra-procédurale sensible au flux" des programmes en bytecode Dalvik. Il est conçu pour analyser la communication entre les applications Android par le biais des *intents*. Le même outil ComDroid est utilisé comme composant d'un autre outil d'analyse appelé Stowaway [48], qui analyse les appels d'API pour déterminer si elles sont sur-privilégiées. ComDroid et Stowaway sont tous les deux des outils d'analyse sophistiqués qui couvrent non seulement le bytecode Dalvik mais aussi des parties importantes de l'API et de la plateforme Android elle-même. Cependant, comme les analyses ne sont pas réellement spécifiées en détail, ni formellement ni informellement, il est impossible d'évaluer leurs points forts et faibles exacts. En effet, il est indiqué dans [48] que Stowaway "éprouve des

difficultés avec les flux de contrôle non linéaires". Ceci souligne le besoin de formaliser à la fois le langage de bytecode Dalvik et l'analyse du flux de contrôle. Dans [99, 100], les auteurs renforcent les politiques de flot d'informations liées aux permissions Android, produisant ainsi un système qui respecte la propriété de non-interférence [1].

3.2.2 Limitations

Parmi les obstacles courants à la conception et à la mise en œuvre des analyses statiques, nous citons :

- *Le recours aux outils d'aide à la rétro-ingénierie*. En pratique, un analyseur statique doit être capable de s'attaquer directement au bytecode Dalvik, mais vu que ce code est ardu à interpréter, il fallait le traduire dans un format plus convivial à l'analyse comme Smali. L'analyse statique du code source Java requiert par contre la conversion de Dalvik vers Java, la compilation en bytecode Java, puis sa reconversion vers Dalvik. Bien qu'ils existent des outils d'aide à la rétro-ingénierie, tels que *dex2jar* [52] et *ded* [53] qui permettent d'effectuer récupérer ces codes, certaines informations du bytecode Java peuvent être perdues. Ces outils doivent déduire donc les détails manquants en se basant sur le contexte environnant, mais cette reconstitution est parfois erronée. Davis et al. [33] ont démontré que même si ces erreurs n'empêchent pas l'analyse statique du bytecode Java converti, elles conduisent souvent à un bytecode Java invalide ou, plus tard, à un bytecode Dalvik invalide. Les auteurs affirment que réécrire directement le bytecode Dalvik permet d'éviter ces problèmes. Jean et al. [101] confirment qu'il n'existe aucun outil de rétro-ingénierie Dalvik-Java qui est 100% robuste et correct. Ceci s'applique aussi au code source Java, il n'y a aucune garantie de récupérer le code source d'origine, mais une reconstitution du code telle qu'interprété par l'outil. D'un autre côté, il se trouve que la plupart des analyses statiques du bytecode Java, qui auraient pu être mis à profit, ne peuvent pas être généralisées pour Android [102]. À titre d'exemple, l'outil FindBugs [103], qui a démontré ses capacités à découvrir des bogues dans le bytecode Java, ne peut pas être exploité facilement pour les programmes Android, une étape supplémentaire est nécessaire pour transformer les APKs Android en JARs.

- *L'absence d'un point d'entrée principal*. Comme nous l'avons expliqué dans le premier chapitre, une application Android ne dispose pas d'un seul point d'entrée. Par conséquent, il est difficile de tous les cerner. Par exemple, pour construire un graphe d'appel global de l'application, un analyseur statique se trouve obligé de rechercher tous les points d'entrée et construire plusieurs graphes d'appels sans aucune garantie sur la façon dont ces graphes peuvent se connecter les uns aux autres.

- *Le cycle de vie des composants*. Dans Android, les différents composants d'une application ont leur propre cycle de vie. Chaque composant implémente ses méthodes qui sont appelées pour le démarrer/arrêter/reprendre selon les besoins de l'environnement. Comme ces méthodes ne sont

1. La non-interférence est exprimée par l'exigence qu'aucune information sur les données privées ne soit divulguée ou déduites en observant le comportement publique d'une application

pas directement connectées au flux d'exécution, elles entravent la solidité de certains scénarios de l'AS.

- *La communication inter-composants (CIC)*. La communication entre les différents composants dans la même application ou d'autres applications est généralement déclenchée par des méthodes spécifiques, appelées méthodes CIC. Les méthodes CIC utilisent un paramètre spécial, contenant toutes les informations nécessaires pour spécifier leurs composants cibles et l'action à effectuer dans un *intent*. Comme les méthodes du cycle de vie, les méthodes CIC sont traitées par le système qui est chargé de les résoudre au moment de l'exécution. Par conséquent, l'AS trouvera hasardeux d'émettre des hypothèses sur la façon dont les composants se connectent les uns aux autres, à moins d'utiliser une heuristique avancée. FlowDroid [97], l'un des analyseurs statiques les plus avancés pour Android, est incapable de suivre les fuites de données entre les composants vu qu'il n'est pas en mesure de connaitre le schéma CIC. Afin de pallier ce problème, les auteurs postulent que toute communication avec une composante externe engendrera une fuite d'informations et que toutes les données échangées sont sensibles. Une telle approximation permettra de détecter la possibilité d'une exfiltration de données, en revanche, elle augmentera certainement le nombre de faux positifs.

- *Les défis hérités de Java*. Comme les applications Android sont principalement écrites en Java, les analyseurs statiques pour ces applications sont confrontés aux mêmes problèmes, notamment la gestion du chargement dynamique du code, la réflexion, le *multithreading*.

    - *Réflexion* : le chargement dynamique du code et des appels réflectifs constituent des contraintes réelles difficiles à traiter de manière statique [49, 50]. En effet, les classes qui sont chargées à l'exécution sont souvent pratiquement impossibles à analyser car elles peuvent être générées à la volée. Sur les 900 applications analysées statiquement par Stowaway [48], 61% utilisent la réflexion dont 59% des appels réflectifs ont pu être détectés. Malgré la puissance de cet outil par rapport aux autres, 105 applications avec des appels réflectifs n'ont été pas capturées, soit 12% des 900 applications. En effet, l'analyse réflective de Stowaway échoue lorsqu'elle se confronte à la création de noms de méthodes basés sur des variables d'environnement non statiques.

    - *Multithreading* : l'analyse des programmes concurrents est difficile car il est compliqué de caractériser l'effet des interactions entre les threads. En outre, l'analyse de tous les entrelacements d'instructions provenant de threads parallèles entraîne généralement des temps d'analyse exponentiels [102].

- *L'obfuscation du code* : une autre barrière difficile à franchir apparaît lorsque le code ciblé est obfusqué. L'obfuscation du code est une technologie utilisée pour rendre sa rétro-ingénierie plus difficile, voire impossible. Les techniques d'obfuscation utilisées dans les logiciels malveillants actuels visent principalement à entraver l'AS. Dans ce cas, il devient difficile aux techniques de décompilation de parvenir à reconstituer le code source. Sihag et al. [104] discutent différentes

méthodes d'obfuscation de code qui sont applicables sur le code source Java, le bytecode Dalvik et même le code natif.

3.3 Analyse dynamique

3.3.1 Acquisition

Les cibles d'intérêt dans un contexte dynamique sont des composants qui peuvent être résolus au moment de l'exécution, c.-à-d. liés à l'environnement d'exécution. Par rapport à Android, celles-ci incluent généralement tout ce que peut être recueilli à partir du SE et de l'instance de la MVD où l'application s'exécute (p. ex. [105, 106]). La plupart de ces analyses portent sur les appels système [2] qui résultent de l'interaction entre l'application et le noyau Linux (p. ex. [107–109]) ou aux appels aux méthodes de l'API Android. Ces méthodes représentent des actions concrètes de l'application qui devraient être effectuées par le SE afin de produire le comportement désiré. Tracer ces données permettra de détecter la possibilité d'un détournement des appels système vers un intermédiaire, identifier des séquences représentant des comportements jugés d'intérêt comme ceux utilisés dans l'envoi d'un message texte, etc.

Les entrées/sorties telles que les données échangées par communication réseaux (SMS, Internet, etc.) ou par périphériques (clavier, Webcam, récepteur radio, etc.) sont des exemples des données qui peuvent être tracées durant l'exécution du programme. De même, les permissions demandées à l'exécution, la mémoire vive et l'ensemble de processus créés à l'exécution sont toutes des données qui peuvent être extraites à des fins d'analyse. Li et al. [110] analysent le mécanisme de permissions d'Android en exécutant plusieurs cas de tests afin de déclencher des comportements inattendus. L'objectif de l'analyse est de prévenir les attaques par escalade de privilèges. DroidForce [111] permet aux utilisateurs de spécifier des contraintes précises sur la manière et le moment où les données peuvent être traitées sur leurs téléphones. Les politiques peuvent être échangées dynamiquement au moment de l'exécution. Elles sont spécifiées à l'aide d'OSL (Obligation Specification Language). Ce sont des politiques complexes, centralisées sur les données (p. ex. interdire l'envoi d'un SMS en fonction du numéro de téléphone) et globales, c.-à-d. qu'elles dictent des contraintes pour une seule application ou pour plusieurs en même temps (p. ex. autoriser l'envoi d'exactement deux SMS par jour et pas plus à un numéro donné, quelle que soit l'application qui l'envoie). DroidForce utilise FlowDroid [97] pour l'analyse statique de l'application. Malgré l'expressivité de la spécification des politiques proposées, ceci n'était pas le cas pour le programme Android qui n'a subi aucune spécification formelle. En effet, DroidForce fonctionne en instrumentant les appels API directement au niveau du bytecode. Bagheri et al. [112] proposent SEPAR, une approche pour le renforcement dynamique des politiques synthétisées, à granularité fine, spécifiquement générées pour une collection particulière d'applications installées. Les auteurs proposent des propriétés de sécurité liées à certaines vulnérabilités qui résultent de la CIC et inter-applications, comme l'escalade des privilèges et la fuite d'information.

2. Une façon pour bénéficier des services offerts par le noyau du SE, telles que les fonctions liées au réseau, au matériel et aux processus.

Les propriétés de sécurité sont spécifiées à l'aide d'un langage de spécification appelé Alloy [113]. Une partie de plateforme Android, qui concerne les APIs relatifs à ce type de propriétés (permission, *intent*, flux de donnée) a été également spécifiée par Alloy. Apex [114] étend les mécanismes de sécurité existant dans Android pour mettre en œuvre un cadre pour le renforcement des politiques qui permet aux utilisateurs d'accorder des permissions de manière précise et de prendre en charge les révocations et les activations des permissions d'une application au moment de l'exécution. Les auteurs ont proposé un formalisme de spécification ainsi qu'une mise en œuvre d'un moniteur pour vérifier le comportement à l'exécution des applications installées. DeepDroid [115] réalise l'instrumentation au niveau natif avec la possibilité d'intercepter les appels système en plus de l'invocation de méthodes pour réguler l'accès aux ressources sensibles. Toutefois, DeepDroid n'utilise aucun langage expressif de politique pour le renforcement.

Ainsi, tous les obstacles qui entravent les analyses statiques comme le cas de chargement dynamique du code, la réflexion sont facilement tracés par les approches dynamiques. Ceci fait de ces analyses des approches complémentaires aux analyses statiques.

3.3.2 Limitations

Les analyses dynamiques sont également assujetties à des contraintes qui réduisent leurs efficacité. Parmi ces contraintes, nous citons le raisonnement sur le code natif. En effet, les méthodes exécutées dans les bibliothèques natives, accessibles via JNI, et qui peuvent contenir des données sensibles, pourraient échapper à ce type d'analyse. Certains choisissent d'empêcher l'exécution du tout code natif pour résoudre ce problème. Cependant, cela compromettra la fonctionnalité de l'application et violera la propriété de transparence qui exige la préservation de la sémantique de toute exécution qui peut être valide.

Comme expliqué dans le chapitre précédent, le monitorage requiert en général plus de temps pour assembler des données qui ne sont accessibles que durant l'exécution de l'application. Les mêmes contraintes se manifestent en l'appliquant sur le système Android. L'AD est incapable aussi d'assurer la sûreté de l'application, vu qu'elle repose sur un échantillon d'exécution ou quelques cas de test.

Afin de renforcer des politiques de sécurité plus flexibles, plusieurs approches ont recours à la modification de l'environnement d'exécution ou du SE Android afin d'insérer des modules de monitorage au niveau des interfaces clés pour assurer l'interception des activités malveillantes lorsqu'elles se produisent sur l'appareil [105, 114]. En d'autres termes, afin d'intégrer le moniteur d'exécution qui peut être en effet un programme ou une application qui s'exécute séparément de l'application surveillée. Cette approche présente un certain nombre d'inconvénients. Premièrement, il faudrait créer un micro-logiciel et un code de plateforme personnalisés -ce qui est fastidieux- et les déployer, ce qui nécessite généralement l'enracinement («*rooting*» en anglais) de l'appareil. Ceci entraîne dans certains cas l'annulation de la garantie de l'utilisateur et affecte la stabilité et la portabilité du système. Deuxièmement, la maintenance de la plateforme modifiée est un défi, surtout pour différents appareils et différentes

versions de la plateforme. La mise à niveau vers une version plus récente d'Android nécessite de réimplémenter les modifications personnalisées dans la nouvelle source de la plateforme, de reconstruire et de déployer sur le dispositif. Le remplacement de la MVD par ART, à titre d'exemple, a rendu invalides les techniques et outils qui reposaient sur les mêmes composantes pour fonctionner, comme le cas du système TaintDroid [105]. Troisièmement, il est difficile de demander à un utilisateur normal d'appliquer le correctif source d'un cadre de sécurité et de compiler l'arbre source d'Android pour son propre appareil. Ces problèmes empêcheront de nombreux projets de sécurité Android basés sur le SE d'être suffisamment adoptés par les utilisateurs normaux. En effet, les applications sont limitées aux politiques de sécurité qui sont prises en charge par le cadre Android modifié. Les problèmes de construction et de déploiement rendent compliqué pour un utilisateur l'ajout ou la mise à jour de nouvelles politiques de sécurité au fil du temps [33].

3.4 Réécriture de programmes

3.4.1 Acquisition

Comme les techniques de réécriture réécrivent le programme statiquement, leurs cibles d'intérêt sont les mêmes utilisés par l'AS. Conséquemment, cette approche peut être appliquée sur le fichier *manifest*, les ressources (p. ex.[116]), le fichier OAT et le code dans tous ses niveaux. Une autre façon de faire est réécrire directement le bytecode Dalvik (p. ex. [1, 33]). Ceci permet d'épargner les efforts de décompilation de Dalvik vers Java et ensuite la compilation de Java vers Dalvik, qui pourrait dans certains cas échouer [10]. Jeon et al. [116] proposent Dr. Android et Mr Hide. Ces outils permettent d'implémenter des permissions plus fines pour l'accès à des fonctionnalités sensibles (Internet, contacts et paramètres du système) et les renforcer dans l'application Android par réécriture du bytecode Dalvik, du fichier manifest et des fichiers ressources. Costa et al. [117] se concentrent sur le bytecode Java. Ils proposent PolEnA, une extension de leur cadre de sécurité Android présenté dans [118] pour renforcer des politiques de permissions dans Android. Les auteurs définissent un langage formel pour les politiques de sécurité. Pour assurer la compatibilité avec l'outil, ils proposent un cadre de réécriture. Un APK est d'abord décompilé, puis le *framework* réécrit son bytecode Java. Enfin, l'application est assemblée et signée.

Bien qu'elle se base sur les données obtenues statiquement, la réécriture de programmes a pu surmonter certaines contraintes qui limitent l'efficacité des approches statiques. Premièrement, les analyses statiques sont en quelques sortes très restrictives, binaires et moins permissives que les analyses dynamiques. On a vu qu'une seule exécution qui viole une politique de sécurité privera l'application de s'exécuter, et ce à jamais. La réécriture de programmes en revanche, offre à l'application la possibilité de s'exécuter en transformant tout comportement suspect en un comportement qui respecte la politique de sécurité. Cette technique a pu trouver un compromis entre la laxité et la restriction excessive des mécanismes existants.

La deuxième contrainte est liée aux utilisations courantes de constructions Java potentiellement pro-

blématiques telle que la réflexion et la répartition[3] dynamique des appels. La sémantique de ces constructions dépend d'informations qui ne sont disponibles que lorsque le programme s'exécute. Les techniques de réécriture de programmes réécrivent les applications afin d'inclure des vérifications (ou des tests dynamiques) à l'exécution pour tenir compte des cas où l'AS échoue. Davis et al. [33] considèrent que ces appels à la bibliothèque de réflexion sont des invocations de méthodes normales. Les auteurs procèdent alors de la même façon qu'une invocation d'une méthode normale. Appguard [1] est un autre exemple de technique de réécriture IRM qui permet de gérer la réflexion, le code chargé dynamiquement et le code natif. Backes et al. définissent une politique de sécurité spéciale appelée *ReflectionPolicy*. Cette politique vise à protéger le moniteur intégré contre toute application malveillante qui s'appuie sur la réflexion pour y accéder. Pour contourner ce type d'attaque, ils interceptent les appels de fonctions à l'API *Reflection*. En particulier, ils surveillent les opérations qui accèdent aux classes et aux champs Java comme *java.lang.Class->forName()* ou *java.lang.Class->getField()* et empêchent ainsi l'accès au package du moniteur intégré. Contrairement aux approches statiques qui reposent sur des hypothèses conservatrices, cette façon de procéder permet de réduire le nombre de faux positifs en produisant du code auto-surveillé capable d'anticiper et éviter toute action potentiellement malicieuse. En ce qui concerne l'exécution du code natif potentiellement malicieux qui représente un défi pour les analyses dynamiques, au lieu de bloquer l'exécution de tout code natif non fiable en interceptant les appels à *System.loadLibrary()*, AppGuard interrompt la procédure de réécriture et avertit l'utilisateur à chaque fois qu'une application tente d'exécuter du code natif non fiable.

Les résultats atteints par quelques chercheurs pionniers tels que Schneider, Ligatti et Hamlen ont simulé d'autres chercheurs, qui ont tiré profit de ces approches et des avantages liés à l'utilisation des méthodes formelles, en particulier l'algèbre de processus, pour renforcer des politiques de sécurité dans des systèmes séquentiels et concurrents (p. ex. [76, 80–85, 119]). Dans cette direction de recherche, nous avons défini dans [13] un opérateur de renforcement qui prend en entrée un programme parallèle spécifié par une version étendue de l'algèbre de processus *ACP* (Algebra of Communicating Process), une politique de sécurité présentée comme une formule logique, et une valeur de risque fixée par l'utilisateur. Le résultat est un nouveau processus qui respecte la politique de sécurité sans dépasser la valeur de risque spécifiée. L'approche a été appliquée sur une application Android réelle, à savoir *LinkedIn*.

Par rapport à Android, la plupart des études se sont focalisées sur une partie ou un seul aspect dans ce SE, que ce soit au niveau de l'analyse ou le renforcement de politiques de sécurité. Ceci revient, selon eux, à la complexité de ce système qui est impossible à modéliser tout entier. C'est pourquoi, chacun s'est intéressé à une ou deux parties (ou aspects) telles que le système de permission (p. ex.[120]), l'interaction inter-applications (p. ex. [121]), l'aspect événementiel (p. ex. [122, 123]), le mécanisme multitâche (p. ex. [124, 125]) ou seulement des constructions Android spécifiques (p. ex.

3. En anglais «dispatching», c'est le processus de sélection de l'implémentation d'une opération (méthode ou fonction) à appeler au moment de l'exécution.

[126]). Schlegel [127], souligne que les applications ont besoin de politiques de sécurité individuelles avec une sémantique spécifique à l'application. L'auteur propose AppSPEAR, une architecture de sécurité adaptée au renforcement de politique de sécurité au niveau des applications, sur laquelle il se concentre sur la politique de contrôle d'accès. Talegaon et al. [128] présentent une spécification formelle du système de contrôle d'accès Android. Durant le processus de formalisation, les auteurs découvrent certains comportements suspects dans le système de contrôle d'accès d'Android, mais ne présentent pas d'alternative pour les renforcer. Betarte et al. [129, 130] présentent une spécification formelle du modèle de permission d'Android. Le langage formel résultant a été utilisé pour renforcer des politiques de contrôle d'accès basées sur les permissions.

Chen et al. [35] propose Pegasus, un système pour la spécification et le renforcement des propriétés de sécurité relatives à l'ordre d'utilisation des permissions et des API spécifiques à la gestion des événements au sein des applications Android. L'utilisateur introduit une spécification du comportement attendu de l'application en utilisant des formules logiques temporelles, et le modèle Pegasus vérifie ces spécifications par rapport à une abstraction appelée graphe d'événements de permission (PEG) construit suite à une analyse statique. Pegasus prend en entrée un fichier DEX et la spécification de la propriété de sécurité en Java. Le bytecode Dalvik est traduit en premier lieu en un byte code Java puis transformé en PEG. Le PEG est ensuite vérifié à l'aide d'un outil de vérification pour la conformité avec la spécification. Si le PEG satisfait la politique alors l'application la satisfait également, sinon une nouvelle version réécrite est introduite contenant, pour chaque violation de la politique de sécurité, un contre exemple de trace.

Bien que les approches ci-dessus reposent sur une base formelle du système Android, elles sont limitées à une partie (essentiellement le système de permission). Par conséquent, les politiques de sécurité appliquées sont limitées à cette partie et ne peuvent pas traiter les autres vulnérabilités liées à d'autres aspects d'Android. Toutefois, la formalisation des fichiers *manifest* et *Smali* permet de renforcer diverses politiques de sécurité, pratiquement tout le comportement de l'application se trouve dans ces fichiers.

Plusieurs mécanismes de sécurité (p. ex. [1, 33, 109, 116]) ont adopté l'approche de réécriture IRM initiée par Erlingsson et Schneider [71]. En l'appliquant sur Android, l'idée fondamentale consiste à réécrire une application non fiable de telle sorte que le code qui surveille l'application (c.-à-d. le moniteur ou le *rewriter*) soit directement intégré à son code. Cela permet au moniteur d'observer une trace des événements relatifs à la sécurité, qui correspondent généralement à des invocations de méthodes. Pour renforcer réellement une politique de sécurité, le moniteur contrôle l'exécution de l'application en supprimant ou en modifiant les appels aux méthodes pertinentes pour la sécurité, voire en mettant fin au programme si nécessaire. Théoriquement, ceci est possible en redirigeant les appels de méthode vers le moniteur intégré et de vérifier si l'exécution de l'appel est autorisée par la politique. Techniquement, ceci est réalisé en modifiant les références des méthodes dans la MVD. AppGuard [1] est un système IRM pour Android. Comme l'illustre Figure 3.1, il prend en entrée une application non fiable *App* et des *politiques de sécurité* définies par l'utilisateur. Le *rewriter* édite le

DEX et intègre le moniteur dans l'application. Les références de la DVM sont modifiées de manière à rediriger les appels de méthode vers le moniteur de sécurité. Le composant *Gestionnaire* conserve un journal détaillé de tous les événements relatifs à la sécurité, pour permettre à l'utilisateur de surveiller le comportement de l'application.

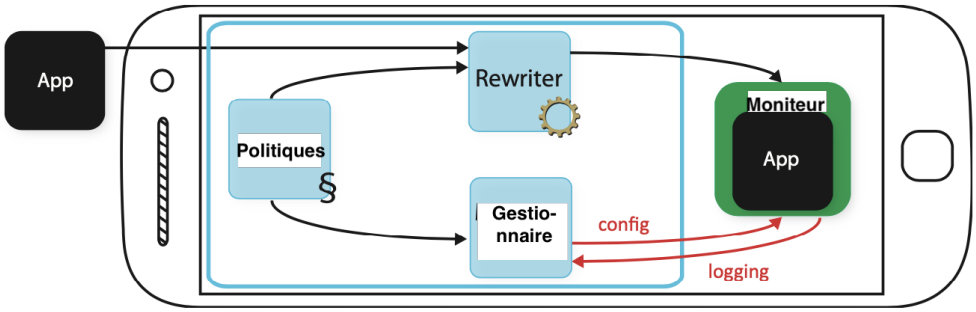


FIGURE 3.1 – Renforcement de politiques de sécurité par AppGuard [1]

Contrairement aux approches dynamiques, les auteurs affirment que le processus de réécriture est rapide, même sur le matériel limité d'un téléphone mobile. Le temps de réécriture est déjà raisonnable et peut être encore réduit avec quelques optimisations. La même logique a été adoptée par Davis et al. [33]. I-ARM-Droid intercepte les appels que l'utilisateur souhaite surveiller. Chaque API est remplacée par une nouvelle méthode incluant la politique de sécurité et retournant une valeur de retour fictive. À titre d'exemple, pour un accès capteur, la nouvelle méthode pourrait renvoyer de fausses données de capteur sans jamais accéder au capteur réel de l'appareil. Une fois ces méthodes sont réécrites, tous les appels aux anciennes méthodes sont remplacés par des appels vers les nouvelles méthodes réécrites. Dans le cas d'appels natifs, le processus de réécriture est interrompu et l'utilisateur est averti. Jeon et al. [116] préconisent de placer le moniteur dans une application à part. Ils suppriment ainsi toutes les permissions de l'application surveillée. Toutefois, cette approche présente quelques inconvénients. Si une politique de sécurité dépend de l'état de l'application surveillée, ceci entraîne une surcharge, vu que toutes les données pertinentes doivent être transférées au moniteur. En outre, il se peut que le moniteur ne soit pas encore initialisé lorsque l'application tente d'effectuer des opérations pertinentes en termes de sécurité. De plus, cette approche ne respecte pas le principe du moindre privilège puisque le moniteur doit disposer des permissions de toutes les applications surveillées.

3.4.2 Limitations

Les techniques de réécriture qui se limitent à une partie de l'application (les permissions, la CIC, la gestion d'événements, etc.) permettent de renforcer des politiques de sécurité qui se restreignent à ces aspects. Par la suite, elles ne permettent pas de résoudre d'autres comportements à risque. En effet, l'utilisation des fichiers *manifest* et *Smali* comme bases pour le renforcement permet de renforcer différentes politiques de sécurité et traiter diverses failles. Ces fichiers contiennent quasiment

tout le comportement d'une application, incluant les interactions utilisateurs possibles (le code des composants des interfaces graphiques tels que les boutons, les boîtes de dialogue, etc.), les appels aux méthodes API et Java (incluant les méthodes Java liées au *multithreading* et à la réflexion), les permissions, les points d'entrée et la CIC.

Nous avons vu également que la satisfaction de deux propriétés de correction et de complétude par un mécanisme de sécurité basé sur la réécriture est primordiale. La violation de ces deux conditions a un impact négatif sur la fiabilité du mécanisme adopté, et fait de lui une méthode de renforcement conservatrice de la sécurité.

En somme, quelque soit l'importance des résultats atteints sur les applications testées et quelques soit la technique de vérification utilisées, de telles approches basées sur des outils de test ne peuvent garantir que des systèmes qui satisfont aux exigences des cas de test. La vérification et la preuve de la solidité demeurent un obstacle majeur pour ces approches. D'ailleurs, aucune preuve ne peut démontrer leurs capacités effectives à satisfaire les politiques de sécurité qu'elles sont censées renforcer.

3.5 Différentes approches pour la formalisation des applications Android

La formalisation est une étape cruciale dans le développement d'un système donné. Elle permet la simplification du langage formalisé en enlevant toute ambiguïté et toute redondance. Elle ouvre les portes à d'autres projets d'optimisations, de vérifications et de renforcement de la sécurité ou tout autre objectif qui n'est pas forcément lié à la sécurité. Généralement, les travaux qui visent à formaliser Android ont un objectif derrière. Quelque soit le motif : des fin d'analyses ; servir comme une base pour une certification, une détection des vulnérabilités ou de comportement malveillant potentiel, ou bien comme dans notre cas, un renforcement de la sécurité. Par ailleurs, nous considérons que cette formalisation est une contribution qui peut faire la base pour différents projets ultérieurs qui s'intéressent à ce système.

Les travaux [91–93, 101, 131–133] se basent sur la formalisation du bytecode Dalvik. Cependant, ils ignorent la nature concurrente de ce dernier. Wognsen et al. [92] ont présenté la première formalisation du bytecode Dalvik avec des fonctionnalités Java. Bien que les résultats de l'étude mettent en évidence la concurrence comme étant la fonctionnalité la plus souvent utilisée par les applications Android, avec un pourcentage (90,18%), cette fonctionnalité a été omise tant dans les analyses que dans la sémantique pour se concentrer à la place sur la réflexion, les exceptions et la répartition dynamique. Sur la même lignée, Payet et al. [132] définissent une sémantique opérationnelle pour un sous-ensemble des instructions Dalvik. Les règles sémantiques sont relativement complexes. Un programme Android a été modélisé comme un graphe de blocs où chaque bloc comprend une ou plusieurs instructions. Les blocs sont liés de manière à exprimer le flux de contrôle passant d'un bloc à l'autre. La sémantique est définie comme étant la base d'analyses statiques, notamment, celles qui prennent en compte le cycle

de vie des activités. Malgré l'importance de la relation thread-activité dans la sémantique d'Android, le *multithreading* a été dissocié de la sémantique des activités et totalement ignoré.

SymDroid [101] est un interprète [4] du bytecode Dalvik pour la détection d'éventuelles vulnérabilités. Il applique une exécution symbolique pour un langage intermédiaire simplifié d'une fraction des opcodes Dalvik, nommé *μ-Dalvik*. Ce sous-langage est défini à partir de 16 instructions considérées comme les plus pertinentes pour effectuer l'analyse du code. Ensuite, il est traité par un noyau d'exécution symbolique pour générer des traces comme résultat intermédiaire. Enfin, le post-analyseur inspecte les traces de sortie et détermine le résultat final. Bien que ce travail formalise, outre les instructions de bytecode, les bibliothèques systèmes, le cycle de vie des composants Android, les services et les vues, il ignore la nature concurrente du système, que ce soit dans le langage formel adopté ou au niveau de l'exécution symbolique du programme, qui est supposée séquentielle.

Le bytecode Dalvik formalisé par l'outil Julia [91] contient les instructions qui obscurcissent le code et qui modifient son ordre d'exécution (invocation de méthode, instructions de branchement et saut). Toutefois, les applications concurrentes n'ont pas été incluses dans ce travail et les gestionnaires d'événements sont supposés exécuter par un seul thread. De plus, nous ne sommes pas au courant de l'existence d'une preuve de solidité pour son extension de Java vers Android. Gunadi et al. [133] proposent une sémantique opérationnelle du bytecode DEX pour la certification des propriétés de non-interférence par le système de type. Cette étude comprend un processus de traduction de la sémantique du bytecode Java développée dans [66] vers une nouvelle sémantique pour le bytecode Dalvik. Les auteurs concluent que les vérificateurs du bytecode existants pour le bytecode Java pourraient certifier également les propriétés de non-interférence pour le bytecode Dalvik. Chaudhuri [126] présente une étude de sécurité sur Android en utilisant une sémantique opérationnelle et un système de types pour des constructions Android spécifiques. Cependant, la sémantique ignore tous les construits Java qui peuvent apparaître dans les applications Android (pas de modélisation formelle des classes et des méthodes), pour se concentrer à la place sur les composants Android, en particulier les *intents* et toutes les fonctionnalités liées à ces composants. Qin et al. dans [123] utilisent une sémantique opérationnelle non standard qui décrit l'exécution symbolique dynamique (appelée aussi exécution concolique) du programme. La sémantique est basée sur Acteve [122]. Elle décrit l'exécution d'un programme en réponse à une séquence d'événements générée automatiquement à partir d'un environnement externe. Toutes les autres fonctionnalités et instructions gérées par Android ont été négligées pour se concentrer à la place sur le paradigme événementiel, ce qui est peu expressif pour formaliser une application Android. Armando et al. [119] ont proposé un modèle formel d'Android qui permet d'exprimer formellement des objectifs de sécurité de haut niveau ainsi que de vérifier si ces objectifs sont atteints ou non, et d'identifier les éventuelles vulnérabilités de sécurité, le cas échéant. Les auteurs adoptent un formalisme de type algèbre de processus, appelé *History expressions* [134].

Si les contributions sus-citées ont connu les avantages de l'utilisation des méthodes formelles dans

4. Appelé souvent, à tord, " interpréteur" par mauvaise traduction de l'anglais, est un outil permettant d'analyser, de traduire et d'exécuter un programme écrit dans un langage informatique. De tels langages sont dits langages interprétés.

la spécification du système Android, elles ont toutes ignoré la programmation concurrente qui est omniprésente dans ce dernier.

En revanche, une minorité des travaux a tenu compte du *multithreading* en formalisant Android. Parmi ces contributions, il y a ceux qui le combinent avec la gestion des événements (p. ex. [135–137]), d'autres considèrent les principales méthodes API qui y sont liées (p. ex. [138]). Kanade [136] propose une sémantique d'un modèle concurrent combinant les threads et l'ensemble d'événements. Toutefois, tout l'accent a été mis sur la nature événementielle d'Android et sa relation avec les threads. En conséquence, tous les autres états que la sémantique pourrait atteindre, tels que ceux résultant de l'exécution d'instructions de base (appel de méthode, branchement, instructions de retour, etc.) n'ont pas été considérés. Calzavara et al. [138] proposent une analyse statique basée sur un modèle formel de la plateforme Android. La sémantique couvre les principales instructions de Dalvik et traitent le *multithreading*. Cet article est une extension de [139], avec des changements dans la sémantique pour gérer les exceptions et l'aspect concurrent. Néanmoins, l'ordonnancement des threads n'a pas été abordé et le lancement des threads est laissé à la machine virtuelle qui est supposée l'exécuter à un moment imprévisible. Mariuca et al. [140] et Rosangela et al. [141, 142] modélisent les contrôles des applications Android contre l'attaque par collusion d'applications. Au lieu d'analyser une seule application, l'approche est appliquée à un ensemble d'applications. La première étape consiste à désassembler le code DEX de chaque application. Ensuite, analyser le code Smali obtenu à l'aide de l'outil $\mathbb{K}$ Framework. La compilation avec $\mathbb{K}$ génère une théorie de réécriture dans Maude [143] représentant les applications Android. Dans la dernière étape, l'évaluateur de modèles vérifie la propriété (collusion) pour l'ensemble des applications Android en entrée.

Dans l'ensemble, aucune des études mentionnées ci-dessus n'offre une sémantique complète couvrant tous les états qu'un thread peut atteindre, ni représente tous les éléments essentiels du *multithreading*. La quasi-majorité des études qui formalisent le bytecode Dalvik, tout en traitant la concurrence, ne comprennent que les deux instructions Dalvik relatives à l'utilisation du moniteur : *monitor-enter* et *monitor-exit*. Toutefois, la sémantique d'un programme Android ne doit pas se limiter à ces instructions. En effet, les instructions relatives à la communication, la signalisation et à l'ordonnancement des threads font partie du comportement concurrent du programme.

En somme, la plupart des sémantiques proposées dans cette section sont des sémantiques dédiées, qui ont été développées "manuellement", c.-à-d. sans l'utilisation d'aucun environnement, guide ou langage pour la formalisation. En effet, les séamtniques générées ne sont pas exécutables, et sont sujettes aux erreurs. En outre, la vérification et la preuve de correction de la formalisation nécessitent le développement de manuel d'outils personnalisés (tels que des interprètes et des compilateurs), généralement sans garantie. Le programme qui en résulte peut finir par présenter des comportements inattendus et, dans certains cas, entraîner des conséquences irréversibles. Certains environnements sont destinées pour la tâche de formalisation. Ils sont dotés d'un guide pour la formalisation. $\mathbb{K}$ *Framework*, à titre d'exemple, résout tous les problèmes liés à la formalisation "manuelle". Il comprend un ensemble d'outils permettant de valider la sémantique à l'aide d'un compilateur-sémantique, de

simuler des exemples de programmes par le moyen d'un interprète et un générateur de cas d'essai, vérifier des propriétés par des modèles spécifiés, tels que l'évaluateur de modèle. C'est un moyen qui permet d'éviter de gaspiller des ressources pour concevoir et mettre en œuvre des outils personnalisés coûteux. Différentes approches définissent plusieurs sémantiques pour un même langage, chacune étant conçue dans un but précis (par exemple, la vérification de programmes, interprétation, débogage, etc.), ce qui n'est ni économique, ni fiable. Par la suite, ces initiatives manquent des fonctionnalités pareilles qui assurent la correction et qui augmentent la fiabilité de la sémantique générée.

3.6 Tableau comparatif

Dans Table 3.2, nous exhibons un tableau comparatif qui récapitule et compare de manière plus structurée les différents travaux connexes. Les approches sont évaluées et classées selon les critères suivants :

Spécification formelle : ce critère indique si l'approche a adopté une démarche formelle pour spécifier une application Android. Dans le cas où ce critère n'est pas rempli, nous ne pouvons pas prouver la solidité de l'approche. Si oui, d'autres critères s'imposent sur la formalisation résultante.

- *Sémantique de traduction et Sémantique dédiée* : deux façons sont adoptées pour formaliser un langage donné. La première est de faire correspondre sa syntaxe à celle d'un autre langage formel existant ayant une sémantique initiale bien définie. La deuxième consiste à développer une sémantique dédiée au langage. L'avantage de la première approche réside dans l'utilisation d'une sémantique de traduction initiale, telle que *BPA* (*Basic Process Alebra*) ou *ACP* (*Algebra of Communicating Process*). Ceci permettra d'épargner d'effort de développer toute une nouvelle sémantique à part entière dans un formalisme différent, qui est une tâche ardue, sujette aux erreurs et fastidieuse. Un autre point fort est que le langage utilisé peut s'adapter aux techniques et outils développés pour le langage destination. Toutefois, lorsque les deux langages n'ont pas le même niveau d'abstraction, la correspondance entre les deux pourrait être difficile à comprendre et peut aboutir à une structuration compliquée avec une signification non intuitive. Par ailleurs, lorsque la version traduite est exécutée et que des erreurs se produisent, il sera difficile de remonter jusqu'au langage d'origine. En somme, bénéficier d'un langage de spécification existant pour définir un nouvel, requiert un langage de base qui remplit plusieurs critères à la fois : une intuition solide permettant de définir des caractéristiques complexes à un système donné, qui supporte naturellement le non-déterminisme, la concurrence, tout en étant générique et modulaire afin d'assurer la flexibilité et la réutilisation. En revanche, développer une sémantique de zéro permet généralement d'obtenir une sémantique plus compréhensible et intuitive. En fait, les règles sémantiques sont conçues dès le départ pour correspondre exactement au sens intuitif des constructions de langage. En outre, dans le cadre de cette approche, le débogage d'un programme est généralement plus simple que dans le cas de la sémantique

de traduction. Néanmoins, des erreurs peuvent y glisser. Les avantages et les inconvénients de chaque méthode sont résumés dans Table 3.1.

	Sémantique de traduction	Sémantique dédiée
Bénéficier d'une spécification formelle existante	✓	
S'adapter aux techniques et outils développés pour le langage de traduction	✓	
Capacité d'exprimer des caractéristiques complexes		✓
Sémantique claire et intuitive		✓
Sémantique non sujette à des erreurs	✓	

TABLE 3.1 – Sémantique de traduction vs sémantique dédiée

- *Outils de validation* : nous avons vu que le développement d'une sémantique dédiée est une tâche sujette à l'erreur bien qu'elle permette d'obtenir une sémantique plus intuitive. Cependant, l'utilisation d'un outil qui permet de définir formellement des langages à partir de leurs syntaxes et sémantiques permettra de résoudre tous les problèmes liés à cette façon. L'idéale que la sémantique générée par cet outil soit exécutable, afin d'être testée, compilée et vérifiée et par la suite s'assurer que tous les détails syntaxiques et sémantiques sont bel et bien défini.

- *Concurrence :* la programmation concurrente est un facteur clé dans une application mobile. Elle permet entre autres de faire fonctionner plusieurs programmes ou plusieurs parties d'un programme en parallèle. Ceci fait de la concurrence un critère important pour évaluer la formalisation du langage Android. Ignorer cette fonctionnalité dans le formalisme et considérer le programme comme étant séquentiel est une hypothèse qui rabaisse la valeur de l'approche adoptée.

- *Expressivité* : le formalisme qui se limite à une seule partie dans Android comme le système de permission ou un seul aspect comme l'aspect événementiel, en ignorant d'autres fonctionnalités est considéré comme moins expressif. Néanmoins, nous sommes conscient que formaliser un système complexe comme Android dans toute son entièreté et réunir tous ces facteurs dans une seule sémantique s'avèrent très ardu. Toutefois, nous considérons que plus la sémantique tient compte des aspects et des fonctionnalités (réflexion, code natif, *multithreading*, etc.), plus le langage formel obtenu est expressif. Nous exprimons une spécification formelle suffisamment expressive par ●, peu expressive par ◐ et pas expressive par ○.

Renforcement : Les approches de renforcement sont classées selon la technique de vérification utilisée.

Approches	Spécification formelle						Renforcement		
	Non	Sémantique de traduction	Sémantique dédiée	Outils de validation	Concurrence	Expressivité	Analyse statique	Analyse dynamique	Réécriture de programmes
Wognsen et al. [92]			✓			◐			
Payet et al. [132]			✓			◐			
Jeon et al. [101]			✓			◐			
Cousot et al.[91]			✓			◐			
Kim et al. [93]			✓			◐			
Armando et al. [119]			✓			◐			
Gunadi et al. [133]			✓			◐			
Kanade et al.[136]			✓		✓	◐			
Calzavara et al. [138]			✓		✓	●			
Calzavara et al. [139]			✓			◐			
Bagheri et al.[120]		✓				◐			
Khan et al.[121]			✓		✓	◐			
Jinlong et al.[124]			✓			◐			
Chaudhuri [126]			✓			◐			
Qin et al. [123]			✓			◐			
Mariuca et al. [140]			✓	✓	✓	●			
Arzt et al. [97]	✓								
Nisi et al.[108]	✓								
Felt et al. [48]	✓								
Chin et al. [144]	✓								
Chen et al. [99]		✓				◐	✓		
Davis et al. [33]	✓								✓
Schlegel[127]	✓								✓
Cotterell et al. [145]	✓								✓
Backes et al.[1]	✓								✓
Jeon et al. [116]	✓								✓
Xu et al. [109]	✓								✓
Rasthofer et al. [111]	✓							✓	
Wang et al. [115]	✓							✓	
Bauer et al. [146]	✓							✓	
Huang et al. [147]	✓								✓
Nauman et al. [114]	✓							✓	
Betarte et al. [129]			✓	✓		◐			✓
Bai et al. [148]	✓							✓	
Chen et al.[35]			✓			◐			✓
Bagheri et al. [112]		✓				◐		✓	

TABLE 3.2 – Tableau comparatif des approches connexes

En somme, selon Table 3.2, les approches formelles constituent une majorité pour l'analyse et la détection ou pour une base d'analyse des applications Android. Elles manquent certainement le côté défense, qui est représenté avec l'approche de renforcement de politique de sécurité. Au moment de la rédaction de ces lignes, aucune approche pour le renforcement de la sécurité n'a été publiée par ces travaux. En plus, la plupart de ces approches ont eu recours à une sémantique dédiée pour la formalisation d'une application Android sans être validée à l'aide d'un outil automatisé. L'autre catégorie présente des approches qui ont tenté de combler cette déficience, mais qui n'adoptent pas des approches de renforcement dans un cadre formel. Dans ce cas, aucune preuve formelle ne peut démontrer que ces mécanismes renforcent réellement les politiques qu'ils prétendent instaurer. La troisième classe correspond à quelques tentatives qui proposent un cadre formel de renforcement, un langage formel pour décrire les politiques de sécurité et l'application Android. Cependant, elle ignore des critères importants liés à l'expressivité de la sémantique définie pour Android, et se contente de spécifier une partie dans Android (le système de permission dans la plupart de cas). La concurrence, à titre d'exemple, est parmi ces critères et qui a été totalement omise dans ces travaux.

Le tableau comparatif a confirmé l'absence d'un travail de recherche qui focalise à la fois sur le renforcement formel et automatique de politiques de sécurité et qui se base sur un formalisme de principaux composants, fonctionnalités et aspects d'Android. Ceci nous ramène aux besoins de bien choisir et filtrer les techniques de vérification adéquates et de voir comment l'adapter formellement pour renforcer automatiquement des politiques de sécurité au sein d'un programme Android. Par conséquent, il serait intéressant de proposer une nouvelle approche dans ce contexte qui réunit la plupart des critères importants, et qui présente dans un cadre **formel** la possibilité de **renforcer automatiquement** des politiques de sécurité dans une spécification formelle suffisamment **expressive**.

3.7 Conclusion

Dans ce chapitre, nous avons discuté l'application de différents mécanismes de renforcement sur le système Android. Les différentes approches mentionnées ont démontré certaines capacités mais aussi des limites. Ces contributions ont confirmé encore une fois le grand potentiel des techniques de réécritures dans le renforcement de politiques de sécurité dans Android. Ces techniques ont pu surmonter les obstacles qui entravent les approches statiques et dynamiques, que ce soit ceux liés au système Android (code natif, réflexion, modification de SE, etc.) ou qui résultent de la technique elle-même (durée et coût d'application, sûreté du programme, laxité, restriction excessive, taux de faux positifs, etc.). Tous ces facteurs ont favorisé l'adoption de la technique de réécriture comme étant l'alternative idéale aux deux autres approches. Ce chapitre souligne également le rôle des méthodes formelles dans le processus complet de renforcement commençant par spécification, la vérification jusqu'à la preuve. Nous avons constaté une minorité de travaux formels qui renforcent de manière efficace et précise de politiques de sécurité. Le reste des travaux proposés dans la littérature, qui représentent une majorité, offrent des solutions non formelles. Dans les prochains chapitres, nous présentons un cadre formel pour le renforcement de politiques de sécurité dans les applications Andoird par réécriture.

Deuxième partie

Contribution

Chapitre 4

Smali$^+$: Sémantique opérationnelle pour Smali

4.1 Introduction

Bien que plusieurs programmeurs négligent l'importance de la sémantique formelle, celle-ci joue un rôle majeur dans la programmation quotidienne. Le besoin d'une sémantique formelle provient de la nécessité de pouvoir garantir qu'un programme se comporte conformément à ce qu'il est censé faire, en toutes circonstances, et pas seulement dans l'environnement de test.

La plupart des programmeurs se focalisent sur la syntaxe du langage où chaque construction linguistique a une signification précise, formulée en langage naturel dans les manuels, à titre d'exemple "*if c then r*". Toutefois, il arrive au programmeur de constater que, bien que la structure du programme soit correcte, ce dernier ne s'exécute toujours pas comme il le devrait. Ceci est dû au fait que la sémantique prévue ne parle qu'en termes généraux, et ne donne aucun moyen de vérifier réellement que le programme est bien celui souhaité.

Il est important que la sémantique soit formelle, afin de fournir i) à l'utilisateur une description non ambiguë de l'effet d'un programme ; ii) un critère de référence pour la mise en œuvre ; ii) une base pour l'analyse, l'optimisation et la vérification de programmes.

La sémantique opérationnelle, à titre d'exemple, est définie mathématiquement et non sur la base du sens du langage naturel. Dans le présent chapitre, nous proposons une sémantique opérationnelle dédiée pour Smali. Le langage formel résultant est un sous-langage de Smali, appelé *Smali*$^+$. *Smali*$^+$ couvre l'ensemble d'instructions Dalvik [149] les plus utilisées selon une étude conduite dans [92]. Ces instructions ont été généralisées en 12 instructions sémantiquement différentes. En plus de cet ensemble, la sémantique inclut les instructions liées à la programmation concurrente.

4.2 Notations

Dans ce qui suit, nous adoptons les notations suivantes :

— Une pile est de la forme $A :: B$, où A constitue le sommet de la pile, B est la partie restante de la pile. Une pile vide se présente par ϵ ;

— $\perp$ désigne une valeur indéfinie ;

— Si M est un Map, alors :

- $M(x)$ sera la valeur de x de M ;
- $M[x \mapsto v] = M'$, tel que : $\begin{cases} M'(y) = M(y) & si\ x \neq y \\ M'(x) = v & sinon \end{cases}$
- $dom(M)$ est le domaine de M, tel que $dom(M)$ égale à l'ensemble de toutes les variables $\mathcal{X}$ dans M, i.e. $dom(M) = \bigcup_{x \in \mathcal{X}} x$, tel que $M(x) \neq x$;

— La fonction $[-]_B$ évalue une expression booléenne. Elle est définie comme suit :

$$[-]_B : Exp_B \longrightarrow \{true, false\}$$

avec Exp_B est une expression booléenne.

4.3 Exécution séquentielle

Chaque application Android est composée d'au moins un thread qui définit le chemin d'exécution du code. Si aucun autre thread n'est créé, tout le code sera exécuté selon un seul chemin. Dans cette section, nous supposons une exécution séquentielle avec un modèle de programmation simple et un ordre d'exécution déterministe. Ceci signifie qu'une instruction donnée ne peut être exécutée qu'à la fin de toutes les instructions qui la précèdent.

4.3.1 Syntaxe

Table 4.1 présente les catégories syntaxiques de base ainsi que la syntaxe des instructions Dalvik sélectionnées. La syntaxe est définie avec la notation conventionnelle Backus-Naur Form (BNF). Les termes terminaux sont représentés en gras.

Un APK dans $Smali^+$ est constitué de classes de l'application, qui correspondent aux fichiers .smali. Le fichier *Androidmanifest* et le reste des fichiers d'un APK désassemblé ne sont pas inclus dans la définition de $Smali^+$. La définition d'un *package* Android *Pckg* est composée du mot clé ".package" suivie par son nom *packagename* et d'un ensemble de classes.

(Package)	*Pckg*	::= **.package** *packagename* {*Cl**}	
(Classe)	*Cl*	::= **.class** (*Acc-flg*)* *Cfn* **.super** *Sc* **.implements** *Intf** {*Fld**, *Mtd**}	
(Super-classe)	*Sc*	::= *Cfn* | $\top$	
(Interface)	*Intf*	::= **.interface** (*Acc-flg*)* *Inf* **.super** *Sinf** {*CstFld**, *MtdSign**}	
(Super-interface)	*Sinf*	::= *Inf*	
(Champs)	*Fld*	::= **.field** (*Acc-flg*)* *Fn* **:** *Type*	
(Champ-constant)	*CstFld*	::= **.field** **public final static** *Fn* **:** *Type*	
(Méthode)	*Mtd*	::= **.method** (*Acc-flg*)* *MtdSign* **.locals** *Loc* {*LabelInst**}	
(Signature d'une méthode)	*MtdSign*	::= *M* (*Type*$_1$, ..., *Type*$_n$) *retType*	
(modificateurs)	*Acc-flg*	::= **public** | **private** | **protected** | **final** | ...	
(Corps d'une méthode)	*LabelInst*	::= *Lab Inst*	
(Instructions)	*Inst*	::= **goto** *Lab*	(saut inconditionnel)
		| **if**$_{\lessgtr}$ *Rg Rg* *Lab*	(saut conditionnel)
		| **move** *Rg Rg*	(déplacement entre registres)
		| **binop**$_{\oplus}$ *Rg Rg Rg*	(opération binaire)
		| **unop**$_{\odot}$ *Rg Rg*	(opération unaire)
		| **new-instance** *Rg Cfn*	(création objet)
		| **invoke-static** *Cfn MtdSig Rg**	(invocation méthode statique)
		| **invoke-virtual** *Rg MtdSig Rg**	(invocation méthode d'instance)
		| **move-result** *Rg*	(déplacement de la valeur résultat)
		| **return** *Rg*	(retour à partir d'une méthode non-void)
		| **return-void**	(retour à partir d'une méthode void)
(Opérateurs)	$\oplus$	::= **+** | **-** | ...	(opérateur binaire)
	$\odot$	::= $\neg$ |++| ...	(opérateur unaire)
	$\lessgtr$	::= **<**|**>**| ...	(opérateur de comparaison)
(Type)	*Type*	::= *Prim* | *Ref*	
	Prim	::= *Single* | *Double*	
	Ref	::= *Cfn*|*ArrayType*	
	ArrayType	::= *ArrayTSingle*|*ArrayTDouble*	
	ArrayTSingle	::= **array** (*Single*|*Ref*)	
	ArrayTDouble	::= **array** *Double*	
	Single	::= **boolean** | **char** | **byte** | **short** | **int** | **float**	
	Double	::= **long** | **double**	
(Type de retour)	*retType*	::= *Type* | **void**	
(Label et nombre de registre)	*Lab, Loc*	::= **int**	
(Noms de registres)	*Rg*	::= *v* | *p* | *p*0 | *ret*	
(Noms)	*Cfn*	::= **L***packagename*/*C***;**	(nom complet classe)
	Inf	::= **L***packagename*/*Itf* **;**	(nom complet interface)
	*v, p, p*0, *ret*	::= **string**	(registres)
	packagename, C		(package, classe)
	Itf, Fn, M		(interface, champ, méthode)

TABLE 4.1 – *Smali*$^+$: Syntaxe du programme séquentiel

La définition d'une classe *Cl* comprend ses modificateurs ou indicateurs d'accès *Acc-flg*. Il s'agit d'un mot-clé définissant la visibilité de la classe. Une classe est définie aussi par le nom qualifié complet *Cfn*. Ce nom indique commence par la lettre L suivie par le nom de son *package* et du nom de la classe *c* (nous supposons une suite illimitée des noms distincts). La définition d'une classe inclut également le nom complet qualifié de sa super-classe directe (un seul héritage). Une classe peut ne pas avoir une classe mère telle que la classe *Object* ou la classe *Thread*. Ce type de super-classes sont symbolisées par $\top$. L'ensemble des interfaces implémentées *Intf* (si existe), les champs *Fld* et les méthodes *Mtd* font partie de la définition d'une classe.

Une interface est spécifiée par son nom complet *Intf*, ses indicateurs d'accès, un ensemble de super-interfaces *Sinf* (0 ou plusieurs), ses méthodes abstraites, qui consistent en leurs signatures de méthodes *MtdSign* et ses champs constants. La définition d'un champ comprend son nom *Fn*, ses indicateurs d'accès et un type *Type*. Un type primitif représente un champ statique, tandis qu'un type de classe *Cfn* correspond à un champ d'instance.

La définition d'une méthode comprend un ensemble d'indicateurs d'accès qui détermine sa portée, suivi par la signature, le nombre de registres locaux (de type entier) sur lesquels elle agit désigné par *Loc*, et une séquence d'instructions étiquetées *LabelInst* représentant le corps de la méthode. Une étiquette ou un compteur du programme est un label du type entier qui pointe sur une instruction donnée dans une méthode *M*. Une étiquette 0 pointe toujours sur la première instruction dans *M*. La deuxième instruction est étiquetée 1 et ainsi de suite jusqu'à la dernière instruction dans *M* (*return* ou *return-void*). La signature d'une méthode comprend son nom *M*, le type d'argument(s) *Type* (si existe) et le type de retour *retType* qui peut être un type void, primitif ou de classe.

Smali$^+$ comprend les instructions qui capturent des fonctionnalités principales de bytecode Dalvik selon une étude élaborée dans [92], portant sur 1 700 applications Android. Comme expliqué dans le premier chapitre, Smali comprend de nombreuses variantes d'instructions qui sont "sémantiquement" similaires mais qui ne diffèrent que par le nombre de bits réservés aux opérandes et le type d'opérande dans l'opérateur. À titre d'exemple, on trouve dans Dalvik, 9 variantes de l'instruction *move*, qui ont théoriquement la même fonctionnalité, qui consiste à déplacer une valeur à partir d'un registre source vers un registre destination, mais qui diffèrent par le nombre de bits utilisé pour représenter les indices des registres. Parmi ces instructions, nous citons *move* vA, vB ; *move/from*16 vAA vBBBB et *move/16* vAAAA, vBBBB. Les registres vA, vAA et vAAAA exigent, respectivement, 4, 8 et 16 bits. Avec la même instruction *move*, on trouve également des variantes ayant différents types d'opérandes dans l'opérateur, par exemple *move* et *move-object*, qui visent à déplacer le contenu d'un registre contenant, respectivement, une valeur et un objet. Les différentes variantes citées ont été généralisées dans *Smali*$^+$ en une seule instruction *move* pour tous types d'opérande (tout en le permettant de varier), ainsi que pour tous nombres de bits réservés pour ce dernier. Le processus de généralisation est présumé dans Table 4.2. Le tableau représente les exemples de quelques variantes Smali et leurs instructions *Smali*$^+$ généralisées correspondantes.

Opcode	Instructions Smali	Instructions $Smali^+$ généralisées
01	*move*	***move***
02	*move/from16*	
03	*move /16*	
04	*move-wide*	
05	*move-wide/from16*	
06	*move-wide/16*	
07	*move-object*	
08	*move-object/from16*	
09	*move-object/16*	
0f	*return*	***return***
10	*return-wide*	
11	*return-object*	
28	*goto*	***goto***
29	*goto/16*	
2A	*goto/32*	
32	*if-eq*	$\boldsymbol{if}_{\otimes}$
32	*if-ne*	
34	*if-lt*	
35	*if-ge*	
36	*if-gt*	
37	*if-le*	
38	*if-eqz*	
39	*if-nez*	
3a	*if-ltz*	
3b	*if-gez*	
3c	*if-gtz*	
3d	*if-lez*	
6e	*invoke-virtual*	***invoke-virtual***
74	*invoke-virtual/range*	
71	*invoke-static*	***invoke-static***
77	*invoke-static/range*	

TABLE 4.2 – Généralisation des instructions Smali

La plupart des instructions présentées agissent sur des registres. Le nom d'un registre est désigné par Rg. Ce nom sera spécifié de la même manière que dans Smali. La notation v pour les registres locaux, p pour les registres paramètres, $p0$ un registre paramètre spécial destiné pour les références et la notation ret est adaptée pour le registre de retour.

Les instructions considérées incluent le saut inconditionnel et conditionnel avec, respectivement, les instructions *goto* et $if_{\otimes}$. L'instruction *move* exprime le déplacement d'une valeur à partir d'un registre source vers un registre destination. Les deux registres peuvent contenir une valeur dans *Val*, un champ d'instance $l.f$ dans H ou un champ statique $Cfn.f$ de la classe *Cfn*. Table 4.3 donne les définitions de ces valeurs.

Les instructions *binop* et *unop* expriment respectivement des opérations binaires et unaires et agissent sur des registres. Nous considérons également les instructions exprimant la création d'un nouvel objet d'une classe *Cfn* avec *new-instance*. Le registre *Rg* dans ce cas est un registre destination qui recevra la référence de l'objet, une fois créé.

Les instructions *return-void* et *return* expriment le retour à partir d'une méthode void et non void, respectivement.

L'invocation d'une méthode statique avec *invoke-static* fait référence au nom de la méthode, aux types et nombre d'arguments, au type de retour et aux registres (inclut dans la signature *MtdSign*), le nom complet de sa classe et un ensemble de registres qui coorespond aux paramètre de la méthode (si existe). Les méthodes d'instance sont dynamiquement invoquées avec l'instruction *invoke-virtual*. Cette instruction comprend toujours un registre additionnel pour la référence de la classe de l'instance.

4.3.2 Sémantique

Table 4.3 définit les domaines utilisés par notre sémantique opérationnelle. Une exécution séquentielle est modélisée avec une configuration locale désignée par σ. Elle décrit l'état complet d'un programme Android séquentiel. Une configuration locale σ est un triplet incluant une pile d'appel C_s, un tas H et un tas statique S.

(Configuration locale)	σ	$::= \langle C_s, H, S \rangle$
(Pile d'appel)	C_s	$::= \epsilon \mid F_m \mid C_s :: C_s$
(Structure d'une méthode)	F_m	$::= \langle M, Lab, R \rangle$
(Tableau de registres)	R	$::= (Rg \mapsto (Val \mid L))^*$
(Tas)	H	$::= \epsilon \mid (L \mapsto (Obj \mid Arr))^*$
(Objet)	Obj	$::= (Cfn, (Fn \mapsto Val)^*)$
(Tableau)	Arr	$::= ArrayType[^* Val$
(Tas statique)	S	$::= \epsilon \mid (Fn \mapsto Val)^*$
(Valeurs)	Val	$::= Type \mid \perp$
(Référence)	L	$::=$ **heap locations** | **null**

TABLE 4.3 – $Smali^+$: Domaines sémantiques du programme séquentiel

La pile d'appel C_s permet de garder une trace de toutes les méthodes invoquées dans le programme. Elle contient le nom de la méthode m, une étiquette vers une instruction i et l'ensemble de registres R utilisé par la méthode (incluant les registres locaux, de paramètres et de retour). Ces champs sont configurés dans la structure Fm. L'invocation d'une méthode résulte en *empilement* de sa structure Fm dans la pile C_s, tandis que sa terminaison (ou retour) se traduit en *dépilement* de la pile. De cette façon, le sommet de la pile correspond toujours à la structure Fm de la méthode qui est en cours d'exécution.

L'ensemble de registres R est un Map entre les noms de registre et valeurs. Les valeurs peuvent être des primitives ou des emplacements dans le tas. Un tas H est un Map entre emplacements l (nous supposons un nombre arbitraire d'emplacements uniques) et objets Obj ou tableaux Arr. Les objets enregistrent leurs classes et une correspondance entre les champs (de classe) et les valeurs.

Type	**Valeur initiale par défaut**
int	0
long	0|
short	0
char	'$\backslash u$0000'
byte	(byte) 0
float	0.0f
double	0.0d
Obj	null
boolean(int)	false (0)

TABLE 4.4 – Valeurs par défaut des types primitifs

Table 4.5 présente les règles sémantiques des instructions définies syntaxiquement dans Table 4.1.

Une règle sémantique est écrite sous la forme d'une règle logique $\frac{condition}{conclusion}$, qui s'écrit comme une série de conditions préalables au-dessus d'une ligne horizontale et la conclusion en dessous. La relation $\sigma \xrightarrow{m(i)} \sigma'$ modélise l'évolution d'une configuration de départ σ en une nouvelle configuration σ' suite à une étape de calcul $m(i)$. $m(i)$ représente l'instruction à la position i dans une méthode m, toujours pour la méthode qui se trouve au sommet de la pile C_s dans σ. Par conséquent, l'instruction $m(i)$ fait partie toujours des conditions dans la règle.

Nous donnons par la suite le sens intuitif de chaque règle.

- R_{goto} : ayant une configuration $<< m, i, R >:: C_s, H, S >$, si l'instruction courante $m(i)$ correspond à $goto\ i'$, alors l'instruction suivante sera celle ayant l'étiquette i', sans affecter les autres valeurs de la configuration.

- $R_{move\text{-}Rg}$, $R_{move\text{-}Sf}$ et $R_{move\text{-}Instf}$: si l'instruction courante $m(i)$ est $move\ v_{des}\ v_{src}$, alors trois règles s'appliquent qui dépendent de la valeur incluse dans le registre destination v_{des}.
  - Si le registre v_{des} contient une valeur dans Val, alors la règle $R_{mv\text{-}reg}$ s'applique. Dans ce cas, v_{des} reçoit la valeur dans le registre source v_{src} et le label i est incrémenté pour avancer à l'instruction suivante ;
  - Si le registre v_{des} contient le nom d'un champ statique f (c'-à-d appartient à $Cfn.Fn$), alors la règle $R_{move\text{-}Sf}$ s'applique. Dans ce cas, le champ dans le tas statique S sera modifié par la valeur dans le registre source v_{src} et le label i est incrémenté pour avancer à l'instruction suivante ;
  - Si le registre v_{des} contient un champ d'instance f ayant la référence l dans le tas (c.-à-d appartient à $L.Fn$), alors la règle $R_{move\text{-}Insf}$ s'applique. Dans ce cas, le champ f ayant la référence l dans le tas H sera modifié par la valeur dans le registre source v_{src} et le label i est incrémenté pour avancer à l'instruction suivante.

- $R_{new\text{-}ins}$: si l'instruction courante $m(i)$ est *new-instance* v cfn, alors la règle fait appel à la fonction $CreateObject()$ pour la création d'un nouvel objet dans le tas H. La fonction $CreateObject()$ prend en paramètre le nom complet de la classe cfn et charge les champs (ayant comme modificateurs *static* ou *final*) à partir de cette classe ou de sa super-classe sc et les initialise via une fonction $init()$. Cette fonction prend l'ensemble des champs et les initialise chacun par sa valeur initiale par défaut comme le montre Table 4.4. Ensuite, la fonction $CreateObject()$ réserve une zone de mémoire ou un emplacement l fraîchement généré dans H. Une fois créé, la fonction retourne l'objet nouvellement créé o' ainsi que son emplacement l'. Ainsi, le registre de destination v reçoit cette référence. La fonction $CreateObject()$ est définie comme suit :

$$CreateObject(cfn) = \begin{cases} (l',\ init(Fields)) & l' \notin dom(H)\ et\ Fields \in cfn\ et\ AccessFlags(Fields) = \{static, final\} \\ CreateObject(sc) & \text{avec sc est la super-classe de } cfn \end{cases}$$

- $R_{b\text{-}op}$ et $R_{u\text{-}op}$: les deux règles calculent une expression binaire ou unaire respectivement des valeurs incluses dans les registres sources, et stockent le résultat dans le registre destination v.

- $R_{if\text{-}true}$ et $R_{if\text{-}false}$: les deux règles modélisent le saut conditionnel suite à l'exécution de l'instruction $if_{\otimes}$. Si la garde est évaluée à *true*, un branchement est effectué vers le label indiqué i' ($R_{if\text{-}true}$), sinon le label est incrémenté pour aller à l'instruction suivante $i+1$ ($R_{if\text{-}false}$).

- $R_{inv\text{-}st}$: la règle s'applique si l'instruction courante est *invoke-static*. Cette instruction invoque la méthode m' à partir de la classe cfn. À partir de ces informations, une fonction $lookup()$ pour la recherche de la méthode appropriée est appelée. Elle cherche une méthode qui correspond à

la signature introduite dans la classe indiquée et en remontant jusqu'à la chaîne de toutes ses super-classes. Une fois localisée, elle retourne la signature de la méthode recherchée avec le numéro de ses registres locaux *loc*. Nous supposons que la classe et la méthode identifiées existent dans l'ancêtre du package et de la classe, avec un tableau de registres locaux. Nous considérons également que tous les contrôles de vérification sont effectués par la machine virtuelle. Par exemple, il est vérifié que la méthode peut être accédée légalement par la classe. La définition de cette fonction se présente comme suit :

$$lookup(MtdSign, cfn) = \begin{cases} m(Type_1, \ldots, Type_n) retType \;\; loc & si \;\; m \in cfn \\ \\ lookup(MtdSign, cfn.sc) & sinon \end{cases}$$

avec sc est la super-classe de *cfn*.

Une fois la méthode localisée, selon le nombre *loc* déclaré par la méthode, des registres locaux sont créés et initialisés à des valeurs indéfinies $\perp$. Ainsi : $(v_j)^{j<loc} \mapsto \perp$. Par exemple, si $loc = 3$ alors 3 registres locaux sont créés, tel que $v_0 \rightarrow \perp,\ v_1 \rightarrow \perp\ et\ v_2 \rightarrow \perp$.

Les registres paramètres reçoivent les valeurs de ceux de la méthode appelante. Une nouvelle structure *Fm* de la méthode appelée est créée à partir de ces informations et empilée au sommet de la pile d'appel C_s. Elle comprend son nom m', une étiquette 0 qui pointe vers sa première instruction et le tableau de registres R'. Le compteur du programme de la méthode appelante est incrémentée afin de reprendre l'exécution à partir de la bonne instruction une fois la méthode appelée retourne.

- $R_{inv\text{-}st}$: dans le cas d'un appel dynamique avec *invoke-virtual*, la classe de la méthode est récupérée à partir de tas par l'intermédiaire de l'emplacement de l'objet *l* qui est transmis au registre v_{ref}. L'exécution de cette instruction affecte la configuration du programme comme avec la règle $R_{inv\text{-}st}$.

- $R_{ret\text{-}nv}$ et $R_{ret\text{-}v}$: les deux règles s'appliquent suite à l'exécution de l'instruction *return* et *return-void*, respectivement. Elles affectent la configuration comme suit : la structure de la méthode en tête de la pile est dépilée et la valeur de retour de la méthode appelée m, incluse dans le registre *ret*, est transmise vers le registre de la méthode appelante m'.

4.4 Exécution concurrente

L'écosystème Android est fortement concurrent, avec plusieurs composants s'exécutant dans la même application au même moment et partageant une partie du tas. La sémantique d'un programme concurrent inclut la sémantique à un thread unique pour chaque thread en cours d'exécution séparément.

$$R_{goto}\ \frac{m(i) = \textbf{goto}\ i'}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i',R>::C_s,\ H,\ S>}$$

$$R_{move\text{-}Rg}\ \frac{m(i) = \textbf{move}\ \ v_{des}\ v_{src} \quad R(v_{des}) \in Val}{<<m,i,R>::C_s\ ,H,\ S> \xrightarrow{m(i)} <<m,i+1,R[v_{des} \mapsto R(v_{src})]>::C_s,\ H,\ S>}$$

$$R_{move\text{-}Sf}\ \frac{m(i) = \textbf{move}\ v_{des}\ \ v_{src} \quad R(v_{des}) = f \in Cfn.Fn}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,\ H,\ S[f \mapsto R(v_{src})]>}$$

$$R_{move\text{-}Instf}\ \frac{m(i) = \textbf{move}\ v_{des}\ \ v_{src} \quad R(v_{des}) = l.f \in L.Fn}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,\ H[l \mapsto (f \mapsto R(v_{src})],\ S>}$$

$$R_{new\text{-}ins}\ \frac{m(i) = \textbf{new-instance}\ v\ \ cfn \quad CreateObject(cfn) = (l', o')}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R[v \mapsto l']>::C_s,\ H[l' \mapsto o'],\ S>}$$

$$R_{b\text{-}op}\ \frac{m(i) = \textbf{binop}_{\oplus}\ \ v\ v_1\ v_2}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R[v \mapsto R(v_1) \oplus R(v_2)]>::C_s,\ H,\ S>}$$

$$R_{u\text{-}op}\ \frac{m(i) = \textbf{unop}_{\odot}\ v\ \ v_1}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R[v \mapsto \odot(R(v_1))]>::C_s,\ H,\ S>}$$

$$R_{if\text{-}true}\ \frac{m(i) = \textbf{if}_{\otimes}\ v_1\ v_2\ \ i' \quad [R(v_1) \otimes R(v_2)]_B = \textbf{true}}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i',R>::C_s,\ H,\ S>}$$

$$R_{if\text{-}false}\ \frac{m(i) = \textbf{if}_{\otimes}\ v_1\ v_2\ \ i' \quad [R(v_1) \otimes R(v_2)]_B = \textbf{false}}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,\ H,\ S>}$$

$$R_{inv\text{-}st}\ \frac{\begin{array}{c} m(i) = \textbf{invoke-static}\ cfn\ \ m'(t_1,\ldots,t_n)retT\ \ v_1,\ldots,v_n \\ lookup(m'(t_1,\ldots.t_n)retT,\ cfn) = m'(t1_1,\ldots,t_n)retT\ \ loc \\ R' = \{(v_j)^{j<loc} \mapsto \bot, p_1 \mapsto R(v_1),\ldots,p_n \mapsto R(v_n)\} \end{array}}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m',0,R'>::<m,i+1,R>::C_s,\ H,\ S>}$$

$$R_{inv\text{-}inst}\ \frac{\begin{array}{c} m(i) = \textbf{invoke-virtual}\ v_{ref}\ \ m'(t_1,\ldots,t_n)retT\ \ v_1,\ldots,v_n \\ R(v_{ref}) = l\ \ \ H(l) = (cfn,(f \mapsto v)^*) \\ lookup(m'(t_1,\ldots,t_n)retT, cfn) = m'(t_1,\ldots,t_n)retT\ \ loc \\ R' = \{(v_j)^{j<loc} \mapsto \bot, p_0 \mapsto l, p_1 \mapsto R(v_1),\ldots,p_n \mapsto R(v_n)\} \end{array}}{<<m,i,R>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m',0,R'>::<m,i+1,R>::C_s,\ H,\ S>}$$

$$R_{ret\text{-}nv}\ \frac{m(i) = \textbf{return}\ v}{<<m,i,R>::<m',i',R'>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m',i',R'[ret \mapsto R(v)]>::C_s,\ H,\ S>}$$

$$R_{ret\text{-}v}\ \frac{m(i) = \textbf{return-void}}{<<m,i,R>::<m',i',R'>::C_s,\ H,\ S> \xrightarrow{m(i)} <<m',i',R'[ret \mapsto R(ret)]>::C_s,\ H,\ S>}$$

TABLE 4.5 – *Smali*$^+$: Règles sémantiques du programme séquentiel

Une configuration globale $\Sigma =< C_s, S_{rbl}, H, S >$ modélise l'état complet d'une application Android dans son implémentation de bas niveau. La configuration est un quadruplet. Le premier attribut correspond à la pile d'appel du thread exécutant C_s, le deuxième attribut est un ensemble de threads prêts à s'exécuter (ou à être sélectionnés par l'ordonnanceur) S_{rbl}, le troisième attribut est le tas H, et le quatrième attribut est le tas statique S.

Chaque thread dans le programme a sa propre pile d'appel C_s pour les méthodes invoquées, leurs arguments et variables locales, ayant la même syntaxe utilisée lors de la définition d'un programme séquentiel, présentée dans Table 4.3. H et S sont des tas dynamiques et statiques qui sont partagés entre tous les threads du programme. Aucun changement n'est effectué sur ces deux structures de données.

Définition 4.4.1. *(Classe thread)*

Une classe Cl est dite classe Thread si et seulement si elle est une instance de la classe Java Thread. Sa super classe Sc correspond soit au nom complet de la classe supérieure Thread (Cfn = $LJava/lang/Thread;$) ou une autre classe qui hérite de cette classe.

4.4.1 Syntaxe des attributs

Table 4.6 représente la syntaxe de différents attributs utilisés pour formaliser un programme Smali à multi-thread.

— S_{rbl} contient un ensemble de piles d'appel, dont chaque pile correspond à un thread dans le programme. La pile contient toujours dans un tableau de registre R, un registre spécial dénoté par $p0$ pour la référence du thread.

— La définition d'un objet Obj a subi quelques modifications pour s'adapter au nouveau contexte multi-thread. Nous avons ajouté quelques champs liés à la synchronisation des threads sur les objets partagés. Le champ acq indique si le moniteur associé à son objet est acquis par un autre thread. Si la valeur de ce champ est égale à la référence d'un thread, ceci signifie que le moniteur est réservé par ce thread. Sinon, une valeur indéfinie $\perp$ signifie que l'objet est libre. La définition de l'objet inclut également l'attribut S_{blck}. Cet ensemble comprend les threads bloqués qui sont en attente de la libération de l'objet en question. Chaque thread dans cet ensemble est présenté par sa pile d'appel. L'ensemble S_{wait} contient les piles d'appels des threads qui sont en attente de notification (c.-à-d. ceux ayant exécuté l'instruction *wait*). L'état initial d'un nouvel objet, dans un contexte multi-thread, sera initialisé comme dans le cas d'un environnement mono-thread. Les primitifs sont initialisés avec leurs valeurs par défaut selon leurs types comme le montre Table 4.4. Les nouveaux attributs sont initialisés comme suit :

  - $acq \mapsto \perp$, initialisé à une valeur indéfinie, ce qui signifie qu'initialement son objet est dans un état libre ;
  - $S_{blck} \mapsto \emptyset$, un ensemble vide de threads bloqués, ce qui signifie qu'initialement aucun thread n'attend la libération du moniteur de l'objet en question ;

- $S_{wait} \mapsto \emptyset$, un ensemble vide de threads en attente de notification.

— Chaque objet thread est doté d'un champ booléen *finished* indiquant si le thread a terminé son exécution ou non, un ensemble de threads en attente de le rejoindre S_{join} et un attribut appelé *State* indiquant son état actuel. L'état "$Running(t_s)$" représente un thread au cours d'exécution, avec t_s est le temps d'exécution alloué pour ce thread. Le statut "*Runnable*" signifie que le thread est prêt à s'exécuter et correspond à l'état initial d'un thread nouvellement créé (pas encore démarré).

  Les attributs des threads sont initialisés comme suit :

  - $finished \mapsto false$
  - $S_{join} \mapsto \emptyset$
  - $State = Runnable$

Σ	$::= \langle C_s, S_{rbl}, H, S \rangle$	(Configuration globale)
S_{rbl}	$::= \emptyset \mid C_s \mid \{S_{rbl}, S_{rbl}\}$	(Ensemble des threads exécutables)
Obj	$::= (Cfn,\ (Fn \mapsto Val)^*,\ acq \mapsto Val,\ S_{blck} \mapsto S_b,\ S_{wait} \mapsto S_w)$	(Objet)
Th	$::= (Cfn,\ (Fn \mapsto Val)^*,\ finished \mapsto \textbf{booelan},\ S_{join} \mapsto S_j,\ State)$	(Objet thread)
S_b	$::= \emptyset \mid \{C_s\} \mid \{S_b \cup S_b\}$	(Ensemble des threads bloqués)
S_w	$::= \emptyset \mid \{C_s\} \mid \{S_w \cup S_w\}$	(Ensemble des threads en attente)
S_j	$::= \emptyset \mid \{C_s\} \mid \{S_j \cup S_j\}$	(Ensemble des threads à joindre)
$S_{wait}, S_{blck}, S_{join}$	$::= \textbf{string}$	(Noms)

TABLE 4.6 – $Smali^+$: Domaines sémantiques d'un programme concurrent

Notre formalisme couvre les méthodes principales de l'API Java Thread [150] ainsi que d'autres méthodes de la classe Java Object [151]. Dans ce qui suit, nous définissons les macro-instructions qui couvrent les méthodes de l'API Java Thread. Ces APIs incluent la méthode *start*() pour le lancement d'un thread, et *join*() pour joindre un thread. Dans les deux cas, les registre v_{ref} contiennent la référence du thread en question.

Nous définissons également des macro-instructions qui couvrent les méthodes de l'API Java Object liées à la signalisation comme *notify*() pour réveiller un seul thread qui attend le moniteur de l'objet en question, *notifyAll*() pour réveiller tous les thread qui sont en attente du moniteur de cet objet et la méthode *wait*() qui fait en sorte que le thread actuel attende qu'un autre thread invoque la méthode *notify*() ou *notifyAll*() pour l'objet référé. Nous donnons aussi la sémantique des instructions Dalvik relatives à la synchronisation des threads avec les instructions *monitor-enter* et *monitor-exit*. Les registres v_{ref} contiennent dans ces cas, la référence de l'objet en question. La syntaxe de ces instructions et des macro-instructions est illustrée dans Table 4.7.

Inst	::=	**start** *Rg*	(lancer le thread dans *Rg*)
	|	**monitor-enter** *Rg*	(acquérir le moniteur de l'objet dans *Rg*)
	|	**monitor-exit** *Rg*	(libérer le moniteur de l'objet dans *Rg*)
	|	**join** *Rg*	(joindre le thread dans *Rg*)
	|	**wait** *Rg*	(libérer le moniteur de l'objet dans *Rg* et suspendre le thread courant)
	|	**notify** *Rg*	(signaler un seul thread parmi ceux qui sont en attente de l'objet dans *Rg*)
	|	**notifyAll** *Rg*	(signaler tous les threads qui sont en attente de l'objet dans *Rg*)

TABLE 4.7 – $Smali^+$: Syntaxe du programme concurrent

4.4.2 Sémantique

Table 4.8 présente les règles sémantiques liées au lancement (R_{start}) et l'ordonnancement des threads (R_{select} et R_{stop}). Afin de simplifier la lecture des règles sémantiques, nous indiquerons seulement les valeurs des champs qui sont pertinentes (c.-à.d. que la règle en question utilise). Les autres valeurs seront notées "_".

- R_{start} : si l'instruction $m(i)$ correspond à la macro-instruction $start\ v_{ref}$ alors son exécution est gérée comme suit :
  - Appel implicite à la méthode $run()$ du thread référencé dans v_{ref}. Cette méthode sera exécutée dans ce thread séparément une fois sélectionné par l'ordonnanceur. Une procédure $lookup()$ pour la recherche la méthode $run()$ du thread en question est déclenchée tout d'abord ;
  - Création d'une nouvelle pile d'appel séparée pour le nouveau thread constituée de la structure $F_m =< @run, 0, R' >$. Cette dernière contient toutes les informations sur la méthode $run()$ de ce thread : le nom de la méthode ; un pointeur vers la première instruction de la méthode $run()$ et un nouveau tableau de registre R'. Tous les registres locaux dans R' sont initialisés à des valeurs indéfinies. Il n'y aura pas de passage de valeurs entre registres de paramètres puisque la méthode $run()$ est sans paramètres. Cependant, le registre paramètre $p0$ reçoit la référence du thread l associé à cette méthode ;
  - Ajout de la nouvelle pile d'appel créé pour le thread lancé à l'ensemble des threads prêts à s'exécuter S_{rbl}. Lorsqu'il aura la chance de s'exécuter, sa méthode cible $run()$ sera exécutée.
- R_{select} : la configuration de départ $< \epsilon,\ S_{rbl}, H, S >$ signifie qu'il n'existe aucun thread en cours d'exécution, dans ce cas la règle R_{select} s'applique. Une fonction *selectThread()* prend l'ensemble des threads qui sont prêt à s'exécuter S_{rbl} et retourne la pile d'appel du thread sélectionné $F_m :: C_s$ ainsi que le temps d'exécution t_s alloué à ce thread. La référence de ce thread est récupéré du registre $R(p0)$ à partir de sa pile. Ensuite, son état $State$ est mis à jour à $Running(t_s)$.

Ce thread sera aussi enlevé de la liste S_{rbl} et placé dans la première position de la configuration pour qu'il commence l'exécution. La relation $\xrightarrow{\tau}$ exprime l'exécution d'une action silencieuse τ.

- R_{stop} : cette règle s'applique si la tranche de temps t_s, allouée pour l'exécution d'une tâche par le thread exécutant, est expirée. Nous modélisons l'aspect temporel dans notre formalisme par la fonction *clock*(), qui représente le chronomètre adopté par l'ordonnanceur pour chaque tâche affectée à un thread. La condition $clock() > t_s$ exprime le fait que le temps t_s est fini. Ce thread sera donc arrêté dans un mode monitoring M (c.-à-d. un mode où le temps d'exécution d'un thread est sous surveillance) et ensuite ajouté à l'ensemble S_{rbl}.

$$R_{start}\ \frac{\begin{array}{c} m(i) = \textbf{start}\ v_{ref} \\ R(v_{ref}) = l \quad H(l) = (cfn, \_,\ finished \mapsto false,\ \ S_{join} \mapsto \emptyset\ State \mapsto \text{"}Runnable\text{"}) \\ lookup(run()void, cfn) = run()void\ \ loc \\ R' = \{(v_j)^{j<loc} \mapsto \bot, p_0 \mapsto l\} \end{array}}{<<<m,i,R>::C_s,\ S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i{+}1,R>::C_s,\ S_{rbl} \cup \{<@run,0,R'>\},\ H,\ S>}$$

$$R_{select}\ \frac{\begin{array}{c} selectThread(S_{rbl}) = (F_m :: C_s,\ t_s),\ \ F_m =< m, i, R > \\ R(p_0) = l \end{array}}{<\epsilon,\ S_{rbl},\ H,\ S> \xrightarrow{\tau} <F_m::C_s,\ S_{rbl} \backslash \{F_m::C_s\},\ H[l \mapsto (state \mapsto \text{"}Running(t_s)\text{"})],\ S>}$$

$$R_{stop}\ \frac{\begin{array}{c} R(p_0) = l \\ H(l) = (cfn, \_,\ \ finished \mapsto \_,\ \ \ S_{join} \mapsto \_\ State \mapsto \text{"}Running(t_s)\text{"}) \\ clock() > t_s \end{array}}{<<m,i,R>::C_s,\ S_{rbl},\ H,\ S>_{\mathbf{M}} \xrightarrow{\tau} <\epsilon,\ S_{rbl} \cup \{C_s\},\ H,\ S>_{\mathbf{M}}}$$

TABLE 4.8 – Sémantique du programme concurrent : ordonnancement

Table 4.9 présente les règles relatives à la synchronisation. Deux conditions sont à vérifier. La première est liée à l'accès mutuel exclusif aux objets partagés dans le tas par différents threads. La seconde concerne la coopération entre ces threads. La coopération est modélisée par un ensemble de threads qui attendent une notification suite à la libération de l'objet en question. Le seul thread qui s'exécute et qui détient le moniteur se trouve dans une section critique.

- $R_{acq-mntr}$ et $R_{mntr-block}$: représentent un thread qui tente d'accéder à la section critique en acquérant le moniteur de l'objet dont la référence est stockée dans un registre v_{ref} avec l'instruction *monitor-enter*. Il vérifie d'abord si le moniteur est acquis par un autre thread. Si c'est le cas, le thread en cours sera bloqué (condition d'accès mutuel exclusif) et ajouté à l'ensemble des threads bloqués S_{blck} (condition de coopération). Ce cas est modélisé par la règle $R_{mntr-block}$. Sinon ($acq \mapsto \bot$), le thread courant peut acquérir le verrou du moniteur. L'attribut acq est alors mis à jour avec la référence de ce thread dans $R(p0)$. Ce thread peut reprendre son exécution dans la section critique. Ce cas est représenté par la règle $R_{acq-mntr}$.

- $R_{Rls-mntr}$: représente la sémantique de l'instruction *monitor-exit*. Il s'agit d'un thread qui atteint la fin de la section critique en libérant le moniteur acquis pour l'objet dans v_{ref} afin qu'un autre thread prenne le relais, ce qui remplit parfaitement la condition de coopération. Le thread courant doit d'abord posséder le moniteur de cet objet (c.-à-d acq est égale à sa référence l'). Une fois cette condition est remplie, l'attribut acq est mis à jour à une valeur indéfinie (indiquant que l'objet est désormais libre). Ensuite, tous les threads en attente dans S_{blck} sont retirés de cet ensemble (c.-à-d. débloqués) et ajoutés à l'ensemble des threads qui sont prêts à s'exécuter S_{rbl}. Il appartient à l'ordonnanceur de sélectionner le thread à exécuter (il n'y a pas d'ordre ou de priorité entre les threads bloqués).

- R_{wait} : fournit la sémantique de la macro-instruction *wait*. Un thread pourrait volontairement renoncer à la détention du moniteur avant d'atteindre la fin de la section critique en exécutant la macro-instruction $wait$. Ce thread libère le moniteur de l'objet dans v_{ref} et reste dans un état d'attente (c.-à-d. suspendu ou inactif jusqu'à ce qu'il soit notifié par un autre thread). Le thread appelant (ayant la référence l') doit tout d'abord être propriétaire du moniteur de cet objet (c.-à-d. qu'il doit exécuter $wait$ à l'intérieur d'un bloc synchronisé) puis le quitter. Une fois que le moniteur associé à cet objet est libéré, le thread courant est placé dans l'ensemble des threads en attente de notification S_{wait}. Cette règle libère également tous les threads qui sont bloqués sur cet objet S_{blck}.

$$R_{acq-mntr}\ \frac{m(i) = \textbf{monitor-enter}\ v_{ref} \quad R(v_{ref}) = l \quad H(l) = (cfn, \_,\ acq \mapsto \bot, S_{blck} \mapsto \_,\ S_{wait} \mapsto \_)}{<<m,i,R>::C_s, S_{rbl}, H, S> \xrightarrow{m(i)} <<m,i+1,R>::C_s, S_{rbl}, H[l \mapsto (acq \mapsto R(p_0))], S>}$$

$$R_{mntr-block}\ \frac{m(i) = \textbf{monitor-enter}\ v_{ref} \quad R(v_{ref}) = l \quad R(p_0) = l' \quad H(l) = (cfn,\ \_,\ acq \mapsto l',\ S_{blck} \mapsto S_b,\ S_{wait} \mapsto \_)}{<<m,i,R>::C_s,\ S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i,R>::C_s,\ S_{rbl},\ H[l \mapsto (S_{blck} \mapsto S_b \cup \{<m,i,R>::C_s\})],\ S>}$$

$$R_{Rls-mntr}\ \frac{m(i) = \textbf{monitor-exit}\ v_{ref} \quad R(p_0) = l' \quad R(v_{ref}) = l \quad H(l) = (cfn,\ \_,\ acq \mapsto l', S_{blck} \mapsto S_b, S_{wait} \mapsto \_)}{<<m,i,R>::C_s,\ S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,\ S_{rbl} \cup S_b,\ H[l \mapsto (acq \mapsto \bot,\ S_{blck} \mapsto \emptyset)],\ S>}$$

$$R_{wait}\ \frac{m(i) = \textbf{wait}\ v_{ref} \quad R(p_0) = l'\ R(v_{ref}) = l \quad H(l) = (cfn,\ \_,\ acq \mapsto l',\ S_{blck} \mapsto S_b,\ S_{wait} \mapsto S_w)}{<m,i,R>::C_s,\ S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i,R>::C_s,\ S_{rbl},\ H[l \mapsto (acq \mapsto \bot,\ S_{blck} \mapsto \emptyset,\ S_{wait} \mapsto S_w \cup \{<m,i,R>::C_s\})],\ S>}$$

TABLE 4.9 – Sémantique du programme concurrent : synchronisation

Table 4.10 présente les règles R_{notify} et $R_{notifyAll}$ exprimant le mécanisme de signalisation.

- R_{notify} : représente la sémantique de la macro-instruction notify v_{ref}. Il s'agit intuitivement de réveiller un seul thread dans S_{wait}, parmi ceux qui attendent le moniteur de l'objet dans v_{ref}. Ce thread sera choisi aléatoirement via la fonction *random*(). Cette fonction prend l'ensemble S_{wait} et retourne la pile d'appel du thread choisi. Cette dernière sera déplacée de l'ensemble en attente S_{wait} vers l'ensemble exécutable S_{rbl}.

- $R_{notifyAll}$: cette règle est similaire à la règle R_{notify}, à l'exception du fait qu'elle réveille tous les threads dans S_{wait}, qui seront déplacés vers l'ensemble S_{rbl}. Notez que les règles R_{notify} et $R_{notifyAll}$ libèrent, en plus du ou des threads en attente dans S_{wait}, tous les threads bloqués sur le même objet dans S_{blck}. Les deux ensembles ont les mêmes privilèges en ce qui concerne l'acquisition du moniteur. En d'autres termes, les threads en attente n'ont aucune priorité sur les threads potentiellement bloqués qui veulent également se synchroniser sur cet objet.

$$R_{notify}\ \frac{\begin{array}{c} m(i) = \textbf{notify}\ \ v_{ref} \\ R(p_0) = l \quad R(v_{ref}) = l' \\ H(l') = (cfn,\ \_,\ acq \mapsto l,\ S_{blck} \mapsto S_b,\ S_{wait} \mapsto S_w) \\ random(S_w) = C'_s \end{array}}{<<m,i,R>::C_s,\ S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,\ S_{rbl} \cup \{C'_s\} \cup S_b,\ H[l' \mapsto (S_{blck} \mapsto \emptyset, S_{wait} \mapsto S_w \setminus \{C'_s\})],\ S>}$$

$$R_{notifyAll}\ \frac{\begin{array}{c} m(i) = \textbf{notifyAll}\ \ v_{ref} \\ R(p_0) = l \quad R(v_{ref}) = l' \\ H(l') = (cfn,\ \_,\ acq \mapsto l,\ S_{blck} \mapsto S_b,\ S_{wait} \mapsto S_w) \end{array}}{<<m,i,R>:::C_s, S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i+1,R>::C_s,\ S_{rbl} \cup S_w \cup S_b,\ H[l' \mapsto (acq \mapsto \bot,\ S_{blck} \mapsto \emptyset,\ S_{wait} \mapsto \emptyset)],\ S>}$$

TABLE 4.10 – Sémantique du programme concurrent : signalisation.

Table 4.11 présente la sémantique de l'instruction *join*, ainsi d'un thread en état de terminaison (c.-à-d. qu'il a terminé l'exécution de la méthode *run()*).

- $R_{Join\text{-}exec}$ et $R_{Join\text{-}wait}$: parmi les pré-conditions de ces deux règles est de vérifier si le thread référencé dans le registre v_{ref} a terminé son exécution. Si c'est le cas, le thread courant reprend son exécution et avance à l'instruction suivante ($R_{Join\text{-}exec}$). Dans le cas contraire, la règle $R_{Join\text{-}wait}$ s'applique. Le thread courant est bloqué dans S_{join}.

- R_{finish} : cette règle représente un thread qui termine son exécution. L'instruction courante dans ce cas est *return-void* de la méthode *run*. La règle libère tous les threads en attente de ce thread dans S_{join} en les déplaçant vers l'ensemble S_{rbl} et le champ *finished* prend la valeur *true*.

Figure 4.1 illustre le cycle de vie et les différents états qu'un thread peut avoir. Un thread passe d'un état à un autre suite à un événement, comme à sa sélection par l'ordonnanceur ou suite à la libération de moniteur d'un objet particulier. Ce passage est illustré par une flèche étiquetée par les différentes règles sémantiques qui gèrent le cycle de vie du thread.

$$R_{Join\text{-}exec}\ \frac{m(i) = \textbf{join}\ \ v_{ref} \quad R(v_{ref}) = l \quad H(l) = (\mathit{finished} \mapsto \mathit{true},\ \ )}{<<m,i,R>::C_s,\ S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i{+}1,R>::C_s,\ S_{rbl},\ H,\ S>}$$

$$R_{join\text{-}wait}\ \frac{m(i) = \textbf{join}\ v_{ref} \quad R(v_{ref}) = l \quad H(l) = (\mathit{cfn},\ \_,\ \mathit{finished} \mapsto \mathit{false},\ \ S_{join} \mapsto S_j)}{<<m,i,R>::C_s, S_{rbl},\ H,\ S> \xrightarrow{m(i)} <<m,i,R>::C_s,\ S_{rbl},\ H[l \mapsto (S_{join} \mapsto S_j)],\ S>}$$

$$R_{finish}\ \frac{@run(i) = \textbf{return-void} \quad R(p_0) = l \quad H(l) = (\mathit{cfn},\ \_\ \mathit{finished} \mapsto \mathit{false},\ \ S_{join} \mapsto S_j)}{<<@run,i,R>,\ S_{rbl},\ H,\ S> \xrightarrow{@run(i)} <\epsilon,\ S_{rbl} \cup S_j,\ H[l \mapsto (\mathit{finished} \mapsto \mathit{true},\ \ S_{join} \mapsto \emptyset)],\ S>}$$

TABLE 4.11 – Sémantique du programme concurrent : join

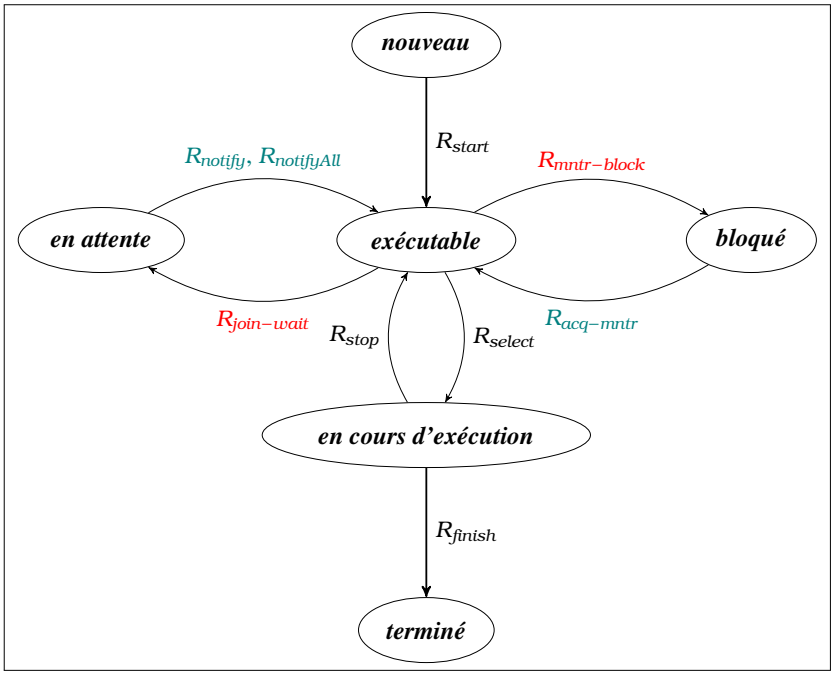


FIGURE 4.1 – États et cycle de vie d'un thread

4.5 Conclusion

Dans ce chapitre nous avons présenté *Smali*$^+$, une sémantique opérationnelle dédiée pour un sous ensemble du langage Smali. La sémantique modélise le comportement d'un programme séquentiel et concurrent. Mise à part la probabilité d'inclure des erreurs, la vérification, l'interprétation du programme et la simulation des programmes types nécessite le développement des outils spécialisés pour la sémantique. Il serait intéressent d'utiliser un environnement qui offre ces outils ainsi qu'un compilateur qui aide à corriger toutes erreurs possible, sans avoir à gaspiller des ressources dans leurs conception et mise en œuvre. En plus du coût supplémentaire engendré, il s'agit d'une solution non vérifiée. Le chapitre suivant présente un environnement qui remplit tout ces critères.

Chapitre 5

$\mathbb{K}$-*Smali* : Sémantique Exécutable pour Smali

5.1 Introduction

« Les outils d'analyse pour tout langage de programmation doivent être basés sur sa sémantique formelle plutôt que sur sa spécification informelle ». Nous approuvons cette affirmation conclue par Stefanescu et al. [152]. Il est pratiquement impossible d'évaluer l'efficacité ou la déficience d'un système ou même de prouver sa validité en l'absence d'une spécification formelle. En générale, une spécification informelle est prédisposée à des problèmes d'ambiguïtés, d'incohérences, voire des parties du langage non définies. De plus, une spécification en langage naturel peut avoir plusieurs interprétations, ce qui fait qu'il n'y a aucune garantie que ces interprétations soient cohérentes avec la spécification ou même entre elles.

Les méthodes de spécification formelles en revanche résout le problème de la variété d'interprétations par la rigueur du formalisme, l'abstraction, la syntaxe et la sémantique mathématiques bien indéfinies. Toutefois, malgré leurs efficacités, les sémantiques formelles sont rarement utilisées directement pour la vérification de programmes et les développements des preuves. La plupart des approches relevées dans la littérature définissent multiples sémantiques pour le même langage, chacune étant conçue dans un objectif différent (la vérification de programmes, l'exécution symbolique, le débogage, etc.), ce qui est moins rentable, demande beaucoup de travail, et parfois sans preuves de solidité par rapport à la sémantique de référence. Par conséquent, même en présence d'une spécification formelle, les cadres pour la définition formelle des langages sont indispensables. Ils offrent un environnement propice à la définition, la vérification et l'analyse par le moyen d'un ensemble d'outils spécialisés intégrés. Ce que nous trouvons un moyen astucieux pour ne pas gaspiller beaucoup de ressources indûment dans la conception et la mise en œuvre d'outils personnalisés, souvent très coûteux.

Néanmoins, ces cadres de définition formelle des langages doivent répondre à certains critères rigoureux. Premièrement, il est important que la sémantique formelle générée soit *lisible* et *compacte*, afin

qu'elle puisse être facilement inspectée. Deuxièmement, cette sémantique devrait être *exécutable* et par la suite elle peut être rigoureusement testée contre une variété des programmes, ce qui augmente sa fiabilité. Troisièmement, l'environnement utilisé doit être suffisamment adapté d'une façon permettant d'effectuer un raisonnement formel et de fournir des preuves automatisées. Il doit aussi être en mesure de modéliser des aspects relatifs au langage, tels que la concurrence. Il est important qu'il soit *générique* afin de ne pas être lié à un langage spécifique, et *modulaire* afin de ne pas avoir à modifier des fonctionnalités déjà formalisées en cas d'ajout de nouvelles. Idéalement, cet environnement devrait inclure son propre analyseur pour le langage pour que le recours à un analyseur externe ne soit pas nécessaire.

Dans le chapitre précédent, nous avons proposé Smali^+, une sémantique opérationnelle modélisant une application au bas-niveau du code Smali. La sémantique a été définie sans l'utilisation d'environnement pareil. Les règles sémantiques ont été développées "manuellement" en suivant les règles de bon sens de construction d'un langage. Toutefois, cette démarche est exposée aux erreurs et ne dispose pas d'un cadre d'ingénierie de sémantiques présentant les caractéristiques mentionnées ci-dessus.

$\mathbb{K}$ *Framework* est un exemple de cadre définitionnel des sémantiques qui répond à la plupart des critères énoncés. Il est armé d'un ensemble d'outils, certains sont utiles pendant le processus de formalisation, d'autres durant la vérification et l'analyse syntaxique. En utilisant l'interprète, par exemple, la sémantique peut être testée contre un nombre potentiellement important de programmes afin de gagner en confiance quant à sa correction. Cela permet d'éviter les erreurs humaines et les omissions qui peuvent glisser, et la possibilité d'échapper à des détails en définissant la sémantique.

Dans ce chapitre, nous nous basons sur la même idée de la formalisation et nous l'améliorons avec l'utilisation de $\mathbb{K}$ *Framework*. Nous appelons le sous-langage formel obtenu $\mathbb{K}$-*Smali*. Dans ce qui suit, nous présentons quelques notations, aspects et fonctionnalités liés à $\mathbb{K}$, qui faciliteront la compréhension du reste de la thèse. Nous nous appuyons sur quelques exemples simples pour décrire la démarche adaptée par $\mathbb{K}$ pour la définition d'un langage formel. Ensuite, nous reprenons la même méthodologie pour la définition de $\mathbb{K}$-*Smali*. Nous finirons le chapitre par montrer l'utilité de cet environnement, non seulement pour la définition formelle d'un langage, mais aussi pour la simulation des programmes types et la vérification de quelques propriétés.

5.2 Introduction à $\mathbb{K}$ *Framework*

$\mathbb{K}$ est un cadre définitionnel de sémantique des langages de programmation, basé sur la logique de réécriture. Figure 5.1 représente son architecture. Différents modules dérivent automatiquement de la même définition formelle du langage. Ils incluent la vérification de modèles (*model-checking*), l'exécution symbolique, la vérification déductive de programmes, etc. Cet environnement offre également une série d'outils, qui ne sont pas spécifiques à un langage mais qui s'appliquent à tout langage ayant une sémantique formelle $\mathbb{K}$. Ces outils comprennent un analyseur syntaxique, un compilateur, un explorateur d'espace d'états et un générateur de cas d'essai. $\mathbb{K}$ supporte également divers backends, tels

que Maude et, expérimentalement, Coq. C'est-à-dire qu'il peut traduire la sémantique d'un langage défini dans $\mathbb{K}$ en définitions Maude ou Coq.

Pour la conception et la spécification d'un langage donné, $\mathbb{K}$ *Framework* impose une méthodologie complète. Elle consiste en trois étapes principales et consécutives : la définition de la syntaxe ; la définition de la configuration ; la définition de la sémantique. Une fois ces étapes terminées et les définitions enregistrées dans des fichiers portant l'extension *k*, la commande *kompile* est utilisée pour compiler chaque définition. Ensuite, la commande *krun* invoque un interprète avec qui des modèles de programme peuvent être simulés et testés. Plusieurs autres options peuvent être ajoutées à cette commande pour générer des modèles sur lesquels des outils de vérification formelle se basent pour l'analyse syntaxique, la vérification formelle déductive et l'exécution symbolique.

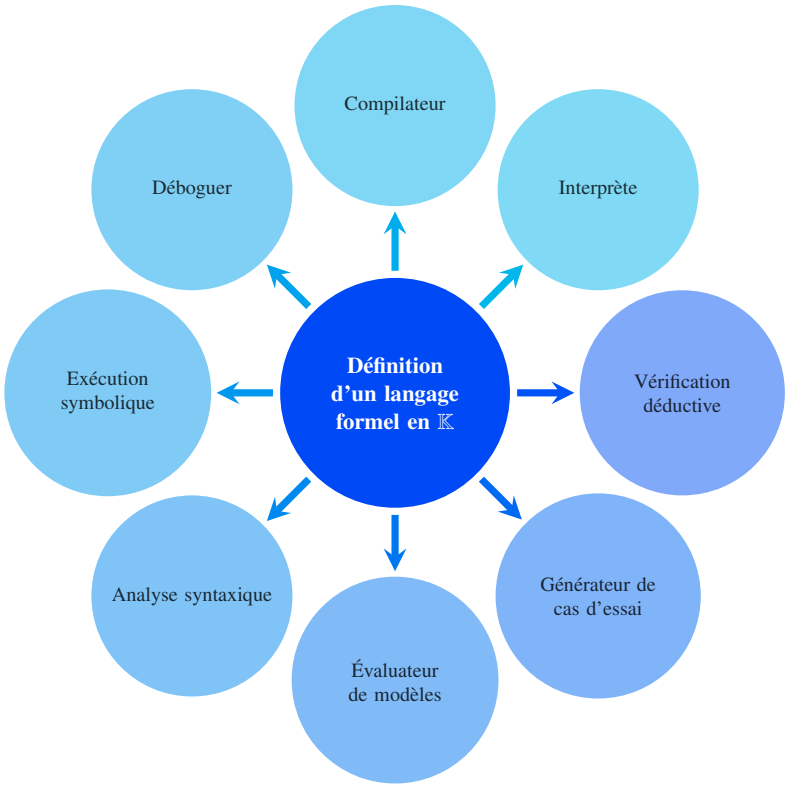


FIGURE 5.1 – Architecture de $\mathbb{K}$ *Framework*

Définition de la syntaxe La définition de la syntaxe d'un langage en $\mathbb{K}$ est contenue dans un module dont le nom est suffixé par le mot "-SYNTAX". Elle utilise la notation conventionnelle Backus-Naur Form (BNF). Listing 5.1 représente un exemple de fichier source $\mathbb{K}$ contenant une définition syntaxique d'un programme *P*. Les non-terminaux commencent par des lettres majuscules et sont précédés du mot-clé *syntax*, tandis que les terminaux sont représentés entre deux guillemets. Par exemple, à la ligne 2, un programme *Pgm* est défini comme une liste d'instructions séparées par un point-virgule.

Les déclarations syntaxiques peuvent être marquées par des attributs. Ces attributs sont spécifiés entre crochets à la fin de chaque définition et sont destinés à fournir des informations supplémentaires à l'analyseur syntaxique. La contrainte de rigueur "strict", par exemple, spécifie comment les arguments de la construction du langage devraient être évalués. À la ligne 3, l'attribut "strict(2)" indique que le deuxième argument (c.-à-d. *Exp*) devrait être évalué en premier. Lorsque aucun nombre n'est fourni avec cet attribut, comme à la ligne 4, toutes les positions des arguments sont considérées comme strictes (c.-à-d. qu'elles sont évaluées dans n'importe quel ordre). D'autres annotations comme "*left*", "*right*", "*token*", "*bracket*" peuvent être aussi utilisées.

En plus des catégories sémantiques prédéfinies (ensembles, listes, mappes, etc.), $\mathbb{K}$ fournit un certain nombre de catégories syntaxiques/types de données intégrés, et des opérations sémantiques pour ceux-ci. Par exemple, les types de base tels que les booléens (*Bool*), les entiers (*Int*), les flottants (*Float*), les chaînes de caractères (*String*) et les identifiants (*Id*) sont prédéfinis dans $\mathbb{K}$.

```
1 module P-SYNTAX
2   syntax Pgm ::= List "{" Inst," ; " "}"
3   syntax Inst ::= Id " " Exp [strict (2)]
4   syntax Exp ::= "mul" "(" Int Int")" [strict]
5 endmodule
```

Listing 5.1 – Exemple de la définition d'une syntaxe $\mathbb{K}$

Définition de la configuration Avant de définir les règles sémantiques, $\mathbb{K}$ exige d'établir la structure de l'état du programme en définissant sa configuration. Celle-ci fournit des informations supplémentaires (outre la syntaxe) sur le langage défini afin de mieux comprendre sa sémantique. Les états de programme dans les configurations $\mathbb{K}$ sont organisés en unités appelées cellules. Les cellules sont étiquetées et éventuellement imbriquées. Chaque cellule contient des informations sémantiques sur le programme, telles que son contexte, sa mémoire, son environnement, etc. Le contenu des cellules diffère en fonction de ces informations et peut contenir différentes structures de données telles que les arbres, les listes, les ensembles et les correspondances (*maps* en anglais).

Les cellules sont définies dans une syntaxe du type XML et introduites par le mot-clé *configuration*. À partir de cette définition, $\mathbb{K}$ peut générer une représentation graphique de la configuration comme celle représentée dans Figure 5.2. Elle correspond à la configuration du programme P dont la syntaxe est donnée dans Listing 5.1. La notation à l'intérieur des cellules représente leur état initial. Par exemple, le "$\bullet_{Map}$ " dans la cellule $\langle\ \rangle_{Memory}$ est la notation de $\mathbb{K}$ pour une correspondance vide. Le point "$\bullet$", lu "rien" et éventuellement marqué de son type en tant qu'indice, comme unité de toutes les structures de données utilisées.

La configuration de P se compose d'une cellule supérieure étiquetée $\top$, contenant deux sous-cellules : une cellule variable ***$PGM*** du type k. La cellule k, par convention, représente toujours une liste de calculs en attente d'exécution, une cellule ***Memory***, représentant une mémoire qui contient une

correspondance entre les variables du programme et leurs valeurs, initialement vide. Le symbole d'astérisque "*" utilisé dans l'étiquette de la sous-cellule *Inst* indique sa multiplicité. Il indique qu'à un moment donné, la cellule peut avoir zéro ou plusieurs instances.

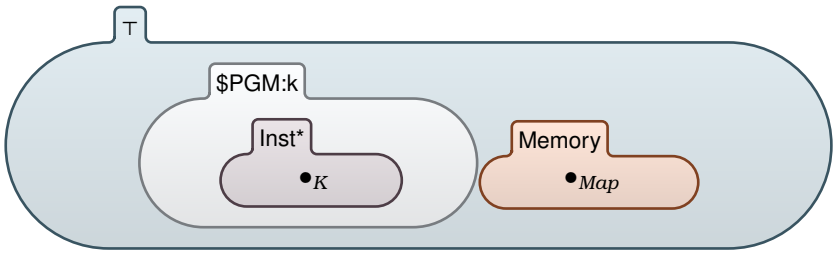


FIGURE 5.2 – Exemple d'une configuration $\mathbb{K}$

Définition de la sémantique Une fois la syntaxe et la configuration définies, il convient de définir les règles sémantiques. En effet, la configuration est initialisée en plaçant le programme cible à sa position spécifiée et en initialisant toutes les autres cellules avec leurs contenus déclarés. À partir de cette étape, les règles sémantiques $\mathbb{K}$ donnant la sémantique du langage (de manière non déterministe et concurrente) se correspondent et s'appliquent, générant tout comportement possible du programme cible.

La définition de la sémantique du langage consiste en un ensemble de règles de réécriture $\mathbb{K}$ qui pilote l'exécution du programme. On peut décrire une règle de réécriture $\mathbb{K}$ comme une transition entre des configurations, qui commence par une configuration contenant le programme original et se termine par une nouvelle configuration maintenant le résultat. Chaque règle dans $\mathbb{K}$ est précédée du mot-clé *rule* et a la forme suivante :

$$\text{rule} \;\; lhs \Rightarrow rhs$$

où lhs représente le côté gauche de la règle et rhs le côté droit. Intuitivement, cela signifie que si lhs est un fragment de l'état actuel de la configuration, alors la règle peut s'appliquer et le fragment correspondant à lhs dans la configuration actuelle serait remplacé par le nouveau fragment de configuration rhs. Listing 5.2 fournit un exemple de définition de la sémantique. Le module P-SEMANTICS représente la définition de la sémantique du programme P. Ce module importe les deux modules P-SYNTAX et INITIALIZE-CONFIGURATION contenant la définition de la syntaxe et de la configuration. La ligne 4 représente la définition d'une règle de réécriture de l'opération de multiplication.

$\mathbb{K}$ fourni également un ensemble d'opérateurs pour l'évaluation des expressions arithmétiques et logiques. Par exemple, les opérateurs $+Int$, $-Int$, $*Int$ sont utilisés pour additionner, soustraire et multiplier des valeurs entiers.

```
1 module P-SEMANTICS
2  imports P-SYNTAX
3  imports INITIALIZE-CONFIGURATION
4  rule <⊤> <PGM> <Inst> I1:Int * I2:Int ⇒ I1 *Int I2 </Inst> </PGM>... </⊤>
5 endmodule
```

Listing 5.2 – Exemple de la définition des règles sémantiques $\mathbb{K}$

Cette règle de réécriture affecte une seule cellule de la configuration du programme *P* (c.-à-d. la cellule *Inst*) comme suit :

$$\text{rule} \quad \left\langle \frac{\text{I1} : \mathit{Int} * \text{I2} : \mathit{Int}}{\mathit{I1} *_{\mathit{Int}} \mathit{I2}} \right\rangle_{\mathit{Inst}}$$

La ligne horizontale exprime une réécriture. Les termes au-dessus de la ligne représentent le côté gauche (*lhs*) de la règle, tandis que ceux au-dessous représentent le côté droit (*rhs*) de la règle. Le reste du contexte de configuration est déduit automatiquement. En effet, dans la définition des règles $\mathbb{K}$, seules les cellules pertinentes sont configurées. Les cellules qui n'ont aucun impact sur une règle R et qui ne sont pas affectées par R n'apparaissent pas explicitement dans la règle. Les cellules tête et queue (potentiellement vides), qui ne sont pas modifiées par une règle, peuvent être dénotées par "...". Remarquez, dans Listing 5.2, l'ellipse "..." utilisée dans la définition de la règle (ligne 3). Elle remplace la partie volatile que la règle ne prend pas en compte et qui correspond à la cellule *Memory*. Graphiquement, ces cellules terminent par une ligne en courbure. Ces caractéristiques rendent les règles de réécriture $\mathbb{K}$ assez compactes et modulaires [153]. Cette option est très utile pour la définition des langages concurrents. Par exemple, dans Listing 5.3, deux threads T1 et T2 sont censés s'exécuter simultanément. La règle permet de créer le premier thread T1 et de le faire tourner jusqu'à ce qu'il se bloque. Ensuite, un deuxième thread T2 est lancé (ligne 2). Grâce à ce raisonnement, les threads sont séparés et les deux sous-modules fonctionnent en parallèle.

```
1  rule <thread>...<K> T1: Proc | T2: Proc ⇒ T1...</K></thread>
2  ( .Bag ⇒ <thread> ... <K> T2: ...</K> </thread>)
```

Listing 5.3 – Exemple d'une exécution concurrente dans $\mathbb{K}$

Comme le montre l'exemple précédent, $\mathbb{K}$ permet l'interprétation du multithreading, mais ne gère pas automatiquement plusieurs fonctionnalités importantes telles que l'ordonnancement, la communication entre les threads et la synchronisation. C'est en fait la responsabilité du concepteur de définir ces caractéristiques à travers la configuration et les règles sémantiques.

Une règle de réécriture $\mathbb{K}$ peut être conditionnelle. Dans ce cas, elle sera accompagnée par une condition qui débute toujours par le mot clé "requires".

5.3 Définition de $\mathbb{K}$-*Smali*

5.3.1 Syntaxe

Listing 5.4 correspond à un fichier source $\mathbb{K}$ contenant la définition de la syntaxe de $\mathbb{K}$-*Smali*. Le module SMALI-SYNTAX fournit les catégories syntaxiques de base du langage ainsi que la syntaxe des instructions sélectionnées.

Suite au processus de désassemblage, toutes les classes Java internes de la source sont séparées des classes qui les incluent, chacune dans un fichier *.smali* séparé. Une application Android est considérée comme une liste de fichiers smali *SmaliFile* et un fichier manifest *ManifestFile*. Ce dernier sert à identifier le point d'entrée de l'application. Sa syntaxe consiste au mot-clé *.manifest* suivi d'une référence de la méthode à partir de laquelle l'application se lance *MethodRef* (ligne 78). La référence est fondamentalement le nom complet qualifié de la classe qui l'inclut ainsi que sa signature (ligne 34). Le point d'entrée du programme est une caractéristique qui n'a pas été prise en compte lors de la formalisation en *Smali*$^+$. Dans ce formalisme, seulement l'ensemble des classes Smali qui constituent le code de l'application a été considéré. Autrement dit, l'identification de la méthode qui lance l'application a été omise de la sémantique.

Chaque classe du fichier Smali est définie par un en-tête de classe *ClassHeader* indiquant toutes les informations relatives à la classe : les éventuels commentaires, son nom complet qualifié, commençant par "L" et se terminant par " ;" (ligne 33), le nom entièrement qualifié de sa super-classe directe (si elle existe), les indicateurs d'accès indiquant sa visibilité, sa classe source Java correspondante (identifiée par le mot-clé *.source*) et enfin un ensemble d'interfaces implémentées. Un commentaire *Comment* est une expression régulière r"<regExp>" qui commence par # et est suivie d'un caractère quelconque (.) zéro ou plusieurs fois (*). L'attribut "token" utilisé lors de la définition d'un commentaire (ligne 10) et du nom complet d'une classe (ligne 33). Il signale que le tri associé sera habité par des valeurs de domaine, qui est un ensemble de valeurs littérales (chaîne et entier). La définition d'une classe inclut également ses champs et ses méthodes.

Une méthode est définie par un ensemble d'indicateurs d'accès qui détermine sa portée, un nom complet, une signature et un corps. La signature d'une méthode est constituée des types d'entrée *MethodIntypes* et des types de retour *MethodRetTypes*. Les champs identifiés par le mot-clé *.field*, des indicateurs d'accès, un nom, un type et une valeur (si elle existe).

Le corps de la méthode est une liste des déclarations étiquetées par un identifiant *Id*, séparées par des espaces. Les déclarations sont soit des directives, soit des instructions. Une directive peut être *.locals* suivie d'un nombre entier, indiquant le nombre de registres locaux alloué par la méthode ou bien la directive *.registers*, indiquant le nombre total de registres dans la méthode (y compris les registres locaux et paramètres). $\mathbb{K}$-*Smali* couvre plus que 40 instructions considérées comme les plus pertinentes pour modéliser un programme Smali.

Les instructions considérées comprennent les sauts inconditionnels et conditionnels avec, respective-

ment, les instructions *goto* et *if*. Tous les branchements sont dirigés vers une étiquette donnée (*:Label*) identifiant l'instruction cible. L'instruction *sparse-switch* utilise un registre à tester et une table de consultation *Switchtab*. Cette table contient une liste triée de paires clé-valeur, où chaque clé est une valeur de test d'une étiquette de cas, et les valeurs sont des offsets de saut. En fonction de la valeur dans le registre, si elle correspond à la valeur clé, elle saute à son offset de saut correspondant. S'il n'y a pas de correspondance dans la table, l'exécution continue avec l'instruction suivante (c.-à-d. le cas par défaut). L'instruction *nop* est une instruction nulle ou une opération vide qui ne correspond à aucune action.

$\mathbb{K}$-*Smali* comprend également les instructions permettant le déplacement d'une constante dont la valeur est une chaîne de caractères ou un entier vers un registre destination avec les instructions *const-string* et *const*. L'échange entre registres est effectué par l'instruction *move* à partir d'un registre source (en deuxième position) vers un registre destination (en première position).

La création d'objets et de tableaux sont représentées respectivement par les instructions *new-instance* et *new-array*. L'instruction *array-length* stocke dans le registre destination donné la longueur du tableau référé dans le deuxième registre. Les instructions arithmétiques telles que l'addition, la multiplication, la soustraction sont exprimées par les instructions *add*, *mul* et *sub*. Le premier registre correspond au registre destination, recevant le résultat. La définition de ces instructions est annoté par l'attribut "left". Cet attribut est utilisé, à titre d'exemple, avec l'opération d'addition pour indiquer qu'elle est associative de gauche à droite (ligne 59).

$\mathbb{K}$-*Smali* comprend aussi des instructions de lecture/écriture à partir des champs statiques (*sget, sput*), des champs d'instance (*iget, iput*) et d'éléments de tableau (*aget, aput*).

L'invocation de méthode fait également partie du sous langage $\mathbb{K}$-*Smali*. L'instruction *invoke-static* invoque une méthode de classe, tandis que l'instruction *invoke-virtual* invoque une méthode d'instance. Les deux instructions ont pratiquement la même syntaxe. La différence entre les deux apparaîtra par la suite lors de la définition de leurs règles sémantiques.

La valeur de retour de la méthode (s'il y en a une) doit être obtenue par l'instruction spéciale *move-result* qui stocke le résultat (la valeur de retour) dans un registre. Les instructions de retour sont exprimées par *retrun* et *return-void* pour un retour à partir d'une méthode void et non void.

Le formalisme inclut aussi les instructions liées au *multithreading*, telles que les instructions de synchronisation *monitor-enter* et *monitor-exit* suivies du nom de registre *RegName*, qui contient la référence de l'objet désignée. Nous allons voir par la suite l'instruction *invoke-virtual*, qui est utilisée également pour le lancement d'un thread. Elle invoque la méthode *start* à partir de la classe *LJava/-lang/Thread ;* et passe comme premier registre paramètre l'objet thread à lancer.

Dans ce qui suit, nous utilisons $\mathcal{P}$ pour désigner un programme $\mathbb{K}$-*Smali* et $\mathcal{A}$ pour désigner l'ensemble des actions (instructions) offertes par $\mathcal{P}$.

```
module SMALI-SYNTAX
 syntax Program     ::= SmaliFiles ManifestFile
 syntax SmaliFiles  ::= List{SmaliFile," "}
 syntax SmaliFile   ::= Class
 syntax Class   ::= ClassHeader Fields Methods
 syntax ClassHeader::= Comments ".class" AccessFlags ClassName SuperClass SourceClass Intf
 syntax SuperClass  ::= Comments ".super" SuperClassName | Empty
 syntax SourceClass::= Comments ".source" String | Empty
 syntax Comments ::= List{Comment," "}
 syntax Comment   ::= r"\\#.*"                                          [token]
 syntax Fields ::= List{Field," "}
 syntax Field ::= Comments ".field" AccessFlags FieldName ":" Type ValueOp
 syntax ValueOp ::= Value | Empty
 syntax Methods ::= List{Method," "}
 syntax Method::=Comments ".method" AccessFlags MethodNameSign MethodBody ".endmethod"
 syntax MethodNameSign ::= MethodName MethodSignature
 syntax MethodSignature ::= MethodInTypes MethodRetType
 syntax MethodInTypes ::= "(" Types ")" | "(" ")"
 syntax MethodRetType ::= Type  | VoidType
 syntax Type::= PrimitiveType | ObjectType | ArrayType
 syntax PrimitiveType ::= "Z" | "B" | "C" | "D" | "F" | "I" | "J" | "S"
 syntax VoidType   ::= "V"                /* void type*/
 syntax ObjectType ::= LName             /* Object reference*/
 syntax ArrayType ::="["PrimitiveType|"["ObjectType|"[ArrayType"
 syntax Value ::= Bool | Int | Float | String
 syntax AccessFlags ::= List{AccessFlag," "}
 syntax AccessFlag::= "public"|"private"|"protected" |"final"|"abstract"|"static"
 syntax ClassName ::= LName
 syntax SuperClassName::= LName
 syntax MethodName ::= Name | "constructor" "<init>"
 syntax FieldName::=Name
 syntax Name   ::= Id
 syntax LName ::= r"L[_a-zA-Z0-9]*[_a-zA-Z0-9]*;"                       [token]
 syntax MethodRef   ::= ClassName "->" MethodNameSignature
 syntax FieldRef    ::= ClassName "->" FieldName
 syntax Parameters ::= List{Parameter,","}
 syntax Parameter   ::= RegName
 syntax MethodBody ::= List{Statement," "}
 syntax Statement   ::= Directive | Instruction
 syntax Directive   ::= ".locals" Int | ".registers" Int
 syntax Instruction::= "goto" ":" Label
                      |":" Label
                      |"nop"
                      |"sparse-switch" RegName "," ":" Switchtab
                      |"const" RegName "," Val
                      |"const-string" RegName "," String
                      |"move" RegName "," RegName
                      |"new-instance" RegName "," ClassName
                      |"new-array" RegName "," RegName "," ArrayType
                      |"array-length" RegName "," RegName
                      |Sget RegName "," FieldRef
                      |Sput RegName "," FieldReference
                      |"iget" RegName "," RegName "," FieldRef
                      |"iput" RegName "," RegName "," FieldRef
                      |"aget" RegName "," RegName "," RegName
                      |"aput" RegName "," RegName "," RegName
                      |"if-eq" RegName "," RegName "," ":" Label
                      |"if-lt" RegName "," RegName "," ":" Label
                      |BinOp RegName "," RegName "," RegName                [left]
                      |UnOp RegName "," RegName
                      |"invoke-static" "{"Parameters"}" "," MethodRef
                      |"invoke-virtual" "{"Parameters"}"","  MethodRef
                      |"move-result" RegName
                      |"retrun-void"
                      |"return" RegName
                      |"monitor-enter" RegName
                      |"monitor-exit" RegName
 syntax Sput   ::= "sput" | "sput-object"
 syntax Sget   ::= "sget" | "sget-object"
 syntax Binop ::= "add" | "sub" | "mul" | "div" |...
 syntax Unop   ::= "neg" | "not" | "int-to-long" |...
 syntax Val   ::= Int
 syntax Switchtab ::= ".sparse-switch" Tablecases ".end sparse-switch"
 syntax Tablecases ::= List {Tablecase," "}
 syntax Tablecase ::= Value "→" ":" Label
 syntax StringId , Label ::= Id
 syntax Empty ::= " "
 syntax ManifestFile::=".manifest" MethodRef
endmodule
```

Listing 5.4 – $\mathbb{K}$-*Smali* : syntaxe

5.3.2 Configuration

Figure 5.3 illustre la configuration $\mathbb{K}$-*Smali* dans un aperçu global. La configuration utilisée pour l'exécution d'un programme Smali consiste en une cellule de niveau supérieur $\top$ contenant quatre cellules principales : ***Threads***, ***Classes***, ***RegisterMethods*** et ***Heap***.

La cellule ***Threads*** représente le comportement concurrent du programme. Elle se compose du thread en cours d'exécution représenté par la sous-cellule *Thread* et d'une liste de threads exécutables (ou prêts à s'exécuter) maintenue dans la sous-cellule *Scheduler*. Toutes les informations nécessaires au *multithreading* (synchronisation, ordonnancement et communication), y compris les détails de l'exécution en cours, se trouvent dans cette sous-cellule. Chaque thread est identifié par un identifiant *Id*, un temps d'exécution *runTime* pour chaque instruction exécutée par le thread, et un statut *status* représentant son état. L'état d'un thread peut être "run" pour un thread qui est en cours d'exécution, une référence d'objet "Ref" si le thread est bloqué en attente de la libération de l'objet *Ref* et "runnable" pour ceux qui sont prêts à s'exécuter et en attente d'être sélectionnés par l'ordonnanceur.

La cellule ***Classes*** accueille une ou plusieurs classe(s). La cellule ***RegisterMethods*** est une cellule indépendante (puisque les registres sont réservés au début de l'exécution de la méthode et libérés à chaque retour). La cellule ***Heap*** représente la partie dynamique principale de l'environnement d'exécution. Elle correspond à une mémoire partagée (ou tas) utilisée pour stocker les objets et les tableaux créés dynamiquement.

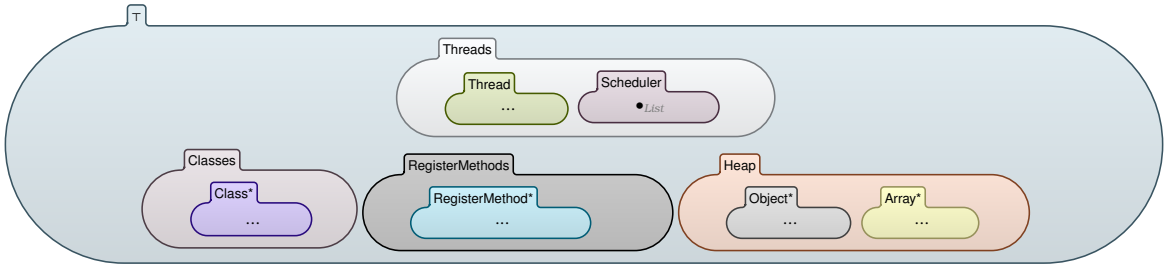


FIGURE 5.3 – $\mathbb{K}$-*Smali* : configuration globale

Figure 5.4 détaille la configuration des sous-cellules. Un thread exécutant est identifié par un identifiant *Id*, une cellule *k* pour le code à exécuter et une cellule *ReturnResult* pour sa valeur de retour.

Chaque classe de la cellule *Class* est définie par un nom de classe entièrement qualifié, le nom complet de sa super-classe directe, un indicateur d'accès indiquant sa visibilité et une cellule *Fieldsclass* faisant correspondre les noms des champs de la classe aux valeurs. La cellule *Class* comprend une cellule *Methods* pour toutes les méthodes (zéro ou plusieurs) de la classe. Une cellule *method* comprend son nom complet, des indicateurs d'accès indiquant son mode d'accès et un corps. Le corps d'une méthode est désigné par la cellule *code* et consiste en une correspondance d'identifiants *Ids* vers des instructions ou directives.

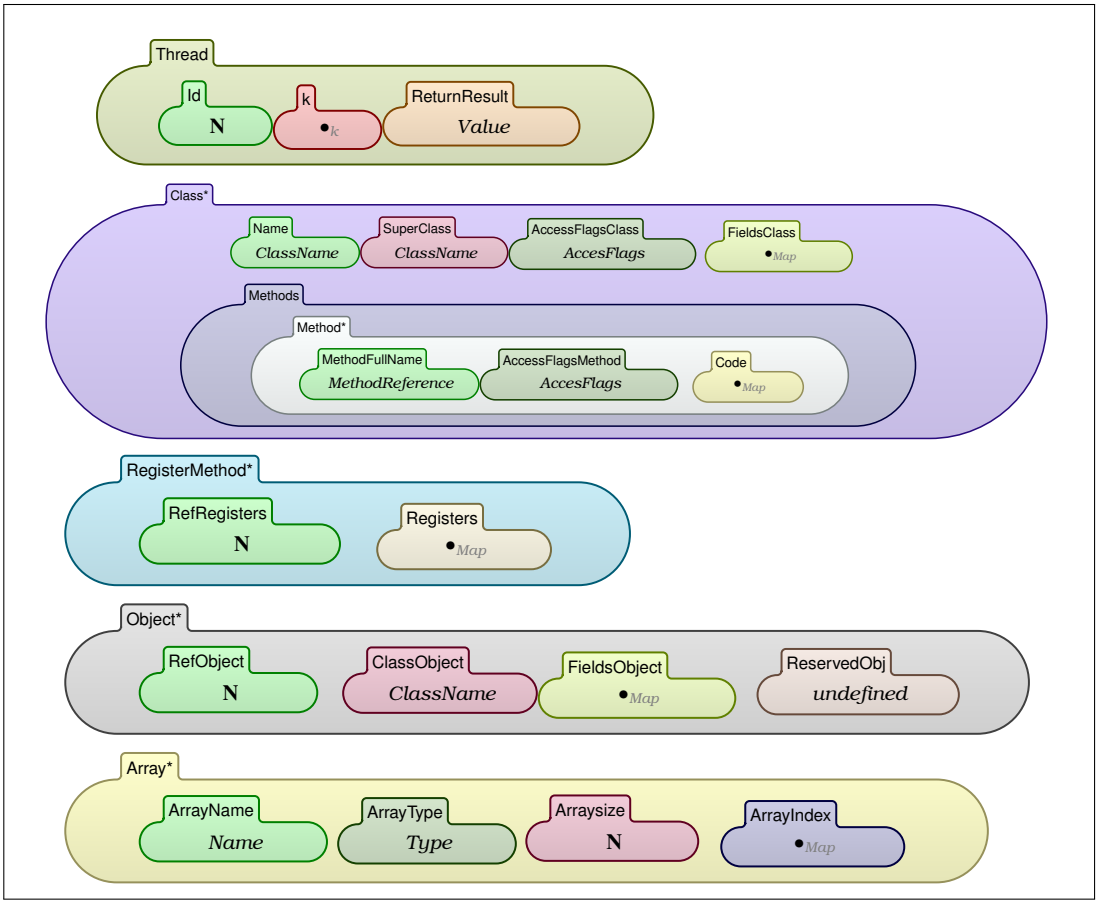


FIGURE 5.4 – $\mathbb{K}$-*Smali* : configuration des sous-cellules

La cellule *RegisterMethod* contient deux sous-cellules, *RegisterRef* portant la référence de registre et *Registers* contenant une correspondance entre les noms de registre et les valeurs. Une cellule *Object* enregistre la référence de l'objet dans le tas, son nom complet de classe, une correspondance des champs (de classe) vers des valeurs, et une cellule *Reservedobject*, initialement indéfinie et utilisée pour la synchronisation des threads. Une valeur "undefined" indique un objet libre (dont le moniteur associé est non acquis par aucun thread), tandis qu'une valeur *Id* désigne un objet réservé par l'identifiant du thread qui détient son moniteur. Enfin, la cellule *Array* enregistre le nom et la taille du tableau, ainsi qu'une forme de correspondance entre les noms d'index et les valeurs.

5.3.3 Initialisation

Nous avons conçu un module appelé "INITIALIZE-CONFIGURATION". Ce module initialise la configuration et remplit chaque cellule de la configuration avec sa valeur correspondante selon la syntaxe définie. Par exemple, pour les cellules de classe, dans chaque classe, les sous-cellules sont remplies avec les informations de la classe (nom, super-classe, champs, etc.), les informations de la méthode et le code. Pour l'initialisation des champs, par exemple, une fonction *defautValue()* initialise les champs avec leurs valeurs par défaut en fonction de leurs types comme, l'illustre Listing 5.5.

```
syntax Value::= "null"|"\u0000"|"undefined"|"0.0d"
syntax Value ::= defaultValue(Type) [function]
rule defaultValue(Ot:ObjectType) ⇒ null
rule defaultValue(Z) ⇒ false
rule defaultValue(C) ⇒ \u0000
rule defaultValue(D) ⇒ 0.0d
rule defaultValue(I) ⇒ 0
rule defaultValue(J) ⇒ 0
rule defaultValue(S) ⇒ 0
rule defaultValue(At:ArrayType) ⇒ undefined
```

Listing 5.5 – $\mathbb{K}$-*Smali* : initialisation des champs

5.3.4 Sémantique

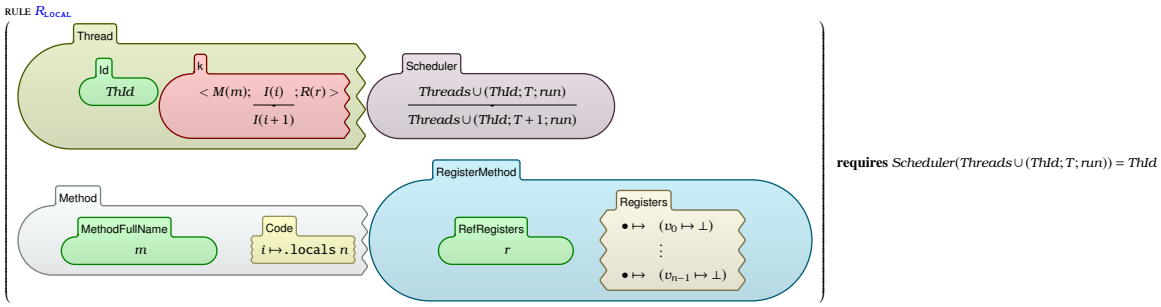
La sémantique opérationnelle de Smali en $\mathbb{K}$ est représentée comme un ensemble de règles de réécriture indépendantes. Elle englobe plus de 50 règles. Dans cette sous-section, nous avons sélectionné les règles qui modélisent les fonctionnalités les plus importantes dans le langage Smali.

Dans chaque règle, nous pouvons capturer trois phases principales d'exécution répétitive : (1) l'exécution de l'instruction sélectionnée dans la sous-cellule *code*, (2) la sélection de l'instruction suivante à exécuter dans la sous-cellule *k*, (3) la vérification que le thread exécutant l'instruction courante, est sélectionné par l'ordonnanceur (retourné par la fonction *Scheduler*()). Cela signifie qu'il doit avoir l'état "run" et le même identifiant. Cette condition est vérifiée par la condition latérale de chaque règle.

Les lignes en courbures dans une cellule signifient que d'autres cellules ont été omises vu qu'elles ne sont pas affectées par la règle en question.

5.3.5 Instructions basiques

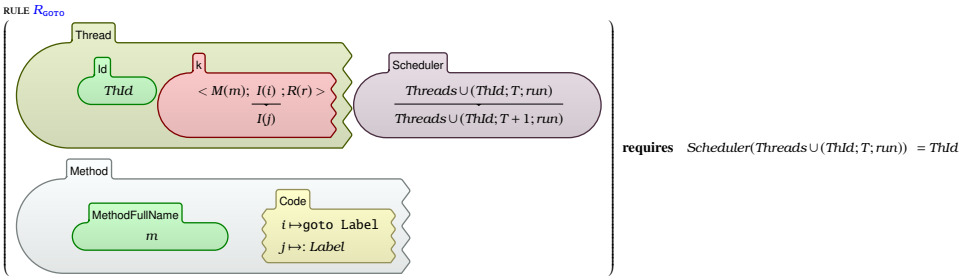
La règle R_{local} représente la sémantique de la directive *.locals*. Suite à l'exécution de cette action, *n* registres locaux sont créés dans la sous-cellule $\langle\ \rangle_{Registers}$ (la nouvelle entrée • expriment la création d'un nouveau registre) et initialisés à des valeurs indéfinies. Dans $\langle\ \rangle_k$, l'instruction courante est réécrite par l'instruction suivante.



Les règles $\mathbb{K}$ qui traitent un branchement (conditionnel ou non), avec les instructions *goto* ou *if* vers un label, mettent à jour le pointeur du programme vers ce label. Cela peut être capturé dans la cellule

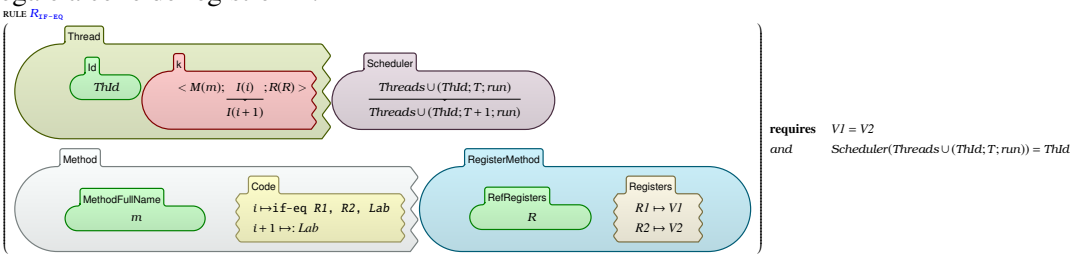
$\langle\ \rangle_k$ où l'instruction courante est réécrite par une nouvelle instruction *j*, qui correspond à l'instruction *:Label*. Afin d'exécuter l'instruction étiquetée par ce label, la règle R_{Label} est déclenchée.

La règle R_{GOTO} permet de sauter inconditionnellement vers l'étiquette *:Label*.

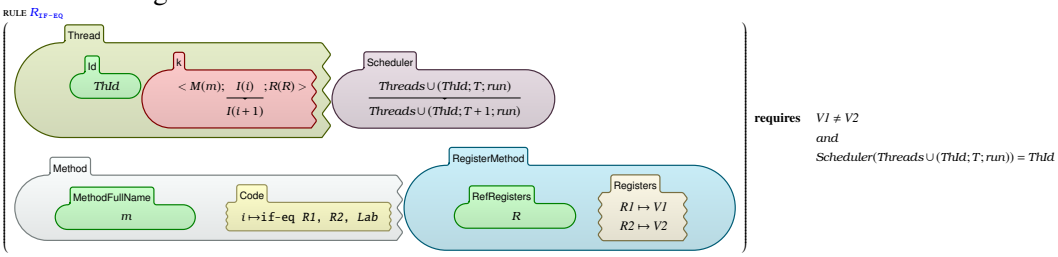


Les deux règles suivante représentent la sémantique de l'instruction *if-eq*.

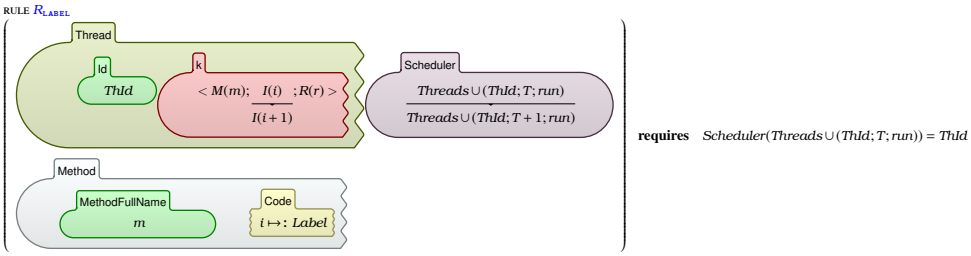
La règle $R_{\texttt{if-eqTRUE}}$ permet de brancher vers le label *Lab* dans le cas où la valeur de regitre *R1* est égale à celle de registre *R2*.



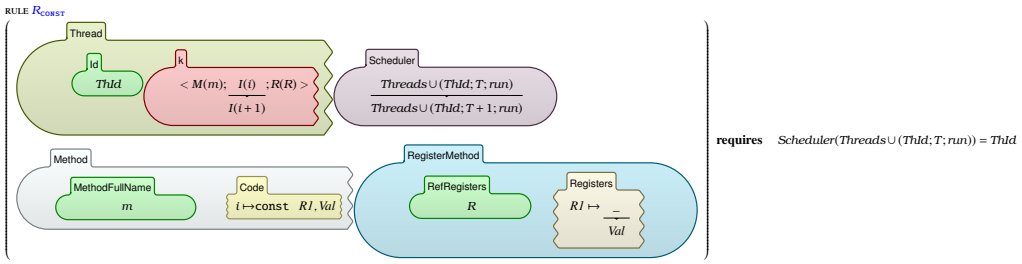
La règle $R_{\texttt{if-eqFalse}}$ permet d'avancer à l'instruction suivante si la valeur de registre *R1* est différente de celle de registre *R2*.

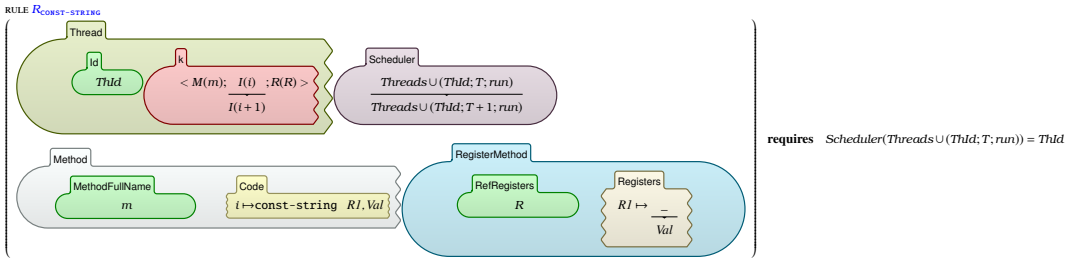


La règle R_{LABEL} permet d'avancer à l'instruction qui suit l'étiquette (ou le label).

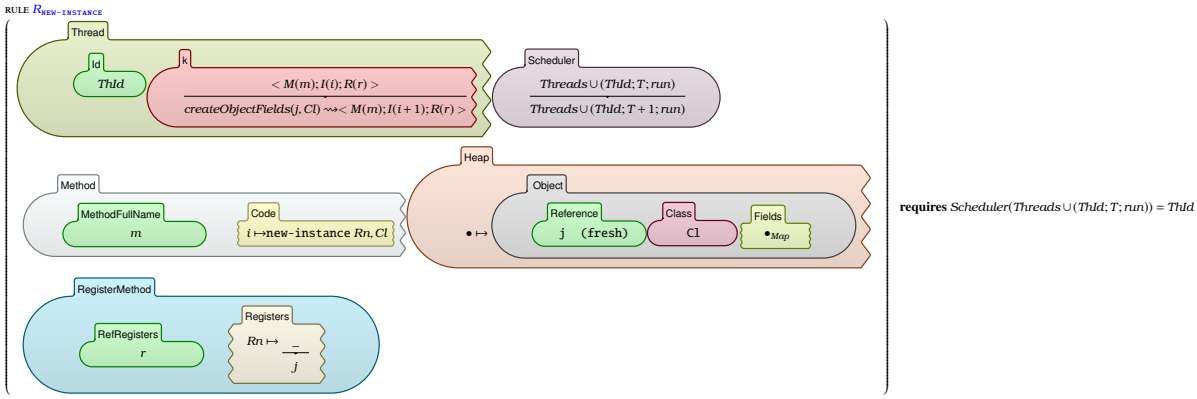


Les deux règles suivantes représentent respectivement la sémantique des instructions *const* et *const-string*.

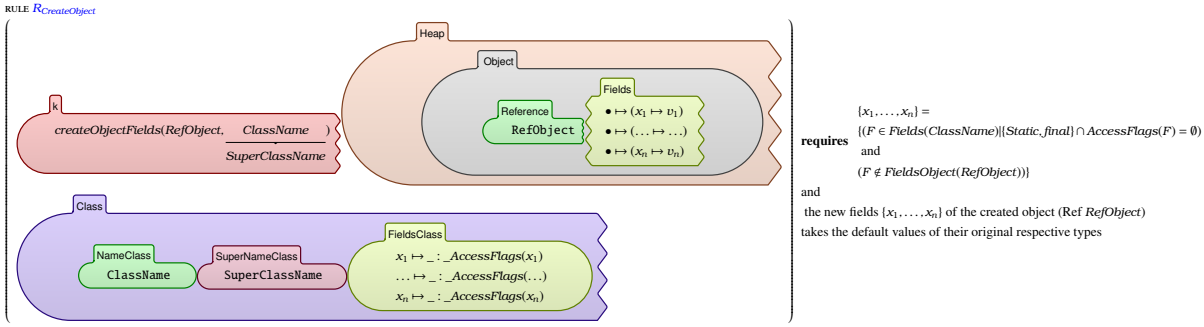


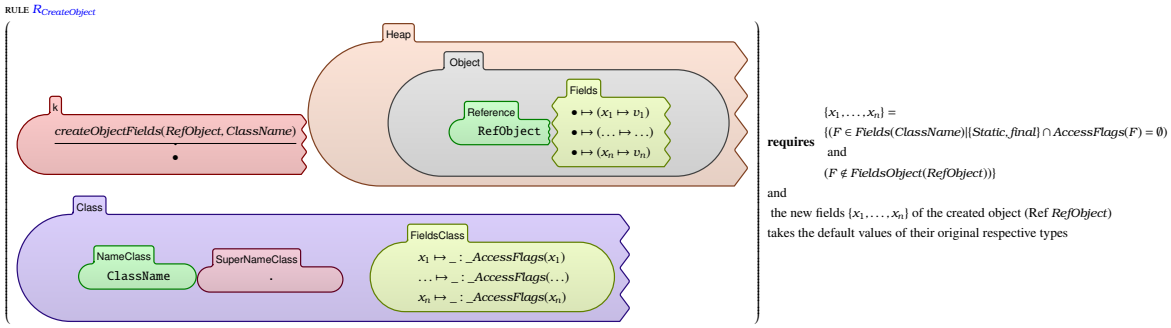


La règle $R_{NEW-INSTANCE}$ est appelée pour la création d'un nouvel objet instancié de la classe *Cl* avec l'instruction *new-instance Rn, Cl*. À l'intérieur de la cellule $\langle\ \rangle_k$, l'instance d'exécution courante est réécrite par une séquence de calculs qui seront exécutés par la règle suivante $R_{Createobject}$. Le compteur est incrémenté pour la prochaine utilisation. Dans $\langle\ \rangle_{Heap}$, une nouvelle entrée est créée (représentée par •) et une référence fraîchement générée est associée à l'objet nouvellement créé dans $\langle\ \rangle_{Object}$. Une fois créé, le registre destination *Rn* reçoit cette référence.

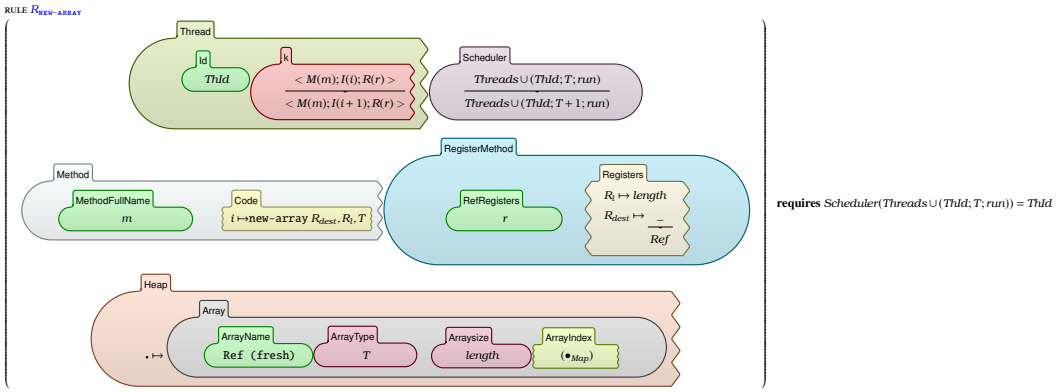


La règle $R_{CreateObject}$ permet de charger tous les champs statiques et finaux (spécifiés par *final* ou *static* dans $\langle\ \rangle_k$) depuis la classe *Cl* et sa super classe (avec les deux règles $R_{Createobject}$). Dans $\langle\ \rangle_{FieldsClass}$, ces champs sont initialisés avec leurs valeurs par défaut selon leurs types, comme montré dans Listing 5.5.

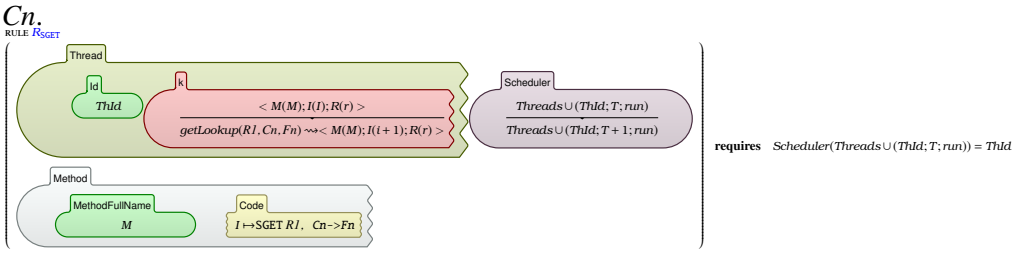




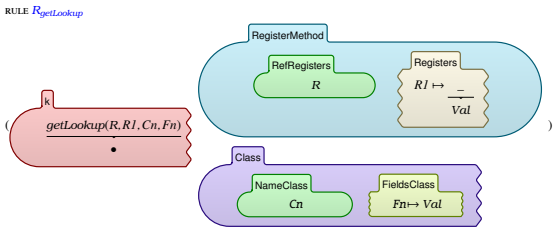
La règle $R_{\texttt{NEW-ARRAY}}$ créé un nouveau tableau dans $\langle\ \rangle_{Heap}$, avec le type T et la taille indiquée dans le registre R_l et pousse sa nouvelle référence dans le registre R_{dest}.



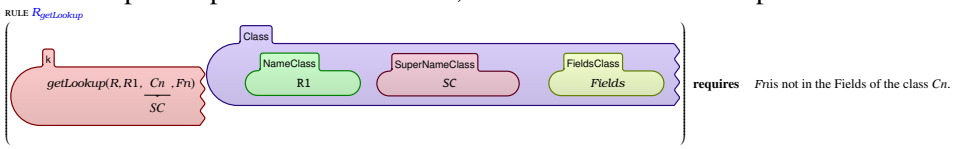
La règle $R_{\texttt{SGET}}$ fait appel à la règle $R_{getLookup}$ pour la recherche du champ statique Fn dans la classe Cn.



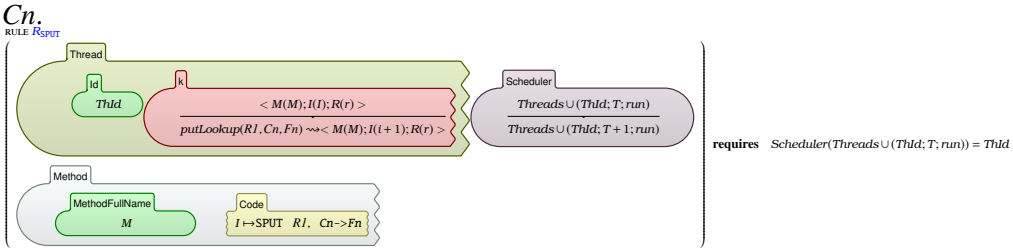
La règle $R_{getLookup}$ récupère le contenu du champ statique Fn de la classe Cn et le place dans le registre $R1$.



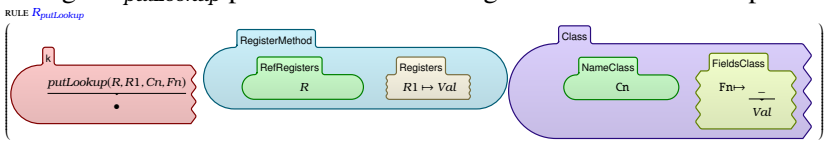
Si le champ n'est pas dans la classe *Cn*, alors chercher dans la super-classe *SC*.



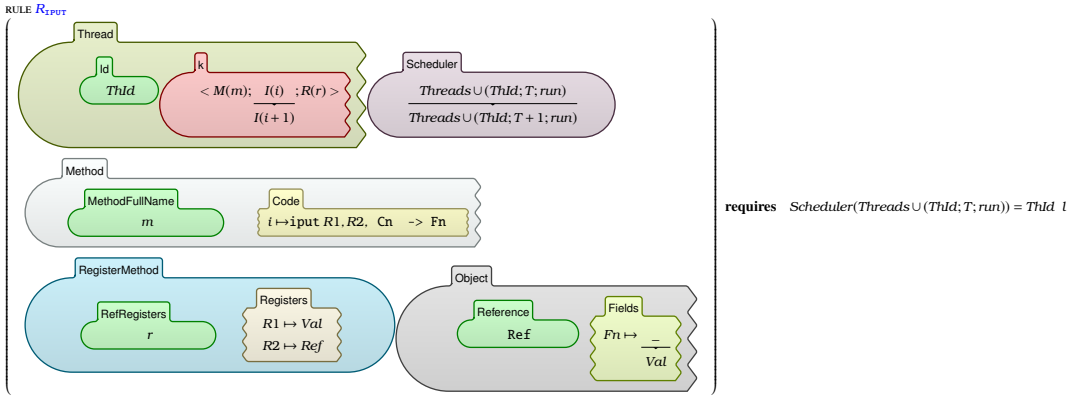
La règle R_{SPUT} fait appel à la règle $R_{putLookup}$ pour la recherche du champ statique *Fn* dans la classe *Cn*.



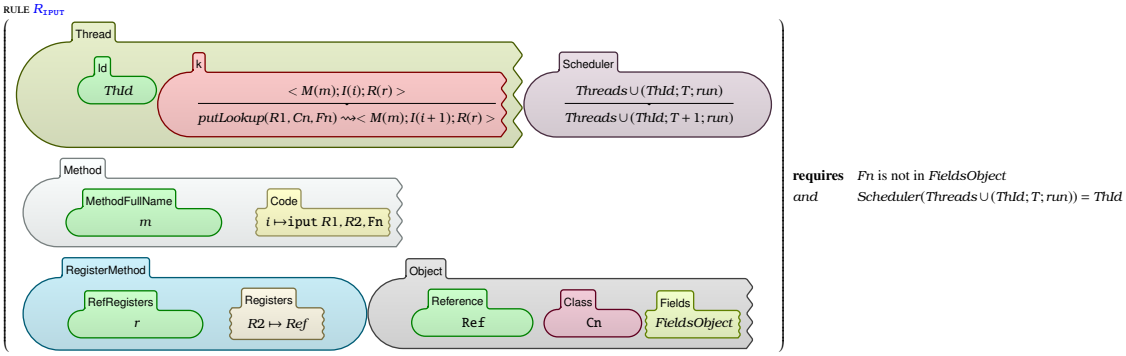
La règle $R_{putLookup}$ place le contenu du registre R1 dans le champ Fn de la classe Cn.



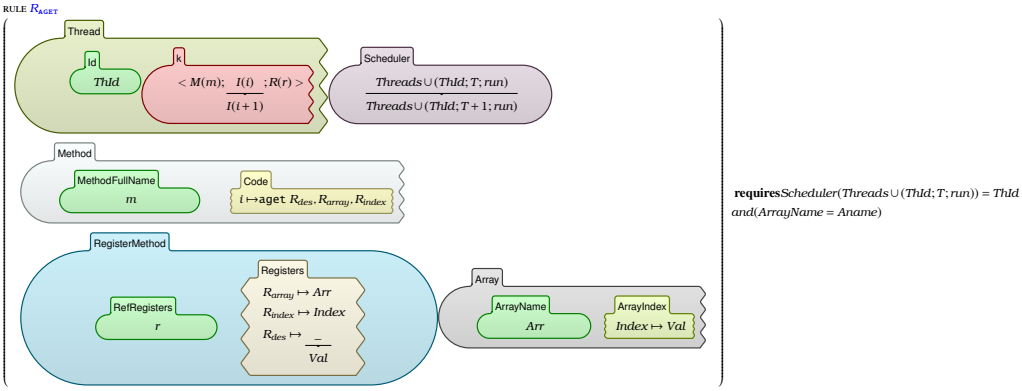
La règle R_{IPUT} est activée lorsque la cellule *k* d'un thread commence par une instruction *iput*. *R*1 est résolu en une valeur *Val* et *R*2 est résolu à une référence *Ref*. Enfin, le champ *Fn* qui correspond à cette référence reçoit la valeur *Val*.



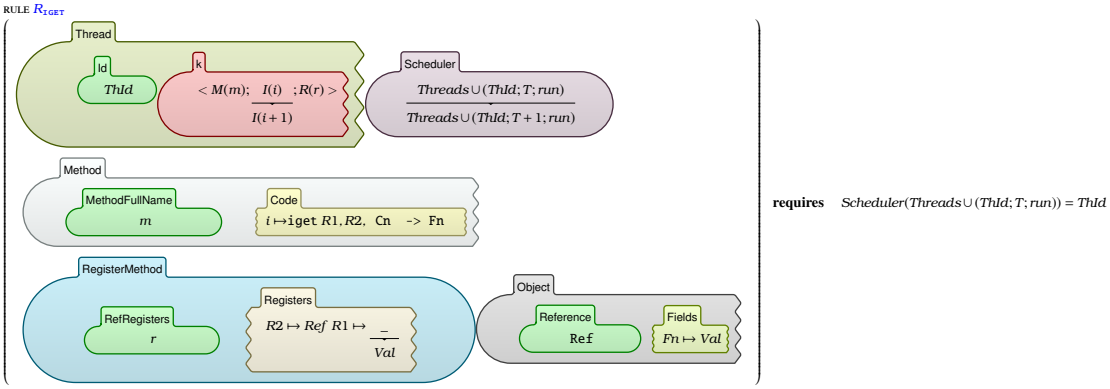
La règle $R_{iputLookup}$ est appelée pour rechercher le champ d'instance *Fn* ayant la référence *Ref* dans la classe *Cn*.



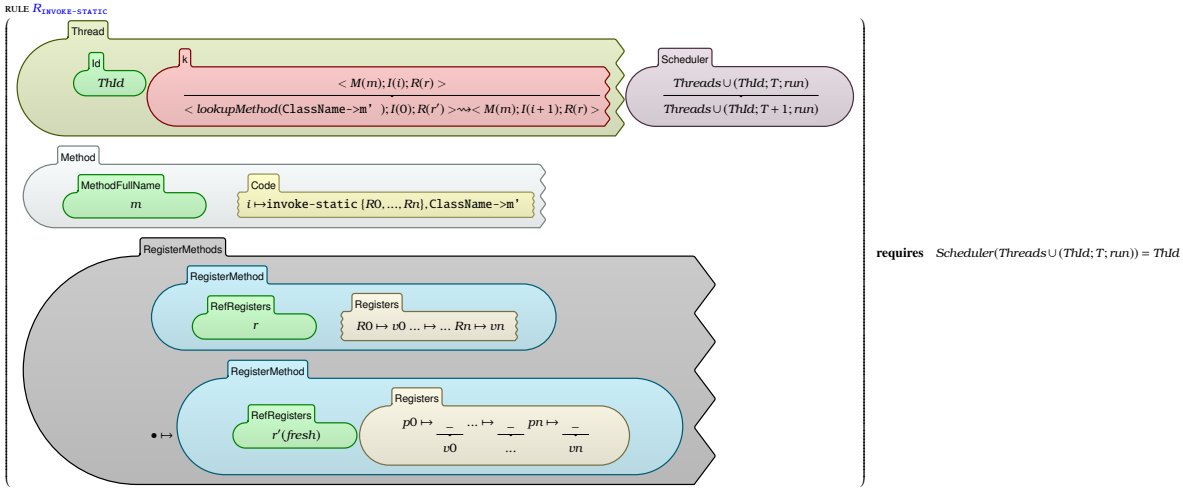
La règle R_{aget} charge le contenu de la case dont l'index est donné dans le registre R_{index} à partir du tableau R_{array} vers le registre destination R_{des}.



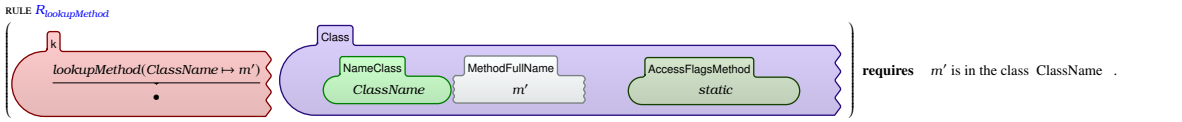
Dans la règle R_{IGET}, le registre $R2$ est tout d'abord résolu en une référence Ref d'u champ d'instance Fn. Ensuite, la valeur Val du champ récupérée à partir de classe objet qui correspond à cette référence est copiée dans le registre $R1$.

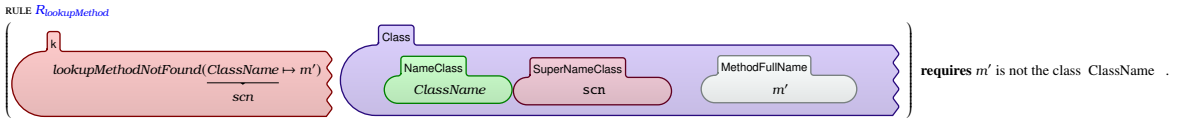


La règle $R_{INVOKE-STATIC}$ concerne l'invocation des méthodes statiques. Dans ce cas, la classe de base est déjà connue. La règle $R_{lookupMethod}$ est déclenchée pour la recherche de la méthode m' dans la classe $ClassName$. Dans $\langle\ \rangle_{RegisterMethod}$, une nouvelle entrée est créée avec une fraîche référence et dont les contenus des registres sont réécrits par ceux de la méthode appelante.



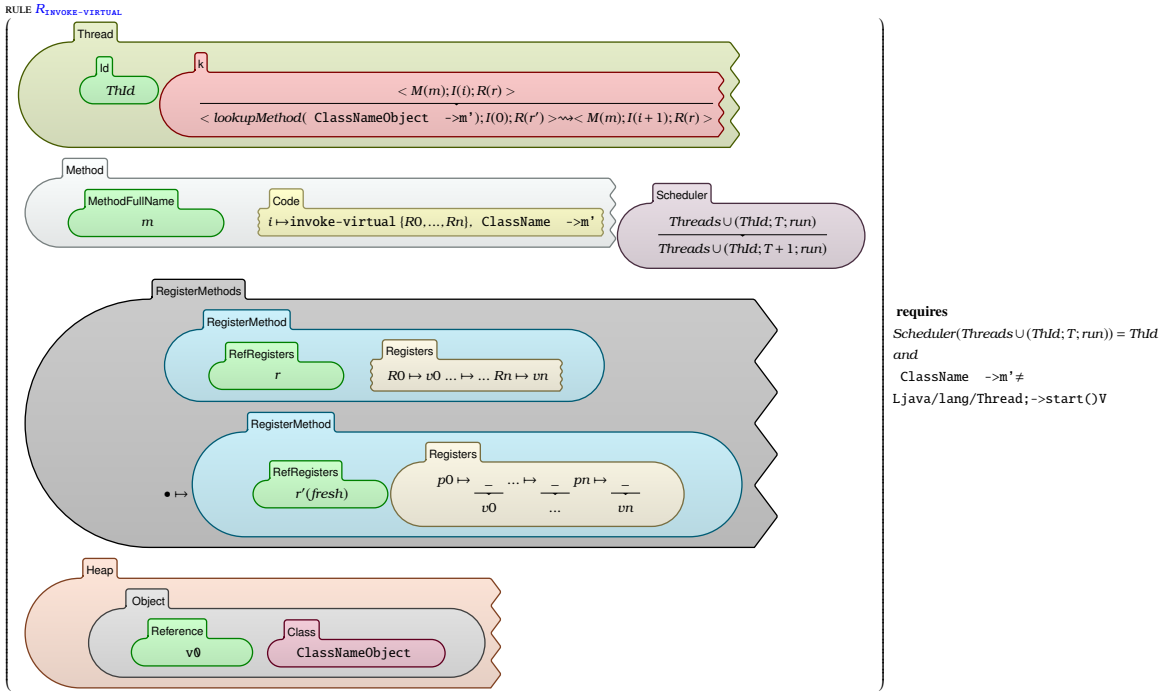
La règle $R_{lookupMethod}$ recherche une méthode statique appelée m' dans la classe Cl. Si elle ne la trouve pas dans cette classe, elle remonte jusqu'à sa chaîne de super-classes.



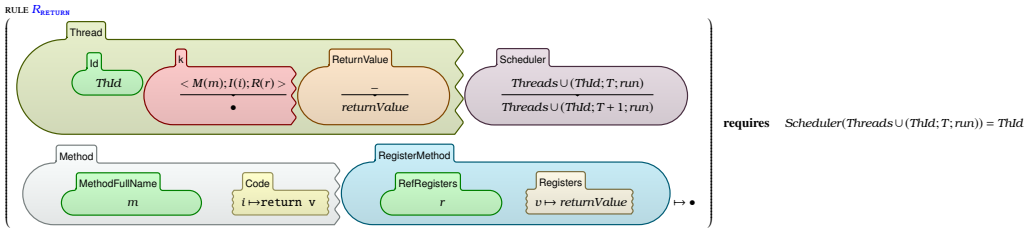


La règle $R_{INVOKE-VIRTUAL}$ s'applique lors de l'invocation d'une méthode d'instance. La classe de l'instance est d'abord récupérée à partir de tas (*heap*) par le biais de sa référence qui se trouve dans le premier argument v_0. Cette règle fait également appel à la règle $R_{lookupMethod}$ pour la recherche de la méthode m' dans la classe de l'objet référé et en remontant jusqu'à sa chaîne de super-classes, si elle ne se trouve pas dans cette classe. Le passage des arguments se fait de la même façon que lors de l'invocation d'une méthode statique.

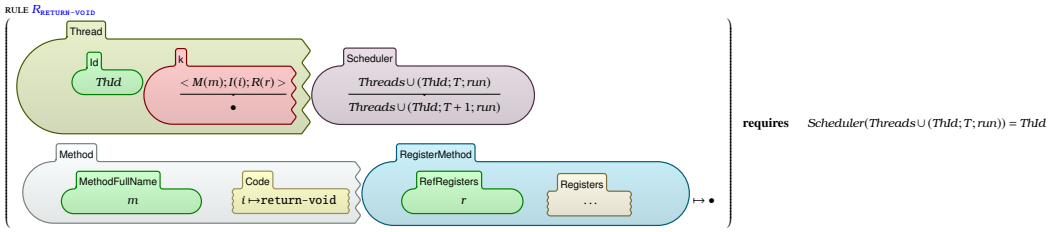
La condition qui accompagne la règle $R_{INVOKE-VIRTUAL}$ vérifie si la méthode invoquée est différente de la méthode *start()* de la classe *Thread*, qui est utilisée pour démarrer un thread, et traitée séparément par la règle $R_{CREATE-THREAD}$.



Les deux règles suivantes représentent respectivement la sémantique des instructions de retour à partir d'une méthode non-void et une méthode void. La règle R_{RETURN} place le contenu du registre R dans la cellule "ReturnValue", termine l'exécution de la méthode courante et libère ses registres.



La règle $R_{\text{RETURN-VOID}}$ achève l'exécution de la méthode courante et libère ses registres.



5.3.6 Instructions liées au mutli-threading

Afin de gérer les différents threads dans un programme Smali, nous avons implémenté un module qu'on a appelé SMALI-SCHEDULER. Listing 5.6 présente le code de ce module. Les différentes règles et fonctions définies dans ce module permettent de gérer les différents threads dans le programme.

```
1  module SMALI-SCHEDULER
2  imports SMALI-CONFIGURATION
3  /*---------------------Scheduler---------------------------------------------------*/
4  syntax Int::="CfsScheduler" "(" ScheduledThreads ")"  [function]
5  rule CfsScheduler((ThreadId:Int, _:Int, "run":String)) =>ThreadId
6  rule CfsScheduler((ThreadId1:Int, RT1:Int,"run":String) (ThreadId2:Int, RT2:Int,"run") ST:
       ScheduledThreads )=> CfsScheduler( (ThreadId2, RT2, "run":String)ST) requires RT1 >=
       Int RT2
7  rule CfsScheduler( (ThreadId1:Int, RT1:Int, "run":String) (ThreadId2:Int, RT2:Int,"run":
       String) ST:ScheduledThreads )=> CfsScheduler(  (ThreadId1, RT1,"run") ST) requires RT1
       <Int RT2
8  rule CfsScheduler( (_:Int, _:Int,State:String) ST:ScheduledThreads)=>CfsScheduler(ST)
       requires State =/=String "run"
9  rule CfsScheduler((ThreadId1:Int, RT1:Int,"run") (_:Int, _:Int,State:String) ST:
       ScheduledThreads)=>CfsScheduler((ThreadId1, RT1,"run") ST) requires State =/=String "
       run"
10 /*-------------------------Update-----------------------------------------------------*/
11 syntax ScheduledThreads::= "SchedulerUpdate" "(" Int "," String "," String ","
       ScheduledThreads")" [function]
12 rule SchedulerUpdate(ThreadId:Int, ThreadNewState:String, StateToUnlock:String,.
       ScheduledThreads)=> .ScheduledThreads
13 rule SchedulerUpdate(ThreadId:Int, ThreadNewState:String,  StateToUnlock:String,(ThreadId:
       Int, RunTime:Int,State:String) ST:ScheduledThreads )=>(ThreadId, RunTime+Int 1,
       ThreadNewState) SchedulerUpdate(ThreadId, ThreadNewState, StateToUnlock, ST)
14 rule SchedulerUpdate(ThreadId:Int, ThreadNewState:String,  StateToUnlock:String,(ThreadId1
       :Int, RunTime:Int, StateThread:String) ST:ScheduledThreads )=>(ThreadId1, RunTime,"run
       ") SchedulerUpdate(ThreadId, ThreadNewState,  StateToUnlock, ST) requires (ThreadId=/=
       K ThreadId1) andBool (StateToUnlock==K StateThread)
15 rule SchedulerUpdate(ThreadId:Int, ThreadNewState:String, StateToUnlock:String,(ThreadId1:
       Int, RunTime:Int, StateThread:String) ST:ScheduledThreads )=> (ThreadId1, RunTime,
       StateThread) SchedulerUpdate(ThreadId, ThreadNewState,  StateToUnlock, ST) requires (
       ThreadId=/=K ThreadId1) andBool (StateToUnlock=/=K StateThread)
16 /*-------------------------Remove-----------------------------------------------------*/
17 syntax ScheduledThreads::= "SchedulerRemove" "(" Int ","  ScheduledThreads")" [function]
18 rule SchedulerRemove( ThreadId:Int, .ScheduledThreads) =>.ScheduledThreads
19 rule SchedulerRemove( ThreadId:Int, (ThreadId:Int, _:Int,_:String) ST:ScheduledThreads)=>
       ST
20 rule SchedulerRemove( ThreadId:Int, (ThreadId1:Int, RunTime:Int, StateToUnlock:String) ST:
       ScheduledThreads )=> (ThreadId1, RunTime,StateToUnlock) SchedulerRemove(ThreadId, ST)
21 requires (ThreadId=/=K ThreadId1)
22 syntax Bool::= "ThreadInScheduler" "(" Int ","ScheduledThreads")"  [function]
23 rule ThreadInScheduler(ThreadId:Int, .ScheduledThreads)=> false
24 rule ThreadInScheduler( ThreadId:Int, (ThreadId:Int, _:Int,_:String) ST:ScheduledThreads)
       =>true
25 rule ThreadInScheduler( ThreadId:Int, (ThreadId1:Int, _:Int, _:String) ST:ScheduledThreads
       )=>ThreadInScheduler(ThreadId, ST) requires (ThreadId=/=K ThreadId1)
26 endmodule
```

Listing 5.6 – $\mathbb{K}$-*Smali* : gestion des threads

- La fonction *CfsScheduler* correspond en effet à l'ordonnanceur *CFS* utilisé dans Android pour gérer les différents threads dans le programme. Il permet de sélectionner le thread le plus prioritaire, c.-à-d. celui que s'est moins exécuté (ayant la plus petite valeur "runtime"), c'est la valeur "nice" utilisée pour sélectionner un thread par le *CFS*. La fonction *CfsScheduler* prend en paramètre la liste des threads dans le programme *ScheduledThreads* et retourne l'identifiant du thread sélectionné.
- La fonction *SchedulerUpdate* permet de mettre à jour les champs d'un thread à chaque exécution et à chaque tentative de synchronisation. Cette fonction prend comme paramètre l'identifiant du thread à modifier *ThreadId*, le nouveau statut *ThreadNewState*, le statut concernée *STateToUnlock* et une liste des threads *ScheduledThreads*.
- À la ligne 13, la fonction *SchedulerUpdate* incrémente le temps d'exécution "runtime" d'un

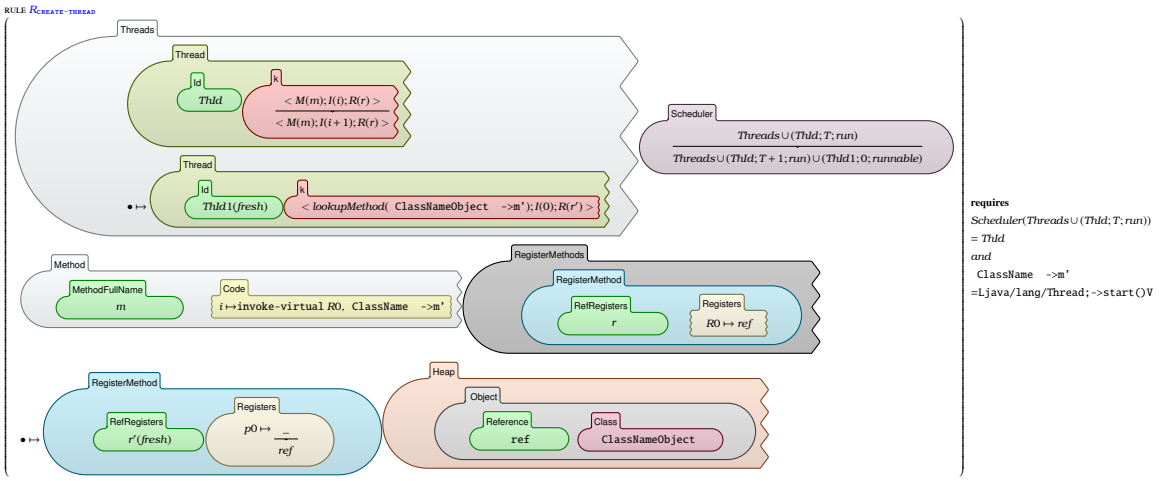
thread à chaque fois qu'il exécute une instruction. Les deux règles à la ligne 14 et 15 assurent les mises à jour de statut du thread "status", qui n'est pas en cours d'exécution, mais qui a le même état que celui donné dans le *StateToUnlock*.

- La fonction *SchedulerRemove* permet d'enlever un thread de la liste des threads à gérer. Elle prend en paramètre l'identifiant du thread à supprimer et une liste des threads et retourne la nouvelle liste mise à jour.
- La fonction *ThreadInScheduler* permet de vérifier si un thread existe dans la liste des threads. Elle prend l'identifiant de thread et la liste des threads et retourne une valeur booléenne confirmant ou non l'appartenance de ce thread à la liste.

Ces fonctions représentent principalement les différentes conditions qui accompagnent chaque règle. Aussi, elles gèrent tout le travail fait dans la cellule *Scheduler* : donner la main à un thread, lui allouer un temps d'exécution, l'arrêter dès que ce temps expire, ajouter et enlever un thread de la liste, etc.

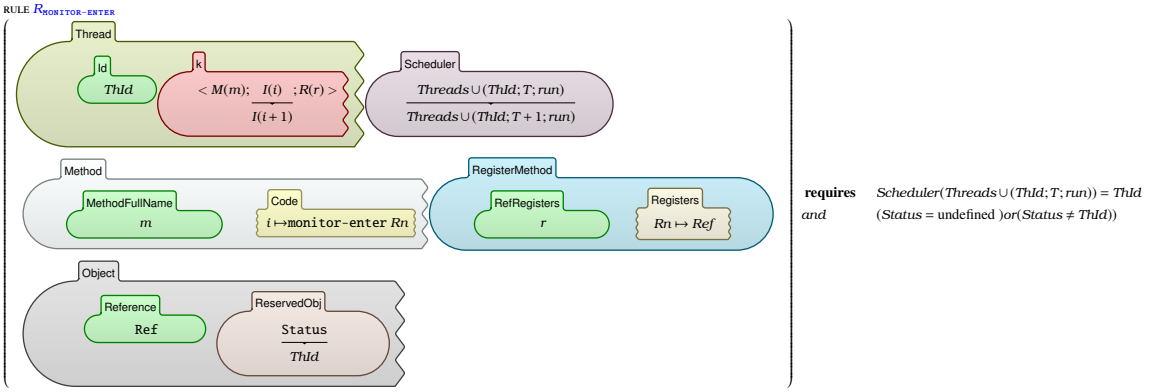
Comme nous l'avons expliqué précédemment, l'instruction *invoke-virtual* de la méthode *start()* à partir de la classe *java/lang/Thread* est gérée différemment qu'une instruction *invoke-virtual* de toute autre méthode. Cette spécificité est représentée par la condition adjacente à la règle qui gère cette instruction. Si elle est vérifiée, la règle $R_{CREATE-THREAD}$ s'applique. Sinon, la règle $R_{INVOKE-VIRTUAL}$ sera appliquée.

La référence du thread à lancer se trouve dans le premier argument. La méthode *lookupMethod* est appelée à la recherche de la méthode *start()*. Un nouvel objet thread est fraîchement créé et ajouté à la liste des threads prêts à s'exécuter. Cela est traité dans la cellule *Scheduler* où un nouveau thread ayant l'identifiant *ThId1*, un temps d'exécution (ou *runtime*) à 0 et un statut *"runnable"* est joint au reste des threads prêts à s'exécuter.

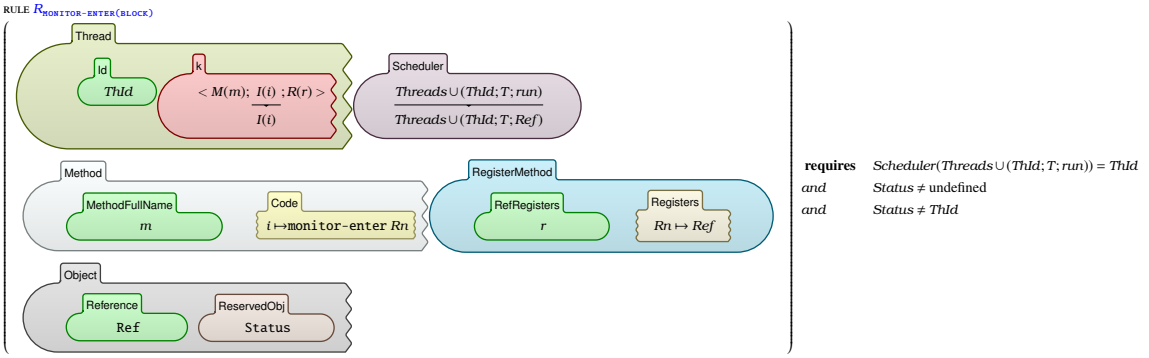


La règle $R_{MONITOR-ENTER}$ représente la sémantique de l'instruction *monitor-enter*. Cette règle s'applique quand l'objet dans le registre *Rn* est libre, c.-à-d. que le moniteur qui y associé n'est pas détenu par aucun thread. Cette condition est vérifiée par la condition latérale de la règle. Dans la cellule

$\langle\ \rangle_{ReservedObject}$, le statut de l'objet réservé est réécrit par la référence du thread. Si le moniteur est détenu par un autre thread, alors la règle $R_{MONITOR-ENTER(BLOCK)}$ s'applique.

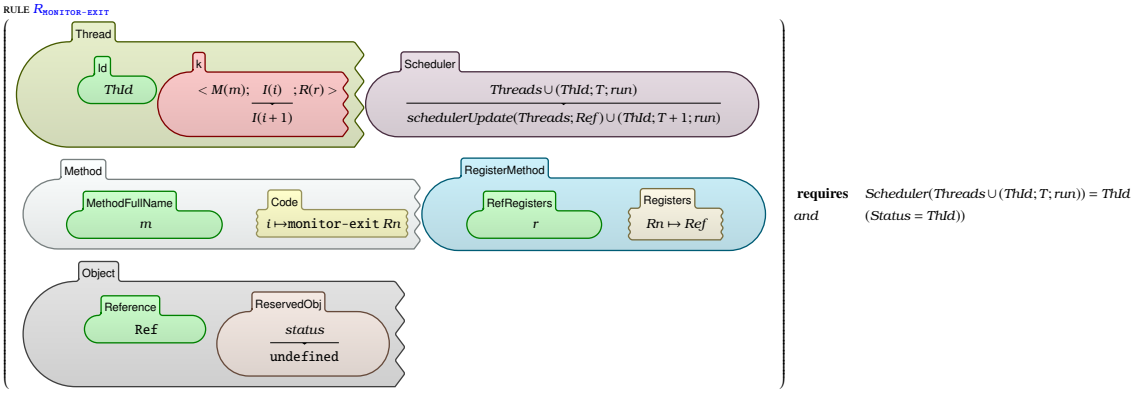


La règle $R_{MONITOR-ENTER(BLOCK)}$ représente également la sémantique de l'instruction *monitor-enter*, mais s'applique seulement quand l'objet à partager est réservé par un autre thread, c.-à-d. ayant comme statut l'identifiant d'un autre thread. Cela est vérifié par la condition adjacente à la règle. Dans ce cas, le thread sera bloqué. Cela peut être constaté dans $\langle\ \rangle_k$ où l'instruction courante est réécrite par la même instruction. Autrement dit, l'exécution du programme est bloquée. Durant cette situation, le système de réécriture fera progresser l'état en évaluant d'autres threads. Cela peut être constaté dans la sous-cellule $\langle\ \rangle_{Scheduler}$ où le thread bloqué se joint à la liste des threads en attente de l'objet référé (*Ref*). Le statut de ce thread est mis à jour tout d'abord par la référence de l'objet qu'il a tenté de réserver. Ceci exprime que ce thread il a le statut bloqué sur l'objet ayant la référence *Ref*.



Le rôle de la règle $R_{MONITOR-EXIT}$ est de libérer le verrou de moniteur associé à l'objet dans *Rn* symétriquement à la façon dont il a été acquis par les deux précédentes règles. Dans cette règle, deux cellules responsables de la synchronisation des threads sont mises à jour : $\langle\ \rangle_{ReservedObject}$ et $\langle\ \rangle_{Scheduler}$. Dans la première cellule, le statut de l'objet ayant comme valeur la référence de ce thread est réécrit par la valeur "undefined" indiquant que l'objet est désormais libre. La libération de l'objet *Ref* affecte la cellule $\langle\ \rangle_{Scheduler}$ où la liste des threads en attente de la libération de cet objet (ayant comme statut la référence de cet objet) est mise à jour avec la fonction *SchedulerUpdate*. Cette fonction

permet de débloquer ces threads. Elle met à jour le statut des threads bloqués "Ref" par un nouveau statut "runnable", indiquant qu'ils sont désormais prêts d'être sélectionné par l'ordonnanceur puisque l'abojet ayant cette référence a été libéré.



where :

$$schedulerUpdate(\{\}; Ref) = \{\}$$
$$schedulerUpdate(\{(ThId; T; Ref)\} \cup Threads; Ref) = \{(ThId; T; run)\} \cup schedulerUpdate(Threads; Ref)$$
$$schedulerUpdate(\{(ThId; T; Status)\} \cup Threads; Ref) = \{(ThId; T; Status)\} \cup schedulerUpdate(Threads; Ref)$$
$$If\ Status \neq Ref$$

Les définitions $\mathbb{K}$ comprennent également des fonctions. La plupart de ces fonctions sont utilisées pour gérer les conditions qui accompagnent les règles de réécriture, en particulier la définition de la logique des prédicats utilisée.

5.4 Simulation des programmes types

Après avoir compilé la sémantique $\mathbb{K}$-*Smali*, plusieurs opérations peuvent être effectuées. Par exemple, nous enregistrons la définition du $\mathbb{K}$-*Smali* dans un fichier principal appelé "*kSmali.k*". Ce fichier importe tous les modules nécessaires à la définition du langage (syntaxe, configuration, règles sémantiques et initialisation du programme). La commande « *kompile kSmali.k* » permet de compiler le langage défini et offre la possibilité d'invoquer un interprète (alias moteur d'exécution, ou réécrivant). Ce dernier est invoqué avec le « *krun programme.smali* ». Cette commande permet d'exécuter des programmes de test enregistrés dans des fichiers *.smali*.

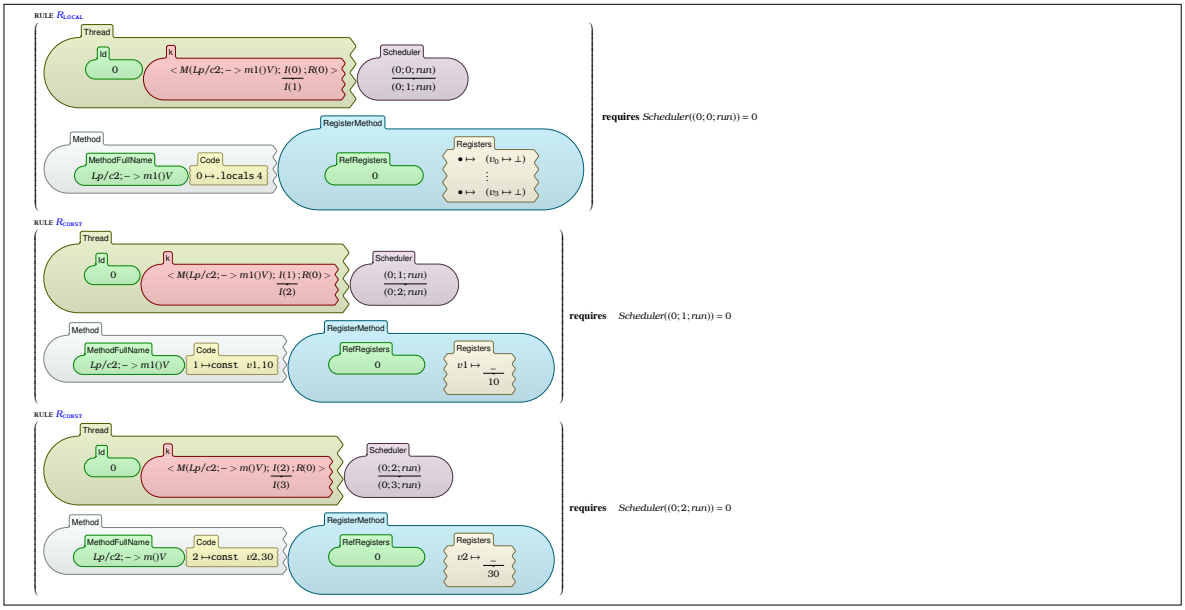
Chaque programme $\mathbb{K}$-*Smali* présenté ci-après est simulé en générant une séquence de traces. Ces traces sont illustrées sous forme de représentations graphiques de règles de réécriture montrant la progression de la configuration du programme pour chaque instruction exécutée.

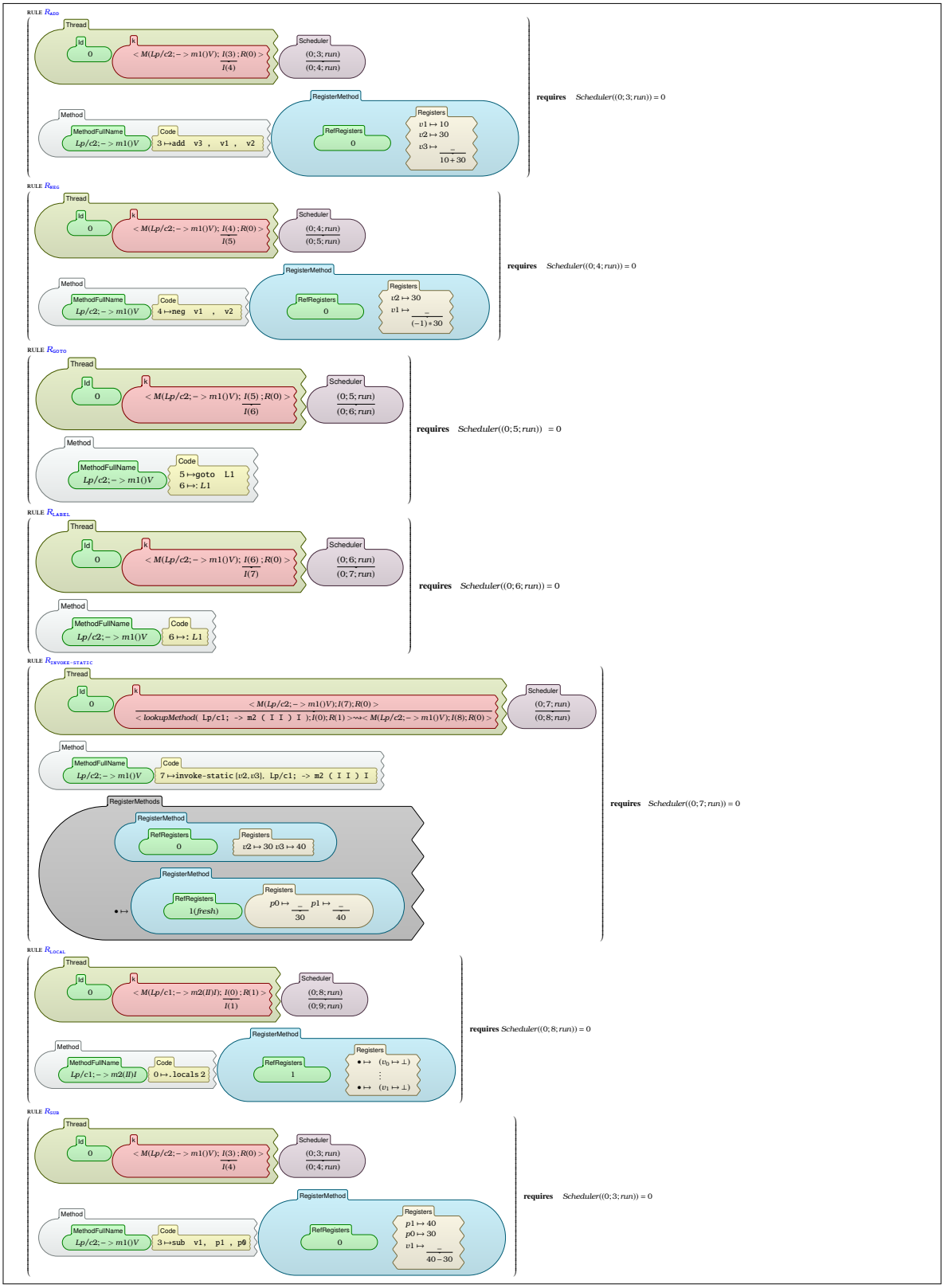
Listing 5.7 représente un exemple de programme $\mathbb{K}$-*Smali* séquentiel, composé de deux classes $c1$ et $c2$, où $c1$ est la super-classe de $c2$. Le ".manifest" de la ligne 28 indique le point d'entrée du programme qui est la méthode $m1$ de la classe $c2$. Ainsi, l'exécution commence à partir de la première instruction de cette méthode. Le programme termine son exécution lorsque la cellule k de la configuration est vide. Autrement dit, il n'y a plus de code à exécuter.

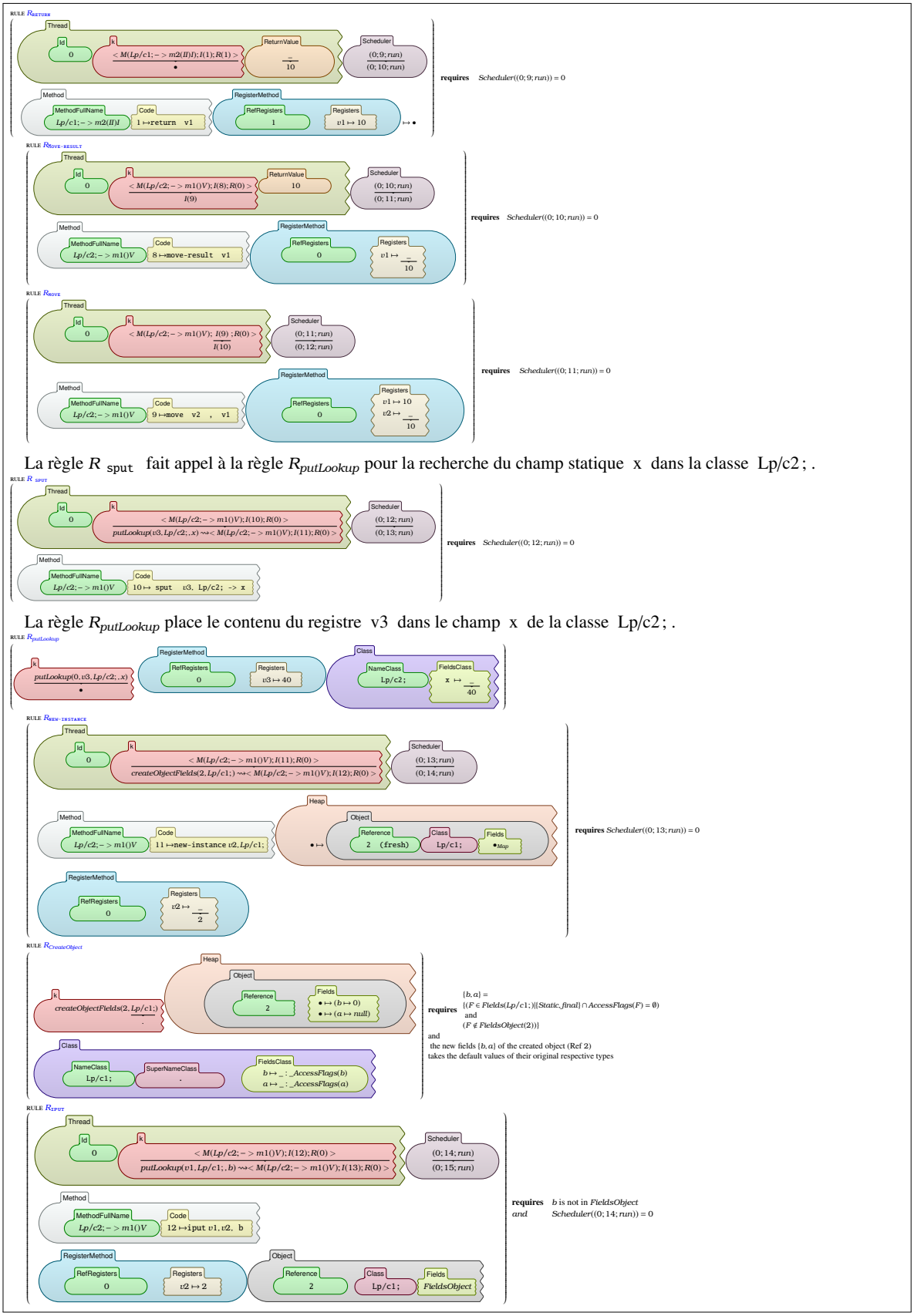
```
1  .class public Lp/c2;
2  .super Lp/c1;
3  .field public x: I
4  .method public static m1()V
5   .locals 4
6   const v1, 10
7   const v2, 30
8   add v3, v1, v2
9   neg v1, v2
10  goto :L1
11  :L1
12  invoke-static {v2, v3}, Lp/c1;->m2(II)I
13  move-result v1
14  move v2, v1
15  sput v3, Lp/c2;->x
16  new-instance v2, Lp/c1;
17  iput v1, v2,  Lp/c1;->b
18  return-void
19 .endmethod
20 .class public Lp/c1;
21 .field public a: Ljava/lang/String;
22 .field private final c: C
23 .method public static m2(II)I
24  .locals 2
25  sub v1, p1, p0
26  return v1
27 .endmethod
28 .manifest Lp/c2;->m1()V
```

Listing 5.7 – Exemple d'un programme séquentiel $\mathbb{K}$-*Smali*

Figure 5.4 représente les traces générées par l'interprète $\mathbb{K}$ suite à l'exécution du programme donné dans Listing 5.7.




RULE $R_{\texttt{ADD}}$
Thread
Id
0
k
$< M(Lp/c2; -> m1()V); I(3); R(0) >$
$I(4)$
Scheduler
$(0;3;run)$
$(0;4;run)$
Method
MethodFullName
$Lp/c2; -> m1()V$
Code
3 ↦ add v3 , v1 , v2
RegisterMethod
RefRegisters
0
Registers
$v1 \mapsto 10$
$v2 \mapsto 30$
$v3 \mapsto \_$
$10+30$
requires $Scheduler((0;3;run)) = 0$
RULE $R_{\texttt{NEG}}$
Thread
Id
0
k
$< M(Lp/c2; -> m1()V); I(4); R(0) >$
$I(5)$
Scheduler
$(0;4;run)$
$(0;5;run)$
Method
MethodFullName
$Lp/c2; -> m1()V$
Code
4 ↦ neg v1 , v2
RegisterMethod
RefRegisters
0
Registers
$v2 \mapsto 30$
$v1 \mapsto \_$
$(-1)*30$
requires $Scheduler((0;4;run)) = 0$
RULE $R_{\texttt{GOTO}}$
Thread
Id
0
k
$< M(Lp/c2; -> m1()V); I(5); R(0) >$
$I(6)$
Scheduler
$(0;5;run)$
$(0;6;run)$
Method
MethodFullName
$Lp/c2; -> m1()V$
Code
5 ↦ goto L1
6 ↦ : L1
requires $Scheduler((0;5;run)) = 0$
RULE $R_{\texttt{LABEL}}$
Thread
Id
0
k
$< M(Lp/c2; -> m1()V); I(6); R(0) >$
$I(7)$
Scheduler
$(0;6;run)$
$(0;7;run)$
Method
MethodFullName
$Lp/c2; -> m1()V$
Code
6 ↦ : L1
requires $Scheduler((0;6;run)) = 0$
RULE $R_{\texttt{INVOKE-STATIC}}$
Thread
Id
0
k
$< M(Lp/c2; -> m1()V); I(7); R(0) >$
$< lookupMethod($ Lp/c1; -> m2 (I I) I $); I(0); R(1) > \rightsquigarrow < M(Lp/c2; -> m1()V); I(8); R(0) >$
Scheduler
$(0;7;run)$
$(0;8;run)$
Method
MethodFullName
$Lp/c2; -> m1()V$
Code
7 ↦ invoke-static $[v2, v3]$, Lp/c1; -> m2 (I I) I
RegisterMethods
RegisterMethod
RefRegisters
0
Registers
$v2 \mapsto 30$ $v3 \mapsto 40$
RegisterMethod
RefRegisters
$1(fresh)$
Registers
$p0 \mapsto \_$ $p1 \mapsto \_$
30 40
requires $Scheduler((0;7;run)) = 0$
RULE $R_{\texttt{LOCAL}}$
Thread
Id
0
k
$< M(Lp/c1; -> m2(II)I); I(0); R(1) >$
$I(1)$
Scheduler
$(0;8;run)$
$(0;9;run)$
Method
MethodFullName
$Lp/c1; -> m2(II)I$
Code
0 ↦ .locals 2
RegisterMethod
RefRegisters
1
Registers
$\bullet \mapsto (v_0 \mapsto \bot)$
$\vdots$
$\bullet \mapsto (v_1 \mapsto \bot)$
requires $Scheduler((0;8;run)) = 0$
RULE $R_{\texttt{SUB}}$
Thread
Id
0
k
$< M(Lp/c2; -> m1()V); I(3); R(0) >$
$I(4)$
Scheduler
$(0;3;run)$
$(0;4;run)$
Method
MethodFullName
$Lp/c2; -> m1()V$
Code
3 ↦ sub v1, p1 , p0
RegisterMethod
RefRegisters
0
Registers
$p1 \mapsto 40$
$p0 \mapsto 30$
$v1 \mapsto \_$
$40-30$
requires $Scheduler((0;3;run)) = 0$



La règle $R_{\texttt{sput}}$ fait appel à la règle $R_{putLookup}$ pour la recherche du champ statique x dans la classe Lp/c2 ; .

La règle $R_{putLookup}$ place le contenu du registre v3 dans le champ x de la classe Lp/c2 ; .

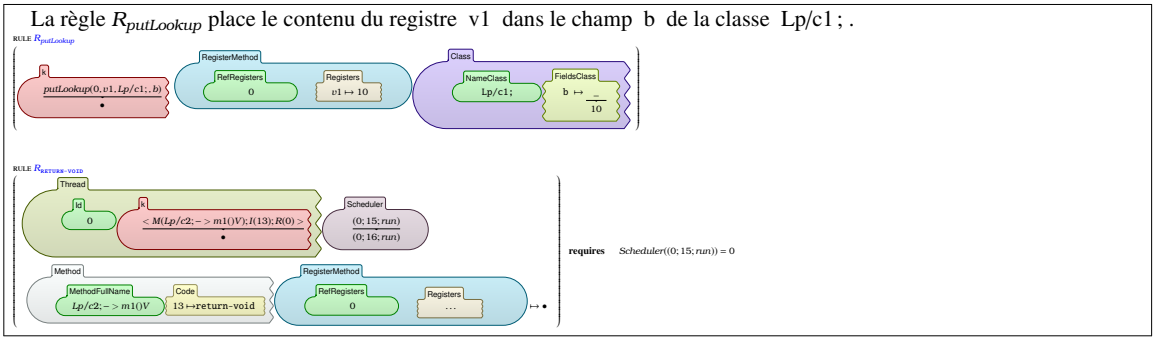


FIGURE 5.4 – Traces d'exécution du programme $\mathbb{K}$-*Smali* présenté dans Listing 5.7

5.5 Vérification du programme en utilisant $\mathbb{K}$ *Framework*

$\mathbb{K}$ supporte naturellement la vérification des programmes pour les langages définis. Pour vérifier des propriétés, nous avons besoin d'un *programme* P, une *propriété* S à prouver, et enfin de tester si P satisfait ou pas la propriété S en utilisant la commande *krun* et l'option appropriée.

Pour la spécification des propriétés, $\mathbb{K}$ offre un large éventail d'options. En résumé, le fondement logique de l'infrastructure de vérification du système $\mathbb{K}$ est la logique de correspondance (*matching logic*) pour les propriétés statiques [154] et la logique d'accessibilité (*reachability logic*) pour les propriétés dynamiques (à partir de la version 4) [155]. Les propriétés peuvent être spécifiées comme des assertions de logique d'accessibilité où chaque assertion est écrite comme une règle de réécriture $\mathbb{K}$. Elles peuvent également être écrites comme des pré-conditions et post-conditions dans des triples d'Hoare [152], des formules de logique temporelle LTL, de logique modale, des formules de logique du premier ordre, ou tout autre formalisme logique. $\mathbb{K}$ offre un contrôle de modèle en logique temporelle linéaire (LTL) par compilation dans des programmes Maude. Cela est assuré par le biais de son backend Maude [143, 156] disponible à partir de la version 3.5.

Dans cette section, nous prenons comme exemples deux propriétés à vérifier par un programme Smali ayant la sémantique $\mathbb{K}$. Ces deux propriétés sont liées à des API sensibles dans Android, à savoir les méthodes des API permettant de prendre une photo ou enregistrer une vidéo et l'API permettant d'envoyer un SMS. Ces API peuvent être exploités pour espionner l'utilisateur ou lui causer des pertes financières. Au lieu de surveiller à chaque fois ces API sont invoqués, ce qui augmente le taux de faux positifs, les formules surveillent l'ordre temporel avec lequel ils sont appelés et qui peut mener à des attaques pareilles.

La logique temporelle (LT) regroupe les modalités LTL et ptLTL [157]. Le LTL avec des opérateurs passés est exponentiellement plus succinct que le LTL avec des opérateurs futurs purs [158]. Cette logique nous a beaucoup aidé à exprimer des propriétés de manière concise. Les opérateurs tels que "précédemment" et "éventuellement dans le passé", permettent de dicter un ordre temporel dans le

passé. Par exemple, une formule temporelle peut exprimer l'ordre suivant : "si une action A est exécutée maintenant, elle doit être *précédée* par l'action B dans le passé". En ce qui concerne Android, cela nous a permis d'imposer un ordre temporel dans l'invocation des API, qui exprime un comportement sûr de l'application ciblée.

5.5.1 Logiciels espions

Prendre des photos, enregistrer des sons et des vidéos à l'insu de l'utilisateur est parmi les comportements qui caractérisent les logiciels espions sur Android.

- *Programme à vérifier :*
  Étant donné un programme *P* qui permet de prendre une photo avec l'instruction *invoke-virtual* de la méthode *TakePicture* du nom de classe entièrement qualifié *Landroid/hardware/Camera;*. L'invocation de la méthode *setPreviewDisplay* ou *setPreviewTexture* de la même classe avant celle de l'API permet d'afficher l'aperçu de l'appareil photo à l'écran. De cette manière, l'utilisateur saura que l'appareil photo est ouvert et qu'il tente de prendre une photo. Les APIs représentant ces deux fonctionnalités (affichage de l'aperçu et prise de photo) sont principalement invoquées par l'instruction *invoke-virtual* comme illustré dans l'exemple affiché dans Listing 5.8.

```
1 invoke-virtual{v1,v2},Landroid/hardware/Camera;↦setPreviewDisplay(Landroid/view/
      SurfaceHolder;)V
2 invoke-virtual{v1,v2},Landroid/hardware/Camera;↦setPreviewTexture (Landroid/view/
      SurfaceHolder;)V
3 invoke-virtual{v0,v2,v2,v1},Landroid/hardware/Camera;↦takePicture(Landroid/hardware/Camera$
      ShutterCallback;Landroid/hardware/Camera$PictureCallback;Landroid/hardware/Camera$
      PictureCallback;)V
```

Listing 5.8 – Exemple des appels API

- *Spécification des propriétés :*
  Une formule LTL peut exprimer le comportement souhaité et requiert l'ordre d'invoquer l'API *SetPreviewDisplay* ou *SetPreviewTexture* **avant** l'API *takePicture*. De cette façon, nous pouvons vérifier si le programme peut espionner l'utilisateur ou non. Pour définir cette propriété, nous exprimons la logique du passé en utilisant les modalités de la logique LTL passée (ptLTL) et de la logique LTL future (LTL). Une formule LTL représentant ce comportement pourrait être définie comme suit :

$$\Box(takePicture \rightarrow \odot(setPreviewDisplay \vee setPreviewTexture))$$

La formule LTL commence par l'opérateur LTL $\Box$ qui signifie "*toujours*". L'opérateur $\odot$ représente la logique du passé (ptLTL) et signifie "*précédemment*". L'opérateur $\vee$ exprime la disjonction. Intuitivement, la formule indique que "*Si takepicture se produit maintenant, setPreviewDisplay ou setPreviewTexture doit (toujours) avoir eu lieu (précédemment)*".

De même, si la méthode *setPreviewDisplay* de *Landroid/media/MediaRecorder;* n'est pas utilisée avant d'enregistrer une vidéo/audio, l'utilisateur ne sera pas averti lorsque l'application tentera d'enregistrer subrepticement une vidéo ou un audio. La formule LTL suivante exprime ce comportement :

$$\Box(setVideoSource \vee setAudioSource \rightarrow \odot setPreviewDisplay)$$

5.5.2 Chevaux de Troie SMS

Les chevaux de Troie SMS causent des pertes financières aux utilisateurs en envoyant des messages SMS à des numéros surtaxés sans le consentement de l'utilisateur. La dissimulation des messages SMS reçus est possible en annulant les diffusions qui y sont liées, c.-à-d. les messages de notification retournés. En effet, après avoir invoqué l'API *sendTextMessage* avec un numéro surtaxé, l'attaquant intercepte et appelle la fonction *abortBroadcast* pour supprimer les messages de notification liés à la facturation des fournisseurs de services respectifs. De cette façon, l'attaquant peut s'assurer que l'utilisateur ne sera pas en mesure de détecter qu'un SMS a été envoyé.

- *Spécification des propriétés :*
  Cette propriété peut être exprimée par la formule ci-dessous :

  $$\Box(\neg abortBraodcast \rightarrow \diamond sendTextMessage)$$

  L'opérateur ptLTL $\diamond$ signifie "*éventuellement dans le passé*". Afin de détecter la possibilité d'un cheval de Troie SMS, la formule garantit que "*chaque fois que la fonction abortBroadcast est précédée d'une méthode sendTextmessage, cette action ne sera pas autorisée*". Intuitivement, elle garantit que l'utilisateur sera notifié chaque fois qu'il recevra un SMS ou tout simplement pas d'annulation des SMS entrants.
- *Commande Krun* :
  La commande *Krun* est utilisée pour exécuter un programme ayant la sémantique $\mathbb{K}$ du langage. Les formules LTL peuvent également être vérifiées par le biais du model checking LTL avec cette commande plus l'option "−−ltlmc" comme suit :

  $$krun\ P.smali\ --\text{ltlmc}\ LTLformula$$

  L'option "−−ltlmc" est utilisée avec la commande *krun* pour indiquer que le programme spécifié ($P.smali$) est évalué par rapport à la formule LTL suivante ($LTLformula$). Le résultat est *True* si la propriété est vérifiée. Sinon, un contre-exemple représentant une exécution violant la propriété est exhibé.

5.6 Conclusion

Nous avons été extrêmement motivés par les différentes sémantiques $\mathbb{K}$ de plusieurs langages réels tels que Java [159], PHP [160], C [161] et Javascript [162]. Nous avons vu comment elles ont été utilisées comme modèles de références fiables pour le langage défini. $\mathbb{K}$-Java, à titre d'exemple, est une sémantique générique qui a été utilisée dans plusieurs projets liés à Java. Il en va de même pour $\mathbb{K}$-*Smali*, la sémantique formelle est exécutable et convient au raisonnement formel et à la génération automatique de preuves. Dans notre cas, nous profitons de la formalisation $\mathbb{K}$-*Smali* pour renforcer des politiques de sécurité dans des applications Android.

En utilisant l'environnement $\mathbb{K}$, nous avons pu améliorer plusieurs points non couverts dans *Smali*$^+$, tels que le point d'entrée du programme, l'étape d'initialisation, et d'autres détails manquants découverts lors de la compilation de la définition du langage. Table 5.1 compare les deux sémantiques. Les symboles indiquent que le langage couvre les fonctionnalités en question entièrement (●), partiellement (◐) ou complètement (○). L'exécution, le débogage de la sémantique sont tous pris en charge par l'environnement $\mathbb{K}$. De plus, grâce à ses outils intégrés, $\mathbb{K}$ permet de vérifier facilement les propriétés des programmes $\mathbb{K}$-*Smali*.

Fonctionnalité	*Smali*$^+$	$\mathbb{K}$-*Smali*
Instructions basiques	●	●
Switch	○	●
Sémantique des directives (.locals, .registers)	○	●
Mode d'accès aux méthodes	◐	●
Initialisation des champs (instances et statiques)	●	●
Chargement des champs d'instance (final et static) lors de la création d'un objet	○	●
Notions de base sur les tableaux	◐	●
Longueur des tableaux	○	●
Initialisations de tableaux	○	●
Valeurs par défaut des tableaux	○	●
Multi-threading	●	●
Point d'entrée de l'application	○	●
Initialisation du programme	◐	●
Compilation de la définition de la sémantique	○	●
Exécution de programmes types	○	●
Vérification du programme	○	●
Sémantique Exécutable	○	●

TABLE 5.1 – *Smali*$^+$ vs $\mathbb{K}$-*Smali*

Chapitre 6

Renforcement formel et automatique de politiques de sécurité dans des applications Android par réécriture

6.1 Introduction

La vérification de la sécurité joue un rôle essentiel en tant que moyen de fournir aux utilisateurs l'assurance de sécurité nécessaire dans de nombreuses applications. Il existe différentes solutions comme la cryptographie, les produits informatiques dédiés (p.ex. pare-feu, antivirus, anti-spam, etc.), le renforcement des politiques de sécurité, etc. En général, la difficulté de ce problème est proportionnelle à la complexité du système.

Android est un système complexe qui fait intervenir multiples aspects, tels que la sécurité, la concurrence. Tenir compte de tous ces aspects lors de la spécification de ce système, tout en se basant seulement sur les méthodes formelles, est une tâche difficile à réaliser. De surcroît, les méthodes formelles ne sont pas à la protée de tout le monde, et ce même pour les experts du domaine.

Le paradigme orienté aspect [163] complète ce côté manquant. En effet, ce paradigme vise à ajouter automatiquement un aspect, tel que la sécurité, à partir d'une spécification. Cela permet de réduire le coût et le temps du développement des systèmes. Par ailleurs, la tâche de maintenance se réduit tout simplement à régénérer automatiquement un système à partir de sa nouvelle spécification. De leur côté, les méthodes formelles assurent que les systèmes résultants respectent leurs spécifications. La combinaison entre les deux méthodes permettra ainsi d'économiser les coûts et réduire les erreurs qui peuvent être causées suite à l'intervention humaine.

Les techniques de réécriture de programme dérivent de la synergie entre le paradigme orienté aspects et les méthodes formelles. L'état de l'art présenté dans la première partie de la thèse a confirmé leur capacité à renforcer plus de politiques de sécurité que les autres techniques existantes, tout en assu-

rant une réduction des surcoûts engendrés par leur application. Dans ce qui suit, nous utilisons ces techniques pour le renforcement de politiques de sécurité au sein des applications Android.

Dans le chapitre précédent, nous avons présenté $\mathbb{K}$-*Smali*, une sémantique dédiée d'une application Android, plus précisément de son bas-niveau du code, Smali. Cette formalisation sera notre point de départ pour appliquer des politiques de sécurité.

Dans le présent chapitre, nous présentons notre approche [12], qui consiste en l'élaboration d'un cadre formel pour le renforcement automatique de politiques de sécurité. Comme le programme Smali, la politique de sécurité subira une spécification formelle. Ensuite les deux spécifications seront adaptées en ajoutant certaines actions de contrôle qui permettent à la politique de sécurité de piloter le programme.

Figure 6.1 illustre le processus de renforcement. L'approche consiste à définir un opérateur "$\sqcap$" qui prend une application Android et une politique de sécurité et génère une intersection P'. Les flèches en pointillés signifient une spécification formelle pour l'application Android en un programme $\mathbb{K}$-*Smali*, P et la politique de sécurité en une formule LTL, φ. P' est une nouvelle version de P qui respecte φ.

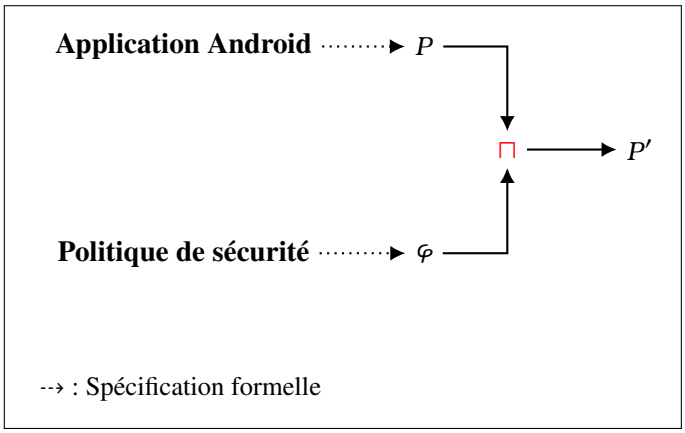


FIGURE 6.1 – Processus de renforcement

La nouvelle version du programme $P' = P \sqcap \varphi$ doit vérifier les deux propriétés suivantes :

— *Propriété de correction :* La version générée P' respecte la politique de sécurité. Ceci équivaut à dire que toutes les traces de P' respectent φ. Formellement :

$$P' \vdash \varphi \tag{6.1}$$

où la notation $\vdash$ signifie intuitivement "satisfait" ou "respecte". Sa définition formelle sera donnée plus loin dans le chapitre.

— *Propriété de complétude :* P' préserve toutes les traces émanant du programme original qui respectent la politique. Ceci est équivalent à dire que toutes les traces de P qui respectent φ sont des traces possibles de P'. Formellement :

$$\forall \tau \in \mathcal{T} : ((\tau \vdash \varphi) \wedge (\tau \in P)) \Rightarrow \tau \in P' \tag{6.2}$$

Par la suite, nous présentons le processus de renforcement détaillé. Dans la deuxième partie du chapitre, nous énonçons et prouvons les deux théorèmes de correction et de complétude.

6.2 Notations

Nous rappelons tout d'abord, les notations suivantes :

- $\mathcal{P}$ est l'ensemble de programmes possibles de $\mathbb{K}$-*Smali* ;
- $\mathcal{A}$ représente l'ensemble de toutes les actions possibles dans $\mathcal{P}$;
- $\mathcal{A}^+ = \mathcal{A} \setminus \{nop\}$ représente l'ensemble de toutes les actions possibles excepté l'action nop ;
- Th est l'ensemble des threads dans $\mathcal{P}$;
- $\mathcal{E}$ est l'ensemble des environnements possibles dans $\mathcal{P}$;
- $\mathcal{T}$ représente l'ensemble des traces possibles construites à partir des actions atomiques, tel que pour $\tau \in \mathcal{T}$, τ peut être :
  - ϵ, la trace vide ;
  - a, une action atomique ;
  - $\tau.\tau$, la concaténation de deux traces. Par exemple, la trace τ=$a.b.c$ représente l'exécution de l'action atomique a suivie de l'action b, suivie de c ;
  - τ^*, une réplication finie de τ ;
  - τ^ω, une réplication infinie de τ ;
- $\forall \tau \in \mathcal{T},\ Prefx(\tau) = \{\tau_1 \mid \exists \tau_2 \in \mathcal{T}\ et\ \tau_1 \tau_2 = \tau\}$.

Comme expliqué dans le chapitre précédent, la configuration d'un programme $\mathbb{K}$-*Smali* est constituée d'un ensemble de cellules. Parmi ces cellules, il y'en a celles qui représentent l'environnement et d'autres qui constituent les threads (voir Figure 5.3). Un processus $P \in \mathcal{P}$ est dénoté par :
$P =< T,\ E >$, où :

- $T \in Th$ est le thread en exécution qui représente la cellule $\langle\ \rangle_{Thread}$;
- $E \in \mathcal{E}$ est l'environnement. Il inclut toutes les cellules et les sous-cellules d'environnement, telles que $\langle\ \rangle_{Heap}$ et $\langle\ \rangle_{Registers}$, $\langle\ \rangle_{Object}$, etc.

Dans ce qui suit, seules les cellules ou les sous-cellules pertinentes dans T et E seront mentionnées.

6.3 Idée de renforcement

L'idée de renforcement consiste à créer un canal entre le programme et la politique qui leur permet de se communiquer et se synchroniser. À travers ce canal de synchronisation, nous voulons créer une sorte de connexion entre les deux à partir de laquelle la politique peut contrôler (bloquer et débloquer) certaines actions du programme.

Pour obliger une action a à ne pas s'exécuter que lorsqu'elle reçoit une autorisation de la part de la politique de sécurité, il suffit de la précéder d'une action bloquante qui sera tout simplement débloquée par la politique de sécurité. Cette tâche est simple si le langage de programmation nous offre une action $receive(c)$ bloquante sur un canal c et une action $send(c)$ qui permet de la débloquer. Dans ce cas, il suffit de remplacer le a par $receive(c)$ a $send(c)$. L'action $receive(c)$ restera bloquée jusqu'à ce qu'elle sera débloquée par une action $send(c)$ provenant de la politique de sécurité pour autoriser le début de a. Quand à l'action $send(c)$, elle signalera à la politique de sécurité la fin de a, une information souvent utile pour débloquer d'autres actions.

Malheureusement, le langage Smali n'a nativement ni $send$ ni $receive$. Il va falloir donc les émuler avec les moyens de bord et particulièrement via des écritures bloquantes dans des zones mémoires partagées. Plus précisément, la création d'un canal de synchronisation a été inspirée de la notion de file bloquante *BlockingQueue*, utilisée en Java dans le cadre d'exécution concurrente [164]. Il s'agit d'une file d'attente dans laquelle plusieurs threads peuvent insérer et retirer des éléments simultanément. Le blocage survient lorsqu'un thread tente d'ajouter ou de retirer un élément qui ne se trouve pas dans la file, il est alors bloqué jusqu'à ce qu'il y ait un élément à saisir. Avec le même raisonnement, le canal de synchronisation que nous proposons est équivalent à la file d'attente bloquante. Le programme et la politique partagent ce canal. Le programme ne peut pas avancer tant qu'il n'a pas reçu de confirmation de la part de la politique. De même, la politique ne peut pas procéder au traitement suivant tant qu'elle n'a pas reçu une notification indiquant la fin de l'action qu'elle a autorisée de la part du programme.

Une même politique de sécurité peut contrôler plusieurs actions appartenant à un ou plusieurs thread. Chaque action doit savoir quand est ce qu'elle pourrait commencer et elle doit signaler sa fin une fois terminée. Le signal de début doit être reçu sur un canal (zone mémoire) connue par l'action et le signal de la fin doit être envoyé sur un canal connu par la politique de sécurité. Une solution naïve serait de créer un canal pour chaque action et pour chaque direction de communication (de la politique vers le programme et vice-versa). Cependant, ceci engendre une consommation substantielle de la mémoire par le processus de renforcement. Une autre solution consiste à utiliser seulement un ou quelques canaux par direction. Pour savoir si elle est autorisée à avancer, l'action doit trouver son nom inscrit dans le canal qui arrive au programme. De l'autre côté, pour savoir si une action a terminé, la politique doit trouver le nom de l'action dans le canal qui arrive du programme. Nous pouvons même optimiser davantage en utilisant le même canal pour les deux directions. Dans ce cas, une action peut commencer si elle trouve son nom (par exemple "move") suivi de la chaîne "-s" (une abréviation de *start*) pour indiquer la possibilité de commencer et quand elle termine, elle écrit dans le canal son nom

suivi de "-e" (une abréviation de *end*) pour signaler sa fin à la politique de sécurité. En d'autres mots, une instruction comme "move" doit recevoir sur un canal c la chaîne "move-s" pour qu'elle puisse commencer et elle doit envoyer sur c "move-e" pour signaler sa fin à la politique de sécurité.

Concrètement, l'envoie et la réception sont des actions de synchronisation émulés par des fonctions appelées $send$ et $receive$. Ces actions sont complémentaires et seront ajoutées au programme et dans la politique de façon qui permettent leurs synchronisations une fois exécutés en parallèle. Cette notion a été inspirée du travail [64]. Prenons un exemple pour clarifier ce concept. Considérons le programme $P = a \parallel b$ et une politique de sécurité $\varphi = a.b$. Pour renforcer φ dans P, nous commençons par transformer la politique en un processus contrôleur. Pour ce faire, la formule φ sera transformée en $send(\text{"a-s"}, c)\ receive(\text{"a-e"}, c)\ send(\text{"b-s"}, c)\ receive(\text{"b-e"}, c)$. Le programme de son côté doit être réécrit de façon à inclure les actions de synchronisation complémentaires à celles ajoutées à la formule de la façon suivante $receive(\text{"a-s"}, c)\ a\ send(\text{"a-e"}, c) \parallel receive(\text{"b-s"}, c)\ b\ send(\text{"b-e"}, c)$. Ensuite, le programme et la politique sont exécutés en parallèle de la manière suivante : $receive(\text{"a-s"}, c)\ a\ send(\text{"a-e"}, c) \parallel receive(\text{"b-s"}, c)\ b\ send(\text{"b-e"}, c) \parallel send(\text{"a-s"}, c)\ receive(\text{"a-e"}, c)\ send(\text{"b-s"}, c)\ receive(\text{"b-e"}, c)$.Le processus final est un programme qui respecte la politique, il exécute l'action a après l'action b comme l'exige la politique. De cette façon, la politique de sécurité est en mesure de débloquer des actions particulières dans le programme et aussi d'imposer un ordre temporel sur les exécutions de P.

Les actions de synchronisation $send$ et $receive$ sont des abréviations de séquence d'instructions $\mathbb{K}$-*Smali* implémentées et compilées dans $\mathbb{K}$. Leur code sera donné plus loin dans le chapitre. Les différentes sections qui suivent représentent les étapes adoptées pour le renforcement d'une politique de sécurité.

6.4 Spécification formelle de politique de sécurité

L'objectif ici est de se doter d'un formalisme qui nous permettra de spécifier des propriétés de sécurité. Par exemple, lorsqu'un utilisateur clique pour enregistrer un audio, il s'attend à ce que le fait de cliquer sur un bouton de démarrage ou d'arrêt lance ou arrête l'enregistrement, et que ce soit la seule façon dont une application enregistre un audio. Cette attente contient une composante logique et une composante temporelle, et peut être formellement exprimée par une formule de logique temporelle. Une politique de sécurité devrait exiger par conséquent à la fois un ordre temporel particulier, comme appeler le gestionnaire d'événement du bouton d'arrêt avant d'appeler l'API permettant d'arrêter l'enregistrement et que les ressources sensibles ne soient pas accédées par des tâches d'arrière-plan. Tout ça pour empêcher l'application d'enregistrer subrepticement des audios. Comme les propriétés peuvent être plus complexes, il est primordial que la logique soit suffisamment expressive et intuitive afin de spécifier un large éventail de propriétés de sécurité.

Pour cette raison, nous choisissons une variante proche de la logique temporelle linéaire LTL comme langage de spécification des politiques de sécurité. La logique adoptée est dénotée par LTL_{φ}. En ef-

fet, l'ensemble intégré de variables propositionnelles et d'opérateurs logiques et temporels, permet de vérifier des propriétés temporelles sur un modèle linéaire (une séquence d'actions ou d'événements) ainsi que le comportement itératif d'un programme. Outre l'expressivité de la logique , notre choix est justifié par sa facilité d'utilisation et sa capacité à exprimer les propriétés de manière simple et naturelle. De plus, nous voyons qu'elle peut se marier adéquatement avec $\mathbb{K}$-*Smali*, étant donné qu'elle offre des constructeurs syntaxiques qui peuvent être traduit facilement en Smali. Dans ce qui suit nous donnons la syntaxe et la sémantique de LTL_φ.

6.4.1 Syntaxe

Table 6.1 présente la syntaxe de la logique LTL_φ écrite dans la grammaire BNF. Il s'agit d'un ensemble de formules (rangée sur φ_1 et φ_2). *tt* et *ff* représentent respectivement les constantes booléennes vrai et faux. 1 est une action vide. a est une action atomique dans $\mathcal{A}^+$. La formule contient également les connecteurs propositionnels standards de négation($\neg$), de conjonction ($\wedge$) et de disjonction ($\vee$). La logique temporelle linéaire peut être définie de manière équivalente en utilisant divers opérateurs temporels. Dans notre cas, nous employons l'opérateur (.) qui représente la composition séquentielle. L'opérateur de *kleene* (*) [165] est utilisé pour exprimer l'itération.

$\varphi ::= tt \mid ff \mid 1 \mid a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1.\varphi_2 \mid \varphi_1^*\varphi_2$

TABLE 6.1 – Syntaxe de la logique LTL_φ

6.4.2 Sémantique

La sémantique de la logique LTL_φ est donnée par la fonction $[\![-]\!]$, définie dans 6.4.1.

Définition 6.4.1. *(Sémantique d'une formule $[\![-]\!]$)*
La sémantique d'une formule LTL_φ est définie par la fonction $[\![-]\!] : LTL_\varphi \longrightarrow 2^{\mathcal{T}}$ telle que présentée dans Table 6.2.

Dans ce qui suit nous donnons le sens intuitif de chaque équation sémantique dans Table 6.2. Toutes les traces satisfont *tt* (2.1) et aucune trace ne satisfait *ff* (2.2). La trace vide ϵ respecte la séquence vide 1 (2.3). Seule la trace formée de l'action atomique a respecte la formule a (2.4). La sémantique d'une négation de formule est le complément de sa sémantique (2.5).

Une trace respecte la formule $\varphi_1 \wedge \varphi_2$ si elle respecte φ_1 et φ_2 (2.6). Une trace respecte la formule $\varphi_1 \vee \varphi_2$ si elle respecte φ_1 ou φ_2 (2.7). La sémantique d'une composition séquentielle de deux formules φ_1 et φ_2 est la concaténation des traces qui respectent φ_1 avec celles qui respectent φ_2 (2.8).

Une trace appartenant à $[\![\varphi_1^*\varphi_2]\!]$ si elle est une répétition infinie de φ_1 ou bien la composition finie d'une trace τ_1 (zéro ou plusieurs fois) concaténées à un certain nombre de traces qui satisfont φ_2. Étant

donnée une formule $\varphi = (\neg lire)^*(lire.(\neg connecter))$. La trace $\tau = ouvrir.ecrire.lire.copier$ satisfait φ. En fait, τ est composée de deux traces. Une trace $\tau_1 = ouvrir.ecrire$ qui satisfait $(\neg lire)^*$ suivie de la trace $\tau_2 = lire.copier$ qui satisfait $(lire.\neg(connecter))$.

$$[\![tt]\!] = \mathcal{T} \tag{2.1}$$

$$[\![ff]\!] = \emptyset \tag{2.2}$$

$$[\![1]\!] = \{\epsilon\} \tag{2.3}$$

$$[\![a]\!] = \{a\} \tag{2.4}$$

$$[\![\neg\varphi]\!] = \mathcal{T} \setminus \{\varphi\} \tag{2.5}$$

$$[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!] \tag{2.6}$$

$$[\![\varphi_1 \vee \varphi_2]\!] = [\![\varphi_1]\!] \cup [\![\varphi_2]\!] \tag{2.7}$$

$$[\![\varphi_1.\varphi_2]\!] = \{\tau_1.\tau_2 | \tau_1 \in [\![\varphi_1]\!] \ et \ \tau_2 \in [\![\varphi_2]\!]\} \tag{2.8}$$

$$[\![\varphi_1^*\varphi_2]\!] = \begin{cases} \tau^\omega \quad avec \ \tau \in [\![\varphi_1]\!] & si \ \ [\![\varphi_2]\!] = \emptyset \\ \{\tau_1^*.\tau_2 \mid \tau_1 \in [\![\varphi_1]\!] \ et \ \tau_2 \in [\![\varphi_2]\!]\} & sinon \end{cases} \tag{2.9}$$

TABLE 6.2 – Sémantique de la logique LTL_φ

La fonction sémantique $[\![\varphi]\!] \subseteq \mathcal{T}$ prend une formule LTL et retourne l'ensemble de traces qui la satisfait. Dans ce qui suit nous définissons la notion de satisfaction d'une formule par un programme.

Définition 6.4.2. *(Relation de satisfaction $\vdash$)*
Soient un ensemble de traces $T \in \mathcal{T}$ et une formule $\varphi \in LTL_\varphi$, nous définissons la relation $\vdash \subseteq \mathcal{T} \times LTL_\varphi$ comme suit :

$$T \vdash \varphi \ \ si \ \ T \subseteq [\![\varphi]\!]$$

6.5 Transformation de la formule LTL

Afin de renforcer une formule LTL au sein d'un programme donné, il faut la transformer en un moniteur. Le moniteur comme expliqué précédemment est un programme qui s'exécute en parallèle avec le programme supervisé, qui intercepte, analyse, ensuite autorise ou non l'exécution de certaines actions. Dans notre cas, ce moniteur est exprimé par une politique de sécurité qui est à son tour décrite à l'aide de la logique temporelle linéaire LTL_φ. Pour transformer une formule LTL_φ en un processus contrôleur, nous devons dans un premier temps ajuster sa forme syntaxique afin qu'elle s'intègre avec les constructions de ce langage, et ce sans changer sa sémantique. Par la suite, nous présentons les étapes adoptées pour transformer une formule LTL_φ donnée en un moniteur qui génère les mêmes traces.

6.5.1 Forme simplifiée de la logique

- *Élimination de l'opérateur de négation* : La négation ne fait pas partie de constructeurs du langage Smali. Ainsi, nous devons éliminer la forme $\neg\varphi$ de la formule. Intuitivement, l'idée consiste à pousser l'opérateur de négation à l'intérieur de la formule de façon que sa portée soit limitée aux actions atomiques. Ensuite, nous remplaçons la négation des actions élémentaires par la disjonction des actions complémentaires dans $\mathcal{A}$. Table 6.3 présente le système de réécriture qui permet de calculer de telles simplifications.

$\neg tt$	$\longrightarrow$	ff	(3.1)
$\neg ff$	$\longrightarrow$	tt	(3.2)
$\neg 1$	$\longrightarrow$	$\bigvee_{x\in\mathcal{A}^+} x$	(3.3)
$\neg\neg\varphi$	$\longrightarrow$	φ	(3.4)
$\neg a$	$\longrightarrow$	$\bigvee_{x\in\mathcal{A}^+\setminus\{a\}} x$	(3.5)
$\neg(\varphi_1 \wedge \varphi_2)$	$\longrightarrow$	$\neg\varphi_1 \vee \neg\varphi_2$	(3.6)
$\neg(\varphi_1 \vee \varphi_2)$	$\longrightarrow$	$\neg\varphi_1 \wedge \neg\varphi_2$	(3.7)
$\neg(\varphi_1.\varphi_2)$	$\longrightarrow$	$\neg\varphi_1 \vee \varphi_1.\neg\varphi_2$	(3.8)
$\neg(\varphi_1^*\varphi_2)$	$\longrightarrow$	$\varphi_1^*\neg\varphi_2 \vee \neg\varphi_1^*\varphi_2$	(3.9)

TABLE 6.3 – Forme simplifiée de la logique : élimination de l'opérateur de négation

L'exemple suivant montre comment le système de réécriture peut être appliqué pour pousser l'opérateur de négation vers les actions élémentaires.

Exemple 6.5.1.
Soit la formule $\varphi = \neg(getSubscriberId.sendTextMessage)$ qui empêche une application Android de voler une information privée. En effet, une application peut accéder à une information privée comme l'ID de téléphone en appelant la méthode *getSubscriberId* et de l'envoyer ensuite, en appelant la méthode *sendTextMessage* à un autre téléphone. L'élimination de la négation dans ce cas s'effectue comme suit :
$\neg(getSubscriberId.sendTextMessage) \xrightarrow{(3.8)} \neg getSubscriberId \vee getSubscriberId.\neg sendTextMessage$
Ensuite l'action $\neg getSubscriberId$ est remplacée par toute autre action dans le programme sauf l'action *getSubscriberId* en appliquant (3.5). De même pour l'action $\neg sendTextMessage$.

- *Élimination de l'opérateur de conjonction* : L'opérateur de conjonction n'apparaît pas non plus dans la construction Smali. La conjonction peut être éliminée en la transformant en des renfor-

cements successifs après avoir calculé la forme normale conjonctive (FNC) de la formule. Table 6.4 présente le calcul de la FNC d'une formule LTL_{φ}.

$\bigwedge_{i=1}^{n} \varphi_i \vee \bigwedge_{j=1}^{n} \varphi_j$	$=$	$\bigwedge_{i,j=1}^{n} (\varphi_i \vee \varphi_j)$
$\bigwedge_{i=1}^{n} \varphi_i \cdot \bigwedge_{j=1}^{n} \varphi_j$	$=$	$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (\varphi_i \cdot \varphi_j)$
$\bigwedge_{i=1}^{n} \varphi_i^* \bigwedge_{j=1}^{n} \varphi_j$	$=$	$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} (\varphi_i^* \varphi_j)$

Table 6.4 – Forme simplifiée de la logique : élimination de l'opérateur de conjonction

Comme illustré dans Table 6.4, une formule est sous FNC si elle est une conjonction d'un ou plusieurs disjonctions ou compositions d'une ou plusieurs formules qui ne contiennent pas de conjonction φ_i. Ainsi, le renforcement de P par $\bigwedge_{i=1}^{n} \varphi_i$ (noté $P \sqcap \bigwedge_{i=1}^{n} \varphi_i$) sera remplacé par un renforcement successif de φ_i, comme suit :

$$P \sqcap \bigwedge_{i=1}^{n} \varphi_i = (((P \sqcap \varphi_1) \sqcap \varphi_2) \ldots \sqcap \varphi_n))$$

avec $\sqcap$ un opérateur de renforcement défini dans Section 6.7.

En appliquant les deux transformations précédentes, nous obtenons une formule de la forme suivante :

$$\bigwedge_{i=1}^{n} \varphi_i$$

telle que :

(i) La portée de l'opérateur de négation est restreinte aux actions atomiques ;

(ii) φ_i *ne contient pas d'opérateur conjonctif.*

Nous désignons par $LTL_{\Downarrow}$ la logique obtenue après simplification des formules LTL_{φ}. Table 6.5 représente sa syntaxe. Dans le reste du document, toutes les formules auxquelles nous faisons référence sont des formules LTL simplifiées dans $LTL_{\Downarrow}$.

$\varphi ::=$	$ff \mid tt \mid 1 \mid a \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 . \varphi_2 \mid \varphi_1^* \varphi_2$

Table 6.5 – Syntaxe de la logique LTL_{φ} simplifiée $LTL_{\Downarrow}$

6.5.2 De la logique LTL vers un moniteur $\mathbb{K}$-*Smali*

Comme mentionné à la section précédente, afin d'intégrer une formule donnée dans un programme, sa forme syntaxique doit se conformer au langage de spécification de ce dernier. Une fois fait, la prochaine étape consiste à la transformer en un moniteur qui soit en mesure de contrôler des actions spécifiques dans la cible.

La transformation d'une formule LTL nécessite d'introduire en premier temps les notions de premières actions acceptées par une formule, sa partie résiduelle une fois une action est acceptée et la possibilité d'inclure une séquence vide. La définition de ces notions est présentée dans Table 6.6.

Définition 6.5.1. *(δ,∂ et o)*
Soient φ_1, φ_2 deux formules de LTL_φ, nous définissons les fonctions δ,∂ et o comme suit :

$\delta(-) : LTL_{\Downarrow} \longrightarrow 2^{\mathcal{A}}$	$\partial_a(-) : LTL_{\Downarrow} \longrightarrow LTL_{\Downarrow}$	$o(-) : LTL_{\Downarrow} \longrightarrow LTL_{\Downarrow}$
$\delta(tt) = \mathcal{A}$	$\partial_a(tt) = tt$	
$\delta(ff) = \emptyset$	$\partial_a(ff) = ff$	$o(tt) = 1$
$\delta(1) = \emptyset$	$\partial_a(1) = ff$	$o(ff) = ff$
$\delta(a) = a$	$\partial_a(a) = 1$	$o(1) = 1$
$\delta(\varphi_1.\varphi_2) = \begin{cases} \delta(\varphi_1) & si\ o(\varphi_1) = ff \\ \delta(\varphi_1) \cup \delta(\varphi_2) & si\ o(\varphi_1) = 1 \end{cases}$	$\partial_a(b) = ff$	$o(a) = ff$
	$\partial_a(\varphi_1.\varphi_2) = \begin{cases} \partial_a(\varphi_1).\varphi_2 \vee \partial_a(\varphi_2) & si\ o(\varphi_1) = ff \\ \partial_a(\varphi_2) & si\ o(\varphi_1) = 1 \end{cases}$	$o(\varphi_1.\varphi_2) = o(\varphi_1) \wedge o(\varphi_2)$
$\delta(\varphi_1 \vee \varphi_2) = \delta(\varphi_1) \vee \delta(\varphi_2)$		$o(\varphi_1 \vee \varphi_2) = o(\varphi_1) \vee o(\varphi_2)$
$\delta(\varphi_1^* \varphi_2) = \delta(\varphi_1) \vee \delta(\varphi_2)$	$\partial_a(\varphi_1 \vee \varphi_2) = \partial_a(\varphi_1) \vee \partial_a(\varphi_2)$	$o(\varphi_1^* \varphi_2) = o(\varphi_2)$
	$\partial_a(\varphi_1^* \varphi_2) = \partial_a(\varphi_1).\varphi_1^* \varphi_2 \vee \partial_a(\varphi_2)$	

TABLE 6.6 – Définitions de fonctions δ, ∂ et o

- $\delta(\varphi)$ retourne les premières actions permises par la formule φ. Par exemple $\delta(a.(b \vee c)) = a$;
- $\partial_a(\varphi)$ représente la dérivée de la formule φ par rapport à une action a. Intuitivement, c'est la partie résiduelle de φ une fois l'action a est acceptée. Par exemple, $\partial_c(a.b.c) = 1$;
- $o(\varphi)$ détermine si la séquence vide ϵ appartient à $[[\varphi]]$. Elle retourne 1 si oui et ff sinon.

Comme expliqué au début du chapitre, la synchronisation entre le programme et la politique de sécurité se fait via un canal de synchronisation. Il s'agit d'une variable dans la mémoire. Concrètement, cette variable correspond à un tableau de chaîne de caractères c partagé entre le programme et la politique. Ces deux derniers synchronisent en lisant et en écrivant dans ce tableau. Afin de déterminer la taille de ce tableau de canaux, nous avons besoin tout d'abord de prolonger les définitions de δ et de ∂ en $\vec{\delta}(\varphi)$ et $\vec{\partial}(\varphi)$ comme suit.

Définition 6.5.2. *($\vec{\partial}(\varphi)$, $\vec{\delta}(\varphi)$)*
Nous prolongeons les définitions de δ et de ∂ en $\vec{\delta}(\varphi)$ et $\vec{\partial}(\varphi)$ de la manière suivante :

(a) $\vec{\partial}(\varphi)$ représente le plus petit ensemble qui contient les éléments suivants :

(i) $\varphi \in \vec{\partial}(\varphi)$

(ii) *si* $\psi \in \vec{\partial}(\varphi)$ *et* $a \in \delta(\psi)$ *alors* $\partial_a(\psi) \in \vec{\partial}(\varphi)$

(b) $\vec{\delta}(\varphi)$ *représente le plus petit ensemble contenant les éléments suivants :*

(i) *si* $\psi \in \vec{\partial}(\varphi)$ *alors* $\delta(\psi) \in \vec{\delta}(\varphi)$

Exemple 6.5.2.
Soit une formule $\varphi = a \vee b$

D'après la définition 6.5.2 *(a) (i)*, nous avons : $a \vee b \in \vec{\partial}(a \vee b)$
D'après la définition 6.5.2 *(a) (ii)* :
$a \vee b \in \vec{\partial}(a \vee b)$ et $a \in \delta(a \vee b)$ alors $a \in \vec{\partial}(a \vee b)$
$a \vee b \in \vec{\partial}(a \vee b)$ et $b \in \delta(a \vee b)$ alors $b \in \vec{\partial}(a \vee b)$
$a \in \vec{\partial}(a \vee b)$ et $a \in \delta(a)$ alors $1 \in \vec{\partial}(a \vee b)$
$b \in \vec{\partial}(a \vee b)$ et $b \in \delta(b)$ alors $1 \in \vec{\partial}(a \vee b)$
Par la suite $\vec{\partial}(a \vee b)$ représente le plus petit ensemble qui contient les éléments suivants : $\{a \vee b, a, b, 1\}$.

D'après la définition 6.5.2 *(b) (i)*, nous avons :
$a \vee b \in \vec{\partial}(a \vee b)$ alors $\{a, b\} \in \vec{\delta}(a \vee b)$
$a \in \vec{\partial}(a \vee b)$ alors $a \in \vec{\delta}(a \vee b)$
$b \in \vec{\partial}(a \vee b)$ alors $b \in \vec{\delta}(a \vee b)$
$1 \in \vec{\partial}(a \vee b)$ alors $\emptyset \in \vec{\delta}(a \vee b)$
Par la suite, $\vec{\delta}(a \vee b)$ représente le plus petit ensemble contenant les éléments suivants : $\{\{a, b\}, \{a\}, \{b\}, \emptyset\}$.

Nous avons besoin également de définir une fonction $size_\delta(-)$ qui calcule la taille de canal de synchronisation comme suit.

Définition 6.5.3. *($size_\delta(\varphi)$)*
Nous définissons $size_\delta(-) : LTL \longrightarrow \mathbb{N}$ *de la manière suivante :*

$$size_\delta(\varphi) = max_{x \in \vec{\delta}(\varphi)}(|\vec{\delta}(x)|)$$

avec $|-|$ *retourne la cardinalité d'un ensemble.*

Exemple 6.5.3.
Considérons une formule $\varphi = a \vee b$. Dans l'exemple précédent, nous avons trouvé que $\vec{\delta}(a \vee b) = \{\{a, b\}, \{a\}, \{b\}, \emptyset\}$. À partir de ces résultats, nous calculons la taille du canal de synchronisation comme suit :

$size_\delta(a \vee b) = max(|\{\{a, b\}, \{a\}, \{b\}, \emptyset\}|) = 2$

La fonction de transformation d'une formule en un processus contrôleur $\mathbb{K}$-*Smali* s'effectue via une fonction $[\![-]\!]_c$ définie comme suit.

Définition 6.5.4. *(Fonction de transformation d'une formule $[\![-]\!]_c$)*
Soient φ une formule de $LTL_{\Downarrow}$, E un environnement dans $\mathcal{E}$ et c un vecteur de canaux de synchronisation de taille n. La fonction de transformation d'une formule $[\![-]\!]_c$ est définie comme suit :

$$[\![-]\!]_c : LTL_{\Downarrow} \longrightarrow \mathcal{P}$$

$$[\![\varphi]\!]_c = < F_{c,\emptyset}(\varphi),\ < c \mapsto \text{"}undefined\text{"} >_{\textbf{FieldsObject}} \dagger E >$$

avec F définie pour tout c et pour tout $\sigma : \{LTL_{\Downarrow} \longrightarrow String\}$ un Map, dans Table 6.7.

La fonction $[\![-]\!]_c$ ajoute la valeur "undefined" dans chaque case du vecteur de synchronisation c. Les fonctions $\ulcorner -$ et $- \urcorner$ prennent en paramètres une instruction $\mathbb{K}$-*Smali*. Elles convertissent l'instruction en une chaîne de caractères, ensuite la concatènent au caractère "s" (*start)* et "e" (*end*), respectivement. Pour ce faire, les deux fonctions font appel aux fonctions *ToString* et *Concat*. Les définitions de $\ulcorner -$ et $- \urcorner$ se présentent comme suit.

Définition 6.5.5. *($\ulcorner -$ et $- \urcorner$)*
Soit $a \in \mathcal{A}$, nous définissons $\ulcorner - : \mathcal{A} \times String \longrightarrow String$ et $- \urcorner : \mathcal{A} \times String \longrightarrow String$ comme suit :

$$\ulcorner a = Concat(ToString(a), \text{"-}s\text{"}) \ \ et \ \ a \urcorner = Concat(ToString(a), \text{"-}e\text{"})$$

avec :

- *$ToString : \mathcal{A} \longrightarrow String$: une fonction permettant la conversion d'une commande en une chaîne de caractère ;*
- *$Concat : String \times String \longrightarrow String$: une fonction permettant de concaténer deux chaînes de caractères.*

Exemple 6.5.4.
Soit l'instruction $a = move$:
$\ulcorner move = Concat(ToString(a), \text{"-}s\text{"}) = \text{"}move\text{-}s\text{"}$
$move \urcorner = Concat(ToString(a), \text{"-}e\text{"}) = \text{"}move\text{-}e\text{"}$

D'une manière intuitive, une formule *tt* est transformée en un processus $\mathbb{K}$-*Smali* $(\bigvee_{a \in \mathcal{A}} a)^* \vee (\bigvee_{a \in \mathcal{A}} a)^{\omega}$, qui accepte toutes les traces (incluant les traces finies et infinies). Étant donné que l'opérateur $\vee$ ne fait pas partie des constructeurs du langage Smali, nous avons utilisé l'instruction Smali *sparse-switch* pour exprimer le choix. Comme nous l'avons expliqué dans le chapitre précédent, l'instruction *sparse-switch* utilise un registre de test et une table de consultation *Switchtab*. En fonction de la valeur dans ce registre, si elle correspond à la valeur clé, un saut vers son offset de saut correspondant est effectué. S'il n'y a pas de correspondance dans la table, l'exécution continue avec l'instruction suivante. Afin de

parcourir toutes les actions possibles dans $\mathcal{A}$, le registre de test contiendra toujours une valeur choisie aléatoirement (avec la fonction *random()*). Pour chaque valeur dans $\mathbb{N}$, il existe un offset contenant l'action correspondante parmi toutes les actions de $\mathcal{A}$. (6.1).

$F_{c,\sigma} : LTL_{\Downarrow} \longrightarrow \mathbb{K}$-*Smali*			
$F_{c,\sigma(tt)}$	=**const** v, *random(val)* **sparse-switch** v, *:switchtab* :switchtab **.sparse-switch** 1 -> *:switch_1* 2-> *:switch_2* ... n -> *:switch_n* **.end sparse-switch** *:switch_1* a *:switch_2* b ... *:switch_n* z	avec $\{a, b, ..., z\} \in A$	(6.1)
$F_{c,\sigma}(ff)$	$= \top$		(6.2)
$F_{c,\sigma}(1)$	= **nop**		(6.3)
$F_{c,\sigma}(a)$	$= send(\ulcorner a, c)\ receive(a\urcorner, c)$		(6.4)
$F_{c,\sigma}(\varphi_1.\varphi_2)$	$= send(\ulcorner a, c)\ receive(a\urcorner, c)\ F_{c,\sigma}(\partial_{a_1}(\varphi_1.\varphi_2))$	$\delta(\varphi_1.\varphi_2) = \{a_1\}$	(6.5)
$F_{c,\sigma}(\varphi)$	$= send(\ulcorner a, c)\ receive(a\urcorner, c)\ F_{c,\sigma}(\partial_{a_1}(\varphi))$	$\delta(\varphi) = \{a_1\}$	(6.5.1)
$F_{c,\sigma}(\varphi_1 \vee \varphi_2)$	$= send(< \ulcorner a_1, \ldots, \ulcorner a_n >, c)$ $receive(< a_1\urcorner, \ldots, a_n\urcorner >, c)$ **sparse-switch** vx, : *switchtab* : *switchtab* **.sparse-switch** $a_1\urcorner \rightarrow$: *switch_1* ... $a_n\urcorner\rightarrow$: *switch_n* : **.end sparse-switch** : *switch_1* $F_{c,\sigma}(\partial_{a_1}(\varphi_1 \vee \varphi_2))$ ... : *switch_n* $F_{c,\sigma}(\partial_{a_n}(\varphi_1 \vee \varphi_2))$	$\delta(\varphi_1 \vee \varphi_2) = \{a_1, \ldots, a_n\}$	(6.6)
$F_{c,\sigma}(\varphi_1^*\varphi_2)$	$= \begin{cases} \textbf{goto } lab & si\ \{(\varphi_1^*\varphi_2) \mapsto lab\} \in \sigma \\ : lab & sinon \\ F_{c,\sigma\dagger[(\varphi_1^*\varphi_2)\mapsto lab]}(\varphi_1.(\varphi_1^*\varphi_2) \vee \varphi_2) & \end{cases}$	(*lab* est un label fraîchement généré)	(6.7)

TABLE 6.7 – Transformation d'une formule LTL

La formule ff est traduite en un processus bloqué "$\top$" (6.2). Une façon d'exprimer le blocage est un code $\mathbb{K}$-*Smali* qui aboutit vers un blocage. La séquence vide 1 est traduite en l'instruction *nop*, qui correspond à aucune opération dans Smali (elle est équivalente formellement à une trace vide) (6.3).

Une action élémentaire a est transformée en un processus composé de deux actions de synchronisation successives $send(\ulcorner a, c)$ et $receive(a\urcorner, c)$ (l'espace représente la composition séquentielle dans Smali) (6.4). $send$ et $receive$ sont deux méthodes $\mathbb{K}$-*Smali*, implémentées dans $\mathbb{K}$. Leurs définitions est données dans 6.7.3 et 6.7.4.

La transformation d'une formule de la forme $\varphi_1.\varphi_2$ est donnée dans (6.5). L'exemple suivant illustre cette transformation.

Exemple 6.5.5.
Soit une formule LTL $\varphi = a.b$. La transformation de φ s'effectue comme suit :

$$\begin{aligned} F_{c,\emptyset}(a.b) &\overset{(6.5)}{=} send(\ulcorner a, c)\ receive(a\urcorner, c)\ F_{c,\sigma}(\partial_a(a.b)) \qquad \delta(a.b) = \{a\} \\ &= send(\ulcorner a, c)\ receive(a\urcorner, c)\ F_{c,\emptyset}(b) \\ &\overset{(6.4)}{=} send(\ulcorner a, c)\ receive(a\urcorner, c)\ send(\ulcorner b, c)\ receive(b\urcorner, c) \end{aligned}$$

Remarque $\sigma = \emptyset$ puisque la formule ne contient pas de boucle.

La transformation d'une formule de la forme $\varphi_1 \vee \varphi_2$ est donnée dans (6.5). La formule est transformée alors en une séquence d'actions. D'abord, une action $send$ qui permet d'envoyer en parallèle les noms des n actions début, retournées par la fonction δ sur le canal c. Ensuite, l'action $receive$ reçoit le nom d'une seule action parmi les n noms d'actions fin reçus et renvoie ce nom dans un registre spécial vx. Par la suite, en fonction de la valeur dans vx, l'instruction *sparse-switch* permet de se brancher sur l'étiquette correspondante pour continuer avec la partie résiduelle de la formule, calculée avec la fonction ∂.

Exemple 6.5.6.
Soit la formule $\varphi = a \vee b$. La transformation de φ s'effectue comme suit :

$$\begin{aligned} F_{c,\emptyset}(a \vee b) \overset{(6.6)}{=} & send(< \ulcorner a, \ulcorner b >, c) \qquad \delta(a \vee b) = \{a, b\} \\ & receive(< a\urcorner, b\urcorner >, c) \\ & \textbf{sparse-switch}\ vx, : switchtab \\ & : switchtab \\ & \textbf{.sparse-switch} \\ & \quad a\urcorner \rightarrow switch\_1 \\ & \quad b\urcorner \rightarrow switch\_2 \\ & \textbf{.end sparse-switch} \\ & : switch\_1 \end{aligned}$$

$$F_{c,\emptyset}(\partial_a(a \vee b)) = F_{c,\emptyset}(1) \overset{(6.3)}{=} \textbf{nop}$$

$: switch\_2$

$$F_{c,\emptyset}(\partial_b(a \vee b)) = F_{c,\emptyset}(1) \overset{(6.3)}{=} \textbf{nop}$$

Pour la formule $\varphi_1^* \varphi_2$, pour chaque boucle qui apparaît dans la formule, une nouvelle étiquette est créée. L'étiquette est suivie de la transformation de la formule $\varphi_1.(\varphi_1^* \varphi_2) \vee \varphi_2$. Chaque fois que la boucle apparaît dans la formule, si elle appartient à σ (c.-à-d. si elle est déjà traitée et possède une étiquette) alors sauter à son étiquette correspondante. Sinon, une nouvelle étiquette vers le traitement de cette formule est créée et associée à celle-ci puis ajoutée à σ avec l'opérateur "†" (6.7). σ est donc une correspondance de la formule avec un opérateur itératif vers son étiquette correspondante, initialement vide. L'exemple suivant illustre cette transformation.

Exemple 6.5.7.
Soit la formule $\varphi = (a^* b)$. La transformation de φ s'effectue comme suit :

$F_{c,\emptyset}(a^* b)$ $\overset{(6.7)}{=}$ $: lab$ $\{(a^* b) \mapsto lab\} \notin \emptyset$

$F_{c,\emptyset \dagger [(a^* b) \mapsto lab]}(a.(a^* b) \vee b)$

$F_{c,\emptyset \dagger [(a^* b) \mapsto lab]}(a.(a^* b) \vee b)$ $\overset{(6.6)}{=}$ $send(< \ulcorner a, \ulcorner b >, c)$ $\delta(a.(a^* b) \vee b) = \{a, b\}$

$receive(< a\urcorner, b\urcorner >, c)$

sparse-switch $vx,$ $: switchtab$

$: switchtab$

.sparse-switch

$a\urcorner \rightarrow switch\_1$

$b\urcorner \rightarrow switch\_2$

.end sparse-switch

$: switch\_1$

$F_{c,\{(a^* b) \mapsto lab\}}(\partial_a(a.(a^* b) \vee b))$

$= F_{c,\{(a^* b) \mapsto lab\}}(\partial_a(a.(a^* b)) \vee \partial_a(b))$

$= F_{c,\{(a^* b) \mapsto lab\}}((a^* b) \vee ff)$

$= F_{c\{,(a^* b) \mapsto lab\}}(a^* b)$

$\overset{(6.6)}{=}$ **goto** lab $(a^* b) \in \{(a^* b) \mapsto lab\}$

$: switch\_2$

$F_{c,\{(a^* b) \mapsto lab\}}(\partial_b(a.(a^* b) \vee b))$

$= F_{c,\{(a^* b) \mapsto lab\}}(\partial_b(a.(a^* b)) \vee \partial_b(b))$

$= F_{c,\{(a^* b) \mapsto lab\}}(1 \vee 1)$

$= F_{c,\{(a^* b) \mapsto lab\}}(1)$

$\overset{(6.3)}{=}$ **nop**

6.6 Transformation du programme $\mathbb{K}$-*Smali*

Le programme exécuté doit également subir des ajustements afin d'être en harmonie avec la formule. Par conséquent, nous réécrivons un programme $\mathbb{K}$-*Smali* en ajoutant des actions de synchronisation complémentaires à celles ajoutées à la formule. Les positions où ces actions sont ajoutées seront identifiées par l'ensemble A, tel que $A \subseteq \mathcal{A}$ et nous verrons par la suite que ça dépendra aussi de la nature de l'action elle-même. La fonction de réécriture d'un programme $\mathbb{K}$-*Smali* est définie dans 6.6.1.

Définition 6.6.1. *(Fonction de transformation d'un programme $\lceil - \rceil_c \; mod \; -$)*
Soit $< T, E >$ un programme dans $\mathcal{P}$ avec T un thread dans Th et E un environnement dans $\mathcal{E}$. Soit $A \subseteq \mathcal{A}$ l'ensemble d'actions à contrôler. Nous définissons la fonction de transformation d'un programme comme suit :

$$\lceil - \rceil_c \; mod \; - : \mathcal{P} \times 2^A \longrightarrow \mathcal{P}$$

$$\lceil < T, \; E > \rceil_c \; mod \; A = < L_c(S_{c,\emptyset}(T, A)), \; E >$$

S est définie, pour tout c un vecteur de canaux de synchronisation et pour tout $\Gamma : [String \longrightarrow String]$ un Map, comme suit :

$$S_{c,\Gamma} : \mathbb{K}\text{-}Smali \times 2^A \longrightarrow \mathbb{K}\text{-}Smali \times Map$$

$$S_{c,\Gamma}(a, A) = \begin{cases} (a, \Gamma) & si \; a \notin A \\ (receive(\ulcorner a, c) \; a \; send(a\urcorner, c), \; \Gamma) & si \; a \in A \; et \; a \neq goto \; :Label \; et \; a \neq if\text{-}eq \; v1 \; v2 \; : Label \\ (receive(\ulcorner a, c) \; a, \; \Gamma \dagger [: Label \mapsto a\urcorner]) & si \; a \in A \; et \; a = goto \; : Label \\ (receive(\ulcorner a, c) \; a \; send(a\urcorner, c), \; \Gamma \dagger [: Label \mapsto a\urcorner]) & si \; a \in A \; et \; a = if\text{-}eq \; v1 \; v2 \; : Label \end{cases} \quad (7.1)$$

$$S_{c,\Gamma}(P_1 \; op \; P_2, \; A) = S_{c,\Gamma}(P_1, A) \; op \; S_{c,\Gamma 1}(P_2, A) \qquad avec \; (P_1', \Gamma_1) = S_{c,\Gamma}(P_1, A) \quad (7.2)$$

La fonction L est définie pour tout c comme suit :

$$L_c : \mathbb{K}\text{-}Smali \times Map \longrightarrow \mathbb{K}\text{-}Smali$$

$$L_c(a, \Gamma) = \begin{cases} a & si \; a \neq : Label \\ a \; send \; (\Gamma(: Label), \; c) & si \; a =: Label \; et : Label \in dom(\Gamma) \end{cases} \quad (8.1)$$

$$L_c(P_1 \; op \; P_2, \; \Gamma) = L_c(P_1, \; \Gamma) \; op \; L_c(P_2, \; \Gamma) \quad (8.2)$$

Nous discutons dans ce qui suit, d'une façon intuitive, du rôle des fonctions S et L définies dans 6.5.4. La fonction S est essentiellement utilisée pour l'ajout des actions de contrôle aux actions dans A, tandis que la fonction L est utilisée spécialement pour le traitement des instructions de type étiquette (label) dans le programme.

Si $a \notin A$, autrement dit a n'est pas une action à contrôler alors S n'effectue aucune transformation sur a et Γ. Sinon, l'ajout des actions de synchronisation dépend de l'action à contrôler. Pour toute instruction a différente d'une instruction de branchement (*goto* et *if*), S ajoute les actions de contrôle complémentaires à celles ajoutées à la formule avant et après. La première est une action *receive* permettant de recevoir le nom de l'action qui marque son début $\ulcorner a$ et la deuxième est *send* permettant d'envoyer le nom de l'action qui marque sa fin $a \urcorner$ (7.1).

Dans le cas d'une instruction de branchement inconditionnelle (a=*goto Label*), la fonction S ajoute une action de début *receive* avant et ajoute également son étiquette (*Label*) avec le nom de l'action fin correspondant à l'environnement Γ. Dans ce cas, la fonction L prend en charge l'ajout d'un *send* du nom de l'action fin correspond à ce label après (7.1). L'exemple suivant illustre ce cas.

Exemple 6.6.1.
Soient $A = \{goto, move\} \subseteq \mathcal{A}$ et le thread $T \in Th$ tel que :
$T =$
$goto\ :Label1$
...
$:Label1$
$move\ v1\ v2$

La transformation de T s'effectue comme suit :

$S_{c,\emptyset}(goto\ :Label1, \{goto, move\}) \overset{(7.1)}{=} (receive("goto\text{-}s", c)\ goto\ :Label1, [\ :Label1 \mapsto "goto\text{-}e"])$ avec $goto \in A$ et $goto\ :Label1 = goto$

$S_{c,[\ :Label1 \mapsto "goto\text{-}e"]}(\ :Label1, \{goto, move\}) \overset{(7.1)}{=}\ :Label1$ avec $:Label1 \notin A$
$S_{c,[\ :Label1 \mapsto "goto\text{-}e"]}(move\ v1\ v2, \{goto, move\}) \overset{(7.1)}{=} (receive("move\text{-}s", c)\ move\ v1\ v2\ \ send("move\text{-}e", c), [\ :Label1 \mapsto "goto\text{-}e"])$ avec $move \in A$ et $move \neq goto$ et if

$L_c(receive("goto\text{-}s")\ goto\ :Label1, [\ :Label1 \mapsto "goto\text{-}e"]) \overset{(8.1)}{=} receive(as)\ goto\ :Label1$ avec $a \neq: Label$
$L_c(\ :Label1, [\ :Label1 \mapsto "goto\text{-}e"]) \overset{(8.1)}{=}\ :Label1\ \ send("goto\text{-}e", c)$ avec $a =\ :Label1$ et $:Label1 \in \Gamma$ et $dom(\Gamma) = "goto\text{-}e"$
$L_c(move\ v1\ v2, [\ :Label1 \mapsto "goto_e"]) \overset{(8.1)}{=} move\ v1\ v2$ avec $move\ v1\ v2 \neq: Label$

Par la suite :
$L_c(S_{c,\emptyset}(T, A)) =$
$receive("goto\text{-}s", c)$
$goto\ :Label1$
...
$:Label1$
$send("goto\text{-}e", c)$
$receive("move\text{-}s", c)$
$move\ v1\ v2$
$send("move\text{-}e", c)$

En ce qui concerne les instructions de branchement conditionnelle (p. ex.*if-eq, if-ne, if-lt*), S ajoute une action *receive* et *send* avant et après, et met à jour Γ pour traiter l'instruction *:Label* dans le cas où la garde s'évalue à vrai. Ensuite, pour chaque instruction de type (*:Label*) dans le programme, la fonction L ajoute une action de contrôle *send* pour marquer la fin de cet action (8.1). L'exemple suivant illustre ce cas.

Exemple 6.6.2.

Soient $A = \{\textit{if-eq}\}$ et le thread $T \in Th$, tel que :

$T =$

$const\ v1\ 30$

$const\ v2\ 10$

$\textit{if-eq}\ v1\ v2\ \ : Label$

...

$: Label$

La transformation de T s'effectue comme suit :

$S_{c,\emptyset}(const\ v1\ 30, \{\textit{if-eq}\}) \overset{(7.1)}{=} (const\ v1\ 30, \emptyset)$ avec $const \notin A$ et $const \neq goto\ et\ if$

$S_{c,\emptyset}(const\ v2\ 10, \{\textit{if-eq}\}) \overset{(7.1)}{=} (const\ v1\ 10, \emptyset)$ avec $const \notin A\ et\ const \neq goto\ et\ if$

$S_{c,\emptyset}(\textit{if-eq}\ :Label, \{\textit{if-eq}\}, c) \overset{(7.1)}{=} (receive(\textit{"if-eq-s"}, c)\ \ \textit{if-eq}\ \ send(\textit{"if-eq-e"}, c)\ :Label, [\ :Label \mapsto \textit{"if-eq-e"}])$ avec $\textit{if-eq} \in A$ et $\textit{if-eq} = \textit{if-eq}$

$S_{c,[\ :Label \mapsto \textit{"if-eq-e"}]}(\ :Label, \{\textit{if-eq}\}) \overset{(7.1)}{=}\ :Label$ avec $:Label \notin A$

$L_c(const\ v1\ 30, [\ :Label \mapsto \textit{"if-eq-e"}]) \overset{(8.1)}{=} const\ v1\ 30$ $\quad avec\ const \neq\ :Label$

$L_c(const\ v2\ 10, [\ :Label \mapsto \textit{"if-eq-e"}]) \overset{(8.1)}{=} const\ v1\ 10$ $\quad avec\ const \neq\ :Label$

$L_c(\ :Label, [\ :Label \mapsto \textit{"if-eq-e"}]) \overset{(8.1)}{=}\ :Label\ \ send(\textit{"if-eq-e"}, c)$ $\quad avec\ \ :Label \in \Gamma\ \ etdom(: Label) = \textit{"if-eq-e"}$

Par la suite :

$L_c(S_{c,\emptyset}(T, A)) =$

$const\ v1\ 30$

$const\ v2\ 10$

$receive(\textit{"if-eq-s"}, c)$

$\textit{if-eq}\ v1\ v2\ \ :Label$

$send(\textit{"if-eq-e"}, c)$

...

$:Label$

$send(\textit{"if-eq-e"}, c)$

6.7 Renforcement de politique de sécurité par réécriture de programmes

Une fois que la formule φ et le programme P sont spécifiés et transformés, l'étape suivante consiste à définir un opérateur de renforcement qui permet d'intégrer φ dans P de façon qui permet leur synchro-

nisation. Plus clairement, le but du processus de renforcement est de forcer P à exécuter uniquement les actions autorisées par φ. Cet objectif est atteignable en exécutant les deux programmes en parallèle. La définition de l'opérateur de renforcement $\sqcap$ est donnée dans 6.7.1.

Définition 6.7.1. *(Opérateur de renforcement $\sqcap$)*
Soient P un programme dans $\mathcal{P}$, A un ensemble d'actions à contrôler tel que $A \subseteq \mathcal{A}$ et φ une formule dans $LTL_{\Downarrow}$. Nous définissions l'opérateur $\sqcap_A : \mathcal{P} \times A \times LTL_{\Downarrow}$ comme suit :

$$P \sqcap_A \varphi = \lceil P \rceil_c \ mod\ A \parallel [\![\varphi]\!]_c$$

avec c un vecteur frais de canaux de synchrnisation de taille n, tel que $n = size_\delta(\varphi)$.

Si $A = \mathcal{A}$, l'opérateur $\sqcap_A$ est tout simplement noté $\sqcap$. Autrement dit, si l'indice A est omis de $\sqcap$, alors par défaut la réécriture concerne toutes les actions possibles dans P, $\mathcal{A}$.

L'opérateur parallèle $\parallel$ est défini comme suit :

Définition 6.7.2. *(Opérateur $\parallel$)*
Soient $P1$ et $P2$ deux programmes dans $\mathcal{P}$, alors $P1 \parallel P2$ est une abréviation (macro) de la liste d'instructions suivante :

```
1 new−instance v0, Lp/P1;
2 invoke− virtual {v0}, Ljava/lang/Thread;→start()V
3 new−instance v1, Lp/P2;
4 invoke− virtual {v1}, Ljava/lang/Thread;→ start()V
```

avec v0 et v1 sont deux registres locaux dans $\mathbb{K}$-Smali.

Intuitivement, la politique et le programme sont transformés en deux threads qui seront démarrés avec l'invocation de la méthode $start()$. Ils seront gérés de la même façon adoptée par Android pour gérer les différents threads.

Le code de la méthode $send$ et $receive$ est donné dans 6.7.3 et 6.7.4. Concrètement, la politique de sécurité est transformée en un thread qui s'exécute en parallèle avec le programme supervisé. Ces deux partagent un tableau de canaux c. La synchronisation sur ce tableau est établie à l'aide des instructions *monitor-enter* et *monitor-exit*. Ces instructions sont utilisées dans les deux méthodes $send$ et $receive$ afin d'assurer qu'un seul un thread (soit le programme soit la politique) peut lire ou écrire dans c.

Les méthodes $send$ et $receive$ prennent deux paramètres. Le premier est une chaîne de caractère qui représente le nom de l'action à contrôler concaténée à la chaîne début ("-s") ou fin ("-e"). Pour des raisons de simplification, nous pouvons abstraire quelques paramètres de l'instruction dans cette chaîne avec "_". Par exemple, si *a=goto :Lab*, $send$ peut envoyer le nom de l'instruction comme suit $send(\ulcorner goto\ \_, c)$. Le deuxième paramètre est l'objet canal. Il s'agit d'un tableau de chaîne de caractères.

Après avoir réservé 5 registres locaux (R_{LOCAL}), les deux méthodes commencent par récupérer l'objet canal c à partir du registre paramètre $p1$ dans le registre $v0$ (R_{IGET}). Ensuite, si le canal est libre, le thread courant peut acquérir le moniteur pour cet objet ($R_{MONITOR-ENTER}$), sinon il sera bloqué jusqu'à sa libération ($R_{MONITOR-ENTER(BLOCK)}$).

Les deux méthodes utilisent l'instruction *array-length* afin de connaitre la taille du tableau. Un compteur (ou index) est initialisé à zéro. Après avoir lu la première case, le compteur est incrémenté pour passer à la case suivante. L'instruction *if-lt* teste si la fin du tableau est atteinte. Si c'est le cas, le moniteur est libéré. Sinon, boucler et répéter le même traitement pour la case suivante.

Les instructions *aget* et *aput* permettent de lire et écrire à partir d'un tableau. La méthode *send* (6.7.3) teste si elle peut écrire sur le canal (c.à-d. la valeur dans chaque case de c est égale à "undefined") (R_{IF-EQ}). Si oui, le thread qui détient le moniteur peut écrire sur le canal le nom de l'instruction (R_{APUT}). Ce nom se trouve dans le premier registre paramètre $p0$. Sinon, le moniteur est libéré ($R_{MONITOR-EXIT}$) et le thread entre dans une boucle (lignes 17 et 18). La méthode *receive* (6.7.4) compare si le nom de l'action reçu (dans $p0$) à la valeur qui se trouve dans le canal c. En cas d'égalité, elle synchronise en remettant la valeur à "undefined" (R_{APUT}). Dans le cas contraire, le moniteur est libéré et le thread tente du nouveau en bouclant (lignes 16 et 17). Cette méthode retourne le nom de l'action (chaîne de caractère) qu'elle a reçu.

En somme, la synchronisation sur l'objet canal partagé se fait par le moyen des instructions *monitor-enter* et *monitor-exit*. Ces deux instructions assurent l'accès mutuellement exclusif aux objets partagés par différents threads. En d'autres termes, ceci garantit qu'un seul thread (soit le programme, soit la politique) acquiert le moniteur associé au canal partagé c et peut écrire et lire à partir de c. Les deux méthodes se synchronisent également via les valeurs dans c. En effet, le thread exécutant *send* ne peut écrire que si le canal contient une valeur "undefined" et le thread exécutant *receive* ne peut synchroniser (c.-à-d. recevoir) que si ce canal contient le nom de l'action à contrôler.

L'exécution de deux méthodes *send* et *receive* exigent un ordre bien déterminé pour qu'il y'ai une synchronisation. Plus précisément, la méthode *send* devrait s'exécuter avant la méthode *receive* afin d'écrire le nom de l'action début sur le canal. Ceci est possible puisque chaque case du canal est initialisée à "undefined" lors de la transformation de la politique de sécurité (voir la définition 6.5.4). Ensuite, la méthode *receive* ne peut synchroniser qu'en recevant ce nom et ensuite copiant la valeur "undefined" à sa place. Par conséquent, la communication entre ces deux méthodes se fait via l'écriture bloquante sur un canal partagé. Dans le cas où la méthode *receive* s'exécute en premier, le thread exécutant sera bloqué et le moniteur sera libéré, puisque il n'y a pas eu une réception du nom de l'action à contrôler sur le canal. Ainsi, ces deux actions de contrôle sont complémentaires.

Définition 6.7.3. *(Code de la méthode send)*

send est l'abréviation de séquence d'instructions $\mathbb{K}$-Smali suivantes :

```
1  /* send est une méthode de type void qui appartient à la classe Lp/PhiChannel;. Elle prend en paramètres une chaîne de
   caractères et un objet tableau de chaîne de caractères c. Elle réserve le moniteur pour l'objet c. Ensuite, elle parcourt
   le tableau, pour chaque case si la valeur égale à "undefined" elle la remplace par la chaîne de caractères passée en param
   ètre dans le registre p0. Sinon elle libère le moniteur et essaie du nouveau en bouclant */
2  .class Lp/PhiChannel;
3  .field final c:[Ljava/lang/String;
4  .method public static send(Ljava/lang/String; Lp/PhiChannel;)V
5    .locals 5                          // allouer 5 registres locaux
6    iget v0,p1, Lp/PhiChannel;->c   // placer l'objet tableau de canaux passé en paramètre p1 en v0
7    monitor-enter v0                // acquérir le moniteur pour l'objet c
8    array-length v1, v0   // retourner la taille du tableau de canaux c passé en paramètre dans v1
9    const v2, 0      // v2<-0, // initialiser le compteur (l'index) à zéro
10   :Wait
11   aget v3,v0, v2      // placer le contenu de la case indexée dans v2 vers v3
12   const-string v4, "undefined"        // v4 <- "undefined"
13   if-eq v3,v4, :label1  // tester si la valeur dans v3 égale à "undefined", si oui sauter vers :label1
14   monitor-exit v0     // sinon libérer le moniteur
15   goto :Wait        // boucler
16   :label1
17   aput p0, v0, v2   // placer le contenu du permier paramètre p0 à l'index dans v2 du tableau
18   const v4,1          // v4<-1
19   add v2 v2 v4            // incérementer le compteur (l'index)
20   if-lt v2, v1, :Wait     // si v2<v1 brancher vers :Wait
21   monitor-exit v0        // sinon libérer le moniteur
22   retrun-void
23 .end method
```

Définition 6.7.4. *(Code de la méthode receive)*

receive est l'abréviation de séquence d'instructions $\mathbb{K}$-Smali suivantes :

```
1  /* receive est une méthode de la classe Lp/PhiChannel;. Elle prend en paramètres une chaîne de caractères et un objet
   tableau de chaîne de caractère c et retourne une chaîne de caractère. Elle réserve le moniteur pour l'objet c. Ensuite,
   elle parcourt le tableau, pour chaque case si la valeur égale à la valeur passée en paramètre dans p0, elle la remplace
   par la valeur "undefined". Sinon elle libère le moniteur et essaie du nouveau en bouclant. */
2  .class Lp/PhiChannel;
3  .field final c:[Ljava/lang/String;
4  .method public static receive(Ljava/lang/String; Lp/Renforcement;) Ljava/lang/String;
5   .locals 5                          // allouer 5 registres locaux
6   iget v0,p1, Lp/PhiChannel;->c   // placer l'objet tableau de canaux passé en paramètre p1 en v0
7   monitor-enter v0                // acquérir le moniteur pour l'objet c
8   array-length v1, v0   // retourner la taille du tableau de canaux c passé en paramètre dans v1
9   const v2, 0      // v2<-0, // initialiser le compteur (l'index) à zéro
10  :Wait
11  aget v3,v0, v2      // placer le contenu de la case indexée dans v2 vers v3
12  if-eq v3,p0, lab1  // tester si la case courante contient la chaîne de caractère passée en paramètre, si oui sauter à lab1
13  monitor-exit v0  // sinon libérer le moniteur associé à l'objet dans v0
14  goto :Wait        // boucler
15  :lab1
16  const-string v4, "undefined"  // v4 <-"undefined"
17  aput v4, v0, v2        // mettre "undefined" dans la case courante du tableau v0
18  const v4,1          // v4<-1
```

```
19  add v2 v2 v4          // incérementer le compteur (l'index)
20  if-lt v2, v1, :Wait     //  si v2<v1 brancher vers :Wait
21  monitor-exit v0        // sinon libérer le moniteur
22  return v3             // retourner la chaîne de caracère lue dans v3
23 .end method      .
```

6.7.1 Exemple du code ajouté pour le renforcement

Dans cette sous-section, nous présentons un exemple du code $\mathbb{K}$-*Smali* ajouté pour le renforcement. En effet, la définition 6.7.1 de l'opérateur de renforcement stipule l'exécution de la politique et le programme en parallèle afin qu'ils synchronisent. Comme expliqué à la section précédente, le programme et la politique sont des threads qui s'exécutent en parallèle. Listing 6.1 représente un exemple du code qui illustre le concept de renforcement.

```
1  .class public Lp/c1;
2  .method public static start ()V
3    .locals 2
4    const-string v0, "a-s"
5    invoke-static {v0}, Lp/PhiChannel;->receive(Ljava/lang/String; Lp/PhiChannel;)Ljava/lang/String;
6    action
7    const-string v1, "a-e"
8    invoke-static {v1}, Lp/PhiChannel;->send(Ljava/lang/String; Lp/PhiChannel;)V
9    return-void
10 .end method
11
12 .class public Lp/Phi;
13 .method public static start ()V
14   .locals 2
15   const-string v0, "a-s"
16   invoke-static {v0}, Lp/PhiChannel;->send(Ljava/lang/String; Lp/PhiChannel;)V
17   const-string v1, "a-e"
18   invoke-static {v1}, Lp/PhiChannel;->receive(Ljava/lang/String; Lp/PhiChannel;)Ljava/lang/String;
19   return-void
20 .end method
21
22 .class public Lp/Renforcement;
23 .super Lp/c1;
24 .field public c: [Ljava/lang/String;
25 .method public static m1()V
26   .locals 3
27   new-instance v0, Lp/PhiChannel;
28   const-string v1, "undefined"
29   iput v1, v0, Lp/PhiChannel;->c
30   new-instance v0, Lp/c1;
31   invoke-virtual {v0}, Ljava/lang/Thread;->start ()V
32   new-instance v2, Lp/Phi;
33   invoke-virtual {v2},Ljava/lang/Thread;->start ()V
34   return-void
35 .end method
36 .manifest Lp/Renforcement;->m1()V
```

Listing 6.1 – Exemple de renforcement de politique de sécurité

La classe $Lp/c1$; représente un exemple du programme à contrôler, qui est transformé. La transformation du programme se manifeste par l'appel des méthodes $receive$ et $send$ à partir de la classe $Lp/PhiChannel$; avant et après l'action à contrôler. C'est l'action $action$ à la ligne 6. La classe Lp/Phi; représente la politique de sécurité transformée.

La classe $Lp/Renforcement$; représente le traitement équivalent à l'opérateur $\sqcap$. La variable représentant le canal est tout d'abord initialisée à "undefined" (ligne 29). Cette étape est cruciale afin de garantir que la méthode $send$ s'exécute toujours en premier. Ensuite, pour assurer l'exécution parallèle de la politique et le programme, un objet thread est instancié de la classe $Lp/c1$; ($R_{NEW-INSTANCE}$), contenant le programme à contrôler et ensuite lancé ($R_{CREATE-THREAD}$) (lignes 30 et 31). De même, un objet thread est instancié de la classe Lp/Phi; ($R_{NEW-INSTANCE}$) contenant la politique de sécurité et ensuite lancé ($R_{CREATE-THREAD}$) (lignes 32 et 33). Ces deux threads seront exécutés en parallèle selon les règles qui permettent leurs gestion et synchronisation.

6.7.2 Discussion

L'introduction d'une troisième entrée A, contenant un sous ensemble d'actions à contrôler, a contribué à l'optimisation du renforcement. En effet, avec la réduction de l'ensemble de toutes les actions du programme $\mathcal{A}$ en un sous-ensemble d'actions A, le nombre de tests insérés pourrait se réduire substantiellement. Le sens intuitif de "$mod\ A$" est que parmi toutes les actions du programme, seulement les actions dans A sont à contrôler, ce qui évitera une certaine lourdeur qui peut être causée si nous considérons toutes les actions possibles du programme. D'autre part, ceci sera d'une grande utilité si par exemple l'utilisateur choisit ces actions, de cette façon, nous lui donnons plus de contrôle sur l'application qu'il installe. Une autre façon de faire est de considérer dans A seulement les actions à risque [13], qui sont les plus sensibles comme celles qu'on a présentées dans le premier chapitre (voir Section 1.5).

Ainsi, pour renforcer une politique de sécurité dans une application Android, nous parcourons le code de l'application et nous ajoutons les actions de synchronisation avant et après chaque action à contrôler dans A. Ainsi, chaque action $a \in A$ est précédée par $receive(\ulcorner a, c)$ et suivie par $send(a\urcorner, c)$. Par conséquent, le renforcement est linéaire par rapport au nombre de tests insérés, autrement dit le coût de renforcement dépend de la taille de l'ensemble des actions à contrôler dans le programme.

Si au lieu de considérer les actions du programme , nous considérons seulement les actions identifiées par la politique, nous risquons de violer la propriété de correction. Autrement dit, nous pouvons finir par une version renforcée qui ne satisfait pas la politique. Par exemple, considérons un programme $P = c\ a$ et une politique de sécurité $\varphi = a$. La politique φ exige que le programme commence par exécuter l'action a. Si nous considérons seulement les actions spécifiées par φ (c.-à-d. $A = \{a\}$), alors le renforcement s'effectue comme suit : $P \sqcap_A \varphi \overset{(6.7.1)}{=} c\ receive(\ulcorner a, c)\ a\ send(a\urcorner, c) \parallel send(\ulcorner a, c)\ receive(a\urcorner, c)$. Puisque l'action $c \notin A$, elle n'a subie aucune transformation et par la suite elle est autorisée à s'exécuter par le programme. Toutefois,

l'exécution de cette action viole la politique qui exige au programme de commencer par l'action a.

Supposons maintenant que $A = \{a, c\}$. Le renforcement de φ dans P s'effectue comme suit :
$P \sqcap_A \varphi \overset{(6.7.1)}{=} receive(\ulcorner c, c)\ c\ send(c\urcorner, c)\ receive(\ulcorner a, c)\ a\ send(a\urcorner, c) \parallel send(\ulcorner a, c)\ receive(a\urcorner, c)$
L'action $receive(\ulcorner c, c)$ demeure bloquée puisque elle n'a pas été débloquée par un $send(\ulcorner c, c)$ à partir de la politique et par la suite le programme ne peut pas exécuter l'action c, ce qui satisfait parfaitement les exigences de la politique de sécurité.

6.8 Exemples

Afin d'illustrer le fonctionnement de notre approche, nous présentons dans cette section deux exemples intuitifs liés à des vulnérabilités dans Android à savoir les chevaux de Troie SMS et les logiciels espion.

Cheval de Troie SMS : Soit le programme P avec une exécution séquentielle des actions : *écrire* un SMS désignée par p (*PutExtra*), *envoyer* un SMS désignée par s (*SendTextMessage*) et *abandonner* la notification des SMS entrants désignée par a (*abortBroadcast*). Ce séquencement d'actions correspond à un scénario d'attaque, où l'application malveillante envoie un SMS à l'insu de l'utilisateur, puis surveille les SMS entrants et bloque ceux qui proviennent du même numéro afin que l'utilisateur ne soit pas averti.

Pour ne pas encombrer la présentation de cet exemple, nous adoptons la notation suivante de P :

$$P = p\ s\ a$$

Considérons une politique de sécurité φ qui autorise seulement l'exécution de deux actions écrire SMS et envoyer SMS tel que :

$$\varphi = p.s$$

L'ensemble d'actions à contrôler n'est pas introduit donc par défaut la réécriture considère toutes les actions possibles du programme ($A = \mathcal{A}$).

Transformer φ :

$F_{c,\emptyset}(p.s)$
$\overset{(6.5)}{=} send(\ulcorner p, c)\ receive(p\urcorner, c)\ F_{c,\sigma}(\partial^1_p(p.s)) \qquad avec\ \delta(p.s) = p$
$= send(\ulcorner p, c)\ receive(p\urcorner, c)\ F_{c,\emptyset}(s)$
$\overset{(6.4)}{=} send(\ulcorner p, c)\ receive(p\urcorner, c)\ send(\ulcorner s, c)\ receive(s\urcorner, c)$

Réécrire P :

$S_{c,\emptyset}(P, A)$
$\overset{(7.2)}{=} S_{c,\emptyset}(p, A)\ S_{c,\emptyset}(s, A)\ S_{c,\emptyset}(a, A)$
$\overset{(7.1)}{=} (receive(\ulcorner p, c)\ p\ send(p\urcorner, c), \emptyset)\ (receive(\ulcorner s, c)\ s\ send(s\urcorner, c), \emptyset)\ (receive(\ulcorner a, c)\ a\ send(a\urcorner, c), \emptyset)\ avec\ p, s, a \in A\ et\ p, s, a \neq goto\ et\ p, s\ et\ a \neq if\text{-}eq$

$L_c(receive(\ulcorner p, c)\ p\ send(p^{\urcorner}, c)\ \ receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c), \emptyset)$
$\overset{(8.1)}{=} L_c(receive(\ulcorner p, c)\ p\ send(p^{\urcorner}, c), \emptyset)\ L_c(receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c), \emptyset)\ L_c(receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c), \emptyset)$
$\overset{(8.2)}{=} receive(\ulcorner p, c)\ p\ send(p^{\urcorner}, c)\ \ receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c)\ \ avec\ p, s\ et\ a \neq:$
$Label$

Renforcer φ dans P :

$P \sqcap \varphi \overset{6.7.1}{=}$

$receive(\ulcorner p, c)\ p\ send(p^{\urcorner}, c)\ \ receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel send(\ulcorner p, c)\ receive(p^{\urcorner}, c)\ send(\ulcorner s, c)\ receive(s^{\urcorner}, c)$
$\xrightarrow{send(\ulcorner p,c)}$

$receive(\ulcorner p, c)\ p\ send(p^{\urcorner}, c)\ \ receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel\ receive(p^{\urcorner}, c)\ send(\ulcorner s, c)\ receive(s^{\urcorner}, c)$
$\xrightarrow{receive(\ulcorner p,c)}$

$p\ send(p^{\urcorner}, c)\ \ receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel\ receive(p^{\urcorner}, c)\ send(\ulcorner s, c)\ receive(s^{\urcorner}, c)$
$\xrightarrow{p}$

$send(p^{\urcorner}, c)\ \ receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel\ receive(p^{\urcorner}, c)\ send(\ulcorner s, c)\ receive(s^{\urcorner}, c)$
$\xrightarrow{send(p^{\urcorner},c)}$

$receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel receive(p^{\urcorner}, c)\ send(\ulcorner s, c)\ receive(s^{\urcorner}, c)$
$\xrightarrow{receive(p^{\urcorner},c)}$

$receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel send(\ulcorner s, c)\ receive(s^{\urcorner}, c)$
$\xrightarrow{send(\ulcorner s,c)}$

$receive(\ulcorner s, c)\ s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel receive(s^{\urcorner}, c)$
$\xrightarrow{receive(\ulcorner s,c)}$

$s\ send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel receive(s^{\urcorner}, c)$
$\xrightarrow{s}$

$send(s^{\urcorner}, c)\ receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel receive(s^{\urcorner}, c)$
$\xrightarrow{send(s^{\urcorner},c)}$

$receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c) \parallel receive(s^{\urcorner}, c)$
$\xrightarrow{receive(s^{\urcorner},c)}$

$receive(\ulcorner a, c)\ a\ send(a^{\urcorner}, c)$

À la fin, le programme n'a pas exécuté l'action *abortBroadcast*, notée par a. En effet, en exécutant la méthode $receive(\ulcorner a, c)$, le thread demeure bloqué vu qu'il n'a pas reçu une valeur contenant le nom de l'action début $\ulcorner a$ à partir de la politique étant donné qu'elle n'autorise pas l'exécution de cette action. Il libère le moniteur et entre dans une boucle infinie jusqu'à la réception de cette valeur. Cela signifie que la politique a empêché l'application d'interrompre les notifications SMS reçues sans l'approbation de l'utilisateur.

Espionnage : Soit le programme P avec l'exécution séquentielle des actions : *ouvrir caméra* désignée par o et qui correspond à la méthode *Camera.open*() dans Android, *prendre photo* désignée par t et correspond à la méthode *Camera.takepicture*(). Cet enchaînement d'actions peut être exploité par une application d'espionnage pour prendre des photos en arrière-plan, autrement dit sans que l'utilisa-

teur s'en rende compte. Toutefois, si l'action permettant de prendre photo est précédée par une action afficher aperçu du caméra sur l'écran, l'utilisateur sera toujours au courant que la caméra est allumée et essaie de prendre des photos.

$$P = o\ t$$

Soit une politique de sécurité qui exige l'exécution de l'action *afficher* aperçu caméra désignée par d (*Camera.setpreviewDisplay*()) avant l'action prendre photo t.

$$\varphi = o.d.t$$

L'ensemble d'actions à contrôler n'est pas introduit donc par défaut la réécriture considère toutes les actions possibles du programme $\mathcal{A}$.

Pour renforcer φ dans P, nous suivons les étapes de renforcement comme suit :

Transformer φ :

$F_{c,\emptyset}(o.d.t)$
$\overset{(6.5)}{=} send(\ulcorner o,c)\ receive(o\urcorner,c)\ F_{c,\emptyset}(\partial_o(o.d.t)) \qquad avec\ \delta(o.d.t) = o$
$= send(\ulcorner o,c)\ receive(o\urcorner,c)\ F_{c,\emptyset}(d.t)$
$\overset{(6.4)}{=} send(\ulcorner o,c)\ receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ F_{c,\emptyset}(\partial_d(d.t))\ \ avec\ \delta(d.t) = d$
$\overset{(6.4)}{=} send(\ulcorner o,c)\ receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ F_{c,\emptyset}(\partial_d(t))$
$\overset{(6.4)}{=} send(\ulcorner o,c)\ receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ receive(t\urcorner,c)$

Transformer P :

$S_{c,\emptyset}(P,A)$
$\overset{(7.2)}{=} S_{c,\emptyset}(o,A)\ S_{c,\emptyset}(t,A)$
$\overset{(7.1)}{=} (receive(\ulcorner o,c)\ o\ send(o\urcorner,c)\ \ receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \quad o,t \in\ A\ et\ o,t \neq goto\ et\ o,t \neq if\text{-}eq$

$L_c(receive(\ulcorner o,c)\ o\ send(o\urcorner,c)\ \ receive(\ulcorner t,c)\ t\ send(t\urcorner,c),\emptyset)$
$\overset{(8.2)}{=} L_c(receive(\ulcorner o,c)\ o\ send(o\urcorner,c),\emptyset)\ L_c(receive(\ulcorner t,c)\ t\ send(t\urcorner,c),\emptyset)$
$\overset{(8.1)}{=} receive(\ulcorner o,c)\ o\ send(o\urcorner,c)\ \ receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \quad avec\ o\ et\ t \neq: Lab$

Renforcer φ dans P :

$P \sqcap \varphi \overset{6.7.1}{=}$

$receive(\ulcorner o,c)\ o\ send(o\urcorner,c)\ \ receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \parallel send(\ulcorner o,c)\ receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ receive(t\urcorner,c)$
$\xrightarrow{send(\ulcorner o,c)}$

$receive(\ulcorner o,c)\ o\ send(o\urcorner,c)\ \ receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \parallel receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ send(t\urcorner,c)$
$\xrightarrow{receive(\ulcorner o,c)}$

$o\ send(o\urcorner,c)\ \ receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \parallel receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ receive(t\urcorner,c)$
$\xrightarrow{o}$

$send(o\urcorner,c)\ \ receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \parallel receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ receive(t\urcorner,c)$
$\xrightarrow{send(o\urcorner,c)}$

$receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \parallel receive(o\urcorner,c)\ send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ receive(t\urcorner,c)$
$\xrightarrow{receive(o\urcorner,c)}$

$receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \parallel send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ receive(t\urcorner,c)$

$\xrightarrow{send(\ulcorner d,c)}$

$receive(\ulcorner t,c)\ t\ send(t\urcorner,c) \parallel send(\ulcorner d,c)\ receive(d\urcorner,c)\ send(\ulcorner t,c)\ receive(t\urcorner,c)$

À la fin, le programme n'a pas synchronisé avec la politique pour l'exécution de l'action d avec un $receive(\ulcorner d,c)$. Ensuite, quand il tente d'exécuter l'action t pour prendre une photo, il se bloque puisque il n'a pas reçu un signal de la part de la politique pour exécuter cette action avec un $send(\ulcorner t,c)$. Ceci est dû au fait que la politique exige l'exécution de l'action d avant. Cela signifie que la politique a empêché la possibilité de prendre une photo en arrière-plan, sans aviser l'utilisateur.

6.9 Formalisation de propriétés de correction et de complétude

Dans cette section, nous prouvons que l'approche de renforcement proposée est à la fois correcte et complète.

6.9.1 Formalisation de la propriété de correction

Pour prouver la correction de l'approche proposée, nous avons besoin d'introduire un ensemble de notations et de définitions. Nous avons besoin en premier temps d'alléger la définition de $\tau \vdash \varphi$. Pour ce faire, nous introduisons la définition d'une trace qui respecte le préfixe d'une formule. Rappelons que la définition de $\tau \vdash \varphi$ signifie que τ est la trace acceptée par φ et que $[\![\varphi]\!] = \{\tau\}$. Par la suite, nous adoptons la définition de $\tau \Vdash \varphi$ pour exprimer une trace τ qui respecte un certain préfixe de $[\![\varphi]\!]$ ($Prefx([\![\varphi]\!])$). Ainsi, nous prolongeons la notion de préfixe définie au début du chapitre à un ensemble de traces comme suit :

Définition 6.9.1. *(Préfixe d'un ensemble de traces Prefx())*
Soit T un ensemble de traces dans $\mathcal{T}$, nous étendons la définition de la fonction préfixe à un ensemble de traces, tel que $Prefx : \mathcal{T} \longrightarrow \mathcal{T}$ de la manière suivante :

$$Prefx(T) = \bigcup_{t \in T} Prefx(t)$$

Par exemple, si $[\![\varphi]\!] = \{a.b.c\}$ alors son préfixe est $Prefx([\![\varphi]\!]) = \{a, a.b, a.b.c\}$.

La raison derrière l'extension de la notion de satisfaction aux programmes est de pouvoir prendre le maximum possible de traces issues du programme, qui satisfait φ, à partir d'une trace qui ne respecte pas φ. Étant donné une trace $\tau = a\ b\ c$ et une formule $\varphi = \{a\ b\ d\}$. Il est clair que τ ne respecte φ. Notre but est de laisser le programme s'exécuter et de l'arrêter quand la politique de sécurité est sur le point d'être violée. Dans ce cas, nous avons $Prefx([\![\varphi]\!]) = \{a, a\ b\}$. Par la suite, nous prenons le plus long préfixe de τ qui respecte φ qui est $a\ b$. Le programme P est désormais autorisé d'exécuter $a\ b$ et puis s'arrêter. De cette façon, au lieu de bloquer toutes les traces qui violent φ lors de renforcement, nous prenons leurs plus longs préfixes qui coïncident avec φ. La définition de $\Vdash$ est donnée dans 6.9.2.

Définition 6.9.2. *(Notion de satisfaction d'un préfixe d'une formule $\Vdash$)*
Soient φ une formule de la logique $LTL_{\Downarrow}$ et τ une trace dans $\mathcal{T}$, nous introduisons la relation $\Vdash \subseteq \mathcal{T} \times LTL_{\Downarrow}$ comme suit :

$$\tau \Vdash \varphi \ \ si\ \tau \in Prefx([\![\varphi]\!])$$

La définition d'une trace τ qui respecte un programme $\lceil P \rceil_c$ $mod\ A$ requiert une fonction d'abstraction qui permet d'éliminer de τ toutes actions qui ne sont pas dans A. Cette fonction est définie comme suit :

Définition 6.9.3. *(Élimination d'actions d'une trace $\lceil - \rceil_A$)*
Soient τ une trace dans $\mathcal{T}$ et A un ensemble d'actions dans $2^{\mathcal{A}}$, nous définissons la fonction $\lceil - \rceil_A : \mathcal{T} \times 2^{\mathcal{A}} \longrightarrow \mathcal{T}$ inductivement comme suit :

- $\lceil \epsilon \rceil_A = \epsilon$

- $\lceil a \rceil_A = \begin{cases} a & si\ a \in A \\ \epsilon & sinon \end{cases}$

- $\lceil a.\tau \rceil_A = \begin{cases} a.\lceil \tau \rceil_A & si\ a \in A \\ \lceil \tau \rceil_A & sinon \end{cases}$

- $\lceil \tau^* \rceil_A = \lceil \tau \rceil_A^*$

- $\lceil \tau^\omega \rceil_A = \lceil \tau \rceil_A^\omega$

Définition 6.9.4. *(Notion de satisfaction par rapport à A $\Vdash_A$)*
Soient $\tau \in \mathcal{T}$, $A \in 2^{\mathcal{A}}$ et $\varphi \in LTL_{\Downarrow}$, nous définissons la relation $\Vdash_A \subseteq \mathcal{T} \times \varphi$ comme suit :

$$\tau \Vdash_A \varphi \quad si \quad \lceil \tau \rceil_A \Vdash \varphi$$

Nous prolongeons la définition de $\Vdash_A$ à un programme comme suit :

$$P \Vdash_A \varphi \ \ si \ \ \lceil P \rceil_c \ mod\ A \Vdash \varphi$$

Nous avons vu que la définition de la correction concerne toutes les traces de la version renforcée, qui devraient satisfaire la politique de sécurité. Étant donné que ces traces incluent les actions de synchronisation, nous introduisons par la suite une relation de transition $\rightarrowtail$ qui abstrait les transitions de ces actions du programme renforcé. Afin de formaliser cette idée, nous introduisons l'ensemble des actions de synchronisation ajoutées à un programme durant le renforcement, dénoté par A_{sync}, tel que pour $A_{sync} \in \mathcal{A}$:

$$A_{sync} = \bigcup_{a \in \mathcal{A}} \{send(\ulcorner a, c)\ receive(a \urcorner, c), send(a \urcorner, c), receive(\ulcorner a, c)\}$$

Définition 6.9.5. *(Relation d'abstraction $\rightarrowtail$)*
Soient P et $P'' \in \mathcal{P}$, nous définissons la relation de transition $\rightarrowtail \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ de la manière suivante :

$$P \overset{a}{\rightarrowtail} P'' \ \ s'il \ \ existe \ \ P' \in \mathcal{P} \ \ tel\ que \ \ P \xrightarrow{a_s} P' \ \ et \ \ P' \xrightarrow{a} P''$$

avec $a_s \in A_{sync}$.

Nous introduisons les relations raccourcie $\rightsquigarrow$, $\downarrow$, $\downarrowtail$ et $\downsquigarrow$, permettant de faire évoluer un processus tout en exécutant une séquence d'actions, comme suit.

Définition 6.9.6. *(Relation raccourcie $\rightsquigarrow$)*
Soient P, $P' \in \mathcal{P}$, $\tau \in \mathcal{T}$ tel que $\tau = a_1, \ldots, a_{n-1}$, nous définissons $\rightsquigarrow \subseteq \mathcal{P} \times \mathcal{T} \times \mathcal{P}$ comme suit :

$$P \overset{\tau}{\rightsquigarrow} P' \text{ s'il existe } P_1, P_2, \ldots, P_{n-1} \in \mathcal{P} \text{ tel que } P \xrightarrow{a_1} P_1, P_1 \xrightarrow{a_2} P_2, \ldots, P_{n-1} \xrightarrow{a_{n-1}} P'$$

Définition 6.9.7. *(Relations $\downarrow$, $\rotatebox{-90}{$\rightsquigarrow$}$ et $\rotatebox{-90}{$\rightsquigarrow$}$)*
Soient $P \in \mathcal{P}$, $\tau \in \mathcal{T}$ tel que $\tau = a_1, \ldots, a_n$, avec $a_i \in \mathcal{A}$. Nous définissons les relations :

- $\downarrow \subseteq \mathcal{P} \times \mathcal{T}$ *comme suit :*
  - $P \downarrow \tau$ *s'il* $\exists P_1, \ldots, P_n \in \mathcal{P}$ *tels que* $P \xrightarrow{a_1} P_1 \ldots \xrightarrow{a_n} P_n$
- $\rotatebox{-90}{$\rightsquigarrow$} \subseteq \mathcal{P} \times \mathcal{T}$ *comme suit :*
  - $P \rotatebox{-90}{$\rightsquigarrow$} \tau$ *s'il existe* $P_1, \ldots, P_n \in \mathcal{P}$ *tels que* $P \overset{a_1}{\rightsquigarrow} P_1 \ldots \overset{a_n}{\rightsquigarrow} P_n$
- $\rotatebox{-90}{$\rightsquigarrow$} \subseteq \mathcal{P} \times \mathcal{T}$ *comme suit :*
  - $P \rotatebox{-90}{$\rightsquigarrow$} \tau$ *s'il existe* $P_1, \ldots, P_n \in \mathcal{P}$ *tels que* $P \overset{a_1}{\rightsquigarrow} P_1 \ldots \overset{a_n}{\rightsquigarrow} P_n$

Les définitions de relations d'abstraction ci-dessus nous permettent de définir une nouvelle relation dénotée par $\Vdash\!\sim$ pour un programme qui respecte une formule après l'abstraction de certaines actions.

Définition 6.9.8. *($P \Vdash \varphi$ et $P \Vdash\!\sim \varphi$)*
Soient $\tau \in \mathcal{T}$, $P \in \mathcal{P}$ et $\varphi \in LTL_{\Downarrow}$, nous définissons les relations $\Vdash : \mathcal{P} \times LTL_{\Downarrow}$ et $\Vdash\!\sim : \mathcal{P} \times LTL_{\Downarrow}$ comme suit :

- *$P \Vdash \varphi$ si pour tout $\tau \in \mathcal{T}$ tel que $P \downarrow \tau$, nous avons $\tau \Vdash \varphi$;*
- *$P \Vdash\!\sim_A \varphi$ si pour tout $\tau \in \mathcal{T}$ tel que $P \rotatebox{-90}{$\rightsquigarrow$} \tau$, nous avons $\tau \Vdash_A \varphi$*

Intuitivement, après l'abstraction des actions de synchronisation dans A_{sync} de la version renforcée, le renforcement est dit correct, si toute trace de $P \sqcap_A \varphi$ respecte φ. Le théorème de correction s'annonce comme suit.

Théorème 6.9.1. *(Correction)*
$\forall P \in \mathcal{P}$, $\forall \varphi \in LTL_{\Downarrow}$, et $A \subseteq \mathcal{A}$, nous avons :

$$P \sqcap_A \varphi \Vdash\!\sim_A \varphi$$

Preuve :
La preuve est donnée dans Section 6.10, page 160. ■

6.9.2 Formalisation de la propriété de complétude

De la même façon que la formalisation de la propriété de correction, l'objectif ici de formaliser la propriété de complétude en faisant toujours abstraction des actions de synchronisation dans A_{sync}, ajoutées lors de renforcement. En d'autres termes, nous souhaitons prouver que la version renforcée est équivalente mais plus restrictive que l'originale. L'objectif donc est de démontrer que seules les traces dans P qui respectent la politique de sécurité φ sont conservées dans $P \sqcap \varphi$. Pour y parvenir, nous avons besoin de définir une relation d'équivalence entre programmes par rapport à une formule. Intuitivement, deux programmes P et P' sont dites équivalents par rapport à une formule φ, si toutes les traces de P qui respectent φ sont des traces de P' et vice versa.

Définition 6.9.9. *(Équivalence de programmes par rapport à une formule $\approx_\varphi$)*
Étant donné deux programmes P et Q dans $\mathcal{P}$, P et Q sont équivalents par rapport à une formule φ et notés $P \approx_\varphi Q$ pour $\approx_\varphi \subseteq \mathcal{P} \times \mathcal{P}$ si :

(i) $P \overset{a}{\rightarrowtail} P'$ *et* $a \in \delta(\varphi)$*, il existe* $Q' \in \mathcal{P}$ *tel que* $Q \overset{a}{\rightarrowtail} Q'$ *et* $P' \approx_{\partial_a(\varphi)} Q'$ *et*

(ii) $Q \overset{a}{\rightarrowtail} Q'$ *et* $a \in \delta(\varphi)$*, il existe* $P' \in \mathcal{P}$ *tel que* $P \overset{a}{\rightarrowtail} P'$ *et* $Q' \approx_{\partial_a(\varphi)} P'$

Il n'est pas difficile de prouver que la relation $\approx_\varphi$ est une relation d'équivalence réflexive, transitive et symétrique.

Intuitivement, après l'abstraction des actions de synchronisation de la version renforcée, le renforcement est dit complet si toute trace de P qui respecte φ est une trace possible de $P \sqcap_A \varphi$ et vice versa. Le théorème de complétude s'annonce comme suit.

Théorème 6.9.2. *(Complétude)*
$\forall P \in \mathcal{P}$, $\forall \varphi \in LTL_{\Downarrow}$, et $A \subseteq \mathcal{A}$, nous avons :

$$P \approx_\varphi P \sqcap_A \varphi$$

Preuve :
La preuve est donnée dans Section 6.10, page 160. ■

6.10 Preuve de correction et complétude de l'approche proposée

Nous présentons dans cette section les preuves de deux théorèmes 6.10.1 et 6.10.2.

6.10.1 Propriétés liées à la logique $LTL_{\Downarrow}$

Définition 6.10.1. *(Équivalence de deux formules par rapport à une trace $\sim$)*
Soient φ_1 et φ_2 deux formules dans $LTL_{\Downarrow}$, φ_1 est dite équivalente à φ_2 et notée $\varphi_1 \sim \varphi_2$ si :

$$[\![\varphi_1]\!] = [\![\varphi_2]\!]$$

Proposition 6.10.1. *(Lien entre $o(\varphi)$, $\delta(\varphi)$ et $\partial(\varphi)$)*
Soient $\varphi \in LTL_{\Downarrow}$, $\tau \in \mathcal{T}$, $A \subseteq \mathcal{A}$ et $a \in A$, nous avons :

$$a.\tau \Vdash_A \varphi \;\; ssi \;\; a \in \delta(\varphi) \; et \; \tau \Vdash_A \varphi$$

Démonstration. Prouvons les deux sens :

- $a.\tau \Vdash_A \varphi$

$\Rightarrow$ {Définition 6.9.4 de $\Vdash_A$}
$\lceil a.\tau \rceil_A \Vdash \varphi$

$\Rightarrow$ {Définition 6.9.3, avec $a \in A$}
$a.\lceil \tau \rceil_A \Vdash \varphi$

$\Rightarrow$ {La preuve découle directement de la définition 6.5.1 de δ et ∂}
$a \in \delta(\varphi)$ et $\lceil \tau \rceil_A \Vdash \varphi$

$\Rightarrow$ {Définition 6.9.4 de $\Vdash_A$}

$\tau \Vdash_A \varphi$

- $a \in \delta(\varphi)$ et $\tau \Vdash_A \varphi$

$\Rightarrow$ {Définitions 6.5.1 de δ et 6.9.3 de $\lceil - \rceil_A$ avec $a \in A$}

$a.\tau \Vdash_A \varphi$

□

Proposition 6.10.2.
Soient φ une formule dans LTL$_{\Downarrow}$ alors :

$$1.\varphi \sim \varphi$$

Démonstration.
La preuve découle directement de la définition de $[\![-]\!]$ où $[\![1.\varphi]\!] \overset{(2.8)}{=} \{\tau_1.\tau_2 | \tau_1 \in [\![1]\!] \ et \ \tau_2 \in [\![\varphi]\!]\}$. Puisque $[\![1]\!] \overset{(2.3)}{=} \{\epsilon\}$ (la trace vide), alors $[\![1.\varphi]\!] = [\![\varphi]\!]$. D'après la définition 6.10.1 de $\sim$, nous obtenons $1.\varphi \sim \varphi$.

□

Proposition 6.10.3. *(Lien entre $o(\varphi)$, $\delta(\varphi)$ et $\partial(\varphi)$)*
Soient φ une formule dans LTL$_{\Downarrow}$ et $a \in \mathcal{A}$, nous avons :

$$\varphi \sim o(\varphi) \vee (\bigvee_{a \in \delta(\varphi)} a.\partial_a(\varphi))$$

Démonstration.
La preuve se fait par induction structurelle sur φ.

- **Cas 1** ($\varphi = ff$) :

  $\Rightarrow$ {Définition 6.5.1, $o(ff) = ff$, $\delta(ff) = \emptyset$, $\partial_a(ff) = ff$}

  $ff \vee ff \sim ff$

- **Cas 2** ($\varphi = 1$) :

  $\Rightarrow$ {Définition 6.5.1, $o(tt) = 1$, $\delta(tt) = \mathcal{A}$, $\partial_a(tt) = tt$}

  $1 \vee (\bigvee_{a \in \mathcal{A}} a.tt)$

  $\Rightarrow$ {La sémantique (2.1) de tt (toutes les traces respectent tt, y inclus la trace vide), avec la définition 6.10.1 de $\sim$ nous obtenons : }

  $1 \vee (\bigvee_{a \in \mathcal{A}} a.tt) \sim tt$

- **Cas 3** ($\varphi = 1$) :

  $\Rightarrow$ {Définition 6.5.1, $o(1) = 1$ et $\delta(1) = \emptyset$}

  $o(1) \vee (\bigvee_{a \in \delta(1)} a.\partial_a(1)) = o(1)$

  $\Rightarrow$ {Définition 6.10.1 de $\sim$}

  $o(1) \sim 1$

- **Cas 4** ($\varphi = a$) :

  $\Rightarrow$ {Définition 6.5.1, $o(a) = ff$ et $\delta(a) = a$ et $\partial_a(a) = 1$}

  $ff \vee \bigvee_{a \in \delta(a)} a.1 = ff \vee (a.1)$

$\Rightarrow$ {Définition 6.10.1 de $\sim$}
$ff \vee a \sim a$

- **Cas 5** ($\varphi = \varphi_1 \vee \varphi_2$) :

  $\Rightarrow$ {Par induction}

  — $\varphi_1 \sim o(\varphi_1) \vee (\bigvee_{a \in \delta(\varphi_1)} a.\partial_a(\varphi_1))$

  — $\varphi_2 \sim o(\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_2)} a.\partial_a(\varphi_2))$

  — $\varphi_1 \vee \varphi_2 = o(\varphi_1) \vee (\bigvee_{a \in \delta(\varphi_1)} a.\partial_a(\varphi_1)) \vee o(\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_2)} a.\partial_a(\varphi_2))$

  $\Rightarrow$ {Définition 6.5.1 de $o(\varphi_1 \vee \varphi_2)$ et $\delta(\varphi_1 \vee \varphi_2)$, définition 6.10.1 de $\sim$}
  $o(\varphi_1) \vee o(\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_1) \vee \delta(\varphi_2)} a.\partial_a(\varphi_1) \vee \partial_a(\varphi_2)) \sim \varphi_1 \vee \varphi_2$

- **Cas 6** ($\varphi = \varphi_1.\varphi_2$) :

  $\Rightarrow$ {Par induction}
  $\varphi_1.\varphi_2 \sim o(\varphi_1) \vee (\bigvee_{a \in \delta(\varphi_1)} a.\partial_a(\varphi_1)).o(\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_2)} a.\partial_a(\varphi_2))$

  $\Rightarrow$ {Distributivité de . sur $\vee$}
  $\varphi_1.\varphi_2 \sim o(\varphi_1).o(\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_1)} a.\partial_a(\varphi_1)).o(\varphi_2)) \vee (\bigvee_{a \in \delta(\varphi_2)} o(\varphi_1).a.\partial_a(\varphi_2))$

  $\Rightarrow$ {Définition 6.5.1 de $o(\varphi_1.\varphi_2)$ $\delta(\varphi_1.\varphi_2)$ et $\partial_a(\varphi_1.\varphi_2)$}
  $o(\varphi_1.\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_1.\varphi_2)} a.\partial_a(\varphi_1.\varphi_2)) \sim \varphi_1.\varphi_2$

- **Cas 7** ($\varphi = \varphi_1^* \varphi_2$) :

  $\Rightarrow$ {Définition 6.4.1 de $[\![-]\!]$, cas (2.9)}
  $\varphi_1^* \varphi_2 \sim \varphi_1.\varphi_1^* \varphi_2 \vee \varphi_2$

  $\Rightarrow$ {Supposons que $o(\varphi_1) = ff$. Par induction, et d'après la définition 6.5.1 nous avons :$o(\varphi_1.\varphi_1^* \varphi_2 \vee \varphi_2) = o(\varphi_1) \vee o(\varphi_2)$, $\delta(\varphi_1.\varphi_1^* \varphi_2 \vee \varphi_2) = \delta(\varphi_1)$, $\partial_a(\varphi_1.\varphi_1^* \varphi_2 \vee \varphi_2) = \partial_a(\varphi_1)^* \varphi_2 \vee \partial_a(\varphi_2)$}
  $o(\varphi_1).ff \vee (\bigvee_{a \in \delta(\varphi_1)} a.\partial_a(\varphi_1)).\varphi_1^* \varphi_2 \vee o(\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_2)} a.\partial_a(\varphi_2))$

  $\Rightarrow$ {Distributivité de . sur $\vee$, $o(\varphi_1) = ff$}
  $(\bigvee_{a \in \delta(\varphi_1)} (a.\partial_a(\varphi_1).\varphi_1^* \varphi_2) \vee o(\varphi_2) \vee (\bigvee_{a \in \delta(\varphi_2)} a.\partial_a(\varphi_2))$

  $\Rightarrow$ {$\partial_a(\varphi_1) = ff$ si $a \notin \delta(\varphi_1)$, nous obtenons alors :}
  $o(\varphi_2) \vee (\bigvee_{a \in (\delta(\varphi_1) \vee \delta(\varphi_2))} (a.\partial_a(\varphi_1).\varphi_1^* \varphi_2 \vee \partial_a(\varphi_2))$

  $\Rightarrow$ {Définition 6.5.1, $o(\varphi_1^* \varphi_2) = o(\varphi_2)$, $\delta(\varphi_1^* \varphi_2) = \delta(\varphi_1) \vee \delta(\varphi_2))$, $\partial_a(\varphi_1^* \varphi_2) = a.\partial_a(\varphi_1).\varphi_1^* \varphi_2$}
  $o(\varphi_1^* \varphi_2) \vee (\bigvee_{a \in \delta(\varphi_1^* \varphi_2)} a.\partial_a(\varphi_1^* \varphi_2)) \sim \varphi_1^* \varphi_2$

□

Proposition 6.10.4. *(lien entre $\partial_a(\varphi)$ et φ)*
Soient φ une formule dans $LTL_{\Downarrow}$ et a une action dans $\mathcal{A}$, nous avons :

$$a \in \delta(\varphi) \Leftrightarrow \exists \Psi \in LTL_{\Downarrow} \mid \varphi \sim a.\partial_a(\varphi) \vee \Psi \ et \ \ a \notin \delta(\Psi)$$

Démonstration.
Prouvons les deux sens :

- $a \in \delta(\varphi)$

  $\Rightarrow$ {Proposition 6.10.3, soit $\Psi = o(\varphi) \vee (\bigvee_{a' \in \delta(\varphi)-\{a\}} a'.\partial_{a'}(\varphi))$, nous obtenons :}

  $\varphi \sim a.\partial_a(\varphi) \vee \Psi$ *avec* $a \notin \delta(\Psi)$

- $\exists \Psi \mid \varphi \sim a.\partial_a(\varphi) \vee \Psi$

  $\Rightarrow$ {Supposons $\varphi \sim a.\partial_a(\varphi) \vee \Psi$ et $a \notin \delta(\Psi)$, nous obtenons directement à partir de la définition 6.5.1 :}>

  $a \in \delta(\varphi)$

□

6.10.2 Propriétés liées à P

Une définition inductive de $\approx_\varphi$ est donnée par la suite.

Définition 6.10.2. *(Relation d'équivalence par rapport à une formule $\approx^i_\varphi$)*
Soient $P,Q \in \mathcal{P}$, la relation $\approx^i_\varphi \subseteq \mathcal{P} \times \mathcal{P}$ est définie inductivement comme suit :

- $P \approx^0_\varphi Q$ *;*
- $P \approx^{i+1}_\varphi Q$*, si :*

  (i) $P \overset{a}{\rightarrowtail} P'$ *et* $a \in \delta(\varphi)$ *alors* $\exists\, Q' \in \mathcal{P} : Q \overset{a}{\rightarrowtail} Q'$ *et* $P' \approx^i_{\partial_a(\varphi)} Q'$ *et*

  (ii) $Q \overset{a}{\rightarrowtail} Q'$ *et* $a \in \delta(\varphi)$*, alors* $\exists P' \in \mathcal{P} : P \overset{a}{\rightarrowtail} P'$ *et* $Q' \approx^i_{\partial_a(\varphi)} P'$

6.10.3 Propriétés liées au renforcement

Définition 6.10.3. *(Déblocage de $\overset{receive(s,c)}{\twoheadrightarrow}$)*
Étant donné deux programmes P et P' dans $\mathcal{P}$, une chaîne de caractères s, la relation $\twoheadrightarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ est définie comme suit :

$$P \overset{receive(s,c)}{\twoheadrightarrow} P' \quad si \quad P \parallel send(s,c) \overset{receive(s,c)}{\rightsquigarrow} P'$$

avec c est un canal de synchronisation.

Proposition 6.10.5. *($\overset{send(s,c)}{\rightsquigarrow}$)*
Soit $P \in \mathcal{P}$, s une chaîne de caractères, pour tout $E \in \mathcal{E}$, il existe E' tel que :

$$< send(s,c)\ P, < c \mapsto \text{"}undefined\text{"} >_{\textbf{FieldsObject}} \dagger E > \overset{send(s,c)}{\rightsquigarrow} < P, < c \mapsto s >_{\textbf{FieldsObject}} \dagger E' >$$

avec c est un canal de synchronisation de taille 1.

Démonstration.

D'après la définition 6.7.3, $send()$ est dénotée par $<M(send), I(i), R(r)>_{\mathbf{k}}$. Il s'agit une méthode $m = send$ qui consiste en un ensemble d'instructions i et un ensemble de registre r. Dans ce qui suit nous présentons l'application de chaque règle associée à ces instructions.

$\Rightarrow$ { Définition 6.7.3 de $send()$, R_{LOCAL}}
$< <M(send), I(0), R(r)>_{\mathbf{k}}, <0 \mapsto .locals\ 5>_{\mathbf{code}},\ E > \xrightarrow{.locals\ 5} <M(send), I(1), R(r)>_{\mathbf{k}},\ <v0 \mapsto \bot, v1 \mapsto \bot, v2 \mapsto \bot, v3 \mapsto \bot, v4 \mapsto \bot>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{IGET}}
$< <M(send), I(1), R(r)>_{\mathbf{k}}, <1 \mapsto iget\ v0, p1, Lp/PhiChannel;- > c>_{\mathbf{code}}, E > \xrightarrow{iget\ v0,p1,Lp/PhiChannel;->c} <M(send), I(2), R(r)>_{\mathbf{k}}, <v0 \mapsto c>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, $R_{MONITOR-ENTER}$}
$< <M(send), I(2), R(r)>_{\mathbf{k}},\ <2 \mapsto monitor\text{-}enter\ v0>_{\mathbf{code}},\ E > \xrightarrow{monitor-enter\ v0} <M(send), I(3), R(r)>_{\mathbf{k}}, <v0 \mapsto c>_{\mathbf{Registers}}, << s, c >_{\mathbf{Reference}} <status \mapsto "ThID">_{\mathbf{ReservedObject}} >_{\mathbf{Object}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, $R_{ARRAY-LENGTH}$}
$< <M(send), I(3), R(r)>_{\mathbf{k}},\ <3 \mapsto array\text{-}length\ v1, v0>_{\mathbf{code}},\ E > \xrightarrow{array-length\ v1,v0} <M(send), I(4), R(r)>_{\mathbf{k}}, <v1 \mapsto 1>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{Const}}
$< <M(send), I(4), R(r)>_{\mathbf{k}},\ <4 \mapsto const\ v2\ 0>_{\mathbf{code}},\ E > \xrightarrow{const\ v2\ 0} <M(send), I(5), R(r)>_{\mathbf{k}},\ <v2 \mapsto 0>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{Label}}
$< <M(send), I(5), R(r)>_{\mathbf{k}}, <6 \mapsto: Wait>_{\mathbf{code}},\ E > \xrightarrow{:Wait} <M(send), I(6), R(r)>_{\mathbf{k}},\ E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{aget}}
$< <M(send), I(6), R(r)>_{\mathbf{k}}, <6 \mapsto aget\ v3, v0, v2>_{\mathbf{code}} < v0 \mapsto c\ \ v2 \mapsto 0 >_{\mathbf{Registers}} <c[0] \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \xrightarrow{aget\ v3,v0,v2} <M(send), I(7), R(r)>_{\mathbf{k}},\ <v3 \mapsto "undefined">_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, $R_{Const-string}$}
$< <M(receive), I(7), R(r)>_{\mathbf{k}},\ <7 \mapsto const\text{-}string\ v4, "undefined">_{\mathbf{code}},\ E > \xrightarrow{const\text{-}string\ v4,"undefined"} <M(receive), I(8), R(r)>_{\mathbf{k}},\ <v4 \mapsto "undefined">_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, $R_{If-eqTrue}$}
$< <M(send), I(8), R(r)>_{\mathbf{k}}, <8 \mapsto if\text{-}eq\ v3, v4\ :label1\ \ 11 \mapsto: label1>_{\mathbf{code}}, <v3 \mapsto "undefined"\ v4 \mapsto "undefined">_{\mathbf{Registers}}, E > \xrightarrow{if\text{-}eq\ v3,v4,:label1} < <M(send), I(11), R(r)>_{\mathbf{k}},\ E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{Label}}
$< <M(send), I(11), R(r)>_{\mathbf{k}}, <11 \mapsto: label>_{\mathbf{code}},\ E > \xrightarrow{:label} <M(send), I(12), R(r)>_{\mathbf{k}},\ E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{APUT}}
$< <M(send), I(12), R(r)>_{\mathbf{k}},\ <12 \mapsto aput\ p0, v0, v2>_{\mathbf{code}}, <p0 \mapsto s\ v0 \mapsto c>_{\mathbf{Registers}} \dagger E > \xrightarrow{aput\ p0,v0,v2} <M(send), I(13), R(r)>_{\mathbf{k}},\ <c[0] \mapsto s>_{\mathbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{Const}}
$< <M(send), I(13), R(r)>_{\mathbf{k}},\ <13 \mapsto constv4\ 1>_{\mathbf{code}},\ E > \xrightarrow{const\ v4\ 1} <M(send), I(14), R(r)>_{\mathbf{k}},\ <v4 \mapsto 1>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{ADD}}
$< <M(send), I(14), R(r)>_{\mathbf{k}}, <14 \mapsto add\ v2\ v2\ v4>_{\mathbf{code}}, v2 \mapsto 0\ v4 \mapsto 1 >_{\mathbf{Registers}}, \dagger E > \overset{add\ v2\ v2\ v4}{\longrightarrow}$
$<M(send), I(15), R(r)>_{\mathbf{k}}, <v2 \mapsto 1>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, $R_{IF-ltFalse}$}
$< <M(send), I(15), R(r)>_{\mathbf{k}}, <15 \mapsto if\text{-}lt\ v2\ v1\ Wait>_{\mathbf{code}}, < v2 \mapsto 1\ v1 \mapsto 1 >_{\mathbf{Registers}} \dagger E >$
$\overset{if\text{-}lt\ v2\ v1\ Wait}{\longrightarrow} <M(send), I(16), R(r)>_{\mathbf{k}}, <v2 \mapsto 1>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, $R_{MONITOR-EXIT}$}
$< <M(send), I(16), R(r)>_{\mathbf{k}}, <16 \mapsto monitor\text{-}exit\ v0>_{\mathbf{code}}, E > \overset{monitor-exit\ v0}{\longrightarrow}$
$<M(send), I(17), R(r)>_{\mathbf{k}} <v0 \mapsto c>_{\mathbf{Registers}} <status \mapsto "undefined">_{\mathbf{ReservedObject}}, E' >$

$\Rightarrow$ {Définition 6.7.3 de $send()$, R_{RETURN}}
$< <M(send), I(17), R(r)>_{\mathbf{k}}, <17 \mapsto return\text{-}void>_{\mathbf{code}}, E' > \overset{return\text{-}void}{\longrightarrow} <\bullet>_{\mathbf{k}}, E' >$

$\Rightarrow$ {Fin de la méthode $send()$, définition 6.9.6 de $\rightsquigarrow$}
$< send(s, c)\ P, <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \overset{send(s,c)}{\rightsquigarrow} < P, <c \mapsto s>_{\mathbf{FieldsObject}} \dagger E' >$

□

Proposition 6.10.6. $(\overset{receive(s,c)}{\rightsquigarrow})$
Soient $P \in \mathcal{P}$ et s une chaîne de caractères, pour tout $E \in \mathcal{E}$ il existe E', tel que :

$$< receive(s, c)\ P, < c \mapsto s >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(s,c)}{\rightsquigarrow} < P, < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E' >$$

avec c est un canal de synchronisation de taille 1.

Démonstration.

D'après la définition 6.7.4, $receive()$ est dénotée par $<M(receive), I(i), R(r)>_{\mathbf{k}}$. Il s'agit une méthode $m = receive$ qui consiste en un ensemble d'instructions i et un ensemble de registre r. Dans ce qui suit nous présentons l'application de chaque règle associée à ces instructions.

$\Rightarrow$ { Définition 6.7.4 de $receive()$, R_{LOCAL}}
$<M(receive), I(0), R(r)>_{\mathbf{k}}, <0 \mapsto .locals\ 5>_{\mathbf{code}}, E > \overset{.locals\ 5}{\longrightarrow} <M(receive), I(1), R(r)>_{\mathbf{k}}, <v0 \mapsto \bot, v1 \mapsto \bot, v2 \mapsto \bot, v3 \mapsto \bot, v4 \mapsto \bot>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.4 de $receive()$, R_{IGET}}
$< <M(receive), I(1), R(r)>_{\mathbf{k}}, <1 \mapsto iget\ v0, p1, Lp/PhiChannel;- > c>_{\mathbf{code}}, E >$
$\overset{iget\ v0,p1,Lp/PhiChannel;->c}{\longrightarrow} <M(receive), I(2), R(r)>_{\mathbf{k}}, <v0 \mapsto c>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.4 de $receive()$, $R_{MONITOR-ENTER}$}
$< <M(receive), I(2), R(r)>_{\mathbf{k}}, <2 \mapsto monitor\text{-}enter\ v0>_{\mathbf{code}}, E > \overset{monitor-enter\ v0}{\longrightarrow}$
$<M(receive), I(3), R(r)>_{\mathbf{k}}, <v0 \mapsto c>_{\mathbf{Registers}}, << c >_{\mathbf{Reference}} <status \mapsto "ThID">_{\mathbf{ReservedObject}} >_{\mathbf{Object}}$
$\dagger E >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), $R_{ARRAY-LENGTH}$}
$< \quad <M(receive), I(3), R(r)>_{\mathbf{k}}, \quad <3 \mapsto array\text{-}length\ v1, v0>_{\mathbf{code}}, \quad E \quad > \xrightarrow{array-length\ v1,v0}$
$<M(receive), I(4), R(r)>_{\mathbf{k}}, <v1 \mapsto 1>_{\mathbf{Registers}} \dagger E>$

$\Rightarrow$ {Définition 6.7.3 de *send*(), R_{Const}}
$< <M(send), I(4), R(r)>_{\mathbf{k}}, <4 \mapsto const\ v2\ 0>_{\mathbf{code}}, E > \xrightarrow{const\ v2\ 0} <M(send), I(5), R(r)>_{\mathbf{k}}, <v2 \mapsto 0>_{\mathbf{Registers}} \dagger E>$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), R_{Label}}
$< <M(receive), I(5), R(r)>_{\mathbf{k}}, <6 \mapsto: Wait>_{\mathbf{code}}, E > \xrightarrow{:Wait} <M(receive), I(6), R(r)>_{\mathbf{k}}, E>$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), R_{aget}}
$< \quad <M(receive), I(6), R(r)>_{\mathbf{k}}, <7 \mapsto aget\ v3, v0, v2>_{\mathbf{code}} < v0 \mapsto c \quad v2 \mapsto 0 >_{\mathbf{Registers}}$
$<c[0] \mapsto "s">_{\mathbf{FieldsObject}} \dagger E > \xrightarrow{aget\ v3,v0,v2} <M(receive), I(7), R(r)>_{\mathbf{k}}, <v3 \mapsto s>_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), $R_{If-eqTrue}$}
$< <M(receive), I(5), R(r)>_{\mathbf{k}}, <5 \mapsto if\text{-}eq\ v3, v0, : lab1\ 8 \mapsto: lab1>_{\mathbf{code}}, <v3 \mapsto s\ p0 \mapsto s>_{\mathbf{Registers}} \dagger E>$
$if\text{-}eq \xrightarrow{v0,v1,:lab1} < <M(receive), I(8), R(r)>_{\mathbf{k}}, E>$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), R_{Label}}
$< <M(receive), I(8), R(r)>_{\mathbf{k}}, <8 \mapsto: lab1>_{\mathbf{code}}, E > \xrightarrow{:label} <M(receive), I(9), R(r)>_{\mathbf{k}}, E>$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), $R_{Const-string}$}
$< <M(receive), I(9), R(r)>_{\mathbf{k}}, <9 \mapsto const\text{-}string\ v4, "undefined">_{\mathbf{code}}, E > \xrightarrow{const\text{-}string\ v4,"undefined"}$
$<M(receive), I(10), R(r)>_{\mathbf{k}}, <v4 \mapsto "undefined">_{\mathbf{Registers}} \dagger E' >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), R_{APUT}}
$< <M(receive), I(10), R(r)>_{\mathbf{k}}, <10 \mapsto aput\ v4, v0, v2>_{\mathbf{code}}, < v4 \mapsto "undefined"\ v0 \mapsto c\ v2 \mapsto 0 >_{\mathbf{Registers}} <c[0] \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' > \xrightarrow{aput\ v4,\ v0,\ v2} <M(receive), I(11), R(r)>_{\mathbf{k}}, <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), R_{Const}}
$< <M(receive), I(11), R(r)>_{\mathbf{k}}, <13 \mapsto const\ v4\ 1>_{\mathbf{code}}, E' > \xrightarrow{const\ v4\ 1} <M(receive), I(12), R(r)>_{\mathbf{k}}, <v4 \mapsto 1>_{\mathbf{Registers}} \dagger E' >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), R_{ADD}}
$< <M(receive), I(12), R(r)>_{\mathbf{k}}, <14 \mapsto add\ v2\ v2\ v4>_{\mathbf{code}}, v2 \mapsto 0\ v4 \mapsto 1 >_{\mathbf{Registers}}, \dagger E' > \xrightarrow{add\ v2\ v2\ v4}$
$<M(receive), I(13), R(r)>_{\mathbf{k}}, <v2 \mapsto 1>_{\mathbf{Registers}} \dagger E' >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), $R_{IF-ltFalse}$}
$< \quad <M(receive), I(15), R(r)>_{\mathbf{k}}, \quad <15 \mapsto if\text{-}lt\ v2\ v1\ Wait>_{\mathbf{code}}, < \quad v2 \mapsto 1\ v1 \mapsto 1 \quad >_{\mathbf{Registers}} \quad \dagger E' \quad >$
$if\text{-}lt \xrightarrow{v2\ v1\ Wait} <M(receive), I(16), R(r)>_{\mathbf{k}}, <v2 \mapsto 1>_{\mathbf{Registers}} \dagger E' >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), $R_{MONITOR-EXIT}$}
$< \; <M(receive), I(11), R(r)>_{\mathbf{k}}, \; <11 \mapsto monitor\text{-}exit \; v0>_{\mathbf{code}}, \; E' \; > \xrightarrow{monitor-exit\ v0} <$
$<M(receive), I(12), R(r)>_{\mathbf{k}} <v0 \mapsto "ObjectRef">_{\mathbf{Registers}} <status \mapsto "undefined">_{\mathbf{ReservedObject}} \dagger E' >$

$\Rightarrow$ {Définition 6.7.4 de *receive*(), R_{RETURN}}
$< <M(receive), I(12), R(r)>_{\mathbf{k}}, <12 \mapsto return\ v3>_{\mathbf{code}}, < v3 \mapsto s >_{\mathbf{Registers}} \dagger E > \xrightarrow{return\ v1} < < \bullet >_{\mathbf{k}}, < \_ \mapsto$
$s >_{\mathbf{ReturnValue}} \dagger E' >$

$\Rightarrow$ {Fin de la méthode *receive*(), définition 6.9.6 de $\rightsquigarrow$}
$< receive(s,c) \;\; P, \; < c \mapsto s>_{\mathbf{FieldsObject}} \dagger E > \overset{receive(s,c)}{\rightsquigarrow} < P, \; <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' >$

□

Proposition 6.10.7. *($\overset{receive(s,c)}{\twoheadrightarrow}$)*
Soient $P \in \mathcal{P}$, s une chaîne de caractères, pour tout $E \in \mathcal{E}$ il existe E', tel que :

$< receive(s,c) \;\; P, \; < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(s,c)}{\twoheadrightarrow} < P, \; < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E' >$

avec c est un canal de synchronisation de taille 1.

Démonstration.

$\Rightarrow$ {Définition 6.10.3 de $\twoheadrightarrow$, proposition 6.10.5 de $\overset{send(s,c)}{\rightsquigarrow}$}
$< receive(s,c) \; P \parallel send(s,c), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{send(s,c)}{\rightsquigarrow} < receive(s,c) \; P \;\; < c \mapsto$
$s >_{\mathbf{FieldsObject}} \dagger E'' >$

$\Rightarrow$ {Définition 6.10.3 de $\twoheadrightarrow$, proposition 6.10.6 de $\overset{receive(s,c)}{\rightsquigarrow}$}
$< receive(s,c) \; P \;\; < c \mapsto s >_{\mathbf{FieldsObject}} \dagger E'' > \overset{receive(s,c)}{\rightsquigarrow} < P, \; < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E' >$

□

Proposition 6.10.8. *(T versus $\lceil T \rceil_c$ mod A)*
Soient T et T' deux threads dans Th, E et E' deux environnements dans $\mathcal{E}$, $A \subseteq \mathcal{A}$ et $a \in A$, nous avons :
$< T, \; E > \xrightarrow{a} < T', \; E' >$ ssi

$< \; L_c(S_{c,\emptyset}(T,A)), \;\; < \; c \; \mapsto \; "undefined" \; >_{\mathbf{FieldsObject}} \; \dagger E \; > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \xrightarrow{a} \overset{send(a\urcorner,c)}{\rightsquigarrow} < \; L_c(S_{c,\emptyset}(T',A)), \;\; < \; c \; \mapsto$
$a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

avec c est un canal de synchronisation.

Démonstration.
La preuve est par induction structurelle sur $\mathcal{A}$.

— Montrons que si $< T, \; E > \xrightarrow{a} < T', \; E' >$ alors il existe une chaîne de caractère $a\urcorner$.
Cas $T =<< M(m), I(i), R(r) >_{\mathbf{k}}, \; < i \mapsto move \; v1 \; v2 >_{\mathbf{code}}>$ et $a = move \; v1 \; v2$
L'action $a = move \; v1 \; v2$ modifie l'environnement où le registre $v1$ reçoit la valeur de registre $v2$ (la cellule *Registers*) et avance à l'instruction suivante. Soit *inst* l'instruction qui vient juste après l'instruction a. Par hypothèse nous avons :

$<< M(m), I(i), R(r) >_{\mathbf{k}},\ < i \mapsto move\ v1\ v2 >_{\mathbf{code}}>,\ E > \overset{move\ v1\ v2}{\longrightarrow} < M(m), I(i+1), R(r) >_{\mathbf{k}},\ < i \mapsto inst >_{\mathbf{code}}>,\ E' >$

{Définition 6.6.1 : $\lceil < a,\ E > \rceil_c\ mod\ A =< L_c(S_{c,\emptyset}(a, A,), c \mapsto "undefined">$ avec $S_{c,\emptyset}(a, A) \overset{(7.1)}{=} (receive(\ulcorner a, c)\ a\ send(a\urcorner, c),\ \emptyset)$ avec $a \in A$ et $a \neq goto$ et $a \neq if\text{-}eq$ et $L_c(: Label, \emptyset) \overset{(8.1)}{=} receive(\ulcorner a, c)\ a\ send(a\urcorner, c)\ avec\ a \neq: Label$, avec la proposition 6.10.7 de $\overset{receive(s,c)}{\twoheadrightarrow}$, nous obtenons : }

$< receive(\ulcorner a, c)\ a\ send(a\urcorner, c), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} < a\ send(a\urcorner, c), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ $\{R_{MOVE}\}$
$< <M(m), I(i), R(r)>_{\mathbf{k}}, <i \mapsto move\ v1\ v2>_{\mathbf{code}}, < v2 \mapsto val >_{\mathbf{Registers}},\ E > \overset{move\ v1\ v2}{\longrightarrow} < <M(m), I(i+1), R(r)>_{\mathbf{k}}, < v1 \mapsto val >_{\mathbf{Registers}} \dagger E >$

$\Rightarrow$ {Invocation de la méthode $send(a\urcorner, c)$, $R_{INVOKE-STATIC}$}
$< <M(m), I(i+1), R(r)>_{\mathbf{k}}, <i+1 \mapsto invoke\text{-}static\{v0\}, \{v1\}LPhiChannel; \rightarrow send >_{\mathbf{code}}, < v0 \mapsto a\urcorner\ \ v1 \mapsto c >_{\mathbf{Registers}},\ E' > \overset{invoke\text{-}static\ \{v0\},\{v1\},LPhiChannel;->send}{\longrightarrow} << lookupmethod(LPhiChannel; \rightarrow send), I(0), R(r') \rightsquigarrow M(m), I(i+2), R(r) >_{\mathbf{k}}, E' >$

$\Rightarrow$ {Proposition 6.10.5 de $\overset{send(a\urcorner,c)}{\rightsquigarrow}$}
$< send(a\urcorner, c),\ < c \mapsto "undefined" <_{\mathbf{FieldsObject}} \dagger E' > \overset{send(a\urcorner,c)}{\rightsquigarrow} << M(m), I(i+2), R(r) >_{\mathbf{k}}, < i+2 \mapsto inst >_{code}, < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Par hypothèse nous avons $< a,\ E > \overset{a}{\longrightarrow} < inst,\ E' >$. $T' =< M(m), I(i+2), R(r) >_{\mathbf{k}}, < i+2 \mapsto inst >_{code}$ alors $L_c(S_{c,\emptyset}(T', A)) = L_c(S_{c,\emptyset}(inst, A))$, par la suite nous obtenons :}
$< L_c(S_{c,\emptyset}(T, A)), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \overset{a}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)), < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

— Montrons que si $< L_c(S_{c,\emptyset}(T, A)),\ < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \overset{a}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)),\ < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$ alors : $< T,\ E > \overset{move\ v1\ v2}{\longrightarrow} < T',\ E' >$

Par hypothèse : $L_c(S_{c,\emptyset}(T, A)) = receive(\ulcorner a, c)\ move\ v1\ v2\ \ send(a\urcorner, c)$ et
$< L_c(S_{c,\emptyset}(T, A)), <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \overset{move\ v1\ v2}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T, A)),\ <c \mapsto a\urcorner>_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Proposition 6.10.7 de $\overset{receive(\ulcorner a,c)}{\twoheadrightarrow}$}
$< receive(\ulcorner a, c)\ move\ v1\ v2\ \ send(a\urcorner, c)\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} < move\ v1\ v2\ send(a\urcorner, c), <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Par hypothèse nous avons : $< move\ v1\ v2\ send(a\urcorner, c),\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \overset{move\ v1\ v2}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < \lceil T' \rceil_c\ mod\ A,\ <c \mapsto a\urcorner>_{\mathbf{FieldsObject}} \dagger E' >$, avec la règle R_{MOVE} nous obtenons :}
$< move\ v1\ v2,\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \overset{move\ v1\ v2}{\longrightarrow} < inst,\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {La règle R_{MOVE} n'affecte pas la variable de contrôle c, par la suite nous pouvons écrire : }
$< move\ v1\ v2,\ E > \overset{move\ v1\ v2}{\longrightarrow} < inst,\ E' >$

— Des preuves similaires au cas précédents peuvent être faites pour les autres instructions, excepté les instructions *goto* et *if* qui seront traitées ci-après.

— Cas $T =<< M(m), I(i), R(r) >_{\mathbf{k}},\ < i \mapsto goto\ : Label >_{\mathbf{code}}>$, avec $a = goto\ : Label$

Par hypothèse : $<< M(m), I(i), R(r) >_{\mathbf{k}},\ < i \mapsto goto\ : Label\ \ j \mapsto: Label >_{\mathbf{code}}>,\ E > \overset{goto\ :Label}{\longrightarrow} < M(m), I(j), R(r) >_{\mathbf{k}}, E >$ et
$<< M(m), I(j), R(r) >_{\mathbf{k}},\ < j \mapsto: Label >_{\mathbf{code}}>,\ E > \overset{goto\ :Label}{\longrightarrow} < M(m), I(j+1), R(r) >_{\mathbf{k}},\ < j+1 \mapsto inst >_{\mathbf{code}}$
$,\ E >$

$\Rightarrow$ {Définition 6.6.1 : $\lceil < a,\ E > \rceil_c\ mod\ A =< L_c(S_{c,\emptyset}(a, A)), < c \mapsto "undefined >_{\mathbf{FieldsObject}}>$, tel que $S_{c,\emptyset}(a, A) \overset{(7.1)}{=} (receive(\ulcorner a, c)\ a\ ,\ \emptyset \dagger [: Label \mapsto a\urcorner])$, avec $a \in A$ et $a = goto\ : Label$ et $L_c(: Label,\ [: Label \mapsto a\urcorner]) \overset{(8.1)}{=} : Label\ send(a\urcorner, c)$ avec : $Label \in dom(\Gamma)$ avec la proposition 6.10.7 de $\overset{receive(s,c)}{\twoheadrightarrow}$, nous obtenons : }
$< receive(\ulcorner a, c)\ a\ : Label\ send(a\urcorner, c), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} < a\ : Label\ send(a\urcorner, c), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {R_{GOTO}}
$< <M(m), I(i), R(r)>_{\mathbf{k}}, <i \mapsto goto\ : Label\ \ j \mapsto: Label>_{\mathbf{code}},\ E > \overset{a}{\longrightarrow} < <M(m), I(j), R(r)>_{\mathbf{k}},\ E >$

$\Rightarrow$ {R_{LABEL}}
$< <M(m), I(j), R(r)>_{\mathbf{k}}, <j \mapsto: Label>_{\mathbf{code}},\ E > \overset{:Label}{\longrightarrow} < <M(m), I(j+1), R(r)>_{\mathbf{k}},\ E >$

$\Rightarrow$ {Fin de l'action : $Label$}
$<: Label\ \ send(a\urcorner, c),\ \ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger\ E > \overset{:Label}{\rightsquigarrow} <\ send(a\urcorner, c), < <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Invocation de la méthode $send$, $R_{INVOKE-STATIC}$}
$< <M(m), I(j + 1), R(r)>_{\mathbf{k}}, <j + 1 \mapsto invoke\text{-}static\{v0\}, \{v1\}LPhiChannel; \rightarrow send >_{\mathbf{code}}, < v0 \mapsto a\urcorner\ \ v1 \mapsto c >_{\mathbf{Registers}} \dagger E' > \overset{invoke\text{-}static\ \{v0\},LPhiChannel;->send}{\longrightarrow} << lookupmethod(LPhiChannel; \rightarrow send), I(0), R(r') \rightsquigarrow M(m), I(j+2), R(r) >_{\mathbf{k}},\ E' >$

$\Rightarrow$ {Proposition 6.10.5 de $\overset{send(a\urcorner,c)}{\rightsquigarrow}$}
$< send(a\urcorner, c),\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' > \overset{send(a\urcorner,c)}{\rightsquigarrow} << M(m), I(j+2), R(r) >_k,\ <c \mapsto a\urcorner>_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {$T' =<< M(m), I(i+2), R(r) >_{\mathbf{k}}, < i+2 \mapsto inst >_{\mathbf{code}}>$ alors $L_c(S_{c,\emptyset}(T', A)) = L_c(S_{c,\emptyset}(inst, A))$, par la suite nous obtenons :}
$< L_c(S_{c,\emptyset}(T, A)),\ \ < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \overset{move\ v1\ v2}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)),\ < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

— Cas $T =<< M(m), I(i), R(r) >_{\mathbf{k}},\ < i \mapsto if\text{-}eq\ v1\ v2 : Label >_{\mathbf{code}}>$, avec $a = if\text{-}eq\ v1\ v2 : Label$

Par hypothèse nous avons :

- Si la condition est évaluée à vrai ($v1 == v2$)

  $<< M(m), I(i), R(r) >_{\mathbf{k}},\ < i \mapsto \textit{if-eq}\ v1\ v2\ : Label\ \ j \mapsto : Label >_{\mathbf{code}}>,\ E > \overset{\textit{if-eq}\ v1\ v2:Label}{\longrightarrow} < M(m), I(j), R(r) >_{\mathbf{k}}, E >$ et

  $<< M(m), I(j), R(r) >_{\mathbf{k}},\ < j \mapsto : Label >_{\mathbf{code}}>,\ E > \overset{\textit{if-eq}\ v1\ v2:Label}{\longrightarrow} < M(m), I(j+1), R(r) >_{\mathbf{k}},\ < j \mapsto inst1 >_{\mathbf{code}}>,\ E >$

- Si la condition est évaluée à faux ($v1 \neq v2$)

  $<< M(m), I(i), R(r) >_{\mathbf{k}},\ < i \mapsto \textit{if-eq}\ v1\ v2 : Label >_{\mathbf{code}}>,\ E > \overset{\textit{if-eq}\ v1\ v2:Label}{\longrightarrow} < M(m), I(i+1), R(r) >_{\mathbf{k}}, < i+1 \mapsto inst2 >_{\mathbf{code}},\ E >$

$\Rightarrow$ {Définition 6.6.1 : $\ulcorner < a,\ E > \urcorner_c\ \ mod\ \ A =< L_c(S_{c,\emptyset}(a, A)), c \mapsto "undefined" >$ tel que : $S_{c,\emptyset}(a, A) \overset{(7.1)}{=} (receive(\ulcorner a, c)\ a\ send(a\urcorner, c),\ \emptyset \dagger [: Label \mapsto a\urcorner])$ avec $a \in A$ et $a = \textit{if-eq}$ et $L_c(: Label,\ [: Label \mapsto a\urcorner]) \overset{(8.1)}{=} : Label\ send(a\urcorner, c)$ avec $: Label \in dom(\Gamma)$ avec la proposition 6.10.7 de $\overset{receive(s,c)}{\twoheadrightarrow}$, nous obtenons : }

$< receive(\ulcorner a, c)\ \ a\ \ send(a\urcorner, c)\ \ : Label\ \ send(a\urcorner, c), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} < a\ send(a\urcorner, c)\ : Label\ send(a\urcorner, c), < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {**Si** $v1 == v2$ **alors** $R_{if-eqTrue}$}

$< <M(m), I(i), R(r)>_{\mathbf{k}}, <i \mapsto \textit{if-eq}\ v1\ v2\ : Label\ \ j \mapsto : Label>_{\mathbf{code}},\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' > \overset{a}{\longrightarrow} < <M(m), I(j), R(r)>_{\mathbf{k}}, <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {R_{LABEL}}

$< <M(m), I(j), R(r)>_{\mathbf{k}}, <j \mapsto : Label>_{\mathbf{code}},\ E > \overset{:Label}{\longrightarrow} < <M(m), I(j+1), R(r)>_{\mathbf{k}},\ E >$

$\Rightarrow$ {Fin de l'action : $Label$}

$<: Label\ \ send(a\urcorner, c),\ \ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \overset{:Label}{\rightsquigarrow} < send(a\urcorner, c),\ \ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Invocation de la méthode $send(a\urcorner, c)$, $R_{INVOKE-STATIC}$}

$< <M(m), I(j+1), R(r)>_{\mathbf{k}}, <j+1 \mapsto \textit{invoke-static}\{v0\}, \{v1\}LPhiChannel; \rightarrow send >_{\mathbf{code}}, < v0 \mapsto a\urcorner\ \ v1 \mapsto c >_{\mathbf{Registers}} \dagger E' > \overset{\textit{invoke-static}\ \{v0\},\{v1\}LPhiChannel;->send}{\longrightarrow} << lookupmethod(LPhiChannel; \rightarrow send), I(0), R(r') \rightsquigarrow M(m), I(j+2), R(r) >_{\mathbf{k}},\ E' >$

$\Rightarrow$ {Proposition 6.10.5 de $\overset{send(a\urcorner,c)}{\rightsquigarrow}$}

$< send(a\urcorner, c),\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' > \overset{send(a\urcorner,c)}{\rightsquigarrow} << M(m), I(j+2), R(r) >_k, <c \mapsto a\urcorner>_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {$T' =<< M(m), I(j+2), R(r) >_{\mathbf{k}}, <j+2 \mapsto inst1 >_{\mathbf{code}}>$ alors $L_c(S_{c,\emptyset}(T', A)) = L_c(S_{c,\emptyset}(inst1, A))$, nous obtenons :}

$< L_c(S_{c,\emptyset}(T, A)),\ < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \overset{\textit{if-eq}\ v1\ v2:Label}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)),\ < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {**Si** $v1 \neq v2$ **alors** $R_{if-eqFalse}$}

$< <M(m), I(i), R(r)>_{\mathbf{k}}, <i \mapsto \textit{if-eq}\ v1\ v2\ : Label>_{\mathbf{code}},\ <c \mapsto "undefined">_{\mathbf{FieldsObject}},\ E > \overset{a}{\longrightarrow} < <M(m), I(i+1), R(r)>_{\mathbf{k}},\ <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {Invocation de la méthode $send(a\urcorner, c)$, $R_{INVOKE-STATIC}$}
$< \; <M(m), I(i+1), R(r)>_{\mathbf{k}}, <i+1 \mapsto$ invoke-static$\{v0\}, \{v1\}, LPhiChannel; \rightarrow send >_{\mathbf{code}}, < v0 \mapsto "a\urcorner" v1 \mapsto c >_{\mathbf{Registers}} \dagger E' > \overset{invoke\text{-}static\ \{v0\},LPhiChannel;->send}{\longrightarrow} << lookupmethod(LPhiChannel; \rightarrow send), I(0), R(r') \rightsquigarrow M(m), \mathbf{I(i+2)}, R(r) >_{\mathbf{k}}, E' >$

$\Rightarrow$ {Proposition 6.10.5 de $\overset{send(a\urcorner,c)}{\rightsquigarrow}$}
$< \; send(a\urcorner, c), \; <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E' > \overset{send(a\urcorner,c)}{\rightsquigarrow} << M(m), I(i+2), R(r) >_k, \; < <c \mapsto a\urcorner>_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {$T' =<< M(m), I(i+2), R(r) >_{\mathbf{k}}, < i+2 \mapsto inst2 >_{\mathbf{code}}>$ alors $L_c(S_{c,\emptyset}(T', A)) = L_c(S_{c,\emptyset}(inst2, A))$, nous obtenons :}
$< L_c(S_{c,\emptyset}(T, A)), \; < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \overset{if\text{-}eq\ v1\ v2:Label}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)), < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

□

Proposition 6.10.9. *(P versus $\lceil P \rceil_c$ mod A)*
Soient P et P′ deux programmes dans $\mathcal{P}$, $A \subseteq \mathcal{A}$, nous avons :

$$P \xrightarrow{a} P' \;\; ssi \;\; \lceil P \rceil_c \; mod \; A \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \xrightarrow{a} \overset{send(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c \; mod \; A$$

avec $a \in A$ et c un canal de contrôle.

Démonstration.
Soit P un programme dans $\mathcal{P}$ tel que $P =< T, \; E >$. Par définition, nous avons :
$\lceil P \rceil_c \; mod \; A \overset{6.6.1}{=} < L_c(S_{c,\emptyset}(T, A)), \; E >$

Par hypothèse nous avons :
$P =< T, \; E > \xrightarrow{a} P'$

$\Rightarrow$ {Pour $a \in A$: }
$< T, \; E > \xrightarrow{a} < T', \; E' >$

$\Rightarrow$ {Proposition 6.10.8, il existe $\ulcorner a$ et $a\urcorner$ tel que :}
$< L_c(S_{c,\emptyset}(T, A)), \; < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} \xrightarrow{a} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)), \; < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Définition 6.9.7, il existe $T1$, $E1$, $T2$, $E2$, tel que :}
$L_c(S_{c,\emptyset}(T, A)), \; < c \mapsto "undefined" >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} < T1, \; E1 > \xrightarrow{a} < T2, \; E2 > \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)), \; < c \mapsto a\urcorner >_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Proposition 6.10.7 de $\overset{receive(\ulcorner a,c)}{\twoheadrightarrow}$}
$< L_c(S_{c,\emptyset}(T, A)), <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\rightsquigarrow} < T1, <c \mapsto "undefined">_{\mathbf{FieldsObject}} \dagger E1 > \xrightarrow{a} < T2, \; E2 > \overset{send(a\urcorner,c}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T', A)), <c \mapsto a\urcorner>_{\mathbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Définition 6.9.7, abréviation de $\overset{receive(\ulcorner a,c)}{\rightsquigarrow} \overset{a}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow}$}
$< L_c(S_{c,\emptyset}(T,A)),\ <c \mapsto \text{"}undefined\text{"}>_{\textbf{FieldsObject}} \dagger E > \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \overset{a}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} < L_c(S_{c,\emptyset}(T',A)), <c \mapsto a\urcorner>_{\textbf{FieldsObject}} \dagger E' >$

$\Rightarrow$ {Puisque $\lceil P \rceil_c\ mod\ A =< L_c(S_{c,\emptyset}(T,A)),\ <c \mapsto \text{"}undefined\text{"}>_{\textbf{FieldsObject}} \dagger E >$ et $\lceil P' \rceil_c\ mod\ A= < L_c(S_{c,\emptyset}(T',A)),\ <c \mapsto a\urcorner>_{\textbf{FieldsObject}} \dagger E >$, nous obtenons :}
$\lceil P \rceil_c\ mod\ A \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \overset{a}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c\ mod\ A$

□

Proposition 6.10.10. *(φ versus $[\![\varphi]\!]_c$)*
Soient φ une formule $LTL_{\Downarrow}$ avec $\delta(\varphi) = a$, E un environnement dans $\mathcal{E}$, nous avons :

$$[\![\varphi]\!]_c =< send(\ulcorner a,c)\ receive(a\urcorner,c)\ \ F_{c,\emptyset}(\partial_a(\varphi)),\ <c \mapsto \text{"}undefined\text{"}>_{\textbf{FieldsObject}} \dagger E >$$

Démonstration.

$\Rightarrow$ {Proposition 6.10.4, avec $a \in \delta(\varphi)$ et $a \notin \delta(\Psi)$}
$[\![\varphi]\!]_c = [\![a.\partial_a(\varphi) \vee \Psi]\!]_c$

$\Rightarrow$ {Définition 6.5.4 de $[\![-]\!]_c$}
$[\![\varphi]\!]_c =< F_{c,\emptyset}(a.\partial_a(\varphi) \vee \Psi),\ <c \mapsto \text{"}undefined\text{"}>_{\textbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {Définition (6.5.1) de F avec $\delta(\varphi) = a$ }
$send(\ulcorner a,c)\ receive(a\urcorner,c)\ F_{c,\emptyset}(\partial_a(a.\partial_a(\varphi) \vee \Psi))$

$\Rightarrow$ {Définition 6.5.1 de ∂ avec $a \notin \delta(\Psi)$}>
$send(\ulcorner a,c)\ receive(a\urcorner,c)\ F_{c,\emptyset}\partial_a(\varphi)$

□

Nous avons vu lors de la définition de la sémantique d'un programme $\mathbb{K}$-*Smali*, P qu'un thread T ne peut évoluer que s'il est sélectionné par l'ordonnanceur *Scheduler*. Cette condition doit être remplie avant l'application de chaque règle $\mathbb{K}$. Sinon, T reste dans un état d'attente dans la cellule $\langle\ \rangle_{Scheduler}$, c.-à-d. qu'il fait partie toujours de la configuration de P avec le thread qui s'exécute dans la cellule $\mathbb{K}$.

Proposition 6.10.11. *(Évolution d'un thread)*
Soient $T1$ et $T2$ deux threads dans Th, $a \in \mathcal{A}$ tel que $T1 \xrightarrow{a} T1'$, alors :

$$T1 \parallel T2 \overset{a}{\rightsquigarrow} T1' \parallel T2 \quad ssi\ scheduler(T1,T2) = T1$$

Démonstration.
La preuve découle directement de la définition de chaque règle $\mathbb{K}$ qui vient avec une condition qui vérifie que le thread exécutant est celui retourné par la fonction *scheduler()*.

□

Proposition 6.10.12. *(Communication)*
Soient $T1$ et $T2$ deux threads dans Th, tels que $scheduler(T1, T2) = T1$ *et* $T1 \overset{send(\ulcorner a,c)}{\longrightarrow} send(\ulcorner a, c) \parallel T1'$ *et* $T2 \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} T2'$*, alors :*

$$T1 \parallel T2 \overset{send(\ulcorner a,c)}{\rightsquigarrow} \overset{receive(\ulcorner a,c)}{\rightsquigarrow} T1' \parallel T2'$$

avec c est le canal de synchronisation.

Démonstration.
Nous avons :

- $T1 \overset{send(\ulcorner a,c)}{\rightsquigarrow} send(\ulcorner a, c) \parallel T1'$

$\Rightarrow$ {Proposition 6.10.11 avec $scheduler(T1, T2) = T1$}
$T1 \parallel T2 \overset{send(\ulcorner a,c)}{\rightsquigarrow} send(\ulcorner a, c) \parallel T1' \parallel T2$ (1)

- $T2 \overset{receive(\ulcorner a,c)}{\twoheadrightarrow} T2'$

$\Rightarrow$ {Définition 6.10.3 de $\twoheadrightarrow$}
$T2 \parallel send(\ulcorner a, c) \overset{receive(\ulcorner a,c)}{\rightsquigarrow} T2'$ (2)

$\Rightarrow$ {Les résultats (1) et (2) donnent :}
$T1 \parallel T2 \overset{send(\ulcorner a,c)}{\rightsquigarrow} \overset{receive(\ulcorner a,c)}{\rightsquigarrow} T1' \parallel T2'$

□

Proposition 6.10.13. *(P versus $P \sqcap_A \varphi$)*
Soient $P, P' \in \mathcal{P}$, $\varphi \in LTL_{\Downarrow}$, $A \subseteq \mathcal{A}$ et $a \in A$, nous avons :

$$P \overset{a}{\mapsto} P' \text{ et } \delta(\varphi) = a \text{ alors } P \sqcap_A \varphi \overset{a}{\rightarrowtail} P' \sqcap_A \partial_a(\varphi)$$

Démonstration.
Nous avons :

- $\delta(\varphi) = a$

$\Rightarrow$ {Proposition 6.10.10}
$[\![\varphi]\!]_c =< send(\ulcorner a, c),\ receive(a^{\urcorner}, c)\ F_{c,\emptyset}(\partial_a(\varphi)),\ <c \mapsto \text{"}undefined\text{"}>_{\mathbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {Proposition 6.10.5 de $\overset{send(\ulcorner a,c)}{\rightsquigarrow}$}
$[\![\varphi]\!]_c \overset{send(\ulcorner a,c)}{\rightsquigarrow} < receive(a^{\urcorner}, c)\ F_{c,\emptyset}(\partial_a(\varphi)),\ <c \mapsto \ulcorner a>_{\mathbf{FieldsObject}} \dagger E >$ (1)

- $P \overset{a}{\mapsto} P'$

$\Rightarrow$ {Proposition 6.10.9}
$\lceil P \rceil_c \ mod\ A \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \overset{a}{\longrightarrow} \overset{send(a^{\urcorner},c)}{\rightsquigarrow} \lceil P' \rceil_c \ mod\ A$

$\Rightarrow$ {Définition 6.9.7, ils existent $P1$, $P2$, tel que}
$\lceil P \rceil_c \ mod\ A \overset{receive(\ulcorner a,c)}{\rightsquigarrow} P1 \quad P1 \overset{a}{\longrightarrow} P2 \quad P2 \overset{send(a^{\urcorner},c)}{\rightsquigarrow} \lceil P' \rceil_c \ mod\ A$

$\Rightarrow$ {Propositions 6.10.11 et 6.10.12 avec (1)}
$\lceil P \rceil_c \; mod \; A \parallel [\![\varphi]\!]_c \overset{send(\ulcorner a,c)}{\rightsquigarrow} \overset{receive(\ulcorner a,c)}{\rightsquigarrow} P1 \parallel < receive(a\urcorner, c) \; F_{c,\emptyset}(\partial_a(\varphi)) \; < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {$P1 \overset{a}{\mapsto} P2$, proposition 6.10.11, avec $scheduler() = P1$}
$P1 \parallel < receive(a\urcorner, c) \; F_{c,\emptyset}(\partial_a(\varphi)), \; < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E > \xrightarrow{a} P2 \parallel < receive(a\urcorner, c) \; F_{c,\emptyset}(\partial_a(\varphi)), < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {$P2 \overset{send(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c \; mod \; A$ et proposition 6.10.11 avec $scheduler() = P2$}
$P2 \parallel < receive(a\urcorner, c) \; F_{c,\emptyset}(\partial_a(\varphi)), \; < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E > \overset{send(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c \; mod \; A \parallel < receive(a\urcorner, c) \; F_{c,\emptyset}(\partial_a(\varphi)), < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {Proposition 6.10.6 de $\overset{receive(a\urcorner,c)}{\rightsquigarrow}$}>
$\lceil P' \rceil_c \; mod \; A \parallel < receive(a\urcorner, c) \; F_{c,\emptyset}(\partial_a(\varphi)), \; < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E > \overset{receive(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c \; mod \; A \parallel < F_{c,\emptyset}(\partial_a(\varphi)), < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {Les résultats précédents et d'après la définition 6.5.4 de $[\![\partial_a(\varphi)]\!]_c =< F_{c,\emptyset}(\partial_a(\varphi)), < c \mapsto \ulcorner a >_{\mathbf{FieldsObject}} \dagger E >$, on déduit que : }
$\lceil P \rceil_c \; mod \; A \parallel [\![\varphi]\!]_c \overset{send(\ulcorner a,c)}{\rightsquigarrow} \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \xrightarrow{a} \overset{send(a\urcorner,c)}{\rightsquigarrow} \overset{receive(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c \; mod \; A \parallel [\![\partial_a(\varphi)]\!]_c$

$\Rightarrow$ {Définition 6.7.1 de $\sqcap_A$}
$P \sqcap_A \varphi \overset{send(\ulcorner a,c)}{\rightsquigarrow} \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \xrightarrow{a} \overset{send(a\urcorner,c)}{\rightsquigarrow} \overset{receive(a\urcorner,c)}{\rightsquigarrow} P' \sqcap_A \partial_a(\varphi)$

$\Rightarrow$ {Définition 6.9.5 de $\rightarrowtail$, $\{send(\ulcorner a, c), send(a\urcorner, c), receive(\ulcorner a, c), receive(a\urcorner, c)\} \in A_{sync}$}
$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P' \sqcap_A \partial_a(\varphi)$

Proposition 6.10.14. *(P versus $P \sqcap \varphi$)* □
Soient $P, P1$ et $P \in \mathcal{P}$, $\varphi \in LTL_{\Downarrow}$, $A \subseteq \mathcal{A}$, et $a \in A$, nous avons :

$$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P1 \;\; alors \;\; P \overset{a}{\mapsto} P', \; P1 = P' \sqcap_A \partial_a(\varphi) \; et \; \delta(\varphi) = a$$

Démonstration.

$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P1$

$\Rightarrow$ {Définition 6.7.1 de $\sqcap$}>
$\lceil P \rceil_c \; mod \; A \parallel [\![\varphi]\!]_c \overset{a}{\rightarrowtail} P1$

$\Rightarrow$ {D'après la définition 6.9.5 de $\rightarrowtail$ et puisque $[\![\varphi]\!]_c$ ne contient que les actions de contrôle et $a \notin A_{sync}$, nous concluons que l'action a provient de P :}
$P \overset{a}{\mapsto} P'$

$\Rightarrow$ {Proposition 6.10.9}
$\lceil P \rceil_c \; mod \; A \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \xrightarrow{a} \overset{send(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c \; mod \; A$

$\Rightarrow$ {D'après la définition 6.10.3 de $\twoheadrightarrow$, pour que l'action $receive(\ulcorner a, c)$ soit exécutée, $[\![\varphi]\!]_c$ doit exécuter une action $send(\ulcorner a, c)$ sur le même canal avant, et d'après la définition (6.5) de $[\![-]\!]_c$, l'action $send(\ulcorner a, c)$ ne peut s'exécuter que lorsque :}
$\delta(\varphi) = a$

$\Rightarrow$ {Proposition 6.10.10}
$[\![\varphi]\!]_c =< send(\ulcorner a, c),\ receive(a\urcorner, c)\ F_{c,\emptyset}(\partial_a(\varphi)),\ <c \mapsto \text{"}undefined\text{"}>_{\textbf{FieldsObject}} \dagger E >$

$\Rightarrow$ {Proposition 6.10.5 de $\overset{send(\ulcorner a,c)}{\rightsquigarrow}$}
$[\![\varphi]\!]_c \overset{send(\ulcorner a,c)}{\rightsquigarrow} < receive(a\urcorner, c)\ F_{c,\emptyset}(\partial_a(\varphi)),\ <c \mapsto a\urcorner>_{\textbf{FieldsObject}} \dagger E >$ (1)

$\Rightarrow$ {D'après la définition 6.5.4, nous avons : $[\![\partial_a(\varphi)]\!]_c = F_{c,\emptyset}(\partial_a(\varphi)),\ < c \mapsto a\urcorner>_{\textbf{FieldsObject}} \dagger E >$, avec le résultat (1) et les propositions 6.10.5 de $\overset{send(\ulcorner a,c)}{\rightsquigarrow}$ et 6.10.6 de $\overset{receive(\ulcorner a,c)}{\rightsquigarrow}$, nous obtenons : }
$\lceil P \rceil_c\ mod\ A \parallel [\![\varphi]\!]_c \overset{send(\ulcorner a,c)}{\rightsquigarrow} \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \overset{a}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} \overset{receive(a\urcorner,c)}{\rightsquigarrow} \lceil P' \rceil_c\ mod\ A \parallel [\![\partial_a(\varphi)]\!]_c$

$\Rightarrow$ {Définition 6.7.1 de $\sqcap_A$}
$P \sqcap_A \varphi \overset{send(\ulcorner a,c)}{\rightsquigarrow} \overset{receive(\ulcorner a,c)}{\rightsquigarrow} \overset{a}{\longrightarrow} \overset{send(a\urcorner,c)}{\rightsquigarrow} \overset{receive(a\urcorner,c)}{\rightsquigarrow} P' \sqcap_A \partial_a(\varphi)$

$\Rightarrow$ {Définition 6.9.5 de $\rightarrowtail$ avec $\{send(\ulcorner a, c), send(a\urcorner, c), receive(\ulcorner a, c), receive(a\urcorner, c)\} \in A_{sync}$}
$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P' \sqcap_A \partial_a(\varphi)$
Par la suite $P1 = P' \sqcap_A \partial_a(\varphi)$

□

Théorème 6.10.1. *(Correction)*
$\forall P \in \mathcal{P}$, $\forall \varphi \in LTL_{\Downarrow}$, et $A \subseteq \mathcal{A}$, nous avons :

$$P \sqcap_A \varphi \Vdash\!\sim_A \varphi$$

Preuve :
Selon la définition 6.9.8 de $\Vdash\!\sim$: $P \sqcap_A \varphi$ si $\forall \tau \in \mathcal{T}$: $P \sqcap_A \varphi \between \tau$ et $\tau \Vdash_A \varphi$.

La preuve est par induction sur la taille de la trace. Nous supposons qu'elle est vraie pour toute trace de longueur n et nous prouvons qu'elle est vraie pour une trace de longueur $n+1$.

La preuve est triviale pour la trace vide. D'après la définition 6.9.2, nous avons $\epsilon \subset (Prefx([[\varphi]]))$, $\varphi \in LTL_{\Downarrow}$ donc, $\epsilon \Vdash\!\sim_A \varphi$.

Soit la trace τ, tel que $|\tau| = n$. Soit la trace $a.\tau$ de longueur $n+1$, tel que :

$P \sqcap_A \varphi \between a.\tau$

$\Rightarrow$ {Définition 6.9.7 de $\downarrow$ et $\between$. Il existe $P_1 \in \mathcal{P}$ tel que :}
$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P_1$ et $P_1 \between \tau$

$\Rightarrow$ {Proposition 6.10.14, il existe $P1$, tel que :}
$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P_1,\quad P \overset{a}{\mapsto} P',\ P1 = P' \sqcap_A \partial_a(\varphi)\ et\ a \in \delta(\varphi)$

$\Rightarrow$ {$P' \sqcap_A \partial_a(\varphi) \between \tau$ avec $|\tau| = n$, par hypothèse d'induction nous avons $P' \sqcap_A \partial_a(\varphi) \Vdash\!\sim_A \partial_a(\varphi)$. Par la suite :}
$\tau \Vdash_A \partial_a(\varphi)$

$\Rightarrow$ {Proposition 6.10.1 avec $a \in \delta(\varphi)$}
$a.\tau \Vdash_A \varphi$

De ce fait, notre approche satisfait la propriété de correction. ■

Théorème 6.10.2. *(Complétude)*
$\forall P \in \mathcal{P}$, $\forall \varphi \in LTL_{\Downarrow}$ et $A \subseteq \mathcal{A}$, nous avons :

$$P \approx_{\varphi} P \sqcap_A \varphi$$

Preuve :
La preuve est par induction. Supposons qu'elle est vraie jusqu'à i, c.-à-d. $P \approx^{i}_{\varphi} P \sqcap_A \varphi$, pour tout i et prouvons que $P \approx^{i+1}_{\varphi} P \sqcap_A \varphi$ est vrai pour $i+1$.

La preuve est triviale pour $i = 0$ d'après la définition 6.10.2.

1. Supposons que : $P \overset{a}{\rightarrowtail} P'$ et $a \in \delta(\varphi)$

$\Rightarrow$ {Définition 6.9.5 de $\rightarrowtail$ avec $a \notin A_{sync}$}
$P \xrightarrow{a} P'$

$\Rightarrow$ {Proposition 6.10.13}>
$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P' \sqcap_A \partial_a(\varphi)$

$\Rightarrow$ {Par induction}
$P \sqcap_A \varphi \overset{a}{\rightarrowtail} P' \sqcap_A \partial_a(\varphi)$ et $P' \approx^{i}_{\partial_a(\varphi)} P' \sqcap_A \partial_a(\varphi)$ (i)

2. Supposons que $P \sqcap_A \varphi \overset{a}{\rightarrowtail} P1$

$\Rightarrow$ {Proposition 6.10.14}
$P \overset{a}{\mapsto} P'$, $P1 = P' \sqcap_A \partial_a(\varphi)$ *et* $\delta(\varphi) = a$

$\Rightarrow$ {Par induction}
$P \overset{a}{\mapsto} P'$ et $P' \approx^{i}_{\partial_a(\varphi)} P' \sqcap_A \partial_a(\varphi)$ (ii)

$\Rightarrow$ {Les résultats (i) et (ii) avec la définition 6.10.2 de $\approx^{i+1}$ donnent :}
$P \approx^{i+1}_{\varphi} P \sqcap_A \varphi$

De ce fait, notre approche satisfait la propriété de complétude. ■

6.11 Tableau récapitulatif

Table 6.10 représente toutes les fonctions utilisées dans ce chapitre. Chaque fonction est présentée par son nom, un lien vers sa définition formelle et une brève description intuitive.

6.12 Conclusion

Dans ce chapitre, nous avons proposé un cadre formel pour le renforcement de politiques de sécurité au sein des applications Android. L'approche est construite en trois étapes principales basées sur l'exploitation des méthodes formelles. La première étape vise à caractériser formellement la politique de sécurité à l'aide d'une logique LTL. Quant au programme, il a également subi une formalisation dans le cadre de définition formelle de langage, $\mathbb{K}$ *Framework*. La troisième étape consiste à définir une technique permettant de renforcer automatiquement la formule LTL dans le programme. Pour y parvenir, ces deux entrées ont été transformées d'une manière permettant leur synchronisation, une fois exécutés en parallèle. Le résultat est une politique qui pilote le programme comme souhaité.

L'avantage majeur de mécanismes de sécurité proposé est qu'il inclut intrinsèquement dans sa spécification les deux propriétés de complétude et de correction, qui sont fondamentales pour toute technique de réécriture de programmes.

Dans le prochain chapitre, l'automatisation de l'approche proposée en utilisant $\mathbb{K}$ sera notre objectif. Un prototype prend en entrée une spécification en $\mathbb{K}$ d'un programme Android et une formule LTL, applique une séquence de modifications et d'ajustements, et génère une nouvelle version du programme qui atteint les objectifs de la politique.

Fonction	**Définition**	**Description**
$[\![-]\!]$	6.4.1	Retourne les traces acceptées par une formule
$\delta(-)$	6.5.1	Retourne les premières actions acceptées par une formule
$\partial_{-}(-)$	6.5.1	Retourne la partie résiduelle d'une formule après l'acceptation d'une action donnée
$o(-)$	6.5.1	Détermine si la séquence vide appartient au langage accepté par la formule
$\vec{\partial}(-)$	6.5.2	Retourne la fermeture transitive d'une formule et ses dérivées
$\vec{\delta}(-)$	6.5.2	Retourne l'ensemble contenant l'ensemble de premières actions d'une formule ainsi que l'ensemble de premières actions de ses dérivées possibles
$size_{\delta}(-)$	6.5.3	Détermine la taille du vecteur de canaux de synchronisation
$\llbracket - \rrbracket_c$	6.5.4	Transforme une formule LTL en un processus contrôleur dans $\mathcal{P}$
$F_{c,\sigma}(-)$	6.5.4	Transforme une formule LTL en un thread dans Th
$\ulcorner -$	6.5.5	Transforme une instruction $\mathbb{K}$-*Smali* en une chaîne de caractère en ajoutant "-s"
$- \urcorner$	6.5.5	Transforme une instruction $\mathbb{K}$-*Smali* en une chaîne de caractère en ajoutant "-e"
$ToString(-)$	6.5.5	Convertit une instruction $\mathbb{K}$-*Smali* en une chaîne de caractère
$Concat(-)$	6.5.5	Concatène deux chaînes de caractères
$\lceil - \rceil_c \ mod \ -$	6.6.1	Réécrit un processus dans $\mathcal{P}$ en insérant les actions de contrôle
$S_{c,\Gamma}(-,-)$	6.6.1	Réécrit un thread dans Th en fonction des actions de contrôle
$L_{c,\Gamma}(-,-)$	6.6.1	Réécrit un thread dans Th, en particulier les actions étiquettes (labels), en fonction des actions de contrôle
$Prefix(-)$	6.9.1	Retourne le préfixe possible d'une trace ou d'un ensemble de traces
$\lceil - \rceil_A$	6.9.3	Élimine les actions qui ne sont pas dans A d'une trace

TABLE 6.10 – Tableau récapitulatif

Chapitre 7

Sémantique $\mathbb{K}$ pour le renforcement automatique de politiques de sécurité dans des applications Android

7.1 Introduction

La plupart des utilisateurs s'appuient sur les solutions de sécurité adoptées par Android, telles que le système de permission et le *SEAndroid* (Security Enhancements for Android) [25] pour protéger leurs données. Cependant, Android continue de signaler des vulnérabilités liées à ses solutions sans offrir une alternative pour rassurer ses utilisateurs.

Au contraire, lors de l'installation d'une application, l'utilisateur se trouve devant deux choix : soit il accepte toutes les permissions demandées, soit il n'en accepte aucune ; il n'y a pas de juste milieu. Pour prendre cette décision, les utilisateurs (et parfois même les développeurs) ne disposent généralement pas de suffisamment d'informations pour juger si une permission est vraiment nécessaire pour effectuer une certaine tâche, ses véritables implications, les intentions malveillantes (dans le cas d'un logiciel espion) de l'application requérante, et si elle comporte des bogues ou des failles.

Dans le but d'établir une sérénité lors de l'installation d'une application, nous avons proposé dans le chapitre précédent une approche pour le renforcement de la sécurité des applications Android. Nous avons souligné l'efficacité des techniques de réécriture de programmes comme elles découlent de la synergie entre le paradigme orienté aspect et les méthodes formelles. Ces techniques réunissent les avantages de deux : la rigueur et l'absence d'ambiguïté lors de la spécification du système grâce à l'utilisation de méthodes formelles, combinées avec la capacité du paradigme orienté aspect d'ajouter automatiquement l'aspect sécurité à cette spécification.

Dans le présent chapitre, nous approuvons cette idée et l'améliorons en utilisant le cadre d'ingénierie de sémantiques $\mathbb{K}$ *Framework*. En effet, nous étudions l'automatisation de l'ensemble du processus de

renforcement en utilisant $\mathbb{K}$, en gardant toujours le fondement formel.

Notre objectif est d'obtenir automatiquement, à partir d'une politique de sécurité et d'un programme Android, une nouvelle version de ce dernier qui respecte cette politique, tel que tout le code nécessaire sera ajouté automatiquement dans les positions appropriées dans le programme en utilisant les outils offerts par $\mathbb{K}$. À cette fin, nous avons implémenté l'idée dans $\mathbb{K}$, en définissant une syntaxe, une configuration et des règles de réécriture.

7.2 Processus de renforcement

Dans le chapitre précédent, nous avons présenté trois étapes principales pour le renforcement de politiques de sécurité dans les applications Android. Ces étapes consistent à :

1. Caractériser formellement la politique de sécurité en utilisant LTL ;
2. Formaliser une appolication Android. Nous avons présenté $\mathbb{K}$-*Smali*, une sémantique formelle du code de type assembleur généré par la rétro-ingénierie des applications Android, Smali ;
3. Définir d'une technique formelle permettant d'intégrer une formule LTL dans un programme $\mathbb{K}$-*Smali* de façon à le contraindre à la respecter. Pour cette étape, nous avons procédé comme suit :
   - Transformer la formule LTL en un programme $\mathbb{K}$-*Smali* (un moniteur) incluant des actions de contrôle ;
   - Modifier le programme $\mathbb{K}$-*Smali* en insérant la partie complémentaire des actions de synchronisation ajoutées à la formule ;
   - Appliquer la technique de renforcement en exécutant les deux programmes en parallèle pour forcer leur synchronisation.

L'objectif ici est d'automatiser tout ce processus en suivant la même démarche. En d'autres termes, notre but est que toutes les étapes mentionnées ci-dessus se réalisent automatiquement. Nous souhaitons que la transformation de la formule LTL, la réécriture du programme, et enfin sa synchronisation avec la politique s'effectuent toutes par l'environnement $\mathbb{K}$. Pour ce faire, nous définissons une sémantique $\mathbb{K}$, incluant une syntaxe, une configuration et des règles de réécriture. La formule LTL sera également spécifiée à l'aide d'une syntaxe $\mathbb{K}$. En ce qui a trait au langage Smali, nous avons déjà sa sémantique exécutable $\mathbb{K}$-*Smali* définie dans le chapitre 4.

Figure 7.1 résume les principales étapes de l'approche de renforcement de politique de sécurité au sein des applications Android en utilisant $\mathbb{K}$ *Framework*. L'entrée principale est une application Android au format *APK* (*Android Package Kit*). Ce fichier est converti en un format lisible via un outil de rétro-ingénierie. Le résultat est un fichier *Smali* contenant le code de l'application accompagné d'un fichier *AndroidManifest* dans une représentation lisible. La deuxième entrée est une politique de sécurité exprimant un comportement particulier. L'environnement $\mathbb{K}$ prend le programme Smali, le fichier

manifest et la politique de sécurité et génère une sémantique exécutable $\mathbb{K}$-*Smali*, ainsi qu'une logique LTL exprimant la politique de sécurité. $\mathbb{K}$-*Smali* est une formalisation du code Smali de l'application avec toutes les informations nécessaires extraites du fichier *AndroidManifest*. En utilisant le même environnement, une nouvelle version sûre du programme original est générée.

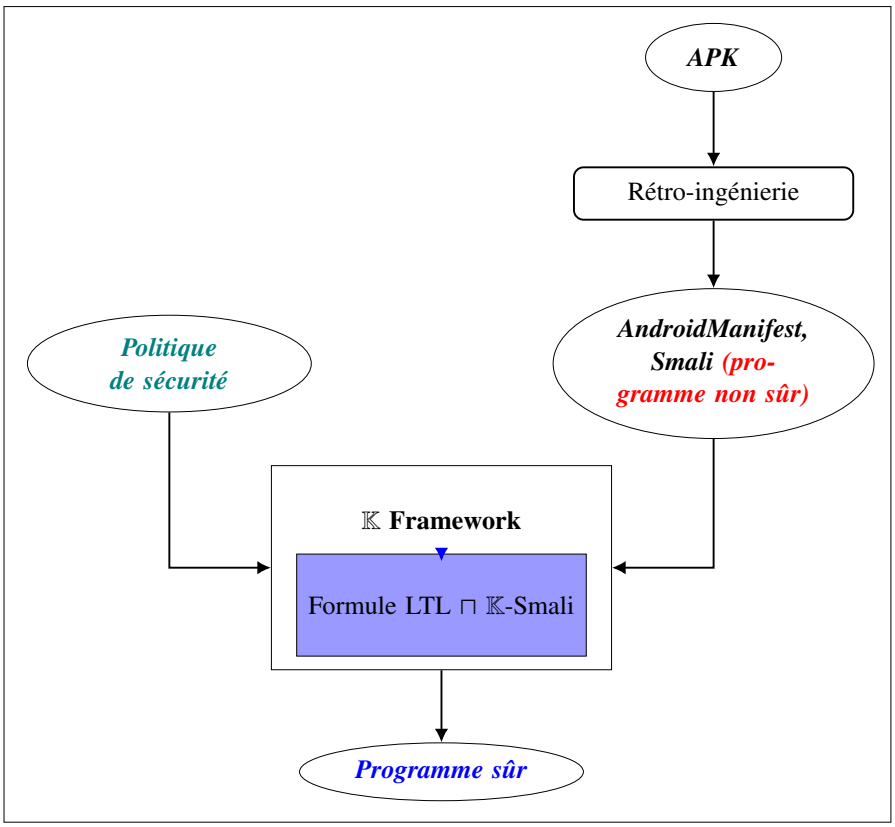


FIGURE 7.1 – Processus de renforcement utilisant $\mathbb{K}$ *Framework*

7.3 Spécification de la formule LTL en $\mathbb{K}$

Listing 7.1 présente la syntaxe $\mathbb{K}$ de la logique LTL. Elle est écrite sous la forme BNF. Une formule LTL peut être les constantes booléennes *True* ou *False*. Elle peut aussi être une action vide 1 ou une action atomique *Action*. Une formule peut être l'union, la composition ou l'intersection de plusieurs formules. L'opérateur de *kleene* (*) est utilisé pour exprimer le comportement itératif. L'attribut "left" annote la définition d'une formule LTL et sert à marquer l'associativité à gauche des opérateurs "+", "." et "$*$". L'attribut "strict" spécifie que les formules LTL doivent être évaluées avant d'évaluer la construction elle-même, en d'autres termes, il est strictement non déterministe dans toutes les formules énumérées. Enfin, l'attribut "bracket" indique qu'une formule LTL peut être placée entre parenthèses, qui sont généralement utilisées pour des raisons de regroupement.

```
module LTLFORMULA-SYNTAX
imports SMALI-SYNTAX
syntax LTL::= "True"
            |"False"
            |"1"
            |Action
            |LTL "." LTL      [left, strict]
            |LTL "+" LTL      [left, strict]
            |LTL "*" LTL      [left, strict]
            |"(" LTL ")"      [bracket]
endmodule
```

7.4 Sémantique $\mathbb{K}$ pour le renforcement de politiques de sécurité dans Smali

Dans cette section, nous montrons comment nous utilisons l'environnement $\mathbb{K}$ pour renforcer une politique de sécurité dans une application Android. Par la suite, nous donnons la fonction en théorie et son équivalent en pratique avec l'utilisation de $\mathbb{K}$ *Framework*.

1. $\lceil P \rceil_c \; mod \; A$ *: Réécrire le programme P en ajoutant des actions de contrôle.*
   Cette étape est implémentée dans $\mathbb{K}$ en ajoutant des appels aux méthodes de synchronisation *send* et *receive* définies dans le chapitre précédent dans leurs positions appropriées.

2. $[\![\varphi]\!]_c$: *Transformer la politique de sécurité φ en un programme $\mathbb{K}$-Smali.*
   Cette étape est mise en œuvre dans $\mathbb{K}$ par la création d'un nouveau thread qui représente la politique de sécurité transformée, c.-à-d. incluant la partie complémentaire des méthodes de synchronisation ajoutées à P.

3. $P \sqcap_A \varphi = (\lceil P \rceil_c \; mod \; A) \parallel [\![\varphi]\!]_c)$: *Exécuter les deux programmes en parallèle pour la synchronisation*. Dans $\mathbb{K}$, cette étape est effectuée en lançant les différents threads du programme, y compris celui qui représente la politique de sécurité.

Pour des raisons de simplification, nous supposons dans le reste de ce chapitre que le canal de synchronisation existe et que les méthodes *send* et *receive* prennent un seul paramètre qui correspond au nom de l'action à contrôler concaténé à la lettre "-s" pour indiquer le début de l'action ou à "-e" qui marque sa fin.

7.4.1 Exemple

Pour mieux clarifier l'idée, nous montrons à travers un exemple comment renforcer une politique de sécurité dans un programme donné en utilisant $\mathbb{K}$. Tout le code nécessaire ajouté pour la politique dans le programme est donné dans cet exemple. Une fois toutes les étapes franchies, nous utilisons

l'interprète fourni par $\mathbb{K}$ pour exécuter le programme résultat, et nous montrons comment ses traces d'exécution sont en harmonie avec la politique de sécurité introduite.

Listing 7.3 représente le programme suspect P et Listing 7.4 représente le programme réécrit P'. P est constitué de trois classes $Lp/c1$;, $Lp/c2$; et $Lp/c3$;. Le *.manifest* (ligne 34) indique le point d'entrée du programme. Il s'agit de la méthode $m1$ de la classe $c3$. La première classe comprend une méthode *start*. Le corps de la méthode consiste à afficher les chaînes de caractères "a" puis "b" en appelant la méthode *println*. La deuxième classe comprend une autre méthode *start* qui invoque la méthode *println* sur les chaînes de caractères "c" puis "d". La classe $Lp/c3$; est une sous-classe de la classe $Lp/c1$;. Deux threads sont instanciés à partir de $Lp/c1$; et $Lp/c2$; (lignes 28 et 30) et lancés (lignes 29 et 31). Une fois lancés et exécutés en parallèle, les chaînes de caractères sont affichées dans l'ordre suivant : "a" "b" "c" d".

Étant donné une politique de sécurité $\varphi = "a.c.b.d"$, qui exige l'affichage des chaînes de caractères dans l'ordre suivant "a" "c" "b" "d" et l'ensemble $A = \{println\}$ (qui est réellement l'instruction *invoke-virtual* de la méthode *println()*) est l'action à contrôler.

Pour faire renforcer φ dans P en utilisant $\mathbb{K}$, nous procédons comme suit :

1. $\lceil P \rceil_c \ mod \ A$
   Modifier le programme P en insérant les actions de synchronisation (*send* et *receive*) dans des points bien déterminés dans le programme. Ces points sont identifiés par l'ensemble d'action à contrôler A, qui correspondent à l'affichage des chaînes constantes "a", "c", "b" et "d" avec l'action *println* dans Listing 7.3). Ainsi, chacune de ces actions dans P sera délimitée par un appel aux méthodes de synchronisation *receive* et *send* à partir de la classe $Lp/PhiChannel$; (voir les définitions 6.7.3 et 6.7.4 de deux méthodes dans le chapitre 6). Cette étape est marquée en bleu dans Listing 7.4. La méthode $receive$ est invoquée avant, permettant de recevoir le nom des actions débuts "a-s", "b-s", "c-s" ou "d-s", tandis que la méthode $send$ est invoquée après pour envoyer les noms des actions fins "a-e", "b-e", "c-e" ou "d-e".

2. $[\![\varphi]\!]_c$
   Transformer la politique de sécurité en un moniteur qui pourrait se synchroniser avec P. La politique est incluse dans une nouvelle classe Lp/Phi; insérée dans le code du programme cible. Cette étape est marquée en vert dans Listing 7.4. La classe comprend la partie complémentaire des actions de synchronisation (*send* suivi d'un *receive*) dans l'ordre requis par la politique ("a", "c", "b", "d").

3. $\lceil P \rceil_c \ mod \ A \parallel [\![\varphi]\!]_c$
   Exécuter le programme modifié et la politique en parallèle. Un nouveau thread instancié de la classe Lp/Phi; (incluant la politique de sécurité) est lancé pour s'exécuter en parallèle avec les threads du programme. Le code présenté en rouge dans Listing 7.4 représente les instructions ajoutées pour exécuter le thread contrôleur en parallèle avec le programme transformé. À la ligne 73, le nouveau thread est instancié à partir de la classe Lp/Phi;, puis lancé à la ligne 74.

Lorsque nous exécutons le programme réécrit P' (Lisitng7.4) en utilisant la commande *Krun* dans $\mathbb{K}$, nous remarquons la synchronisation entre la politique de sécurité incluse dans le thread nouvellement ajouté et le programme réécrit comme prévu. Nous voyons que le nouveau programme respecte la politique appliquée, c.-à-d. l'affichage de "a.c.b.d" comme stipulé par la politique. Le résultat est illustré dans Listing 7.5 à partir des traces générées lors de l'exécution de P'. Pour des raisons de simplicité et pour que le résultat produit soit facile à comprendre, nous avons conservé uniquement les traces affectées par la politique. Les traces présentées aux lignes 9, 18, 27 et 35 où les chaînes "a", "c", "b", "d" sont respectivement mises en évidence et affichées dans l'ordre requis.

7.5 Automatisation de processus de renforcement en utilisant $\mathbb{K}$

Dans l'exemple précédent, nous avons montré comment appliquer en pratique une politique de sécurité dans un programme donné. Le code de cet exemple a été injecté manuellement dans le programme original en suivant les trois étapes mentionnées. Dans ce qui suit, nous étudions l'automatisation de l'ensemble du processus d'application de la politique en utilisant le même environnement $\mathbb{K}$. L'objectif est d'obtenir automatiquement, à partir d'un programme et d'une formule, un nouveau programme qui se comporte conformément à la formule. Ainsi, tout le code nécessaire à l'application de la politique sera ajouté automatiquement aux bons endroits du programme. À cette fin, nous avons implémenté l'idée dans $\mathbb{K}$ en définissant la syntaxe, la configuration et les règles de réécriture.

7.5.1 Syntaxe

Listing 7.2 représente le module ENFORCMEMENT-SYNTAX. Il représente la définition de la syntaxe $\mathbb{K}$ pour le renforcement des formules LTL sur un programme $\mathbb{K}$-*Smali*. Ce module appelle les deux modules SMALI-SYNTAX (Listing 5.4) et LTLFORMULA-SYNTAX (Listing 7.1), importés pour la définition de la syntaxe du programme $\mathbb{K}$-*Smali* et de la formule LTL.

La syntaxe consiste en un programme $\mathbb{K}$-*Smali* "*Programme*" "mod" un ensemble d'actions à contrôler "[*Actions*]", renforcé "⊓" par une formule "LTL".

```
1 module ENFORCEMENT−SYNTAX
2 imports SMALI−SYNTAX
3 imports LTLFORMULA−SYNTAX
4   syntax Actions ::= List "{" Action," ," "}"
5   syntax Action  ::= Instruction
6   syntax LTL     ::= LtlFormula | Action
7   syntax Enforcement ::= Program "mod" "[" Actions "]"  "<"⊓">" LTL
8 endmodule
```

Listing 7.2 – Syntaxe $\mathbb{K}$ du renforcement d'une formule sur un programme $\mathbb{K}$-*Smali*

```
1  .class public Lp/c1;
2  .method public static start()V
3    .locals 2
4    sget-object v0,Ljava/lang/System;->out:Ljava/io/PrintStream;
5    const-string v1, "a"
6    invoke-virtual {v0, v1},Ljava/io/PrintStream;->println(Ljava/lang/
     String;)V
7    const-string v1, "b"
8    invoke-virtual {v0, v1},Ljava/io/PrintStream;->println(Ljava/lang/
     String;)V
9    return-void
10 .end method
11
12 .class public Lp/c2;
13 .method public static start()V
14   .locals 2
15   sget-object v0,Ljava/lang/System;->out:Ljava/io/PrintStream;
16   const-string v1, "c"
17   invoke-virtual {v0, v1},Ljava/io/PrintStream;->println(Ljava/lang
     /String;)V
18   const-string v1, "d"
19   invoke-virtual {v0, v1},Ljava/io/PrintStream;->println(Ljava/lang
     /String;)V
20   return-void
21 .end method
22
23 .class public Lp/c3;
24 .super Lp/c1;
25 .field public x: Ljava/lang/Object;
26 .method public static m1()V
27   .locals 2
28   new-instance v0, Lp/c1;
29   invoke-virtual {v0}, Ljava/lang/Thread;->start()V
30   new-instance v1, Lp/c2;
31   invoke-virtual {v1}, Ljava/lang/Thread;->start()V
32   return-void
33 .end method
34 .manifest Lp/c3;->m1()V
```

Listing 7.3 – Programme P avant renforcement

```
1  .class public Lp/c1;
2  .method public static start()V
3    .locals 2
4    sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
5    const-string v1, "a-s"
6    invoke-static {v1}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
7    const-string v1, "a"
8    invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/
     String;)
9    const-string v1, "a-e"
10   invoke-static {v1}, Lp/PhiChannel;->send(Ljava/lang/String;)V
11   const-string v1, "b-s"
12   invoke-static {v1}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
13   const-string v1, "b"
14   invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/
     String;)
15   const-string v1, "b-e"
16   invoke-static {v1}, Lp/PhiChannel;->send(Ljava/lang/String;)V
17   return-void
18 .end method
19
20 .class public Lp/c2;
21 .method public static start()V
22   .locals 3
23   const-string v1, "c-d"
24   invoke-static {v1}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
25   const-string v1, "c"
26   invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/
     String;)
27   const-string v1, "c-e"
28   invoke-static {v1}, Lp/PhiChannel;->send(Ljava/lang/String;)V
29   const-string v1, "d-s"
30   invoke-static {v1}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
31   const-string v1, "d"
32   invoke-virtual {v0, v1}, Ljava/io/PrintStream;->println(Ljava/lang/
     String;)
33   const-string v1, "d-e"
34   invoke-static {v1},Lp/PhiChannel;->send(Ljava/lang/String;)V
35   return-void
36 .end method
37
38 .class public Lp/Phi;
39 .method public static start()V
40   .locals 2
41   const-string v0, "a-s"
42   invoke-static {v0}, Lp/PhiChannel;->send(Ljava/lang/String;)V
43   const-string v0, "a-e"
44   invoke-static {v0}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
45   const-string v0, "c-s"
46   invoke-static {v0}, Lp/PhiChannel;->send(Ljava/lang/String;)V
47   const-string v0, "c-e"
48   invoke-static {v0}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
49   const-string v0, "b-s"
50   invoke-static {v0}, Lp/PhiChannel;->send(Ljava/lang/String;)V
51   const-string v0, "b-e"
52   invoke-static {v0}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
53   const-string v0, "d-s"
54   invoke-static {v0}, Lp/PhiChannel;->send(Ljava/lang/String;)V
55   const-string v0, "d-e"
56   invoke-static {v0}, Lp/PhiChannel;->receive(Ljava/lang/String;)V
57   return-void
58   .end method
59
60 .class public Lp/c3;
61 .super Lp/c1;
62 .field public x: Ljava/lang/Object;
63 .method public static m1()V
64   .locals 3
65   new-instance v0, Lp/PhiChannel;
66   const-string v1, "undefined"
67   iput v1, v0, Lp/PhiChannel;->a
68   sput-object v0, Lp/c3;->x
69   new-instance v0, Lp/c1;
70   invoke-virtual {v0}, Ljava/lang/Thread;->start()V
71   new-instance v1, Lp/c2;
72   invoke-virtual {v1}, Ljava/lang/Thread;->start()V
73   new-instance v2, Lp/Phi;
74   invoke-virtual {v2},Ljava/lang/Thread;->start()V
75   return-void
76   .end method
77 .manifest Lp/c3;->m1()V
```

Listing 7.4 – Programme P' après renforcement

```
1  <trace>
2   \rconstString { Lp/c3; -> m1 ( ) V } { v1 } { "undefined" }
3   \rconstString { Lp/c1; -> start ( ) V } { v1 } { "a-s" }
4   \rconstString { Lp/c2; -> start ( ) V } { v2 } { "c-s" }
5   \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "a-s" }
6   \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; ) V } { v2 } {"undefined"}
7   \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "a-e" }
8   \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
9   \rconstString { Lp/c1; -> start ( ) V } { v1 } { "a" }
10  \rconstString { Lp/c1; -> start ( ) V } { v1 } { "a-e" }
11  \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; ) V } { v2 } {"undefined"}
12  \rconstString { Lp/c1; -> start ( ) V } { v1 } { "b-s" }
13  \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
14  \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "c-s" }
15  \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; ) V } { v2 } { "undefined"}
16  \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "c-e" }
17  \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
18  \rconstString { Lp/c2; -> start ( ) V } { v2 } { "c" }
19  \rconstString { Lp/c2; -> start ( ) V } { v2 } { "c-e" }
20  \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; ) V } { v2 } {"undefined"}
21  \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
22  \rconstString { Lp/c2; -> start ( ) V } { v2 } { "d-s" }
23  \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "b-s" }
24  \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; ) V } { v2 } {"undefined"}
25  \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
26  \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "b-e"}
27  \rconstString { Lp/c1; -> start ( ) V } { v1 } { "b" }
28  \rconstString { Lp/c1; -> start ( ) V } { v1 } { "b-e" }
29  \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; )V } { v2 } { "undefined"}
30  \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
31  \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "d-s" }
32  \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; ) V } { v2 } {"undefined"}
33  \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
34  \rconstString { Lp/Phi; -> start ( ) V } { v0 } { "d-e" }
35  \rconstString { Lp/c2; -> start ( ) V } { v2 } {"d"}
36  \rconstString { Lp/c2; -> start ( ) V } { v2 } { "d-e" }
37  \rconstString { Lp/PhiChannel; -> send ( Ljava/lang/String; ) V } { v2 } {"undefined"}
38  \rconstString { Lp/PhiChannel; -> receive ( Ljava/lang/String; ) V } { v2 } {"undefined"}
39 </trace>
```

Listing 7.5 – Traces de P' générées par $\mathbb{K}$

7.5.2 Configuration

Figure 7.2 représente la configuration de renforcement d'un programme par une formule. Elle se compose d'une cellule supérieure étiquetée $\top$, contenant deux sous-cellules : la cellule k contient le programme *$ PGM* de type k à renforcer. La seconde cellule étiquetée *EnforcedProgram* contient le programme renforcé. Autrement dit, elle représente le résultat après les modifications apportées aux entrées de renforcement, c.-à-d. le programme, la formule dans la cellule k. Cette configuration peut s'appliquer sur n'importe quel programme ayant une sémantique $\mathbb{K}$ renforcé par une politique de sécurité.

7.5.3 Sémantique

Nous avons défini les règles de réécriture $\mathbb{K}$ nécessaires à la transformation de formules LTL en un moniteur, à la réécriture de programmes en fonction de la politique de sécurité, et à la synchronisation

FIGURE 7.2 – Configuration $\mathbb{K}$ du renforcement d'une formule dans un programme $\mathbb{K}$-*Smali*

entre ceux-ci. Figure 7.3 représente graphiquement quelques règles $\mathbb{K}$ utilisées pour le renforcement.

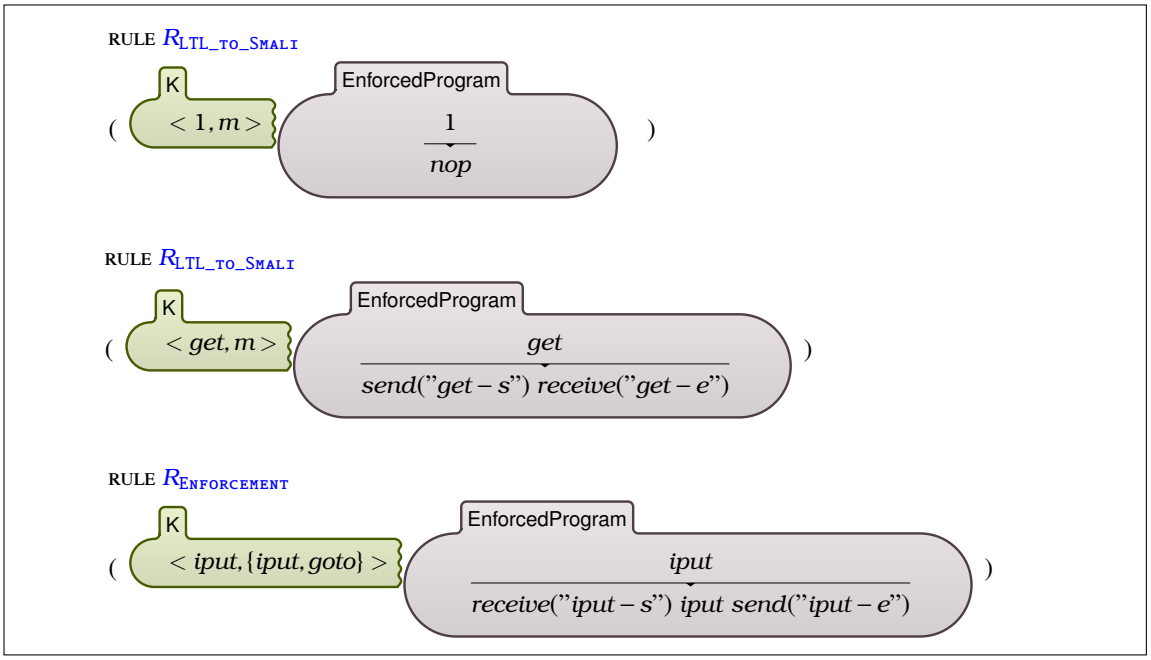


FIGURE 7.3 – Règles de réécriture $\mathbb{K}$ pour le renforcement d'une formule dans un programme $\mathbb{K}$-*Smali*

Les règles $\mathbb{K}$ qui représentent le renforcement de formule LTL visent à réécrire automatiquement le programme introduit en ajoutant les actions de synchronisation appropriées à leurs emplacements correspondants. Ces actions sont contenues dans une classe injectée dans le programme, appelée à chaque fois par son nom complet.

La règle $R_{LTL-to-Smali}$ représente la transformation d'une formule LTL en un programme $\mathbb{K}$-*Smali*. La formule 1 est transformée en son instruction équivalente en $\mathbb{K}$-*Smali*, *nop*. La règle $R_{LTL-to-Smali}$ transforme une action *get* en une nouvelle formule qui peut se synchroniser une fois appliquée dans le programme, en appelant les méthodes *send* et *receive*. La règle $R_{Enforcement}$ réécrit l'instruction en question par une nouvelle séquence d'instructions dans laquelle *receive* et *send* sont ajoutées avant et après l'instruction à contrôler (*iput*).

La transformation des formules incluant un opérateur de disjonction sont traitées de la même manière présentée dans le chapitre précédent (Table 6.7).

Toutes les fonctions nécessaires à la transformation du programme et la formule, ont été définies en $\mathbb{K}$. À titre d'exemple, Listing 7.6 représente la définition de la syntaxe et les règles de la fonction ∂ définies dans le chapitre précédent, appelée "partial". La fonction prend comme paramètres une action et une formule LTL et retourne la dérivée de la formule par rapport à l'action.

```
1 syntax LTL ::=" partial " "(" Action "," LTL ")" [ function ]
2 rule partial (_:Action, False:LTL)=> False
3 rule partial (_:Action, 1_:LTL)=> False
4 rule partial (Act:Action, Act:Action)=> 1_
5 rule partial (Act:Action, Act1:Action)=> False requires Act=/=K Act1
6 rule partial (Act:Action, Phi1:LTL.Phi2:LTL)=>partial(Act, Phi1).Phi2 requires (notBool o(Phi1))
7 rule partial (Act:Action, Phi1:LTL.Phi2:LTL)=>partial(Act, Phi1).Phi2 + partial (Act, Phi2) requires o(Phi1)
8 rule partial (Act:Action, Phi1:LTL+Phi2:LTL)=>partial(Act, Phi1) + partial (Act, Phi2)
9 rule partial (Act:Action, Phi1:LTL*Phi2:LTL)=>partial(Act, Phi1).Phi1*Phi2 + partial (Act, Phi2)
```

Listing 7.6 – Définitions de la fontion partial ∂ en $\mathbb{K}$

De même, les fonctions permettant de retourner les premières actions autorisées par une formule (δ), ou si une formule donnée contient la séquence vide (o) sont toutes définies dans $\mathbb{K}$.

Exemple 7.5.1.
Prenons l'exemple d'une formule comprenant une boucle. Dans le chapitre précédent, nous avons défini sa transformation comme suit :

$$F_{c,\sigma}(\varphi_1^*\varphi_2)=\begin{cases} \textbf{goto} : lab & si\ \{(\varphi_1^*\varphi_2) \mapsto : lab\} \in \sigma \\ : lab & sinon \\ F_{c,\sigma\dagger[(\varphi_1^*\varphi_2)\mapsto:lab]}(\varphi_1.(\varphi_1^*\varphi_2) \vee \varphi_2) & \end{cases} \qquad (: lab \text{ est un label fraîchement généré})$$

Les règles de réécriture $\mathbb{K}$ équivalentes se présentent comme suit :

```
1 rule <k> <Phi1:LTL*Phi2:LTL,Loops:Map (R|->Phi1*Phi2):Map>=>...</k>
2 <renforcedProgram>... .K => String2Id("goto LabStar" +String Int2String (R) +String "~>")
3 </renforcedProgram>
4
5 rule
6 <k><Phi1:LTL*Phi2:LTL,Loops:Map>=><Phi1.(Phi1*Phi2)+Phi2,Loops (!R|->Phi1*Phi2)>...</k>
7 <renforcedProgram>... .K =>String2Id(":LabStar"+String Int2String (!R)) </renforcedProgram>
8 requires notBool(Phi1*Phi2 in (values(Loops)))
```

La première règle représente la transformation de la formule dans le cas où la boucle apparaît pour la première fois. Un nouveau label *LabStar* est créé pointant vers sa transformation. Le "*goto LabStar*" permet de brancher vers ce label. La deuxième règle représente le cas où la boucle est déjà traitée et qu'elle possède déjà un label qui pointe vers sa transformation. Ceci est vérifié avec la condition qui suit le mot clé "requires" qui confirme que la formule comprenant la boucle n'a pas un label associé et par la suite n'a pas été déjà transformée.

Tous les modules sont enregistrés dans des fichiers *.k*. Le fichier principal "*Enforcement*" faisant appel à tous les modules nécessaires est compilé avec la commande *kompile*. Cette commande affichera les

erreurs détectées ou les détails manquants avec les numéros des lignes en question. Si aucune erreur n'est détectée, un nouveau dossier est généré renommé "*Enforcement Kompiled*". Un interprète peut être invoqué avec la commande *krun* pour tester le programme à sécuriser. Dans ce qui suit, nous utilisons l'interprète invoqué pour tester le renforcement d'une formule LTL dans un programme $\mathbb{K}$-*Smali*.

7.6 Exemple

Comme nous l'avons expliqué lors de la définition de la sémantique de renforcement de politique de sécurité, nous avons défini un opérateur d'intersection $\sqcap$ qui prend comme entrées un programme, un ensemble d'actions à contrôler et une formule LTL, et génère un nouveau programme qui satisfait la formule.

---

syntax Enforcement ::= Program "mod" "[" Actions "]" "<" $\sqcap$ ">" LTL

Prenons un exemple pour voir comment le renforcement de la politique dans un programme $\mathbb{K}$-*Smali* a été automatisé. Listing 7.7 présente un exemple de paramètres d'entrée pour le renforcement. Il consiste en un programme $\mathbb{K}$−Smali composé d'une classe $Lp/c2;$. À la ligne 19, l'opérateur modulo "mod" identifie les actions à contrôler. La ligne 22 représente l'opérateur de renforcement "$\sqcap$" suivi de la formule LTL. Il s'agit d'une séquence d'actions, comprenant les opérateurs de composition "." et itératif "*".

Après avoir introduit la formule, le programme et un ensemble d'actions à contrôler, nous utilisons l'interprète $\mathbb{K}$ pour simuler ces entrées. Cette commande génère la configuration illustrée dans Listing 7.8. La cellule $\langle\ \rangle_{EnforcedProgram}$ représente le programme renforcé généré par $\mathbb{K}$. Dans ce programme, nous pouvons constater :

1. La modification du programme original avec l'ajout des actions de contrôle dans les bonnes positions. Cette étape est marquée en bleu.
2. La transformation de la formule en une méthode $\mathbb{K}$-*Smali Phi*(). Elle inclut les actions de synchronisation complémentaires à celles ajoutées au programme. Elle inclut également un nouveau label *LabStar* créé (ligne 25) ainsi que le saut inconditionnel vers ce label avec l'instruction *goto* (ligne 32). Il s'agit d'une nouvelle étiquette générée par les règles sémantiques pour gérer le comportement itératif de la formule *(new-instance.invoke-static.invoke-virtual)**. Cette transformation est marquée en vert.
3. Le renforcement en exécutant le programme et la politique en parallèle. Cette étape est marquée en rouge. Un nouveau thread contenant la formule est instancié de la méthode *Phi*() et démarré.

```
1  .class public Lp/c2;
2  .super Lp/c1;
3  .field public x: I
4  .field public y: C
5  .method public static m1()V
6   .locals 3
7   const v1, 30
8   goto L1
9    ...
10  :L1
11  invoke-static {v0, v1}, Lp/c1;->m2(I C)C
12  sput v0, Lp/c2;->x
13  new-instance v2, Lp/c1;
14  invoke-virtual {v2,v1}, Lp/c1;->m2(I C)C
15  iput v1, v2, Lp/c1;->b
16  return-void
17 .end method
18
19  mod
20  [new-instance, goto, invoke-virtual, invoke-static]
21
22  <⊓>
23
24  (new-instance.invoke-static .invoke-virtual )*
```

Listing 7.7 – Exemple des entrées de renforcement : progamme et formule

```
1  <⊤>
2  <k>•</k>
3  <EnforcedProgram>
4   .locals 3
5   const v1, 30
6   receive("goto-s")
7   goto L1
8    ...
9   :L1
10  send("goto-e")
11  receive("invoke-static-s")
12  invoke-static {v0, v1 .Parameters}, Lp/c1;->m2( I C )C
13  send("invoke-static-e")
14  sput v0,Lp/c2;->x
15  receive("new-instance-s")
16  new-instance v2, Lp/c1;
17  send("new-instance-e")
18  receive("invoke-virtual-s")
19  invoke-virtual {v2,v1 ,.Parameters}, Lp/c1;->m2( I C )C
20  send("invoke-virtual-e")
21  iput v1, v2, Lp/c1; ->b
22  return-void
23  .end method
24  .Comments .method public Phi ( ) V
25   :LabStar
26  send("new-instance-s" )
27  receive("new-instance-e")
28  send("invoke-static-s")
29  receive("invoke-static-e" )
30  send("invoke-virtual-s" )
31  receive("invoke-virtual-e" )
32  goto LabStar
33  .end method
34
35  new-instance v2, Lp/c2;
36  invoke-virtual {v2}, Ljava/lang/Thread;->start()V
37  new-instance v2, Lp/Phi;
38  invoke-virtual {v2}, Ljava/lang/Thread;->start()V
39 </EnforcedProgram>
40 </⊤>
```

Listing 7.8 – Programme renforcé

7.7 Conclusion

Ce chapitre présente l'automatisation du processus complet de renforcement, en utilisant $\mathbb{K}$ *Framewrok*, à partir de la spécification du langage Smali et de la formule, jusqu'à la réécriture du programme. L'environnement $\mathbb{K}$ nous a permis d'établir une approche à la fois formelle et pratique pour le renforcement de politique de sécurité. Nous le considérons comme un prototype de l'approche qui prend la spécification $\mathbb{K}$ de l'application Android et la politique de sécurité, applique une séquence de modifications et d'ajustements, et génère une nouvelle application qui atteint les objectifs de la politique. À partir de la formule introduite et de manière élégante, la sémantique identifie les points pertinents du code dans lesquels les contrôles de sécurité devraient être appliqués. En d'autres termes, $\mathbb{K}$ agit comme un l'environnement de réécriture de programmes. Ce qui distingue cette approche aussi est que le résultat de renforcement peut être vérifié directement grâce à l'interprète offert par $\mathbb{K}$. Il suffit de simuler le programme renforcé pour s'assurer qu'il se comporte conformément à la politique.

Chapitre 8

Comparaison avec des travaux similaires

Dans le présent chapitre, nous reprenons le même tableau comparatif qu'on a exhibé pour comparer les différents travaux de recherche recensés dans la littérature. Nous avons ajouté à la fin (en rouge) notre approche pour le renforcement formel de politique de sécurité dans les applications Android, *RFPA*.

Nous avons constaté que la plupart des approches formelles se contentent de proposer une spécification formelle d'une application Android. Aussi, comme illustré dans Table 8.1, la plupart des sémantiques formelles proposées négligent l'aspect concurrent du langage pour se concentrer à la place sur d'autres aspects, tels que le système de permission, l'aspect événementiel, etc. La plupart des auteurs supposent que l'exécution est séquentielle ou mono-thread. Selon eux, Android est un système complexe, qui est impossible à formaliser dans son entièreté. Une telle hypothèse fait de la sémantique, une formalisation incomplète. En effet, selon l'étude faite par Wognsen et al. [92] sur 1700 applications Android, 88% font appels aux moniteurs à travers les instructions *monitor-enter* et *monitor-exit* et 90,18% contiennent une référence à java/lang/Thread. La sémantique $\mathbb{K}$-*Smali* couvre la plupart des fonctionnalités d'Android et prend en compte la nature concurrentielle du langage. Elle comprend les instructions liées au multithreading telles que la synchronisation, la création et la gestion des threads. La sémantique est assez expressive vu qu'elle formalise le fichier Smali, l'équivalent du fichier *classes.dex*. Ce fichier contient tout le code de l'application.

D'une autre part, les sémantiques dédiées proposées dans la littérature ont été définies sans l'utilisation d'un cadre de définition d'un langage formel. Les règles sémantiques ainsi que la syntaxe ont été conçues sans être validées par un outil automatisé. Par conséquent, commettre des erreurs ou laisser échapper des détails est fort probable. Un environnement, comme celui de $\mathbb{K}$ est doté d'un guide permettant d'obtenir une formalisation plus propre en compilant chaque définition, d'épargner du temps et des efforts en fournissant automatiquement et gratuitement un ensemble d'outils de vérification des propriétés, d'interprétation et d'analyse syntaxique, etc. Ces initiatives manquent de fonctionnalités similaires qui assurent la correction et augmentent la fiabilité de la sémantique générée.

La littérature fait état aussi de plusieurs initiatives utilisant l'assistant de preuve Coq [166] pour définir

des spécifications formelles et pour la vérification de programmes Android. En effet, Coq offre un support mathématique pour construire et vérifier les preuves de correction. Khan et al. [121] proposent un modèle formel pour analyser les flux de données entre les applications Android en utilisant Coq. Betarte et al. [129, 130] ont proposé une spécification formelle du modèle de permission d'Android permettant d'énoncer et de prouver des propriétés de sécurité et d'appliquer des politiques de contrôle d'accès basées sur la permission. En le comparant à Coq, $\mathbb{K}$ *Framework* supporte un interprète qui sert à tester et exécuter des exemples de programmes (sémantique exécutable), un moteur d'exécution symbolique pour le langage, et des analyseurs syntaxiques générés automatiquement à partir de la spécification. Par ailleurs, la définition des programmes avec $\mathbb{K}$ est plus claire et concise avec la notation BNF, par rapport aux définitions inductives purement syntaxiques avec Coq. En résumé, le cadre $\mathbb{K}$ est mieux adapté à la spécification de langages et à la vérification de programmes. Cette tâche est beaucoup plus coûteuse lorsqu'on utilise Coq. Ceci étant dit, Coq reste plus adapté à la modélisation de problèmes mathématiques et à la démonstration de théorèmes.

L'autre partie des travaux constitue une majorité des approches non formelles qui ont proposé une solution pour renforcer des politiques de sécurité dans les applications Android. Indépendamment de la technique de renforcement utilisée, il est impossible de prouver leurs solidités en l'absence d'un fondement formel.

Comme le montre Table 8.1, l'approche proposée à travers cette thèse réunit tous les critères évoqués : un cadre **formel** permettant de **renforcer** des politiques de sécurité **automatiquement** dans une **spécification** formelle **expressive**.

En effet, la sémantique définie est exécutable et générée à l'aide d'un environnement qui a été testé et approuvé dans la définition de plusieurs langages populaires tels que Java [159], PHP [160] et C [161]. Cet environnent nous offre un cadre à la fois efficient et efficace pour la formalisation d'un langage, la vérification des propriétés et nous l'avons utilisé même pour le renforcement de politique de sécurité.

Par ailleurs, notre approche est orientée réécriture, ce qui fait qu'elle hérite tous les avantages liés à cette technique et remplit ses deux critères principaux : la correction et la complétude.

L'aspect automatique est la valeur ajoutée de cette approche. Le code non sécurisé est automatiquement transformé en un code capable de s'auto-surveiller pour éviter toute action à risque. De surcroît, la sémantique permet de modéliser l'appel à n'importe quelle méthode Java. Ce qui fait que notre formalisme peut couvrir plusieurs fonctionnalités telles que la réflexion, le code natif et le multithreading.

Approches	Spécification formelle						Renforcement		
	Non	Sémantique de traduction	Sémantique dédiée	Outils de validation	Concurrence	Expressivité	Analyse statique	Analyse dynamique	Réécriture de programmes
Wognsen et al. [92]			✓			◐			
Payet et al. [132]			✓			◐			
Jeon et al. [101]			✓			◐			
Cousot et al.[91]			✓			◐			
Kim et al. [93]			✓			◐			
Armando et al. [119]			✓			◐			
Gunadi et al. [133]			✓			◐			
Kanade et al.[136]			✓		✓	◐			
Calzavara et al. [138]			✓		✓	●			
Calzavara et al. [139]			✓			◐			
Bagheri et al.[120]		✓				◐			
Khan et al.[121]			✓		✓	◐			
Jinlong et al.[124]			✓			◐			
Chaudhuri [126]			✓			◐			
Qin et al. [123]			✓			◐			
Mariuca et al. [140]			✓	✓	✓	●			
Arzt et al. [97]	✓								
Nisi et al.[108]	✓								
Felt et al. [48]	✓								
Chin et al. [144]	✓								
Chen et al. [99]		✓				◐	✓		
Davis et al. [33]	✓								✓
Schlegel[127]	✓								✓
Cotterell et al. [145]	✓								✓
Backes et al.[?]	✓								✓
Jeon et al. [116]	✓								✓
Xu et al. [109]	✓								✓
Rasthofer et al. [111]	✓							✓	
Wang et al. [115]	✓							✓	
Bauer et al. [146]	✓							✓	
Huang et al. [147]	✓								✓
Nauman et al. [114]	✓							✓	
Betarte et al. [129]			✓	✓		◐			✓
Bai et al. [148]	✓							✓	
Chen et al.[35]			✓			◐			✓
Bagheri et al. [112]		✓				◐		✓	
RFPA			✓	✓	✓	●			✓

TABLE 8.1 – Comparaison avec des travaux connexes

Conclusion

Les téléphones intelligents sont devenus un élément essentiel de notre vie quotidienne. Ils ont changé notre société et notre mode de vie d'une manière drastique, et ce, à jamais. Ces petits appareils qui étaient autrefois essentiellement utilisés pour passer des appels ou envoyer des messages, aujourd'hui, ils dépassent les ordinateurs comme étant le dispositif digital le plus utilisé. Ils gèrent notre agenda, nos e-mails, nos cartes de crédit, nos itinéraires et nos documents professionnels.

Cependant, depuis l'avènement de ces plateformes mobiles, la liste des incidents liés à la sécurité informatique ne cesse de s'allonger. Les applications qui peuvent être offertes par ces appareils intelligents ont non seulement donné d'innombrables nouvelles possibilités attrayantes et à faible coût tant pour les individus que pour les sociétés, mais ont aussi suscité de nombreuses idées chez ceux qui cherchent de l'argent facile, et ceux qui trouvent du plaisir à saccager la vie privé des autres.

Android est un système d'exploitation mobile qui offre des mécanismes proactifs de sécurité (bac à sable applicatif, SELinux, système de permission, etc.) afin de protéger ses utilisateurs. Toutefois, le niveau de sophistication des logiciels malveillants augmente à un rythme alarmant et s'adapte continuellement avec ces mesures de sécurité. Les créateurs de maliciels utilisent généralement les applications comme un moyen pour s'y introduire et mettre les mains sur les données personnelles abondantes qui résident sur ces appareils. Des menaces qui pèsent non seulement sur les données privées des utilisateurs, mais qui peuvent leur causer aussi des pertes financières et qui touchent dans certains cas à l'autonomie de l'appareil.

Les principales contributions de cette thèse peuvent être résumées comme suit :

Comme première contribution, nous avons proposée Smali^{+}, une sémantique opérationnelle dédiée du langage Smali. Les règles sémantiques étaient conçues pour modéliser le comportement séquentiel et parallèle du langage.

Par la suite, durant la thèse, nous avons découvert le $\mathbb{K}$ *Framework* et nous y avons vu beaucoup de bénéfices. L'une de ses particularités est sa capacité à produire une formalisation plus propre. Il est doté d'un paquet d'outils incluant un compilateur permettant de détecter toute possibilité d'erreur ou d'omission, d'un interprète qui permet de tester la sémantique contre une variété des programmes types et d'un vérificateur de programme pour vérifier des propriétés. La formalisation $\mathbb{K}$-*Smali* proposée dans cette thèse est une contribution qui peut faire la base pour des travaux ultérieurs d'analyse,

de vérification et d'optimisation sur le système et les applications Android.

La troisième contribution consiste en l'élaboration d'un cadre formel pour le renforcement automatique de politiques de sécurité dans les applications Android par réécriture. L'idée est de générer automatiquement à partir d'une application potentiellement malicieuse et d'une politique de sécurité, une nouvelle version sécuritaire et en même temps équivalente à la première. Grâce au fondement formel de l'approche, nous avons prouvé que la version générée contrôle toute exécution susceptible de violer la politique de sécurité en question, soit en bloquant cette exécution ou bien en imposant un ordre temporel qui fait en sorte que la politique soit respectée, tout en étant conforme à sa spécification d'origine.

Le même environnement, $\mathbb{K}$ *Framework*, a été utilisé pour renforcer des politiques de sécurité. Dans cette contribution, nous avons défini toute une sémantique $\mathbb{K}$, permettant à partir d'une spécification $\mathbb{K}$ d'un programme Android et d'une formule LTL, de générer un nouveau programme qui respecte la politique. Les tests nécessaires sont insérés par $\mathbb{K}$ aux bonnes positions dans le code afin de le contraindre à respecter la politique. Grâce à l'interprète offert par $\mathbb{K}$, nous avons pu aussi tester si le programme renforcé généré se comporte conformément à la politique.

Cerner un ensemble d'actions à contrôler parmi toutes les actions de l'application est une idée qui a contribué à optimiser l'approche en minimisant autant que possible le nombre de tests insérés et d'éviter d'ajouter du code inutile, qui ne fait qu'alourdir davantage son exécution. Comme perspective, il serait intéressant de mieux affiner cette idée et de penser à donner plus de contrôle à l'utilisateur lors de l'installation d'une application Android sur son dispositif. Nous avons vu que l'utilisateur n'a pas un grand choix en effectuant cette action. Il serait intéressant tout d'abord de lui laisser le choix de sélectionner cet ensemble. Ensuite, de le laisser exprimer ses attentes de l'application en termes de sécurité sous forme d'une politique de sécurité écrite en langage naturel. Cette politique sera transformée par la suite en une formule LTL, et renforcée dans l'application.

Il serait également intéressant si l'utilisateur final choisit un niveau de sécurité ou une valeur risque à ne pas dépasser par l'application lors de son exécution. Nous avons déjà abordé cette idée dans la contribution [13]. Il reste de l'adapter à notre approche. Les entrées dans ce cas sont non seulement une application et une politique de sécurité, mais aussi un seuil de risque. Chaque action sera quantifiée à une valeur de risque selon sa gravité et selon ce qu'elle pèse sur les données privées de l'utilisateur et sur le fonctionnement de l'appareil en général. Le résultat est une application qui satisfait aux exigences de la politique tout en respectant la volonté de l'utilisateur.

Une autre direction prometteuse serait d'explorer la possibilité d'utiliser les automates de Büchi. Il se trouve que toute formule LTL peut être traduite en un automate de Büchi, sous certaines conditions. En ce propos, il existe des algorithmes efficaces qui assurent cette transformation [167, 168]. Les automates de Büchi ont l'avantage d'être utilisés par des algorithmes d'apprentissage qui peuvent aider dans la vérification de modèles [169, 170].

L'utilisation de la logique floue constitue une autre avenue intéressante à considérer. En effet, cette direction s'avère de plus en plus attractive dans la détection des maliciels, vulnérabilités et des failles de sécurité dans les systèmes Andoid [171–173] ainsi que dans les systèmes connexes ayant des enjeux de sécurité similaires [174–176].

Bibliographie

[1] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky, "Appguard - Fine-Grained Policy Enforcement for Untrusted Android Applications," in *Data Privacy Management and Autonomous Spontaneous Security - 8th International Workshop, DPM 2013, and 6th International Workshop, SETOP 2013, Egham, UK, September 12-13, 2013, Revised Selected Papers*, pp. 213–231, 2013.

[2] A. ANNIE, "5 prédictions pour le marché des applications mobiles en 2021." https://www.afjv.com/news/10405_5-predictions-pour-le-marche-des-applications-mobiles-en-2021.htm/. Dernière consultation 03-02-2022.

[3] I. Corporation, "Smartphone Market Share." https://www.idc.com/promo/smartphone-market-share. Dernière consultation 20-01-2021.

[4] M. M. T. R. Q1, "Mcafee Mobile Threat Report." https://www.mcafee.com/content/dam/consumer/en-us/docs/2020-Mobile-Threat-Report.pdf. Dernière consultation 23-11-2021.

[5] S. Gatlan, "Anubis Android Trojan Spotted with Almost Functional Ransomware Module." https://www.bleepingcomputer.com/news/security/anubis-android-trojan-spotted-with-almost-functional-ransomware-module/. Dernière consultation 07-05-2021.

[6] N. Collier, "New Android Trojan malware discovered in Google Play." https://blog.malwarebytes.com/cybercrime/2017/11/new-trojan-malware-discovered-google-play//. Dernière consultation 07-05-2021.

[7] F-Secure, "Trojan :Android/GGTracker.A." https://www.f-secure.com/v-descs/trojan_android_ggtracker.shtml//. Dernière consultation 10-05-2021.

[8] L. Barrett, "Sms-Sending Trojan Targets Android Smartphones." https://www.esecurityplanet.com/trends/article.php/3898041/SMSSending-Trojan-Targets-Android-Smartphones.htm/. Dernière consultation 07-05-2021.

[9] C. Fang, "Fake cryptocurrency mining apps trick victims into watching ads, paying for subscription service." https://www.trendmicro.com/en_us/research/21/h/fake-cryptocurrency-mining-apps-trick-victims-into-watching-ads-.html/. Dernière consultation 01-03-2022.

[10] M. Ziadia, J. Fattahi, M. Mejri, and E. Pricop, "Smali+ : An Operational Semantics for Low-level Code Generated from Reverse Engineering Android Applications," *Information*, vol. 11, no. 3, 2020.

[11] M. Ziadia, M. Mejri, and J. Fattahi, "K-smali : An Executable Semantics for Program Verification of Reversed Android Applications," *The 14th International Symposium on Foundations & Practice of Security. Paris, France, FPS*, pp. 1–17, 2021.

[12] M. Ziadia, M. Mejri, and J. Fattahi, "Formal and Automatic Security Policy Enforcement on Android Applications by Rewriting," in *New Trends in Intelligent Software Methodologies, Tools and Techniques - Proceedings of the 20th International Conference on New Trends in Intelligent Software Methodologies, Tools and Techniques, SoMeT 202, Cancun, Mexico, 21-23 September, 2021* (H. Fujita and H. Pérez-Meana, eds.), vol. 337 of *Frontiers in Artificial Intelligence and Applications*, pp. 85–98, IOS Press, 2021.

[13] M. Ziadia and M. Mejri, "Formal Enforcement of Security Policies on Parallel Systems with Risk Integration," in *Codes, Cryptology, and Information Security - First International Conference, C2SI 2015, Rabat, Morocco, May 26-28, 2015, Proceedings - In Honor of Thierry Berger* (S. E. Hajji, A. Nitaj, C. Carlet, and E. M. Souidi, eds.), vol. 9084 of *Lecture Notes in Computer Science*, pp. 133–148, Springer, 2015.

[14] M. Ziadia, M. Mejri, and J. Fattahi, "K Semantics for Security Policy Enforcement on Android Applications with Practical Cases," in *the 2nd EAI International Conference on Computational Intelligence and Communications November 18-19, 2021 Versailles, France* (E. CICom 2021, ed.), pp. 1–17, EAI CICom 2021, 2021.

[15] J. Kim, I. Kim, C. Min, H. Jun, S. Lee, W. Kim, and Y. I. Eom, "Static dalvik bytecode optimization for android applications," *ETRI Journal*, vol. 37, 09 2015.

[16] L. Davi, A. Dmitrienko, A. Sadeghi, and M. Winandy, "Privilege Escalation Attacks on Android," in *Information Security - 13th International Conference, ISC 2010, Boca Raton, FL, USA, October 25-28, 2010, Revised Selected Papers*, pp. 346–360, 2010.

[17] G. INC, "Security enhancements." https://source.android.com/security/enhancements/index.html. Dernière consultation 08-08-2021.

[18] S. Smalley and R. Craig, "Security enhanced (SE) android : Bringing flexible MAC to android," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, The Internet Society, 2013.

[19] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi, "Seapp : Bringing mandatory access control to android apps," in *Proc. of the 30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, Aug. 2021.

[20] G. Shrivastava, P. Kumar, D. Gupta, and J. J. P. C. Rodrigues, "Privacy issues of android application permissions : A literature review," *Trans. Emerg. Telecommun. Technol.*, vol. 31, no. 12, 2020.

[21] P. Wijesekera, A. Baokar, A. Hosseini, S. Egelman, D. A. Wagner, and K. Beznosov, "Android permissions remystified : A field study on contextual integrity," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA* (J. Jung and T. Holz, eds.), pp. 499–514, USENIX Association, August 12-14, 2015.

[22] G. INC, "Security app-sandbox." https://source.android.com/security/enhancements/index.html. Dernière consultation 16-04-2021.

[23] R. Mayrhofer, J. V. Stoep, C. Brubaker, and N. Kralevich, "The android platform security model," *ACM Trans. Priv. Secur.*, vol. 24, no. 3, pp. 19 :1–19 :35, 2021.

[24] A. Shabtai, Y. Fledel, and Y. Elovici, "Securing android-powered mobile devices using selinux," *IEEE Secur. Priv.*, vol. 8, no. 3, pp. 36–44, 2010.

[25] "SEAndroid & selinux, making devices more secure : A technology primer." https://hsc.com/DesktopModules/DigArticle/Print.aspx?PortalId=0&ModuleId=1215&Article=66.

[26] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, "Xmandroid : A New Android Evolution to Mitigate Privilege Escalation Attacks," in *TR-2011-04*, 2011.

[27] J. Fattahi, *Analyse des Protocoles Cryptographiques par les Fonctions Témoins*. PhD dissertation, Université Laval. Québec. Canada, February 2016.

[28] M. Debbabi and M. Mejri, "Towards the correctness of security protocols," *Electronic Notes in Theoretical Computer Science*, vol. 83, pp. 55–98, 2003. Proceedings of 19th Conference on the Mathematical Foundations of Programming Semantics.

[29] J. Fattahi, M. Mejri, and E. Pricop, *The Theory of Witness-Functions*, pp. 1–19. Cham : Springer International Publishing, 2016.

[30] S. Info, "Le buffer overflow (dépassement de tampon)." https://www.securiteinfo.com/attaques/hacking/buff.shtml. Dernière consultation 19-09-2021.

[31] A. Göransson, *Efficient Android Threading - Asynchronous Processing Techniques for Android Applications*. O'Reilly, 2014.

[32] M. Jerbi, Z. C. Dagdia, S. Bechikh, and L. B. Said, "On the use of artificial malicious patterns for android malware detection," *Computers and Security*, vol. 92, p. 101743, 2020.

[33] B. Davis, B. S, A. Khodaverdian, and H. Chen, "I-arm-droid : A rewriting framework for in-app reference monitors for android applications," in *In Proceedings of the Mobile Security Technologies 2012, MOST '12. IEEE*, 2012.

[34] M. Alhanahnah, Q. Yan, H. Bagheri, H. Zhou, Y. Tsutano, W. Srisa-an, and X. Luo, "Detecting Vulnerable Android Inter-App Communication in Dynamically Loaded Code," in *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*, pp. 550–558, April 2019.

[35] K. Z. Chen, N. M. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. R. Magrino, E. X. Wu, M. Rinard, and D. X. Song, "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*, 2013.

[36] S. K. Sasidharan and C. Thomas, "Prodroid - an android malware detection framework based on profile hidden markov model," *Pervasive Mob. Comput.*, vol. 72, p. 101336, 2021.

[37] P. Weichbroth and L. Lysik, "Mobile security : Threats and best practices," *Mob. Inf. Syst.*, vol. 2020, pp. 8828078 :1–8828078 :15, 2020.

[38] J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones : attacks, implications and opportunities," in *Eleventh Workshop on Mobile Computing Systems and Applications, HotMobile '10, Annapolis, Maryland, USA, February 22-23, 2010*, pp. 49–54, 2010.

[39] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. A. Wagner, "A survey of mobile malware in the wild," in *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA* (X. Jiang, A. Bhattacharya, P. Dasgupta, and W. Enck, eds.), pp. 3–14, ACM, 2011.

[40] G. Kindermans, "Voici le premier cheval de troie par sms pour android !." https://datanews.levif.be/ict/actualite/voici-le-premier-cheval-de-troie-par-sms-pour-android/article-normal-312493.html?cookie_check=1627071528. Dernière consultation 11-01-2021.

[41] E. Mills, "Dog Wars app for Android is Trojanized." https://www.cnet.com/news/dog-wars-app-for-android-is-trojanized/. Dernière consultation 10-05-2021.

[42] M. Scalas, D. Maiorca, F. Mercaldo, C. A. Visaggio, F. Martinelli, and G. Giacinto, "On the effectiveness of system api-related information for android ransomware detection," *Comput. Secur.*, vol. 86, pp. 168–182, 2019.

[43] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, You, Get Off of My Market : Detecting Malicious Apps in Official and Alternative Android Markets," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*, 2012.

[44] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, "Mobile-sandbox : having a deeper look into android applications," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC '13, Coimbra, Portugal, March 18-22, 2013* (S. Y. Shin and J. C. Maldonado, eds.), pp. 1808–1815, ACM, 2013.

[45] R. Fedler, M. Kulicke, and J. Schütte, "Native code execution control for attack mitigation on android," in *SPSM'13, Proceedings of the 2013 ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2013, November 8, 2013, Berlin, Germany* (W. Enck, A. P. Felt, and N. Asokan, eds.), pp. 15–20, ACM, 2013.

[46] A. I. Aysan, F. Sakiz, and S. Sen, "Analysis of dynamic code updating in android with security perspective," *IET Inf. Secur.*, vol. 13, no. 3, pp. 269–277, 2019.

[47] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Krügel, and G. Vigna, "Execute This ! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," in *NDSS*, 2014.

[48] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. A. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011* (Y. Chen, G. Danezis, and V. Shmatikov, eds.), pp. 627–638, ACM, 2011.

[49] J. Fattahi, M. Mejri, M. Ziadia, E. Pricop, and O. Samoud, "Introduction to SinJAR (a New Tool for Reverse Engineering Java Applications) and Tracing Its Malicious Actions Using Hidden Markov Models," in *New Trends in Intelligent Software Methodologies, Tools and Techniques - Proceedings of the 16th International Conference, SoMeT_17, Kitakyushu City, Japan, September 26-28, 2017* (H. Fujita, A. Selamat, and S. Omatu, eds.), vol. 297 of *Frontiers in Artificial Intelligence and Applications*, pp. 441–453, IOS Press, 2017.

[50] J. Fattahi, M. Couture, and M. Mejri, "SinCRY : A Preventive Defense Tool for Detecting Vulnerabilities in Java Applications Integrating Cryptographic Modules," in *New Trends in Intelligent Software Methodologies, Tools and Techniques - Proceedings of the 17th International Conference SoMeT_18, Granada, Spain, 26-28 September 2018* (H. Fujita and E. Herrera-Viedma, eds.), vol. 303 of *Frontiers in Artificial Intelligence and Applications*, pp. 187–200, IOS Press, 2018.

[51] Apktool, "A tool for reverse engineering android apk files." https://ibotpeaches.github.io/Apktool/. Dernière consultation 15-10-2020.

[52] "dex2jar : A tool for converting android's .dex format to java's .class format." https://github.com/pxb1988/dex2jar. Dernière consultation 20-02-2021.

[53] W. Enck, D. Octeau, P. D. McDaniel, and S. Chaudhuri, "A study of android application security," in *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*, USENIX Association, 2011.

[54] J. J. Drake, Z. Lanier, C. Mulliner, P. O. Fora, S. A. Ridley, and G. Wicherski, *Android Hacker's Handbook*. Wiley Publishing, 2014.

[55] L. N. Y. L. C. T. Y. Sun, "Cosc 530 : Semester project dalvik virtual machine." https://docplayer.net/13603048-Cosc-530-semester-project-dalvik-virtual-machine-by-laxman-nathawat-yuping-lu-chunyan-tang-ye-sun.html. Dernière consultation 27-12-2021.

[56] J. Meye, "JASMIN USER GUIDE." http://jasmin.sourceforge.net/guide.html. Dernière consultation 12-08-2020.

[57] K. W. Hamlen, "Security policy enforcement by automated program-rewriting," in *PhD thesis*, University of Texas at Dallas, 2006.

[58] L. Bauer, J. Ligatti, and D. Walker, "More enforceable security policies," 08 2002.

[59] F. B. Schneider, "Enforceable security policies," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 1, pp. 30–50, 2000.

[60] D. Servos and S. L. Osborn, "Current research and open problems in attribute-based access control," *ACM Comput. Surv.*, vol. 49, no. 4, pp. 65 :1–65 :45, 2017.

[61] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières, "Securing distributed systems with information flow control," in *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings* (J. Crowcroft and M. Dahlin, eds.), pp. 293–308, USENIX Association, 2008.

[62] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pp. 11–20, IEEE Computer Society, 1982.

[63] B. Alpern and F. B. Schneider, "Recognizing safety and liveness," *Distributed Comput.*, vol. 2, no. 3, pp. 117–126, 1987.

[64] M. M. Lanjar, "Cadre algébrique pour le renforcement de politique de sécurité sur les systèmes concurents par réécriture automatique de programmes," in *PhD thesis*, Université Laval, 2010.

[65] P. Cousot, "Program Analysis : The Abstract Interpretation Perspective," *SIGPLAN Notices*, vol. 32, no. 1, pp. 73–76, 1997.

[66] G. Barthe, D. Pichardie, and T. Rezk, "A certified lightweight non-interference java bytecode verifier," *Mathematical Structures in Computer Science*, vol. 23, no. 5, pp. 1032–1081, 2013.

[67] I. Konnov, "Edmund m. clarke, thomas a. henzinger, helmut veith, and roderick bloem (eds) : Handbook of model checking - springer international publishing ag, cham, switzerland, 2018," *Formal Asp. Comput.*, vol. 31, no. 4, pp. 455–456, 2019.

[68] G. A. Kildall, "A Unified Approach to Global Program Optimization," in *Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973* (P. C. Fischer and J. D. Ullman, eds.), pp. 194–206, ACM Press, 1973.

[69] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst, "Static Analysis of Implicit Control Flow : Resolving Java Reflection and Android Intents (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pp. 669–679, 2015.

[70] H. Kevin, M. Greg, and B. S. Fred, "Computability classes for enforcement mechanisms," in *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Association for Computing Machinery, 2006.

[71] Ú. Erlingsson and F. B. Schneider, "IRM enforcement of java stack inspection," in *2000 IEEE Symposium on Security and Privacy, Berkeley, California, USA, May 14-17, 2000*, pp. 246–255, IEEE Computer Society, 2000.

[72] Ú. Erlingsson and F. B. Schneider, "SASI enforcement of security policies : a retrospective," in *Proceedings of the 1999 Workshop on New Security Paradigms, Caledon Hills, ON, Canada, September 22-24, 1999* (D. M. Kienzle, M. E. Zurko, S. J. Greenwald, and C. Serbau, eds.), pp. 87–95, ACM, 1999.

[73] J. Ligatti, L. Bauer, and D. Walker, "Edit automata : enforcement mechanisms for run-time security policies," *Int. J. Inf. Sec.*, vol. 4, no. 1-2, pp. 2–16, 2005.

[74] J. Ligatti and S. Reddy, "A theory of runtime enforcement, with results," in *Computer Security - ESORICS 2010, 15th European Symposium on Research in Computer Security, Athens, Greece, September 20-22, 2010. Proceedings* (D. Gritzalis, B. Preneel, and M. Theoharidou, eds.), vol. 6345 of *Lecture Notes in Computer Science*, pp. 87–100, Springer, 2010.

[75] J. Ligatti, L. Bauer, and D. Walker, "Enforcing non-safety security policies with program monitors," in *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings* (S. D. C. di Vimercati, P. F. Syverson, and D. Gollmann, eds.), vol. 3679 of *Lecture Notes in Computer Science*, pp. 355–373, Springer, 2005.

[76] H. Ould-Slimane, M. Mejri, and K. Adi, "Using edit automata for rewriting-based security enforcement," in *Data and Applications Security XXIII, 23rd Annual IFIP WG 11.3 Working Conference, Montreal, Canada, July 12-15, 2009. Proceedings* (E. Gudes and J. Vaidya, eds.), vol. 5645 of *Lecture Notes in Computer Science*, pp. 175–190, Springer, 2009.

[77] C. Talhi, N. Tawbi, and M. Debbabi, "Execution monitoring enforcement for limited-memory systems," in *Proceedings of the 2006 International Conference on Privacy, Security and Trust : Bridge the Gap Between PST Technologies and Business Services, PST 2006, Markham, Ontario, Canada, October 30 - November 1, 2006*, vol. 380 of *ACM International Conference Proceeding Series*, p. 38, ACM, 2006.

[78] P. W. L. Fong, "Access control by tracking shallow execution history," in *2004 IEEE Symposium on Security and Privacy (S&P 2004), 9-12 May 2004, Berkeley, CA, USA*, pp. 43–55, IEEE Computer Society, 2004.

[79] S. Eilenberg, *Automata, languages, and machines., B*. Pure and applied mathematics, Academic Press, 1976.

[80] M. Mejri and H. Fujita, "Enforcing Security Policies Using Algebraic Approach," in *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Seventh SoMeT 2008, October 15-17, 2008, Sharjah, United Arab Emirates* (H. Fujita and I. A. Zualkernan, eds.), vol. 182 of *Frontiers in Artificial Intelligence and Applications*, pp. 84–98, IOS Press, 2008.

[81] M. Langar and M. Mejri, "Optimized enforcement of security policies," in *Foundations of Computer Security 2005*, pp. 37–42, 2005.

[82] G. Sui and M. Mejri, "FASER (formal and automatic security enforcement by rewriting) by BPA algebra with test," *Int. J. Grid Util. Comput.*, vol. 4, no. 2/3, pp. 204–211, 2013.

[83] M. Langar, M. Mejri, and K. Adi, "Formal enforcement of security policies on concurrent systems," *J. Symb. Comput.*, vol. 46, no. 9, pp. 997–1016, 2011.

[84] M. Langar and K. Dahmani, "Formal enforcement of security policies on choreographed services," in *Proceedings of the Formal Methods for Security Workshop co-located with the PetriNets-2014 Conference , Tunis, Tunisia, June 23rd, 2014* (V. Cortier and R. Robbana, eds.), vol. 1158 of *CEUR Workshop Proceedings*, pp. 53–67, CEUR-WS.org, 2014.

[85] K. Adi, L. Hamza, and L. Pene, "Automatic security policy enforcement in computer systems," *Comput. Secur.*, vol. 73, pp. 156–171, 2018.

[86] P. Lantz and B. Johansson, "Towards bridging the gap between Dalvik bytecode and native code during static analysis of Android applications," *2015 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pp. 587–593, 2015.

[87] Z. Lu and S. Mukhopadhyay, "Model-based static source code analysis of java programs with applications to android security," in *36th Annual IEEE Computer Software and Applications Conference, COMPSAC 2012, Izmir, Turkey, July 16-20, 2012*, pp. 322–327, 2012.

[88] Y. Guo, L. Yang, X. Gao, and K. Wu, "The static detection analysis technology of android source codes," in *IEEE International Conference on Network Infrastructure and Digital Content, IC-NIDC 2016, Beijing, China, September 23-25, 2016*, pp. 288–292, 2016.

[89] N. Hoshieah, S. Zein, N. Salleh, and J. Grundy, "A static analysis of android source code for lifecycle development usage patterns," *Journal of Computer Science*, vol. 15, no. 1, pp. 92–107, 2019.

[90] P. Cousot and R. Cousot, "Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977* (R. M. Graham, M. A. Harrison, and R. Sethi, eds.), pp. 238–252, ACM, 1977.

[91] E. Payet and F. Spoto, "Static Analysis of Android Programs," *Inf. Softw. Technol.*, vol. 54, no. 11, pp. 1192–1201, 2012.

[92] E. Wognsen, H. Søndberg Karlsen, M. Chr. Olesen, and R. Hansen, "Formalisation and analysis of Dalvik bytecode," *Science of Computer Programming*, pp. 25–55, 2014.

[93] J. Kim, Y. Yoon, and K. Yi, "Scandal : Static analyzer for detecting privacy leaks in android applications," *In H. Chen, L. Koved, and D. S. Wallach, editors, MoST 2012 : Mobile Security Technologies 2012, Los Alamitos, CA, USA*, 2012.

[94] R. Sato, D. Chiba, and S. Goto, "Detecting Android malware by Analyzing Manifest Files," *Proceedings of the Asia-Pacific Advanced Network http ://dx.doi.org/10.7125/APAN.36.4*, vol. a36, p. 23-31, 2013.

[95] D. Wu, X. Luo, and R. K. C. Chang, "A sink-driven approach to detecting exposed component vulnerabilities in Android apps," *CoRR*, vol. abs/1405.6282, 2014.

[96] A. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid : Automated Security Certification of Android Applications," *Manuscript, Univ. of Maryland, http ://www. cs. umd. edu/avik/projects/scandroidascaa*, vol. 2(3), 2009.

[97] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid : Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *SIGPLAN Not.*, vol. 49, pp. 259–269, jun 2014.

[98] B. Rajesh, P. Reddy, M. Patil, and H. Pareek, "Droidswan :detecting malicious android applications based on static feature analysis," 05 2015.

[99] Z. Xu, H. Chen, A. Tiu, Y. Liu, and K. Sareen, "A permission-dependent type system for secure information flow analysis," *J. Comput. Secur.*, vol. 29, no. 2, pp. 161–228, 2021.

[100] Z. Xu, H. Chen, A. Tiu, Y. Liu, and K. Sareen, "A permission-dependent type system for secure information flow analysis," *J. Comput. Secur.*, vol. 29, no. 2, pp. 161–228, 2021.

[101] J. Jeon, K. K. Micinski, and J. Foster, "Symdroid : Symbolic execution for dalvik bytecode," 2012.

[102] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and Y. L. Traon, "Static Analysis of Android Apps : A Systematic Literature Review," *Inf. Softw. Technol.*, vol. 88, pp. 67–95, 2017.

[103] "Findbugs™ - find bugs in java programs." http://findbugs.sourceforge.net. Dernière consultation 07-06-2021.

[104] V. Sihag, M. Vardhan, and P. Singh, "A survey of android application and malware hardening," *Comput. Sci. Rev.*, vol. 39, p. 100365, 2021.

[105] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid : An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), pp. 393–407, USENIX Association, 2010.

[106] L. Yan and H. Yin, "Droidscope : Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pp. 569–584, 2012.

[107] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid : behavior-based malware detection system for android," in *SPSM'11, Proceedings of the 1st ACM Workshop Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2011, October 17, 2011, Chicago, IL, USA*, pp. 15–26, 2011.

[108] D. Nisi, A. Bianchi, and Y. Fratantonio, "Exploring syscall-based semantics reconstruction of android applications," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2019, Chaoyang District, Beijing, China, September 23-25, 2019*, pp. 517–531, 2019.

[109] R. Xu, H. Saïdi, and R. J. Anderson, "Aurasium : Practical Policy Enforcement for Android Applications," in *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pp. 539–552, 2012.

[110] R. Li, W. Diao, Z. Li, J. Du, and S. Guo, "Android Custom Permissions Demystified : From Privilege Escalation to Design Shortcomings," in *2021 2021 IEEE Symposium on Security and Privacy (SP)*, (Los Alamitos, CA, USA), pp. 70–86, IEEE Computer Society, may 2021.

[111] S. Rasthofer, S. Arzt, E. Lovat, and E. Bodden, "Droidforce : Enforcing Complex, Data-centric, system-wide Policies in Android," in *Ninth International Conference on Availability, Reliability and Security, ARES 2014, Fribourg, Switzerland, September 8-12, 2014*, pp. 40–49, 2014.

[112] H. Bagheri, A. Sadeghi, R. J. Behrouz, and S. Malek, "Practical, Formal Synthesis and Automatic Enforcement of Security policies for android," in *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2016, Toulouse, France, June 28 - July 1, 2016*, pp. 514–525, 2016.

[113] "A guide to alloy." https://www.doc.ic.ac.uk/project/examples/2007/271j/suprema_on_alloy/Web/intro.php. Dernière consultation 07-08-2021.

[114] M. Nauman, S. Khan, and X. Zhang, "Apex : extending Android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2010, Beijing, China, April 13-16, 2010*, pp. 328–332, 2010.

[115] X. Wang, K. Sun, Y. Wang, and J. Jing, "Deepdroid : Dynamically Enforcing Enterprise Policy on Android Devices," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, 2015.

[116] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. D. Millstein, "Dr. Android and Mr. hide : fine-grained permissions in android applications," in *SPSM'12, Proceedings of the Workshop on Security and Privacy in Smartphones and Mobile Devices, Co-located with CCS 2012, October 19, 2012, Raleigh, NC, USA*, pp. 3–14, 2012.

[117] G. Costa, F. Sinigaglia, and R. Carbone, "Polena : Enforcing fine-grained permission policies in android," in *Computer Safety, Reliability, and Security - SAFECOMP 2017 Workshops, ASSURE, DECSoS, SASSUR, TELERISE, and TIPS, Trento, Italy, September 12, 2017, Proceedings* (S. Tonetta, E. Schoitsch, and F. Bitsch, eds.), vol. 10489 of *Lecture Notes in Computer Science*, pp. 407–414, Springer, 2017.

[118] A. Armando, R. Carbone, G. Costa, and A. Merlo, "Android permissions unleashed," in *IEEE 28th Computer Security Foundations Symposium, CSF 2015, Verona, Italy, 13-17 July, 2015* (C. Fournet, M. W. Hicks, and L. Viganò, eds.), pp. 320–333, IEEE Computer Society, 2015.

[119] A. Armando, G. Costa, and A. Merlo, "Formal Modeling and Reasoning about the Android Security Framework," in *Trustworthy Global Computing - 7th International Symposium, TGC 2012, Newcastle upon Tyne, UK, September 7-8, 2012, Revised Selected Papers* (C. Palamidessi and M. D. Ryan, eds.), vol. 8191 of *Lecture Notes in Computer Science*, pp. 64–81, Springer, 2012.

[120] H. Bagheri, E. Kang, S. Malek, and D. Jackson, "Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification," in *FM 2015 : Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, pp. 73–89, 2015.

[121] W. Khan, M. Kamran, A. Ahmad, F. A. Khan, and A. Derhab, "Formal analysis of language-based Android security using theorem proving approach," *IEEE Access*, vol. 7, pp. 16550–16560, 2019.

[122] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, p. 59, 2012.

[123] J. Qin, H. Zhang, S. Wang, Z. Geng, and T. Chen, "Acteve++ : An improved android application automatic tester based on acteve," *IEEE Access*, vol. 7, pp. 31358–31363, 2019.

[124] J. He, T. Chen, P. Wang, Z. Wu, and J. Yan, "Android Multitasking Mechanism : Formal Semantics and Static Analysis of Apps," in *Programming Languages and Systems* (A. W. Lin, ed.), 2019.

[125] J. He, T. Chen, P. Wang, Z. Wu, and J. Yan, "Android Multitasking Mechanism : Formal Semantics and Static Analysis of apps," in *Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings*, pp. 291–312, 2019.

[126] A. Chaudhuri, "Language-based security on Android," in *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security, PLAS 2009, Dublin, Ireland, 15-21 June, 2009*, pp. 1–7, 2009.

[127] M. Schlegel, "Trusted Enforcement of Application-specific Security Policies," *CoRR*, vol. abs/2105.01970, 2021.

[128] S. Talegaon and R. Krishnan, "A Formal Specification of Access Control in Android," in *Secure Knowledge Management In Artificial Intelligence Era - 8th International Conference, SKM 2019, Goa, India, December 21-22, 2019, Proceedings* (S. K. Sahay, N. Goel, V. Patil, and M. Jadliwala, eds.), vol. 1186 of *Communications in Computer and Information Science*, pp. 101–125, Springer, 2019.

[129] G. Betarte, J. Campo, M. Cristiá, F. Gorostiaga, C. Luna, and C. Sanz, "Towards formal model-based analysis and testing of android's security mechanisms," in *2017 XLIII Latin American Computer Conference (CLEI)*, pp. 1–10, 2017.

[130] G. Betarte, J. D. Campo, C. Luna, and A. Romano, "Formal analysis of android's permission-based security model,," *Sci. Ann. Comput. Sci.*, vol. 26, no. 1, pp. 27–68, 2016.

[131] E. Wognsen and S. Karlsen, "Static Analysis of Dalvik Bytecode and Reflection in Android," *Master's thesis, Department of Computer Science, Aalborg University, Aalborg, Denmark*, 2012.

[132] É. Payet and F. Spoto, "An operational semantics for android activities," in *Proceedings of the ACM SIGPLAN 2014 workshop on Partial evaluation and program manipulation, PEPM 2014, January 20-21, 2014, San Diego, California, USA* (W. Chin and J. Hage, eds.), pp. 121–132, ACM, 2014.

[133] H. Gunadi, "Formal Certification of Non-interferent Android Bytecode (DEX Bytecode)," in *20th International Conference on Engineering of Complex Computer Systems, ICECCS 2015, Gold Coast, Australia, December 9-12, 2015*, pp. 202–205, 2015.

[134] M. Bartoletti, P. Degano, G. L. Ferrari, and R. Zunino, "Local policies for resource usage analysis," *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 6, pp. 23 :1–23 :43, 2009.

[135] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for Android applications," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pp. 316–325, 2014.

[136] A. Kanade, "Chapter seven - Event-Based Concurrency : Applications, Abstractions, and Analyses," *Advances in Computers*, vol. 112, pp. 379–412, 2019.

[137] A. Bouajjani, M. Emmi, C. Enea, B. K. Ozkan, and S. Tasiran, "Verifying Robustness of Event-Driven Asynchronous Programs Against Concurrency," in *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pp. 170–200, 2017.

[138] S. Calzavara, I. Grishchenko, A. Koutsos, and M. Maffei, "A sound flow-sensitive heap abstraction for the static analysis of android applications," in *30th IEEE Computer Security Foundations Symposium, CSF 2017, Santa Barbara, CA, USA, August 21-25, 2017*, pp. 22–36, IEEE Computer Society, 2017.

[139] S. Calzavara, I. Grishchenko, and M. Maffei, "Horndroid : Practical and sound static analysis of android applications by SMT solving," in *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, pp. 47–62, IEEE, 2016.

[140] A. Mariuca, J. Blasco, T. Chen, H. Kalutarage, I. Muttik, H. Nguyen, M. Roggenbach, and S. Shaikh, *Detecting Malicious Collusion Between Mobile Software Applications : The Android TM Case*, pp. 55–97. Springer International Publishing, Aug. 2017.

[141] R. Casolare, F. Martinelli, F. Mercaldo, V. Nardone, and A. Santone, "Colluding android apps detection via model checking," in *Web, Artificial Intelligence and Network Applications - Proceedings of the Workshops of the 34th International Conference on Advanced Information*

Networking and Applications, AINA Workshops 2020, Caserta, Italy, 15-17 April (L. Barolli, F. Amato, F. Moscato, T. Enokido, and M. Takizawa, eds.), vol. 1150 of *Advances in Intelligent Systems and Computing*, pp. 776–786, Springer, 2020.

[142] R. Casolare, F. Martinelli, F. Mercaldo, and A. Santone, "Android Collusion : Detecting Malicious Applications Inter-communication through SharedPreferences," *Inf.*, vol. 11, no. 6, p. 304, 2020.

[143] T. Serbanuta and G. Rosu, "K-maude : A Rewriting Based Tool for Semantics of Programming Languages," in *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers* (P. C. Ölveczky, ed.), vol. 6381 of *Lecture Notes in Computer Science*, pp. 104–122, Springer, 2010.

[144] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, (New York, NY, USA), pp. 239–252, 2011.

[145] K. Cotterell, I. Welch, and A. Chen, "An Android Security Policy Enforcement tool," *International Journal of Electronics and Telecommunications*, vol. vol. 61, no. No 4, 2015.

[146] A. Bauer, J. Küster, and G. Vegliach, "Runtime Verification Meets Android Security," in *NASA Formal Methods - 4th International Symposium, NFM 2012, Norfolk, VA, USA, April 3-5, 2012. Proceedings* (A. Goodloe and S. Person, eds.), vol. 7226 of *Lecture Notes in Computer Science*, pp. 174–180, Springer, 2012.

[147] C. Huang, S. Wang, H. Sun, and Z. Qi, "Defdroid : Securing Android with Fine-Grained Security Policy," in *The 27th International Conference on Software Engineering and Knowledge Engineering, SEKE 2015, Wyndham Pittsburgh University Center, Pittsburgh, PA, USA, July 6-8, 2015*, pp. 375–378, 2015.

[148] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen, "Context-Aware Usage Control for Android," in *Security and Privacy in Communication Networks - 6th Iternational ICST Conference, SecureComm 2010, Singapore, September 7-9, 2010. Proceedings* (S. Jajodia and J. Zhou, eds.), vol. 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 326–343, Springer, 2010.

[149] "Dalvik bytecode." https://source.android.com/devices/tech/dalvik/dalvik-bytecode. Dernière consultation 03-01-2022.

[150] "Java Platform Standard Edition 8 Documentation : Class Thread." https://docs.oracle.com/javase/8/docs/.

[151] "Java Platform Standard Edition 8 Documentation : Class Object." https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html/. Dernière consultation 08-01-2021.

[152] A. Stefanescu, D. Park, S. Yuwen, Y. Li, and G. Rosu, "Semantics-based program verifiers for all languages," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands,* (E. Visser and Y. Smaragdakis, eds.), pp. 74–91, ACM, October 30 - November 4, 2016.

[153] G. Rosu and T. Serbanuta, "An overview of the K semantic framework," *J. Log. Algebraic Methods Program.*, vol. 79, no. 6, pp. 397–434, 2010.

[154] G. Rosu and X. Chen, "Matching logic : the foundation of the K framework (invited talk)," in *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020* (J. Blanchette and C. Hritcu, eds.), p. 1, ACM, 2020.

[155] G. Rosu and A. Stefanescu, "Checking reachability using matching logic," in *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA* (G. T. Leavens and M. B. Dwyer, eds.), pp. 555–574, ACM, October 21-25, 2012.

[156] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, vol. 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[157] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[158] A. Cimatti, M. Roveri, and D. Sheridan, "Bounded Verification of Past LTL," in *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings* (A. J. Hu and A. K. Martin, eds.), vol. 3312 of *Lecture Notes in Computer Science*, pp. 245–259, Springer, 2004.

[159] D. Bogdanas and G. Rosu, "K-java : A Complete Semantics of Java," in *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* (S. K. Rajamani and D. Walker, eds.), pp. 445–456, ACM, 2015.

[160] D. Filaretti and S. Maffeis, "An Executable Formal Semantics of PHP," in *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings* (R. E. Jones, ed.), vol. 8586 of *Lecture Notes in Computer Science*, pp. 567–592, Springer, 2014.

[161] C. Hathhorn, C. Ellison, and G. Rosu, "Defining the undefinedness of C," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (D. Grove and S. Blackburn, eds.), pp. 336–345, ACM, 2015.

[162] D. Park, A. Stefanescu, and G. Rosu, "KJS : a complete formal semantics of JavaScript," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015* (D. Grove and S. M. Blackburn, eds.), pp. 346–356, ACM, 2015.

[163] H. Masuhara, S. Chiba, and N. Ubayashi, eds., *Aspect-Oriented Software Development, AOSD*, ACM '13, Fukuoka, Japan, March 24-29, 2013, 2013.

[164] J. P. S. E. 7, "Interface blockingQueue<e>." https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html. Dernière consultation 10-02-2022.

[165] J. A. Bergstra, I. Bethke, and A. Ponse, "Process algebra with iteration and nesting," *Comput. J.*, vol. 37, no. 4, pp. 243–258, 1994.

[166] Y. Bertot and P. Castéran, *Interactive Theorem Proving and Program Development - Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.

[167] S. Mochizuki, M. Shimakawa, S. Hagihara, and N. Yonezaki, "Fast translation from LTL to Büchi automata via non-transition-based automata," in *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings* (S. Merz and J. Pang, eds.), vol. 8829 of *Lecture Notes in Computer Science*, pp. 364–379, Springer, 2014.

[168] Y. Tsay and M. Y. Vardi, "From linear temporal logics to Büchi automata : The early and simple principle," in *Model Checking, Synthesis, and Learning - Essays Dedicated to Bengt Jonsson on The Occasion of His 60th Birthday* (E. Olderog, B. Steffen, and W. Yi, eds.), vol. 13030 of *Lecture Notes in Computer Science*, pp. 8–40, Springer, 2021.

[169] Y. Li, Y. Chen, L. Zhang, and D. Liu, "A novel learning algorithm for Büchi automata based on family of dfas and classification trees," *Inf. Comput.*, vol. 281, p. 104678, 2021.

[170] M. Cai, S. Xiao, and Z. Kan, "Reinforcement learning based temporal logic control with soft constraints using limit-deterministic generalized Büchi automata," *CoRR*, vol. abs/2101.10284, 2021.

[171] J. M. Arif, M. F. A. Razak, S. R. T. Mat, S. Awang, N. S. N. Ismail, and A. Firdaus, "Android mobile malware detection using fuzzy AHP," *J. Inf. Secur. Appl.*, vol. 61, p. 102929, 2021.

[172] X. Pei, L. Yu, S. Tian, H. Wang, and Y. Peng, "Combining multi-features with a neural joint model for android malware detection," *J. Intell. Fuzzy Syst.*, vol. 38, no. 2, pp. 2151–2163, 2020.

[173] A. Altaher, "An improved android malware detection scheme based on an evolving hybrid neuro-fuzzy classifier (EHNFC) and permission-based features," *Neural Comput. Appl.*, vol. 28, no. 12, pp. 4147–4157, 2017.

[174] E. Pricop, S. F. Mihalache, N. Paraschiv, J. Fattahi, and F. Zamfir, "Considerations regarding security issues impact on systems availability," in *2016 8th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, pp. 1–6, 2016.

[175] N. Berjab, H. H. Le, and H. Yokota, "Recovering missing data via top-k repeated patterns for fuzzy-based abnormal node detection in sensor networks," *IEEE Access*, vol. 10, pp. 61046–61064, 2022.

[176] G. Mariappan, A. R. Satish, P. V. B. Reddy, and B. Maram, "Adaptive partitioning-based copy-move image forgery detection using optimal enabled deep neuro-fuzzy network," *Comput. Intell.*, vol. 38, no. 2, pp. 586–609, 2022.