Conception, implémentation et preuve d'un service de transfert de flôt d'exécution au sein d'un noyau de système d'exploitation

Florian Vanhems

Université de Lille

CRIStAL - UMR 9189 ED MADIS - 631

Thèse

présentée et soutenue publiquement le
2 Mars 2023

pour l'obtention du grade de
Docteur de l'Université de Lille
spécialité Informatique

par
Florian Vanhems

Conception, implémentation et preuve d'un service de transfert de flot d'exécution au sein d'un noyau de système d'exploitation

---

Composition du jury :

Julia Lawall	**Rapportrice**	*Directrice de recherche, Inria Paris*
Emmanuel Baccelli	**Rapporteur**	*Professeur des universités, Freie Universität Berlin*
Lionel Rieg	**Examinateur**	*Maître de conférence, Université Grenobles Alpes*
Sylvain Salvati	**Examinateur et Président du jury**	*Professeur des universités, Université de Lille*
Gilles Grimaud	**Directeur de thèse**	*Professeur des universités, Université de Lille*
David Nowak	**Co-Encadrant**	*Chargé de recherche, CNRS*

Abstract

Les travaux présentés dans ce document de thèse sont liés à la vérification formelle de propriétés sur des composants de systèmes d'exploitation. Les premiers travaux piliers de ce domaine sont ceux du projet seL4 ; démontrant que la vérification de propriétés formelles sur un micro noyau est réalisable, malgré un coût élevé. Pour réduire le coût de la preuve, le projet CertikOS a proposé une méthode de preuve plus étagée et plus modulaire, en tirant à l'extrême la méthode de preuve par raffinement. L'équipe du noyau Pip a pris le contrepied de ces travaux, en prônant le minimalisme, en utilisant une méthodologie reposant sur un *shallow embedding* et en prouvant les propriétés désirées directement plutôt qu'en utilisant la méthode par raffinement.

Les travaux présentés dans cette thèse sont plus spécifiquement liés au noyau Pip. Les travaux précédents sur le noyau Pip ont porté sur une preuve de préservation de l'isolation des services fournis par Pip manipulant la mémoire. Cependant, un aspect critique du noyau devait encore être conçu : le transfert de flot d'exécution d'une partition de mémoire à une autre.

La première contribution de cette thèse présente un nouveau service de Pip conçu pour supporter tous les transferts de flots d'exécution possibles au sein d'un système – les interruptions, les fautes, et les appels explicites. Ce service gère de manière unifiée ces transferts de flot d'exécution afin de réduire au minimum l'effort de preuve. Une implémentation est proposée pour le noyau Pip.

La seconde contribution de cette thèse est la première implémentation au monde d'un ordonnanceur *Earliest Deadline First* pour jobs arbitraires muni d'une preuve formelle de sa correction. La preuve garantit que la fonction d'élection respecte la politique *EDF*, garantissant l'optimalité du planning sur les machines mono-processeur. La preuve a été conduite en partie en suivant la méthodologie habituelle de Pip, utilisant un *shallow embedding* et une monade d'état. Elle a cependant été réalisée par raffinement. De plus, l'ordonnanceur se sert du service de transfert de flot d'exécution ; montrant la polyvalence et l'utilisabilité du service.

La dernière contribution présentée dans cette thèse est une preuve de concept libérant le code des services de Pip de ses liens avec le modèle d'isolation. Cette indépendance permet de créer des modèles alternatifs, permettant de raisonner sur le code à propos de nouvelles propriété tout en limitant l'effort de preuve. Cette contribution ouvre de nouvelles perspectives de recherche liées à la réduction du coup de raisonnement sur des propriétés additionnelles sur Pip. Cette preuve de concept n'apporte cependant pas que des avantages : en particulier sur la confiance accordée à la conjonction de propriétés formellement prouvées sur des modèles différents.

Abstract

The work described in this document is related to formal proofs on operating systems.

The first breakthrough in the domain was the SeL4 project ; demonstrating that producing a complete proof on a microkernel was achievable, albeit very costly. In order to bring the proof's cost down, the CertikOS project showcased a more layered and modular approach, leveraging *refinements*. The Pip kernel team tackled the problem from the opposite side, focusing on minimalism, using a *shallow embedding* methodology and getting rid of refinement altogether. This thesis' contributions are more specifically tied to the Pip kernel.

Previous work on the Pip protokernel focused on providing an isolation proof to Pip's services manipulating the system's memory. Yet, another critical aspect of the kernel – handling the execution flow transfer from a partition to another – remained to be designed.

The first contribution of this thesis outlines the design of a single service able to handle all possible control flow transfers in a system ; namely interrupts, faults and explicit control flow transfers. The design focuses on minimalism and code factorization in order to reduce the overall proof effort. An implementation of the service is provided for the Pip kernel. We believe the idea behind the service is general enough to be implemented in other kernels and other architectures.

The second contribution outlined in this thesis is the first formally proven correct userland implementation of an Earliest Deadline First scheduler for arbitrary jobs. The formal proof guarantees that the scheduler's election function respects the Earliest Deadline First scheduling policy, and is guaranteed to be optimal on mono-processor systems. This proof was partly conducted using Pip's usual methodology, leveraging a shallow embedding of the scheduler's code in Coq and a state monad. Nonetheless, while the Pip kernel properties were proven directly, the presented scheduler proofs include three refinement levels ; from the scheduling policy to the actual implementation. Furthermore, the scheduler uses the previously described service in order to pass the control flow to partitions and regain the execution flow through interrupts, showcasing its usability and versatility.

The last contribution presented in this thesis is a proof of concept severing Pip's isolation model from its code. This isolation model independance allows to build alternative models designed to reason on new properties while limiting the proof effort. As such, this contribution opens new research perspectives that were previously too costly to consider. Nonetheless, this proof of concept does not bring the same level of confidence on the composition of properties about the code formally proven on different models.

Remerciements

Ce manuscrit est pour moi la synthèse des mes années d'apprentissage et de recherche. Il n'existe cependant qu'au travers du soutien que j'ai reçu durant ces années, et je souhaite – avant que vous ne commenciez à lire ce document – rendre un hommage tout particulier aux personnes sans qui ce document n'aurait pas abouti.

Je voudrais commencer par remercier mes collègues. À toutes les personnes avec qui j'ai pu travailler, merci pour votre sérieux, votre patience et votre bienveillance. Merci pour ces moments passés à travailler avec vous ; mais aussi pour ces moments de vie commune. Si j'ai pu aller jusqu'au bout de cette thèse, c'est aussi parce qu'il était agréable de venir travailler, même lorsque la fatigue, la lassitude et la morosité auraient dû venir à bout de ma motivation. Par ailleurs, j'ai tant progressé grâce à vous. Gilles, David, Samuel, Vlad, Étienne, Narjes, Damien, Nicolas, Thomas, Alexandre, Pierre, Olivier, Michael, Clément, Giuseppe, merci pour votre aide. Merci de m'avoir expliqué vos travaux. Merci d'avoir partagé votre expérience lors de ces longues discussions. Merci de m'avoir inculqué le doute scientifique. Je me sens privilégié d'avoir pu travailler et d'avoir pu apprendre avec vous.

Je voudrais à présent remercier ma famille, qui m'a apporté un soutien inconditionnel lors de mes études. Si la période de la pandémie a été difficile à vivre, elle aurait été sans nul doute un calvaire sans vous. Merci pour votre écoute. Merci pour vos encouragements. Je voudrais remercier plus particulièrement mes parents et mes grands parents, qui ont toujours fait de mon bien être et de ma réussite leur priorité. Si j'ai pu prétendre à cette thèse et la mener à bien, c'est aussi grâce à vous. Merci du fond du cœur.

Je voudrais ensuite remercier mes amis. Merci pour tous ces moments d'allégresse passés en votre compagnie. J'ai conscience que ces dernières années d'étude et de travail n'auraient pas été les mêmes sans vous. Merci d'avoir été présents pour moi.

Enfin je voudrais remercier ma compagne, Juliette, qui m'a accompagnée au quotidien durant cette période. Merci de m'avoir incité à aller de l'avant. Merci de m'avoir insufflé l'envie de continuer lorsque la motivation me faisait défaut. Merci d'avoir été conciliante, au détriment parfois de tes propres envies, pour que je puisse arriver là où je suis aujourd'hui. Si j'écris ces lignes aujourd'hui, à quelques semaines de ma soutenance, c'est aussi grâce à toi. J'espère pouvoir te rendre à mon tour l'inspiration que tu m'as apportée.

Table des matières

Première partie

Corps du document

1 Introduction

1.1 Contexte

1.1.1 Humain

Cette thèse a été menée à l'Université de Lille, en collaboration avec le *Centre de Recherche en Informatique, Signal et Automatique de Lille* (communement abrégé en laboratoire CRIStAL). Cette thèse a été financée par une dotation de l'Université de Lille.

Cette thèse a été dirigée par Gilles Grimaud, directeur de l'équipe « *eXtra Small, eXtra Safe* » (abrégé 2XS) du CRIStAL. L'équipe se spécialise dans la conception de logiciels et matériels apportant sécurité, fiabilité et efficacité aux systèmes embarqués fortement contraints. Les travaux menés dans l'équipe portent sur la conception d'un noyau de système d'exploitation munis de preuves formelles de propriétés d'isolation de la mémoire, sur les moyens d'attaque physique sur du logiciel (Bluetooth, LoRa, analyse de la consommation, ...), sur la détection de malware et obfuscation d'applications Android, mais aussi sur des objets mathématiques plus théoriques comme par exemple les fonctions corécursives et leur représentation dans un assistant de preuve.

2XS a des relations privilégiées avec d'autres équipes du laboratoire, notamment celles faisant partie du même groupe thématique « *Systèmes embarqués adaptables et sécurisés* ». Cette thèse a notamment tiré profit d'une forte proximité avec l'équipe *SyCoMoRES*, dont les travaux portent sur la conception et l'analyse des systèmes embarqués temps réel, basé sur l'analyse symbolique de composants paramétriques. La seconde contribution de cette thèse est le fruit de cette collaboration.

Par ailleurs, l'équipe *2XS* est hébergée à l'*Institut de Recherche sur les Composants logiciels et matériels pour l'Information et la Communication Avancée* (abrégé IRCICA). L'IRCICA est un établissement conçu pour favoriser la recherche interdisciplinaire, ce qui a notamment permis à l'équipe de saisir de nombreuses opportunités de collaboration avec l'*Institut d'Électronique, de Microélectronique et de Nanotechnologies* (abrégé laboratoire IEMN), et plus particulièrement avec le groupe de recherche *CSAM* notamment sur les travaux relatifs à l'attaque de logiciel au travers de moyens physiques.

Les travaux présentés dans cette thèse sont liés au noyau de système d'exploitation nommé Pip développé dans l'équipe *2XS*.

1.1.2 Pip

Pip est un noyau de système d'exploitation *minimal* dont le seul but est de garantir l'isolation d'applications s'exécutant sur le système. Pour ce faire, Pip est muni de preuves

formelles que ses services préservent les propriétés d'isolation lors de leur exécution. Pip utilise la mémoire virtuelle comme moyen de garantir ces propriétés.

Le projet Pip a démarré avec trois thèses fondatrices :

— La thèse de Narjes Jomaa, soutenue en décembre 2018, a porté sur l'aspect formel du noyau. Narjes a développé une méthodologie permettant de raisonner sur le code des services de Pip, ainsi qu'une méthodologie de co-design du code des services avec les preuves formelles afin d'alléger l'effort de preuve global. Narjes est à l'origine des preuves de préservation de l'isolation fournies par Pip ;

— La thèse de Quentin Bergougnoux, soutenue en juin 2019, a porté sur l'implémentation du noyau sur l'architecture Intel x86, en particulier sur le code des services actuellement présents dans le noyau. Ses travaux ont aussi porté sur des preuves de concept explorant les possibilités de portage de Pip sur un environnement multi-cœur ;

— La thèse de Mahiedinne Yaker, soutenue en décembre 2019, a porté sur l'implémentation de Pip sur une plateforme embarquée basée sur l'architecture Intel, offrant des perspectives de travail sur les systèmes embarqués. Ces travaux ont aussi portés sur des réflexions autour de la conception de systèmes où les entités y demeurant ne se font pas mutuellement confiance.

De ces travaux fondateurs ont émergé de nouvelles opportunités de recherche, dont certains se sont transformés en sujets de thèse. Trois nouvelles thèses ont été pourvues, portant sur des sujets étendants les travaux fondateurs :

— La thèse de Nicolas Dejon, soutenue en décembre 2022, qui porte sur l'application des propriétés d'isolation de Pip aux systèmes dépourvus de mémoire virtuelle, mais pouvant restreindre l'accès à certaines portions de mémoire grâce à une *MPU*. Ces caractéristiques sont courantes sur des systèmes beaucoup plus modestes, et se prêtent particulièrement bien à de l'*IoT* ;

— Les travaux initiaux de Sofia Santiago Fernandez qui portent sur la preuve de préservation de la sémantique du code des services lors de la compilation du code Gallina *shallow-embedded* vers du code C ;

— Mes propres travaux de thèse, présentés dans ce document, portant sur la formalisation du transfert de flôt d'exécution au sein du noyau et de travaux préliminaires relatifs à l'ajout de nouvelles propriétés non relatives à l'isolation.

Les doctorants n'ont pas été les seules personnes recrutées pour participer au développement de Pip : c'est par exemple le cas de Damien Amara, recruté en tant qu'ingénieur de recherche. Damien a contribué de manière significative à l'implémentation de Pip sur l'architecture Armv7, ainsi qu'à la version de Pip pour les systèmes munis d'une *MPU*. Pip a aussi été au cœur de nombreuses collaborations industrielles avec Orange. Il a été principalement financé par le projet européen Celtic+ ODSI, puis par le projet ANR franco-allemand TinyPART.

1.2 Objets d'étude

Cette thèse est développée autour de trois objets d'études principaux qui transparaissent dans l'état de l'art et les chapitres de contribution.

1.2.1 Ordonnancement

Un des objets d'étude de cette thèse est l'ordonnancement. Dans un système où des entités ont besoin de ressources pour accomplir certaines actions et où les ressources disponibles sont limitées, l'ordonnancement est le fait d'arbitrer quelles entités disposeront d'un accès aux ressources ainsi que les périodes à laquelle elles en disposeront. Le logiciel réalisant l'ordonnancement au sein d'un système est appelé l'ordonnanceur. Le processus d'ordonnancement est omniprésent dans nos ordinateurs modernes, par exemple lorsqu'ils doivent choisir – plusieurs centaines de fois par seconde – le prochain programme à exécuter parmi la centaine de programmes attendant leur tour. Cette décision est orientée par la politique d'ordonnancement qui dicte à l'ordonnanceur selon quels critères distribuer les ressources. Certaines politiques s'attachent plus particulièrement au respect de contraintes temporelles : on parle alors de politique d'ordonnancement temps réel.

1.2.2 Transfert de flot d'exécution

Un autre objet d'étude connexe dont il sera abondamment question dans ce document est le transfert de flôt d'exécution. La notion de transfert de flot d'execution est similaire à celle de la commutation de contexte. Lorsqu'un programme est interrompu et qu'un autre s'exécute à sa place, par exemple sous l'effet de l'ordonnancement des programmes du système, il se déroule un transfert de flot d'exécution. Le système se charge de sauvegarder tous les paramètres qui permettront de restaurer le programme interrompu comme s'il n'avait jamais été interrompu, puis charge le programme qui continuera son exécution.

Ce mécanisme et le processus d'ordonnancement au sein d'un système permettent d'exécuter de multiples programmes de manière concurrente et efficace. Ils sont notamment à l'oeuvre lorsque vous travaillez avec de nombreux logiciels simultanément sur votre ordinateur, lorsque vous visionnez une vidéo sur votre téléphone, ou lorsque vous prenez l'avion et qu'il navigue en pilote automatique.

1.2.3 Preuve de programmes

Le dernier objet d'étude principal, omniprésent dans ce document, est la preuve de programmes. La preuve de programmes est une technique permettant d'apporter de très fortes garanties sur le fonctionnement du programme étudié par le biais de démonstrations mathématiques. Apporter de telles garanties sur un programme est néanmoins *extrêmement* coûteux, et n'est généralement entrepris que pour les systèmes dont un dysfonctionnement logiciel pourrait mettre en péril des vies humaines, ou pourrait engendrer la perte

d'une quantité astronomique d'argent. De tels systèmes sont courants dans certains secteurs d'activités tels que l'aérospatial ou le transport.

1.3 Présentation du document

1.3.1 Plan

La lecture de ce document a été découpée en 6 chapitres principaux. Le premier chapitre, que vous êtes en train de lire, est un chapitre d'introduction proposant une mise en contexte des travaux de thèses ainsi qu'une brève introduction aux objets d'étude principaux. Ce chapitre se termine par une exposition du plan et propose différents axes de lecture en fonction des sujets d'intérêts du lecteur.

Le second chapitre fait un état de l'art des sujets abordés dans cette thèse. Cet état de l'art se contente de décrire les notions nécessaires à la compréhension des travaux de thèse. Il est décomposé en trois parties, qui reflètent les objets d'étude principaux. La première partie de l'état de l'art concerne les transferts de flot d'exécution. Elle en donne une définition qui les classifie et décrit ses implications en terme de logiciel et de sécurité, plus particulièrement au travers du prisme de l'architecture Intel x86. La seconde partie de l'état de l'art, plus modeste, est dédiée à l'ordonnancement. Elle en fait une présentation générale, discute des politiques d'ordonnancement en proposant des métriques pour les évaluer, puis discute plus particulièrement des systèmes temps réel, dont elle développe juste la théorie nécessaire à la compréhension des travaux de cette thèse. Cet état de l'art s'achève sur une partie concernant la preuve de programmes. L'état de l'art sur cette partie commence par décrire le processus de raisonnement mathématique dans sa généralité, axée sur la vérification automatique du raisonnement grâce aux assistants de preuve, en prenant l'exemple de l'assistant de preuve Coq. Dans un second temps, cette partie décrit les méthodes particulières permettant de raisonner sur des programmes, puis conclue sur les exemples les plus probants de l'application de ces techniques sur des systèmes d'exploitation.

Le troisième chapitre présente la première contribution décrite dans cette thèse, un service de transfert de flot d'exécution unifié au sein du noyau Pip. Le chapitre commence par donner quelques motivations à l'écriture de ce nouveau service. Il décrit ensuite les idées derrière le service puis en décrit l'implémentation détaillée au sein de l'architecture Intel x86, en montrant comment les différents transferts de flot d'exécution ont été unifiés. Le chapitre se poursuit sur l'établissement de la preuve d'isolation traditionnelle de Pip sur ce service en décrivant les ajouts à l'interface avec la monade et détaillant fortement certaines portions du raisonnement, points clés de la preuve complète. Ce chapitre se termine sur une courte section de retours d'expérience, proposant certaines métriques à propos de ce service et proposant des réflexions sur le résultat produit.

Le quatrième chapitre présente la seconde contribution de cette thèse, un ordonnanceur *Earliest Deadline First* implémenté en espace utilisateur et s'exécutant sur le noyau Pip dont la correction de la fonction d'élection a été formellement prouvée. Le chapitre donne

d'abord une vue d'ensemble de l'ordonnanceur, décrivant la fonction d'élection et sa place au sein de l'ordonnanceur, et décrivant les composants internes de l'ordonnanceur. La section suivante décrit la preuve formelle de correction de la fonction d'élection, décrivant les différentes étapes du raffinement et les hypothèses utilisées. Le chapitre continue sur l'implémentation exécutable de l'ordonnanceur, et des problèmes de distortion entre les modèles et l'implementation. Ce chapitre se termine sur une section de prise de recul, discutant en particulier de la base de confiance des travaux présentés dans ce chapitre et dans les travaux de l'état de l'art.

Le cinquième chapitre présente la dernière contribution de cette thèse, une preuve de concept sur le noyau Pip, séparant la définition des modèles du code des services. Ce chapitre est conçu autour d'une portion de code des services, sur laquelle tous les aspects de Pip sont déclinés. Ce chapitre présente en premier lieu les motivations de cette contribution de manière brève. Il présente ensuite l'architecture monolithique actuelle de Pip, décrivant toutes les dépendances du code avec les modèles d'isolation, puis décrivant le processus de preuve actuel. Dans la section suivante est décrite la preuve de concept de ce chapitre. Elle commence par donner une vue d'ensemble de la preuve de concept et des changements apportés au noyau. Cette section décrit ensuite plus en détail chaque nouvel élément introduit par la preuve de concept, la manière d'extraire l'*AST* du code des services et la méthode de preuve du code des services avec cette preuve de concept. Le chapitre se termine sur les perspectives de recherche apportées par cette contribution.

Le dernier chapitre résume les trois contributions décrites dans ce document, en rappelant ce qui a été établi et les limites des travaux. Ce document s'achève sur les perspectives de recherche à plus long terme. Les dernières pages du document contiennent la bibliographie et les annexes.

1.3.2 Axes de lecture

Cette section tâche d'aiguiller votre lecture sur les parties du document susceptibles de vous intéresser.

Si vous souhaitez plutôt lire les parties du document dédiées aux aspects systèmes du transfert de flôt d'exécution et de l'ordonnancement, je vous invite à lire les sections 2.1 et 2.2 de l'état de l'art. Votre lecture peut se poursuivre dans le chapitre 3 décrivant le service de transfert de flot d'execution de Pip, et plus particulièrement la section 3.2 décrivant de manière détaillée l'implémentation unifiée des différents transferts de flot d'exécution. Enfin, vous pourrez achever votre lecture dans le chapitre 4 décrivant l'ordonnanceur *Earliest Deadline First*, en lisant les sections 4.2 qui donne une vision globale des composants de l'ordonnanceur ainsi que la section 4.4 qui décrit l'implémentation de ces composants.

Si vous êtes plutôt intéressés par la méthodologie relative à la preuve formelle de propriétés sur du code source, je vous invite à lire la section 2.3 de l'état de l'art, qui traite de l'application de méthodes formelles sur du code source. Vous pourrez ensuite lire la section 4.3 du chapitre 4, qui traite de la méthode de preuve de la fonction d'élection de l'ordonnanceur par raffinement. Vous pourrez terminer votre lecture sur le chapitre 5, qui

propose une solution modularisant les modèles au sein du noyau Pip, permettant de créer de nombreux modèles pour les mêmes services.

Si vous êtes plutôt intéressés par les détails de la conduite d'une preuve formelle sur du code, je vous invite à lire aussi la section 2.3 de l'état de l'art, qui traite de la mécanique de raisonnement formel sur du code source, puis la section 3.3.3 du chapitre 3, qui présente les détails d'une portion de la preuve formelle nécessaire à l'établissement de la propriété de préservation de l'isolation de Pip sur le code du service de tranfert de flot d'exécution. Le chapitre 4 et le description de l'établissement de la preuve en section 4.3 pourrait se révéler une lecture intéressante, bien que moins portées sur les détails de l'établissemrnt de la preuve.

2 État de l'art

Ce chapitre a pour intention de définir et préciser les différentes notions nécessaires à la lecture des travaux de thèse, ainsi que de définir le contexte scientifique du travail. Il portera, dans une première section, sur les détails des différents transferts de flot d'exécution dans les systèmes modernes, ainsi que les changements d'états inhérents à ces transferts de flot d'exécution. Cette section abordera ensuite les problèmes de sécurité liés au transfert de flot d'exécution, ainsi que les techniques de mitigation de ces problèmes. Cette section terminera sur les problématiques temps réel touchant au transfert de flot d'exécution. La seconde section de ce chapitre fera un état des lieux de la preuve de programme. Elle commencera par discuter de ce qu'est une preuve et de leur vérification automatique, ainsi que des stratégies de conduite de preuve. Cette section continuera sur la preuve de programme, en particulier comment raisonner sur un programme impératif. Elle abordera aussi les notions de représentation du langage. Enfin, elle terminera sur les exemples de systèmes vérifiés formellement.

2.1 Transfert de flot d'exécution

2.1.1 Définitions

Flot d'exécution Un flot d'exécution est une trame de modifications successives de l'état de la machine, animée par la récupération et l'exécution machinale d'instructions assembleur contigües par le processeur. Cette trame est l'œuvre d'une entité qui a conçu le code et les conditions d'exécution nécessaires permettant d'aboutir à un résultat. Ces conditions d'exécution (ou *contexte d'exécution*) peuvent comprendre n'importe quel sous-ensemble de l'état de la machine, comme l'état de certains registres du processeur ou l'état d'une partie de la mémoire par exemple. Une certaine notion de confiance est liée à chaque fil d'exécution, qui peut se refléter sur les droits accordés à celui-ci.

Transfert de flot d'exécution Le transfert de flot d'exécution est le fait de faire diverger un flot d'exécution vers un autre. Il existe de nombreuses utilisations du transfert de flot d'exécution, mais nous en distinguerons deux types : les transferts de flot d'exécutions internes et les transferts de flot d'exécution externes.

Un transfert est dit *interne* si le flot d'exécution initial et le flot d'exécution ciblé ont été conçus par la même entité. Les deux flots d'exécutions sont conçus pour coopérer, et ce type de transfert se déroule typiquement sans changement de droits. Un premier exemple d'un tel transfert de flot d'exécution, le plus simple, est le saut : il permet de définir de

manière arbitraire l'adresse de la prochaine instruction à exécuter. Similairement, le saut conditionnel permet d'effectuer un saut en fonction du résultat d'une instruction assembleur, divisant le flot d'exécution en deux fils possibles. Les appels de fonctions sont un exemple plus complexe de transferts internes ; contrairement aux sauts, les appels de fonctions permettent de revenir au flot d'exécution initial. Dès lors, à chaque appel de fonction, il est nécessaire de sauvegarder la portion de l'état qui sera modifiée par effet de bord lors de l'exécution de la fonction. La portion de l'état modifié par la fonction est connue par l'entité, qui peut donc réaliser une sauvegarde partielle minimale de l'état sans perte d'information sur l'état.

Les transferts de flot d'exécution *externes* sont ceux qui – par opposition aux transferts de flot d'exécution internes – impliquent un transfert de flot d'exécution vers le fil d'exécution d'une autre entité. Ce type de transfert peut se traduire par un changement de droits liés au fil d'exécution, reflétant les droits de la nouvelle entité. Les transferts externes peuvent aussi engendrer une sauvegarde du contexte d'exécution initial ; cependant, à l'instar des transferts internes, il n'est pas possible de savoir quelles portions de l'état sont susceptibles d'être modifiées par l'autre entité. Ainsi, un tel transfert implique une sauvegarde préventive complète du contexte d'exécution susceptible d'être modifié par l'autre entité.

Réaliser une telle sauvegarde est une opération délicate car il est difficile de déterminer ce qui fait partie de l'état d'un programme. Le contenu des registres du processeur et de la mémoire, mais aussi l'état des composants logiciels ou matériels intéragissant avec le programme (tels que les périphériques, le cache, etc.) font partie intégrante de l'état d'un programme.

De plus, chaque fil d'exécution ciblable par un transfert externe doit être muni d'un cadre d'exécution, qui spécifie le point d'entrée, l'environnement initial, et les droits liés à ce fil d'exécution. Ces cadres servent à s'assurer que le flot d'exécution ne sera exécuté que dans les conditions prévues par l'entité qui l'a conçu. Cela implique que l'intégrité des cadres d'exécution de chaque entité doit être garanti au sein d'un système ; une entité ne doit pas pouvoir altérer les cadres d'exécution d'une autre entité.

Ce document discutera principalement des transferts de flot d'exécution externes. La prochaine sous-section présentera des exemples de transferts de flot d'exécution externes au sein d'un système.

2.1.2 Transferts de flot d'exécution externes au sein d'un système

Appels explicites

Les transferts externes les plus courants sont les transferts de flot d'exécution explicites, c'est-à-dire dont la cible est explicitement fournie lors de l'appel, ou clairement établie dans la documentation.

Par exemple, dans Linux, un processus peut demander l'ouverture d'un fichier avec l'appel système `open()`. Cet appel transfère le flot d'exécution d'un processus non privilégié

vers le noyau Linux disposant du plus haut niveau de privilèges. Les fonctions appelables par des transferts explicites servent d'interface entre des logiciels disposant de droits distincts. Dans le cas de Linux, le cadre d'exécution a été défini par Linux, qui prend les dispositions de sauvegardes nécessaires afin de pouvoir rétablir le contexte d'exécution du processus appelant au moment du retour de l'appel système.

Fautes

Les différentes formes de fautes logicielles constituent aussi une forme de transfert de flot d'exécution externes. Les logiciels sont susceptibles de déclencher des fautes logicielles de différentes façons, par exemple :

— décodage impossible de la prochaine instruction ;
— demande d'exécution d'une instruction impossible (division par zéro...) ;
— demande d'accès à une adresse mémoire protégée, résultat de l'activité d'une MMU ou d'un autre dispositif matériel (MPU, Trustzone, SGX) ;
— demande d'exécution d'une instruction privilégiée en mode non-privilégié.

Dans ces différentes situations, il s'agit de transferts implicites depuis le logiciel en faute vers une fonction d'un logiciel en charge du traitement de cette faute. Les différentes fonctions de gestion des fautes sont généralement définies par des éléments de configuration du matériel. Le plus souvent, ils sont définis par l'intermédiaire d'une table (ou vecteur) dont le nom change d'une architecture de microprocesseur à l'autre. Ce vecteur précise généralement le niveau d'élévation de privilèges associé à l'exécution de la fonction de traitement de la faute. Cette table précise en réalité le cadre d'exécution des fils d'exécutions dédiés à chaque fautes. Sur les architectures Intel cette table est appelée *IDT* (pour *Interrupt Descriptor Table*).

Interruptions matérielles

Les interruptions matérielles sont des transferts non explicites *a priori* non contrôlés par le code non privilégié. Elles sont déclenchées par le matériel, signalant un événement important à traiter, tel que l'arrivée d'un paquet réseau par exemple. Les fonctions de traitement des interruptions matérielles ainsi que leur cadre d'exécution sont aussi définis dans l'*IDT* dans l'architecture Intel x86.

2.1.3 Implications logicielles d'un transfert de flot d'exécution

Changement de droits

Sur l'architecture Intel x86, les différents transferts de flot d'exécution peuvent s'opérer par le biais de différentes instructions et événements matériels. Néanmoins, lorsqu'un changement de droits est requis lors d'un transfert, les différents chemins sont régis par un ensemble de mécanismes de contrôle relativement homogènes.

Changement de droits sous x86 Les privilèges attribués au code s'exécutant actuellement sur la machine sont ceux du *segment* chargé dans le registre `CS` (pour *code segment*). Les segments sont définis dans la *GDT* (pour *Global Descriptor Table*) à l'initialisation de la machine. La *GDT* est une table globale spécifiée par Intel, dont l'adresse est accessible par un registre dédié. Dans cette table par exemple, Linux se contente de définir deux segments de code. Un premier segment associé au niveau de privilèges maximum du microprocesseur, nommé par Intel *ring 0*, utilisé pour le code responsable du système ; et un second segment non-privilégié, associé au niveau de privilèges *ring 3*, pour le reste du code.

Contrairement au code s'exécutant avec le segment privilégié, le code s'exécutant sans privilège ne peut pas changer de segment à sa guise. Des mécanismes de contrôle du processeur déclenchent une faute si du code non-privilégié essaie de modifier son segment de code. Pour y parvenir, il est possible d'utiliser les *gates*, qui sont des tremplins définis dans les tables globales du système (comme l'*IDT* ou la *GDT*), permettant au code non privilégié d'appeler une fonction prédéfinie qui s'exécute avec d'autres droits.

Lorsqu'un changement de segment déclenche un changement de niveau de privilèges, le processeur change de pile. Ce changement de pile permet d'éviter aux routines privilégiées les échecs dus à un manque de place sur la pile, ainsi qu'à les prémunir d'éventuelles interférences avec les procédures non privilégiées [Int19a]. Une pile doit être définie par niveau de privilèges (*ring*) utilisé par le système ; leurs adresses doivent être renseignées dans une structure appelée *TSS* (pour *Task State Segment*). Cette structure est initialisée conjointement avec la *GDT* qui contient son descripteur. Un registre dédié indique au processeur la position de ce descripteur dans la *GDT*.

Appels systèmes sur x86 Afin d'obtenir un transfert de flot d'exécution avec élévation de privilèges, le logiciel appelant non-privilégié peut exécuter des instructions dédiés aux différentes *gates* des tables globales. Tout d'abord, l'instruction `int` permet d'appeler les *gates* situées dans l'*IDT*. Ces *gates* sont soit des *interrupt gates*, des *trap gates* ou des *task gates* [Int19b]. `int` s'accompagne d'un argument correspondant à l'index de la *gate* ciblée dans l'*IDT*. Le code ainsi appelé sera exécuté avec le niveau de privilèges spécifié par le segment de code indiqué dans la gate (et chargé dans le registre `CS`).

D'autres instructions telles que `lcall` ou `ljmp` permettent d'utiliser les *gates* situées dans la *GDT*. Ces gates sont soit des *call gates* ou des *task gates*. Ces gates permettent de copier un nombre fixé d'arguments depuis la pile de l'appelant dans la pile du code privilégié à l'appel de ces instructions. Le nombre d'arguments à copier est renseigné dans la *gate* ciblée par l'instruction. Là aussi, l'élevation de privilèges est spécifiée par le segment de code indiqué dans la gate et chargé dans `CS` lors de l'appel.

La troisième manière de déclencher une élévation de privilèges est l'instruction `sysenter`. Cette instruction ne sollicite aucune *gate* : à la place, elle utilise les *MSR* (pour *model-specific registers*), qui sont des registres de contrôle du processeur. Ces *MSR* sont manipulables grâce aux instructions `wrmsr` et `rdmsr`, qui sont des instructions privilégiées et qui permettent d'écrire et de lire dans ces registres respectivement. Un appel à `sysenter` uti-

lise les MSR `0×174`, `0×175` et `0×176` pour charger `CS EIP SS ESP`. Le système doit donc avoir initialisé ces registres avant l'utilisation de `sysenter`. De plus, `sysenter` ne sauvegarde pas l'adresse de retour ni l'adresse de la pile lors d'un appel, qui doivent être placés dans les registres `ECX` et `EDX` au moment de l'appel à `sysexit` pour retourner dans le code appelant.

Interruptions et fautes sur x86 Les interruptions liées au matériel sur l'architecture x86 étaient autrefois gérées par un coprocesseur (le PIC 8259 *pour Programmable Interrupt Controller* ou plus récemment, l'APIC pour *Advanced Programmable Interrupt Controller*). Ce coprocesseur est maintenant intégré au processeur, mais nous continuerons de parler de coprocesseur pour honorer l'histoire. Ce coprocesseur utilise les *gates* situées dans l'IDT de la même manière que l'instruction `int`. Il est possible de configurer ce coprocesseur pour qu'il utilise une certaine plage de niveaux d'interruption, ou pour qu'il masque temporairement la venue de nouvelles interruptions. Les fautes utilisent elles aussi l'*IDT*, et utilisent les trente-deux premières *gates* de la table, en fonction de la faute à déclencher. Les fautes et interruptions déclenchées par le processeur ou le coprocesseur ont toujours le droit d'utiliser les *gates*, peu importe le niveau de privilèges du code s'exécutant au moment de l'interruption.

Fonctionnement de la *MMU* sur l'architecture Intel 32 bits Sur l'architecture Intel x86, mais aussi sur toutes les autres architectures supportant une *MMU* (pour *Memory Management Unit*), il est possible d'associer des droits d'accès spécifiques à chaque page de mémoire configurée dans l'espace d'adressage virtuel.

Le concept de traduction d'adresse virtuelle vers l'adresse réelle est le suivant. Les bits de poids fort de l'adresse virtuelle servent à traverser les tables de la MMU (sur l'architecture Intel x86, le *Page Directory* et les *Page Tables*). Les bits de poids faible, quant à eux, correspondent à l'emplacement de l'adresse désirée dans la page réelle obtenue après traduction (souvent appelé *offset*).

En particulier, sur Intel x86 et en mode de pagination 32 bits pour des pages de 4 Kio, les espaces d'adressage sont configurés par une structure de données arborescente de pages de 4 Kio. Cette structure de données a deux étages : la racine appelée PD pour *Page Directory*, et les feuilles appelées PT (pour *Page Tables*). Le développeur renseigne l'adresse du Page Directory à utiliser dans le registre `CR3` du processeur ; cette adresse est alignée sur 4 Kio, les 12 bits de poids faible (11-0) sont ignorés ou réservés pour un autre usage.

Plus précisement, le *Page Directory* est constitué de 1024 entrées de 32 bits appelées les *PDE* (pour *Page Directory Entries*). Les 20 bits de poids fort (31-12) de ces entrées déterminent l'adresse de la *Page Table* à utiliser, alignée sur 4 Kio. Les *Page Tables* sont aussi constituées de 1024 entrées de 32 bits appelées *PTE* (pour *Page Table Entries*). De la même manière, les 20 bits de poids fort (31-12) déterminent l'adresse de la page de mémoire réelle, alignée sur 4 Kio. Il est aussi possible de configurer des pages de 4 Mio plutot que des pages de 4 Kio en modifiants certains bits de controle des *PDE*.

Lors de la traduction d'une adresse virtuelle, les 10 bits de poids fort de l'adresse virtuelle (31-22) déterminent le numéro de *PDE* à utiliser, les bits (21-12) déterminent le numéro de *PTE*, et les 12 bits de poids faible restants (11-0) déterminent l'*offset* de l'adresse cible dans la page réelle. [Int19c]

Contrôle d'accès par la MMU sur Intel x86 Dans le mode de pagination 32 bits d'Intel, les droits associés à chaque page sont présents dans les *PTE*, dans les 12 bits de poids faible. Le bit 1 (`R/W`) permet d'empêcher les accès en écriture sur la page. Le bit 2 (`U/S`) permet d'empêcher n'importe quel accès utilisateur à la page - en lecture ou en écriture. Le niveau de privilège de l'accès dépend du *CPL* (pour *Current Privilege Level*) de l'instruction courante, souvent déterminée par le segment de code actuel. L'architecture Intel permet de restreindre la récupération des instructions en discriminant chaque page de mémoire. Cette fonctionnalité n'est cependant pas disponible dans le mode de pagination 32 bits, puisqu'elle nécessite des *PDE* ou *PTE* longues de 64 bits. Elle est par exemple disponible dans le mode de pagination *PAE*. Cette fonctionnalité est activable au travers du bit `NXE` du registre `IA32_EFER`. Pour chaque page de mémoire, mettre le bit 63 (`XD`) à 1 dans une *PTE* empêche la récupération d'instruction depuis cette page mémoire.

Cependant, d'autres fonctionnalités de contrôle d'accès globaux liées à la *MMU* existent. L'architecture Intel propose aussi les mécanismes *SMAP* et *SMEP* comme décrits ci-après dans le paragraphe 2.1.4. Le bit *SMAP* (pour *Supervisor Mode Access Protection* présent dans le registre `CR4` permet d'empêcher du code utilisant un segment de données privilégié d'accéder aux pages mémoire annoncées comme étant des pages mémoire utilisateur (bit `U/S` à 1). Le bit *SMEP* pour *Supervisor Mode Execution Protection* présent dans le registre `CR4` permet d'empêcher la récupération de code lorsque le segment actuel octroie des accès privilégiés et que la page mémoire contenant les instructions est annoncée comme une page mémoire utilisateur (bit `U/S` à 1).

Capture de l'état d'exécution

Lorsqu'un transfert de flot d'exécution externe survient, un changement de contexte d'exécution doit être effectué, assisté par le processeur. En effet, sans l'aide du processeur, le contenu de certains registres (comme le pointeur d'instruction ou le pointeur de pile) seraient instantanément perdus au moment du transfert. Voici comment se passe le transfert pour chaque mode de transfert différent sous l'architecture Intel x86 :

Changement de pile et registres sauvés lors d'un appel au travers d'une *callgate* Lors d'un tel appel, le processeur accède à la *callgate* par la *GDT*. Cette callgate indique entre autres, le segment de code à utiliser (et donc le niveau de privilège associé), l'emplacement du code à exécuter, le nombre d'arguments à copier depuis la pile utilisateur. La pile ainsi que le segment à utiliser sont eux renseignés dans la *TSS* et sont liés au niveau de privilège du segment de code de la *callgate*.

Tout d'abord, le processeur sauve de manière temporaire le segment de pile et le pointeur de pile dans un tampon interne. Il remplace les registres de segment de pile et de pointeur de pile par ceux renseignés dans la *TSS*, changeant de pile. Il pousse ensuite le segment de pile et le pointeur de pile de l'utilisateur dans la nouvelle pile. Il copie ensuite les arguments depuis la pile utilisateur vers la nouvelle pile, leur nombre étant renseigné dans la call gate. Enfin, le processeur pousse sur la pile le registre de segment de code et le pointeur d'instruction, avant de les remplacer par ceux renseignés dans la *callgate*.

Changement de pile et registres sauvés lors d'une interruption, d'une faute ou d'un appel à `int` Lors d'une faute, d'une interruption ou d'un appel à `int`, le processeur accède à une interrupt gate ou une trap gate dans l'*IDT*. Comme pour la call gate, la gate contient le segment de code à utiliser, le code à exécuter. Ce transfert de flot d'exécution n'entraine pas une copie par le processeur d'éventuels arguments depuis la pile utilisateur sur la nouvelle pile.

Tout d'abord, le processeur change de pile, en sauvant temporairement les valeurs utilisateurs du segment de pile `SS` et de la pile `ESP`, et en les écrivants sur la pile renseignée dans la nouvelle pile renseignée dans la *TSS*. Le processeur sauve ensuite l'état des registres `EFLAGS`, du segment de code `CS` et du pointeur d'instruction `EIP`. EFLAGS contient entre autres l'état des drapeaux d'overflow, de retenue, de parité, de conditions, mais aussi d'activation des interruptions. Le processeur peut pousser un code d'erreur supplémentaire sur la pile dans le cas de déclenchement de certaines fautes afin de préciser leur cause.

2.1.4 Sécurité liée aux transferts de flot d'exécution

Les transferts de flot d'exécution (internes ou externes) sont un des points clés de la sécurité d'un système. Pour prendre le contrôle d'un système, une entité pourrait essayer de corrompre les transferts de flot d'exécution se tenant à l'intérieur du système pour réussir à exécuter son propre fil d'exécution dans le système.

Ainsi, ce document distinguera deux catégories majeures d'attaques : les attaques visant les transferts de flot d'exécution internes, et les attaques visant les transferts de flot d'exécution externes.

Sécurité relative aux transferts de flot d'exécutions internes

Ce type de transfert de flot d'exécution, d'apparence assez anodine, est pourtant l'objet d'attaques multiples, dont le but est de faire dévier l'exécution (de préférence en mode privilégié) vers du code choisi par l'attaquant. Pour y arriver, un attaquant doit exploiter une vulnérabilité dans une portion de code, qui lui donnera le contrôle d'une zone de mémoire d'intérêt (la pile, le tas, ou même le code). Une fois qu'il contrôle cette zone mémoire, il lui suffit d'écrire un *shellcode*, et d'exploiter une vulnérabilité dans du code privilégié pour que l'exécution du `return` de la fonction compromise saute dans le shellcode. L'attaquant gagne à ce moment le contrôle de la machine.

Commence alors un jeu du chat et de la souris pour essayer de mitiger l'impact de ces vulnérabilités. Pour compliquer la vie de l'attaquant, et qu'il lui soit plus difficile d'exécuter son shellcode, de nombreuses stratégies ont été entreprises par les fabriquant de matériels ainsi que par les développeurs de systèmes d'exploitation.

Canaries Une première stratégie est l'ajout de *canary* qui visent à détecter les corruptions mémoires. Les canaries sont des valeurs écrites dans la pile ou le tas et qui sont générées aléatoirement à chaque exécution. Lors de la sortie de la frame protégée par le canary, le code vérifie que la valeur du canary correspond bien à celle qui avait été écrite initialement ; si ce n'est pas le cas, c'est qu'une corruption mémoire a eu lieu et une faute est levée.

Une des techniques permettant de vaincre les canaries est de lire la valeur initiale du canary avant de corrompre la mémoire. En effet, la canary **reste la même pour l'intégralité de l'exécution**. Une fois cette valeur récupérée, il suffit de corrompre la mémoire en réécrivant cette valeur au bon endroit pour échapper à la détection. De plus, si l'exploitation de la vulnérabilité permet de corrompre la mémoire de manière fine, il suffit d'éviter d'écrire sur la canary.

Droits fins pour les zones mémoires Une autre stratégie a été de définir des droits fins concernant l'accès aux différentes zones mémoires de l'espace d'adressage des processus. Le mécanisme de mémoire virtuelle permet de définir des droits d'accès propres à chaque page mémoire configurée (lecture, écriture, éxecution, accessible en mode non privilégié). Par exemple, les pages mémoire contenant du code sont typiquement configurées pour des accès en lecture et exécution, alors que les pages contenant des données (pour la pile, le tas, les sections de données d'un binaire) sont configurées pour des accès en lecture/écriture.

Cette stratégie de défense empêche un attaquant d'exploiter une vulnérabilité pour écrire un shellcode dans la mémoire si on considère que chaque page de mémoire est soit exécutable, soit accessible en écriture. Cependant, il existe des cas d'usage légitimes qui violent cette contrainte, par exemple lors de compilation à la volée (ou JIT, pour Just-In-Time). Fatalement, de tels logiciels sont devenus la cible privilégiée des attaquants, on pourra par exemple citer Webkit [Gro18]. Heureusement, il est peu probable que de tels logiciels aient besoin de s'exécuter en mode privilégié.

Pour affaiblir ce vecteur d'attaque, cette stratégie de défense est renforcée par des mécanismes de sécurité supplémentaires tels que le *Supervisor Mode Access Prevention* (SMAP) et le *Supervisor Mode Execution Prevention* (SMEP). SMAP permet au processeur de lever une faute lorsque qu'il exécute du code privilégié et qu'il essaie d'accéder (en lecture ou en écriture) à des données présentes dans l'espace utilisateur. SMEP permet en complément de lever une faute lorsque le processeur essaie d'exécuter du code dans l'espace utilisateur alors qu'il se trouve dans un mode d'exécution privilégié.

Ces mécanismes permettent d'isoler le code privilégié de potentiels shellcodes écrit en espace utilisateur. Ainsi, pour compromettre intégralement un système, l'attaquant doit à

présent exploiter une vulnérabilité dans le code privilégié, ayant à sa disposition des pages mémoire soit accessibles en écriture soit exécutables et qui, de surcrois, ne font pas partie de l'espace utilisateur. Nait alors une nouvelle technique d'exploitation de vulnérabilité.

Return Oriented Programming Le ROP (pour *Return Oriented Programming*) consiste à attaquer du code vulnérable en n'utilisant que le code déjà accessible dans l'environnement d'origine, mais en exécutant des portions arbitraires de celui-ci. L'attaque consiste à repérer des *gadgets* : de brèves portions de code ayant un effet spécifique sur la mémoire ou les registres, suivi d'une instruction `return`. Pour l'attaquant, il suffit de dévier le flot d'exécution sur l'un de ces gadgets et de manipuler la mémoire, de manière à ce que l'exécution du gadget entraine l'exécution du suivant. L'attaquant parvient au final à exécuter son shellcode constitué d'une succession de gadgets, contournant les mécanismes de sécurité mentionnés dans le paragraphe précédent.

Plusieurs contre-mesures ont émergé pour rendre plus difficile le ROP.

Address Space Layout Randomization L'ASLR (pour *Address Space Layout Randomization*) rend imprédictible l'adresse des différentes zones de mémoire au sein d'un espace d'adressage virtuel. Les adresses du binaire, de la pile, du tas, des librairies, du noyau, etc. sont rendus aléatoires à chaque nouvelle exécution. L'ASLR est de ce fait un frein considérable au développement d'un shellcode en ROP, puisqu'il est impossible de prédire où se situeront les gadgets lors de la prochaine exécution.

L'ASLR n'est cependant pas parfait. Les adresses des zones mémoire restantes peuvent être révélées par des pointeurs dans les zones mémoires controlées par l'attaquant [Ngu21], ou grâce à des attaques micro-architecturales [Gru+16 ; Jen22].

Vérification de l'intégrité du flot d'exécution Une autre approche permettant de réduire la marge de manoeuvre de l'attaquant et de vérifier que le flot d'exécution est conforme à celui attendu. À chaque appel et à chaque retour de fonction, le processeur vérifie si la cible du saut est valide. Plusieurs implémentations existent, notamment des implémentations matérielles au sein des processeurs [ASI17 ; Muj21], mais aussi certaines implémentations logicielles notamment provenant de compilateurs [Tic+14]. Windows, macOS, Android, iOS utilisent déjà un mécanisme de vérification du flot d'exécution.

eXecute Only Memory Le XOM (pour *eXecute Only Memory*), est une fonctionnalité de certains processeurs permettant de déclencher une faute lorsqu'un accès en lecture est fait sur les pages mémoires configurées comme étant exécutables. Avant cette fonctionnalité, aucune distinction n'était faite entre le processus de récupération des instructions par le processeur et la lecture de données par l'utilisateur. Cette fonctionnalité rend considérablement plus difficile la recherche de gadgets, puisqu'il est impossible pour l'attaquant de lire le code qu'il souhaite compromettre directement sur la cible. Il est parfois reproché à cette

fonctionnalité qu'elle relève de la sécurité par l'obscurité, et qu'elle n'est pas réellement efficace.

« Mieux vaut prévenir que guérir » Ces contre-mesures, sans cesse contournées par de nouvelles méthodes d'attaque, supposent qu'il existera toujours des vulnérabilités dans le logiciel comme dans le matériel et tentent donc de limiter au maximum leur impact sur les systèmes affectés. Une toute autre classe de mesures essaie de régler le problème en s'attaquant à l'existence même des vulnérabilités, plutôt que d'essayer minimiser leurs conséquences.

Les méthodes formelles, qui feront l'objet d'une section à part entière dans cet été de l'art (voir 2.3), en font partie. On pourrait aussi citer les méthodes d'analyse statique [Bal+06], les méthodes d'exécution symbolique [Bal+18], de fuzzing [Sch+17], et plus particulièrement le langage Rust [MK14 ; Moz] conçu pour éradiquer ces vulnérabilités par conception. Par ailleurs, de récents travaux ont commencé à s'intéresser formellement aux fonctionnalités de Rust [Jun+17].

2.1.5 Sécurité relative aux transferts de flot d'exécution externes

Comme exprimé dans la section 2.1.1, certains transferts de flot d'exécution dits *externes* impliquent un changement de privilèges et d'environnement. Ces changements sont décrits par des cadres d'exécution définis par le concepteur du système.

Une catégorie d'attaque vise à compromettre ces cadres d'exécution en les modifiant de manière avantageuse pour l'attaquant, afin par exemple de s'octroyer des droits supplémentaires. Ainsi, la sécurité d'un système informatique dépend aussi du maintient de l'intégrité des éléments liés aux cadres d'exécution des procédures des différentes entitées présentes au sein du système. Les éléments ciblés par ce type d'attaques sont variés ; sur l'architecture Intel x86, ces attaques pourraient par exemple viser à altérer les structures liées à l'*IDT*, à la *GDT*, à la *TSS* ou encore aux structures liées à l'arborescence de la *MMU*.

D'autres types d'attaques de cette catégorie concernent l'environnement d'exécution du programme ciblé, et tirent profit de portions de l'état du programme contenues dans le matériel pour en extraire de manière illégitime des informations critiques. Les attaques Spectre [Koc+20], Meltdown [Lip+20] et leurs dérivés en sont les représentants les plus emblématiques. Ces attaques ont très récemment eu un retentissement majeur dans le monde de la sécurité informatique de par leur sévérité. Elles sont rendues possibles par une isolation insuffisante au niveau du matériel de ce qui constitue l'état d'un programme.

Cas particulier des interruptions matérielles et des fautes

Puisque les fautes et interruptions déclenchent un changement de privilèges, elles sont un vecteur d'attaque pour un attaquant cherchant à s'octroyer de nouveaux droits au même titre que les autres éléments liés aux cadres d'exécution précédemment décrits. En effet,

des failles peuvent résider dans les routines de gestion de ces portions de logiciel, mais aussi dans la protection des cadres d'exécution de ces routines.

Cependant, elle présentent une caractéristique supplémentaire d'intérêt pour un attaquant : les interruptions et fautes brisent le flot d'exécution de manière intempestive et modifient potentiellement l'environnement d'exécution du code interrompu. Cela offre aux attaquants de nouvelles perspectives d'attaques sur les fils d'exécution pouvant être interrompus, qui tombent alors dans la catégorie des vulnérabilités de *concurrence critique*.

Même en ayant pleinement conscience des différentes interactions et dépendances entre les différents composants d'un système, les vulnérabilités de concurrence critique sont **notoirement difficiles à cerner** [NM92], principalement à cause d'un phénomène d'explosion combinatoire. Il peut s'avérer difficile de détecter une telle vulnérabilité par les tests, puisqu'ils sont souvent effectués dans des environnement très controlés où les mêmes conditions d'exécution sont artificiellement répétées, occultant d'autres fils d'exécution possibles. Malgré cela, si le développeur parvient à exhiber un fil d'exécution contenant un comportement anormal, il peut alors être délicat de reproduire le fil d'exécution ayant conduit à ce comportement. En effet, le fil d'exécution peut etre le résultat de nombreuses interactions – parfois non-déterministes – du programme avec son environnement. De plus, attacher un debuggueur tel que `gdb` peut modifier subtilement ces interactions, de manière telle qu'il soit impossible d'exhiber à nouveau le comportement anormal : on parle alors d'Heisenbug.

Pour illustrer la difficulté à cerner cette catégorie de bugs, on pourrait citer un problème d'incohérence de cache dans le noyau de système d'exploitation de la Nintendo Switch après une interruption matérielle ayant mené à un changement de coeur. Les effets de ce bug avaient été observés dès la sortie de la console ; il n'a cependant été trouvé et corrigé qu'à la sortie du firmware 14.0.0 de la console, soit plus de 5 ans après les premiers rapports [plu22]. On pourrait aussi citer une vulnérabilité exploitée dans la pile IPV6 du noyau FreeBSD de la console Playstation 4 de Sony, profitant d'une situation de concurrence critique pour déclecher un Use-After-Free et compromettant le système d'exploitation. Cette faille présente sur tous les firmwares de la console depuis son lancement en 2013 a été découverte en 2018 puis divulguée et patchée en 2020, soit 7 ans après sa mise sur le marché [Ngu20]. La même vulnérabilité était présente jusqu'à la version 5.00 du firmware de la Playstation 5, soit plus de deux ans après le rapport de bug concernant l'ancienne console [Ngu22].

Certains débuggers (notamment `rr` [OCa+17]) ont implémenté une fonctionnalité "*record and replay*" (enregistre et rejoue), permettant de capturer une trace du programme inspecté, puis de rejouer dynamiquement cette même trace à la demande. Cette fonctionnalité résoud le problème de la reproductibilité des comportement anormaux des programmes, et permet de surcrois de revenir à un état précédent de l'exécution lors d'une session de débuggage, ce qui est impossible avec les débuggers classiques. Certains émulateurs tels que Xen ou Qemu proposent des fonctionnalités "*record and replay*" sur les machines virtualisées. Malheureusement, les fonctionnalités "*record and replay*" pour des programmes

sur plusieurs coeurs sont actuellement extrêmement lents, et profiteraient grandement d'implémentation matérielles si elles venaient à exister [OCa+17].

De nombreux travaux ont été menés afin de détecter les situations de concurrence critique, par exemple par analyse statique [EA03]. D'autres travaux ont développés des méthodes plus particulières permettant de découvrir des situations de concurrence critique liées aux interruptions matérielles [Wan+17]. Parallèlement, dans le monde de la preuve formelle, on pourrait citer les travaux ayant abouti à la logique de séparation [Rey02].

2.2 Ordonnancement

Le transfert de flot d'exécution est au coeur du fonctionnement de systèmes complexes, par exemple lors de l'ordonnancement au sein d'un système informatique. L'ordonnancement dans un système d'exploitation permet à plusieurs programmes de s'exécuter de manière concurrentielle sur le même processeur, en alternant leur exécution. L'ordonnanceur décide quel programme sera le prochain à s'exécuter sur une unité de calcul donnée, qu'il décide selon une *politique d'ordonnancement*. Ces politiques sont multiples et visent à satisfaire des contraintes diverses, pouvant par exemple viser à exécuter les programmes interactifs de manière prioritaire, ou plus simplement à exécuter chaque programme l'un après l'autre. L'ordonnancement permet ainsi d'optimiser l'utilisation du processeur pour un objectif particulier.

2.2.1 Partage équitable du CPU

Une des fonctions principales de l'ordonnancement est le partage du temps CPU. En effet, les programmes s'exécutant à sur un système ne collaborent pas forcément avec le système pour permettre aux autres programmes de s'exécuter. Il revient alors au système d'exploitation d'interrompre les programmes à intervalles réguliers, grâce aux interruptions déclenchées par l'horloge par exemple, pour que chaque programme puisse avoir une chance de bénéficier d'un créneau d'exécution. Lorsque le programme est interrompu, le système appelle l'ordonnanceur, qui choisira le meilleur programme à exécuter selon sa politique. Plusieurs indicateurs permettent d'évaluer les politiques d'ordonnancement dans les systèmes classiques, notamment :

- le débit, mesurant le nombre de tâches terminées sur une certaine période
- le temps d'attente, mesurant le temps moyen entre le moment où la tâche a été créée et le moment où elle a commencé à être exécutée
- l'équité, mesurant la différence entre les temps CPU accordés à chaque processus comparé à leurs besoins
- la latence, mesurant le temps d'attente moyen entre la soumission de la tache et sa terminaison

Exemples de politiques d'ordonnancement

Les politiques d'ordonnancement se divisent en deux catégories : les politiques d'ordonnancement temps-réel et les autres. Les politiques d'ordonnancement temps réel sont dédiées au respect de contraintes temporelles associées aux processus à ordonnancer. Les autres politiques d'ordonnancement privilègient d'autres critères qui peuvent complètement varier selon les objectifs ciblés par la politique d'ordonnancement.

La politique d'ordonnancement la plus simple est celle du tourniquet (*round-robin scheduling*) : une unique file des processus en attente d'exécution est maintenue par l'ordonnanceur, qui exécute le premier processus de la file. Au bout d'un certain temps, l'ordonnanceur arrête l'exécution du processus et le place à la fin de cette file, puis exécute le suivant. Cette politique favorise grandement l'équité entre les processus, en considérant que tous les processus ont la même priorité. Certaines variantes de cette politique existe, comme par exemple la politique *fair-share*, qui divise équitablement le temps CPU entre les utilisateurs ou les groupes plutôt qu'entre les processus [KL88]. La politique *foreground-background* [NW08] permet aux processus nouvellement créés de bénéficier d'une période d'exécution prioritaire, et sont placés dans la file *foreground*. À la fin de cette période d'exécution, le processus est placé dans la file *background* qui est ordonnancée comme tous les autres processus avec une politique d'ordonnancement à tourniquet.

La politique *shortest job next* ou sa version préemptive *shortest remaining time next* exécutera en priorité les processus les plus courts ou ayant presque complété leur exécution. Cette politique maximise le débit, au détriment de l'équité. Cette politique se montre efficace pour ordonnancer le traitement de requêtes sur un serveur web [AY08].

De très nombreuses politiques sont disponibles, chacunes conçues pour maximiser des objectifs particulier. Pour conclure ces exemples avec une politique créé pour maximiser un objectif peu commun, on pourrait par exemple citer la politique d'ordonnancement *YDS* [YDS95], conçue pour minimiser l'énergie dépensée par le processeur à exécuter les processus.

2.2.2 Respect des contraintes de temps

L'ordonnancement est un élément clé des systèmes temps réel. Les systèmes temps réel ont des contraintes de temps associées à chaque unité de travail. Les systèmes temps réel *souples* sont munis de contraintes de temps indicatives, le non respect des contraintes temporelles pouvant mener à une dégradation de la qualité des résultats produits par le système, par exemple dans le cas d'applications multimédia (audio, vidéo, etc.) ou dans le cas de systèmes de surveillance collectant des données (par exemple météorologiques). Nous nous attarderons sur les systèmes temps réel *stricts*, qui doivent impérativement compléter chaque unité de travail demandée avant leur expiration sous peine de dysfonctionnement critique du système. Ces systèmes s'appuient sur des modèles mathématiques décrivant les actions à accomplir par le système.

Tâches et jobs

Les systèmes temps réel sont conçus pour réaliser des actions dans certaines limites temporelles. En toute généricité, ces actions sont uniques au sein des systèmes, et sont désignées en anglais par le terme *job* (utilisé dans la suite du document faute d'un équivalent français adéquat). Les *jobs* ont au minimum une action qui leur est associée, une date à partir de laquelle il est possible de commencer à réaliser l'action (appelée *release date*), une durée (appelée *duration*), et une échéance (appelée *deadline*). Cependant, les actions à réaliser par les systèmes temps réels sont rarement unique en pratique : les systèmes peuvent être amenés à répéter certaines actions (ou *job*) tout au long de leur fonctionnement. On parle alors d'une *tâche* (ou *task* en anglais), produisant un nouveau job unique à chaque fois que l'action doit être répétée. Par exemple, un système temps réel pourrait être amené à vérifier périodiquement la valeur produite par une sonde pour vérifier son bon fonctionnement : on parlerait alors de tâche de vérification des valeurs de la sonde ; chaque vérification indépendante de la valeur produisant un *job*.

Modèles de tâches

Les tâches d'un système temps réel sont traditionnellement décrites par un ensemble de trois modèles mathématiques différents. Les tâches *périodiques* permettent de représenter les tâches qui doivent produire un nouveau job après une période de renouvellement qui leur est propre. Les tâches *sporadiques* permettent de représenter les tâches qui peuvent produire un nouveau job à un moment aléatoire, mais qui ne peuvent produire un nouveau job qu'après un certain délai d'attente. Les tâches *apériodiques* sont des tâches qui peuvent produire un nouveau job à un moment aléatoire, sans délai particulier.

Vérification du respect des échéances

La vérification du respect des contraintes temporelles s'effectue *avant* la mise en fonctionnement du système temps réel. Lors de la conception du système, un modèle du fonctionnement du système doit être créé, associant une durée (ou une borne supérieure associée à la durée) à chaque action réalisée par le système, ainsi qu'un modèle spécifiant à quel moment chaque action doit être effectuée. Une fois ce modèle créé, il doit faire l'objet d'une *analyse d'ordonnançabilité*, vérifiant que chaque action entreprise pourra se terminer dans le temps imparti.

Lorsque l'analyse d'ordonnançabilité a été effectuée, et qu'elle atteste qu'il sera toujours possible d'ordonnancer les différents jobs du système en respectant les échéances, il existe deux méthodes permettant d'ordonnancer les jobs. La première méthode consiste à précalculer l'ordonnancement des jobs, et à l'inclure de manière statique dans le système. Cette méthode de calcul hors ligne (ou *offline* en anglais) du plan d'ordonnancement, a pour avantages d'affranchir le système temps réel du coût de l'ordonnancement "en direct". Le système n'aura pas à choisir lui même la prochaine tâche à effectuer ; elle lui a été précalcu-

lée. Cependant, il n'est pas toujours possible de savoir à chaque instant du fonctionnement d'un système temps réel quels seront les jobs à exécuter par exemple dans le cas de tâches sporadiques ou apériodiques. En effet, certaines actions des sytèmes temps réels peuvent être liées à des stimuli externes, comme par exemple l'action de maintenir l'assiette d'un avion en vol lorsqu'il tangue sous l'effet de turbulences. Dans ce genre de cas, le système temps réel doit pouvoir s'accomoder de telles variations, et en conséquence le plan d'ordonnancement doit pouvoir être modifié pendant le fonctionnement du système. Il n'est alors plus possible de précalculer le plan d'ordonnancement ; le système temps réel doit intégrer un ordonnanceur pour faire face à ces aléas. Il convient alors de vérifier aussi les résultats produits par l'ordonnanceur.

Exemples de politique d'ordonnancement temps réel

On peut distinguer deux types de politiques dans les politiques d'ordonnancement temps réel : les politiques à priorité fixe, c'est à dire dont l'ordre des priorités est préétabli et n'évolue pas avec le temps, et les politiques à priorité dynamiques, dont l'ordre des priorités est recalculé tout au long de l'exécution. On retrouve dans les politiques à priorités fixes l'ordonnancement *rate monotonic* [LL73], qui associe la plus haute priorité aux tâches dont la période est la plus courte. La politique d'ordonnancement *deadline monotonic* [Aud+90] fait aussi partie des politiques d'ordonnancement à priorité fixe et associe la plus haute priorité aux tâches dont l'échéance relative est la plus courte. La politique d'ordonnancement *rate-monotonic* est la politique à priorités fixes la plus efficace pour des tâches strictement périodiques ; le pire taux d'utilisation du processeur étant d'environ 69% sur des ensembles de tâches particuliers, et en moyenne de 88%. La politique *deadline monotonic* est quand à elle optimale sur un ensemble de tâches périodiques ou sporadiques [LW82].

Dans les politiques d'ordonnancement temps réel à priorité dynamique, on retrouve notamment la politique d'ordonnancement *Earliest Deadline First* [LL73], qui priorise les processus ayant actuellement l'échéance la plus proche. La politique *Earliest Deadline First* est optimale sur les tâches périodiques et apériodiques, pouvant atteindre 100% de taux d'utilisation du processeur contrairement aux politiques d'ordonnacement à priorité fixe [But11].

2.3 Preuve de code

Cette section va décrire les notions et travaux nécessaires à la compréhension des contributions sur la preuve formelle de code décrits dans les chapitres suivants. La première sous-section décrira le processus de vérification automatique de preuves, en décrivant d'abord le processus de raisonnement automatique et introduisant les notions d'axiomes, d'hypothèses, ainsi que le déroulement de la preuve avec les règles d'inférence. Cette description sera ensuite illustrée par l'exemple de l'assistant de preuve Coq, outil de l'état de l'art

que j'ai utilisé pour mes contributions. Enfin, la sous-section décrira les deux méthodes principales de raisonnement : la preuve directe et la preuve par raffinement.

La seconde sous-section décrira plus spécifiquement le processus de raisonnement formel sur du code. Elle commencera par discuter du raisonnement sur des programmes impératifs, et en particulier de la *logique de Hoare*. En seconde partie de cette sous section seront décrits les aspects concernant la représentation du programme dans l'assistant de preuve, ainsi que le principe des *monades*, permettant de capturer les effets de bords des programmes impératifs dans les langages fonctionnels des assistants de preuve.

Enfin, la dernière sous-section décrira les travaux les plus reconnus concernant l'application du raisonnement formel sur les systèmes d'exploitation, en finissant par le noyau développé au sein de l'équipe.

2.3.1 Vérification automatique d'une preuve

Qu'est ce qu'une preuve formelle ?

Une preuve ou une démonstration formelle se place dans le cadre d'un système formel, qui définit les règles permettant de formuler des propositions mathématiques valides ainsi que les règles de transformations pouvant s'appliquer aux propositions. Une démonstration est l'ensemble des transformations effectuées sur une proposition initiale pour la transformer en une proposition finale. Cette proposition finale peut porter plusieurs noms suivant l'importance du résultat obtenu : *proposition*, *lemme*, ou encore *théorème*. Une preuve est correcte si chaque transformation fait partie des règles de transformation (ou *règles d'inférence*) du système formel dans lequel se place la démonstration. L'utilisation d'un système formel permet la vérification automatique des preuves par un ordinateur.

Axiomes Tout système formel inclus des *axiomes* qui sont des propositions qui ne peuvent pas être prouvées, et qui servent de point de départ au raisonnement au sein de ce système formel. Pour qu'un système formel soit pertinent, il faut que ses axiomes n'amènent pas à une *contradiction*.

Hypothèses Les hypothèses sont des propositions qui n'ont pas été prouvées. Certaines pourraient être prouvées, et d'autres sont synonymes d'axiomes. Les hypothèses peuvent servir à établir des preuves d'autres propositions. Cependant, toute démonstration utilisant une hypothèse n'est pas complète tant que l'hypothèse elle même n'a pas été prouvée, contrairement aux axiomes qui n'ont pas vocation à être prouvés. Les hypothèses peuvent par exemple servir à structurer les longues preuves.

Raisonnement Le raisonnement au sein d'une preuve est déroulé grâce à l'application de *règles d'inférence*. Une règle d'inférence s'applique sur une ou plusieurs propositions initiales appelées *prémisses* et crée une nouvelle proposition en retour qu'on appelle la *conclu-*

sion. Les règles d'inférence sont traditionnelement écrites verticalement, mais peuvent aussi être écrites horizontalement comme suit :

$$\text{prémisses} \vdash \text{conclusion}$$

Le symbole $\vdash$ est appelé le symbole de dérivation signifiant "démontre". Prenons par exemple la règle d'inférence du *modus ponens*. Elle serait alors écrite :

$$A, \quad A \implies B \quad \vdash \quad B$$

Lien entre preuve et véracité de la proposition finale Il est important de distinguer le fait qu'une preuve soit correcte et la valeur de vérité de la proposition finale. En effet, la valeur de vérité d'une proposition résultant d'une démonstration est liée à la valeur de vérité des prémisses. Entre d'autres termes, toutes les hypothèses utilisées dans la démonstration doivent être valides pour que la proposition finale soit valide.

Assistants de preuve

Les assistants de preuve sont des logiciels conçus sur un système formel permettant de décrire propositions et raisonnements mathématiques. Ces propositions et raisonnements sont établis *manuellement* au travers d'un langage propre à l'assistant de preuve. La fonction principale de l'assistant de preuve est de *vérifier* le déroulé de la preuve depuis les prémisses du raisonnement jusqu'à sa conclusion. En d'autres termes, il va vérifier que l'application successive des règles d'inférence sur les prémisses conduit en effet à la conclusion décrite par l'utilisateur.

Néanmoins, certains assistants de preuve sont parfois capables de trouver automatiquement les termes de preuve permettant d'arriver à la conclusion. Les termes de preuve capables d'être déterminées automatiquement varient entre les différents assistants de preuve. Certains assistants de preuve proposent par exemple des heuristiques permettant de trouver des termes ; certains pouvant même aller jusqu'à trouver certains termes de preuve de manière systématique par construction du langage.

Il existe de nombreux assistants de preuve basés sur des systèmes formels différents. On pourra par exemple citer Coq [Coq+], Isabelle/HOL [NWP02], F* [RI], Agda [BDN09] ou encore Lean [Mou+15].

L'assistant de preuve Coq L'assistant de preuve Coq [Coq+] est un logiciel open-source français initialement développé à l'INRIA permettant de vérifier automatiquement des preuves. Le langage de Coq est Gallina, un langage fonctionnel proche d'OCaml permettant de décrire à la fois preuves, programmes, et prédicats. En effet, Coq repose sur le calcul des constructions : un système formel dérivé du lambda calcul et utilisant l'isomorphisme de Curry-Howard [Cur+58 ; Cur34 ; How80] pour établir un lien entre les démons-

trations et les programmes. Ainsi, le calcul des constructions s'inscrit dans la lignée des logiques intuitionnistes formant des preuves *constructives*.

Logique intuitionniste et preuves constructives La logique intuitionniste et les preuves constructives rejettent le principe de *tiers exclu* qui stipule que soit une proposition est vraie, soit sa négation est vraie. En particulier, il n'est pas possible d'établir une preuve d'une proposition par l'absurde, c'est à dire en montrant que la négation d'une proposition aboutie à une contradiction ($\neg P \implies \bot \vdash P$). Cette règle, uniquement valide en logique classique est à distinguer de la règle de réfutation ($P \implies \bot \vdash \neg P$). Cette règle, valide à la fois en logique classique et en logique intuitionniste, peut par exemple être utilisée pour prouver que $\sqrt{2}$ n'est pas rationnel. Pour en faciliter la compréhension, certaines propositions en logique intuitionniste peuvent être interprétées d'une manière différente de la logique classique. Par exemple, en logique intuitionniste :

- A se lit « A est prouvable »
- $\neg A$ se lit « A est contradictoire »
- $\exists x, A(x)$ se lit « On peut exhiber un élément x tel que $A(x)$ est prouvable »
- $\forall x, A(x)$ se lit « Pour n'importe quel élément x, $A(x)$ est prouvable »

Stratégie de conduite de preuve

Tout comme la conception de logiciel, la conduite de preuve nécessite parfois un effort d'ingénierie pour structurer la preuve. Savoir si une preuve mérite d'être structurée relève de l'expertise : on parle alors de génie de la preuve. Il existe deux méthodes principales permettant de compléter la preuve d'une proposition : la preuve directe et la preuve par raffinement.

Preuve directe La méthode de preuve directe est à opposer à la méthode de preuve par raffinement. Elle consiste à utiliser les lemmes et théorèmes directement fournis dans et par le langage pour arriver à la preuve finale. De plus, la preuve directe permet de structurer la preuve au travers de *lemmes* ou *théorèmes* intermédiaires pour factoriser et rendre plus accessible les éléments de preuve redondants. La méthode de preuve directe est utilisée pour la majorité des preuves concernant le noyau Pip développé dans l'équipe. La principale force de la preuve directe est qu'elle se contente de prouver uniquement la propriété finale : chaque proposition intermédiaire ne dépasse pas le cadre de la preuve initiale. La contrepartie d'une telle approche est qu'il est fastidieux de réutiliser des portions de cette preuve pour les appliquer à une autre conclusion.

Preuve par raffinement La preuve par raffinement [DEB98] est plutôt employée lorsque les étapes de la preuve à conduire ne sont pas immédiates, souvent de par la complexité de la preuve. La méthode de preuve par raffinement consiste à appliquer la stratégie « diviser pour régner » en trouvant des abstractions intermédiaires permettant de découper

la preuve en morceaux indépendants, ce qui permet par ailleurs de rendre la preuve modulaire. Ces avantages viennent au prix d'un effort d'abstraction supplémentaire non requis par la preuve directe. Certaines de ces abstractions intermédiaires peuvent néanmoins avoir été définies par des travaux préliminaires, réduisant le coût de recherche des abstractions. Par ailleurs, nous défendons la thèse que la preuve par raffinement et l'utilisation d'abstractions de manière générale éloigne la preuve conduite de l'objet d'étude initial, et mène plus facilement à l'oubli de certaines contraintes. On pourrait notamment citer le paradoxe du raffinement [Jac88 ; Mor06].

De nombreux travaux munis de preuves formelles sur les systèmes d'exploitation utilisent le raffinement. CertikOS et seL4 en sont les exemples les plus connus. Le projet CompCert, compilateur de code source C garantissant la préservation de la sémantique lors de la compilation, utilise aussi le raffinement pour montrer que le processus de compilation est correct.

2.3.2 Preuve de programme

Cette sous-section traite de la manière de raisonner sur les programmes, en particulier sur les programmes impératifs dont le paradigme est à priori incompatible avec les langages fonctionnels des assistants de preuves tels que Coq et son langage Gallina. La logique de Hoare est une réponse classique à ce problème, permettant de raisonner sur les programmes séquentiels. Les monades d'état sont un moyen de représenter des programmes séquentiels dans les langages fonctionnels utilisés par les assistants de preuve, et servent d'interface entre le monde mathématique et le monde réel. Cependant, prouver des propriétés sur du code source ne garantit pas que ces propriétés seront préservées lors de la compilation ; des projets tels que CompCert tentent de répondre à ce problème.

Logique de Hoare

La logique de Hoare est un système formel permettant de raisonner sur les programmes séquentiels. La logique de Hoare raisonne sur des *triplets de Hoare* qui sont définis comme suit :

$$\{P\}\ c\ \{Q\} \tag{2.1a}$$

où :

- P représente les *préconditions* qui sont les propriétés sur l'état de la machine supposées prouvables *avant* l'exécution du programme à vérifier - ses *prémisses*.
- c représente le code du programme à vérifier.
- Q représente les *postconditions* qui sont les propriétés sur l'état de la machine supposées prouvables *après* l'exécution du code c du programme.

Par définition, un triplet de Hoare est *prouvable si et seulement si*, quel que soit l'état du système initial E_0 satisfaisant les propriétés P alors c fait entrer le système dans un nouvel état E_n dans lequel les propriétés Q sont prouvables. Si le nouvel état E_n engendré par c viole les *postconditions* Q, alors le triplet $\{P\}\ c\ \{Q\}$ est contradictoire. Au travers du prisme intuitionniste, on pourrait lire le triplet de Hoare $\{P\}\ c\ \{Q\}$ comme « Si les propriétés P sur l'état de la machine sont supposées prouvables avant l'exécution du code du programme c, alors les propriétés Q sur l'état de la machine après exécution de c sont prouvables ».

Pour garantir formellement que c respecte les propriétés qu'on souhaite garantir sur le système, on peut décomposer le code du programme c en parties c_i aussi élémentaires que souhaité.
Le triplet devient alors :

$$\{P\}\ c_1; c_2; \dots; c_n\ \{Q\} \tag{2.1b}$$

On pourrait par exemple considérer que ces parties élémentaires sont les instructions assembleur de la machine.

La logique de Hoare nous permet alors de décomposer la preuve en observant un enchaînement d'états intermédiaires E_i résultant de l'exécution des instructions c_i.

$$E_0 \xrightarrow{c_1} E_1 \xrightarrow{c_2} \dots \xrightarrow{c_n} E_n \tag{2.1c}$$

Chaque instruction c_i amène de nouvelles propriétés sur l'état E_i nouvellement créé et permet de créer des triplets de Hoare intermédiaires. Ces nouvelles propriétés servent à la fois de postconditions Q_i pour le triplet concernant l'instruction c_i et de préconditions pour le triplet de l'instruction suivante.

$$\{P\}\ c_1\ \{Q_1\}, \{Q_1\}\ c_2\ \{Q_2\},\ \dots\ , \{Q_{n-1}\}\ c_n\ \{Q_n\} \tag{2.1d}$$

Ainsi, au fur et à mesure de l'exécution du programme et de l'enchaînement des états, les propriétés sur l'état de la machine changent et s'étoffent. Pour montrer que les postconditions Q souhaitées sur l'état final E_n sont prouvables, il faut montrer qu'elles sont prouvables si les postconditions Q_n résultant de l'exécution de c sont prouvables, soit $Q_n \implies Q$.

Modèle de la machine et sémantique opérationnelle Pour pouvoir raisonner grâce à la logique de Hoare, un modèle de la machine et des instructions élémentaires doit être établi. Ces modèles permettent de décrire un système de transition d'état, donnant une signification formelle au programme. Cette sémantique du programme, décrivant des états successifs de la machine, est appelée *sémantique opérationnelle*.

Ces modèles sont des *axiomes* de la logique de Hoare. Par ailleurs, il est important de gar-

der à l'esprit que ces modèles – que l'homme de l'art tient pour vrai – ne sont que des projections subjectives de l'état et des instructions réelles dans le monde des mathématiques et sont, en ce sens, **arbitraires**. Ces modèles, bien que nécessaires au raisonnement, ne sont pas intrinséquement équivalents aux instructions exécutées par la machine. Chaque choix de modèle pour l'étude d'un objet réel vient avec sa part d'incertitude, quelles que soient les précautions prises lors de ce choix. La preuve d'un programme n'échappe pas à cette règle : elle est aussi contestable que les modèles sur lesquels elle repose.

Langage

L'assistant de preuve Coq utilise un langage de programmation fonctionnel nommé Gallina. Étant un langage fonctionnel, il est peu adapté à l'écriture de certains programmes tels que les systèmes d'exploitation qui sont des programmes impératifs séquentiels. La sous-section précédente a introduit la logique de Hoare, permettant de raisonner sur de tels programmes. Il faut alors réussir à représenter les programmes séquentiels dans un langage fonctionnel.

Il existe deux méthodes pour y parvenir. La première consiste à représenter le langage impératif sous forme de structures de données dans le langage de l'assistant de preuve. Cette méthode est appelée le *deep embedding*. La seconde méthode consiste à utiliser le langage de l'assistant de preuve pour implémenter directement les élements logiciels à prouver ; on parle alors de *shallow embedding*.

Deep embedding Le deep embedding permet de représenter un langage cible en modélisant à la fois la syntaxe et la sémantique de ce langage. Les programmes sont ensuite étudiés sous la forme d'un *AST* (pour *Abstract Syntax Tree*), qui est un arbre de leur syntaxe. Le deep embedding permet de ce fait d'exprimer des propriétés sur la structure du programme. Ces propriétés sont parfois intéressantes, notamment pour procéder à la preuve de propriétés concernant la compilation du langage ciblé. Le projet CompCert utilise un deep embedding de C pour garantir la préservation de la sémantique lors de la compilation vers de l'assembleur.

Shallow embedding Le shallow embedding au contraire ne définit pas la syntaxe du langage cible, et se contente de ne définir que la sémantique des programmes à prouver. Cette approche est plus légère, mais ne permet cependant pas de prouver de propriétés relatives à la structure du programme. Un shallow embedding se sert de la syntaxe du langage hôte.

Monade d'état Les langages purement fonctionnels n'ont pas de notion d'état. Ainsi pour modéliser fidèlement des programmes impératifs et leurs effets de bords sur l'état de la machine, on peut le simuler. Une monade d'état est toute indiquée pour implémenter des procédures impératives dans un langage fonctionnel : elle permet en quelque sorte

d'« enrober » des valeurs avec un état. Les fonctions ainsi créées, imitant les procédures impératives, manipulent ces valeurs « enrobées ». Elles indiquent comment produire la valeur de retour mais aussi quelles modifications éventuelles sont apportées à l'état. Ces fonctions sont dites *monadiques*.

Cette construction explicite l'état implicite des programmes impératifs qui transite de fonction en fonction de manière transparente grâce à la fonction `bind`. L'appel à la fonction bind est rendu transparent dans le code grâce à du sucre syntaxique. En plus de la fonction `bind`, chaque monade nécessite un élément neutre (ou *unit*). Les effets des deux fonctions sont expliqués ci-après.

Il est ainsi possible de définir une monade d'état telle que la monade d'état `LLI` présentée en listing 2.1. Tout d'abord, elle comprend un type `state` représentant l'état de la machine qui transitera de fonction en fonction. Elle définit aussi un type `result` représentant soit une valeur `val` soit un comportement indéfini `undef`.

```
Definition LLI (A :Type) : Type := state → result (A * state).

Definition ret {A : Type} (a : A) : LLI A :=
  fun s ⇒ val (a , s) .

Definition bind {A B : Type} (m : LLI A)(f : A → LLI B) : LLI B :=
  fun s ⇒ match m s with
           | val (a, s') ⇒ f a s'
           | undef a s' ⇒ undef a s'
           end.
```

Figure 2.1 – Définition de la monade d'état avec ses fonctions `bind` et `ret` en Coq

Ces types sont utilisés pour définir la monade d'état `LLI`, présentée en figure 2.1. La monade `LLI` est définie comme une fonction sur des types, prenant un type `state` en argument, et renvoyant une valeur de retour de type `result` composé d'un type arbitraire `A` et d'un `state`. Les fonctions partielles sur l'état renvoyant un type `LLI` sont donc *monadiques*. La monade `LLI` « enrobe » des valeurs quelconques avec le type `state`.

Lorsqu'on lui donne une fonction monadique `f` de `A` vers `LLI B` et une valeur monadique de type `LLI A`, la fonction `bind` permet de récupérer la valeur « enrobée » `A` et de lui appliquer la fonction `f`, produisant une valeur monadique `LLI B`. La fonction `ret` sert d'élément neutre, qui à partir d'un état et d'une valeur, renvoie cette même valeur « enrobée » avec l'état.

La littérature scientifique propose des définitions plus formelles des monades. Celle de Saunders Mac Lane [Mac71, p. 134], associée à la théorie des catégories pourrait être traduite comme suit :

« Une monade dans X est un simple monoïde dans la catégorie des endofoncteurs de X,

ayant pour produit × la composition d'endofoncteurs et pour élément neutre l'endofoncteur identité. »

Compilation et préservation de la sémantique

Une problématique subsiste lorsque des propriétés formelles ont été prouvées sur du code source. Une fois compilé, il reste à établir que les propriétés resteront valables sur le code produit. La recherche dans ce domaine a mené au projet CompCert[Ler09], un compilateur de code source C dont la préservation de la sémantique est garantie formellement jusqu'au code assembleur.

La propriété de préservation de la sémantique est énoncée comme suit :

▸ **Théorème 2.1.** Pour tout programme source S et pour tout code C généré par le compilateur, si le compilateur a produit le code C à partir du programme source S sans remonter d'erreur de compilation, alors le comportement observable de C est l'un des comportements observables possibles de S. ◂

CompCert compile le programme source S à partir de son *Abstract Syntax Tree* (ou *AST*) issu de la passe du préprocesseur, de l'analyseur syntaxique et de la phase de vérification des erreurs de type (*type checking*). CompCert produit le code C sous la forme d'un *AST* du langage assembleur ciblé.

À ce jour, le projet CompCert a prouvé 90% de la chaîne de compilation, notamment les algorithmes d'optimisation et de génération de code assembleur. Les 10% restants à prouver incluent l'assemblage et la phase d'édition des liens [PL].

Les projets ayant recours à la vérification de programme par *deep embedding* ont l'avantage de pouvoir aisément reconstruire l'*AST* du programme et de le compiler, notamment grâce à CompCert. Cependant, pour les projets utilisant un *shallow embedding* et n'explicitant pas l'*AST* du programme vérifié, l'étape de reconstruction du programme n'est pas triviale.

Digger [HO] est un outil écrit en Haskell qui transforme le code shallow embedded écrit en Gallina en code source C. Digger n'est pas muni de preuve formelle de préservation de la sémantique. Cet outil ne permet pas de prouver l'équivalence entre le code shallow embedded et le code finalement exécuté.

∂x [Hym ; Yua+22], un autre outil récemment paru, permet de transformer le code shallow embedded écrit en Gallina en un AST attendu par CompCert. *∂x* comporte deux phases de fonctionnement : la première phase consiste à extraire l'*AST* du programme écrit en Gallina sous la forme d'une représentation intermédiaire ; la seconde transforme cet *AST* en *AST* CompCert `Csyntax`. La première phase est écrite en Elpi [Dun+15] car Coq ne propose pas nativement de mécanisme de *réflexion*. La seconde phase est écrite directement dans Coq. Il est de ce fait désormais possible de raisonner sur la phase de transformation de l'*AST*. *∂x* permet en outre d'afficher le code source C produit, en utilisant le *pretty-printer* de CompCert.

2.3.3 Illustration système

seL4

seL4 [seL22b] est un noyau de système d'exploitation de la famille des noyaux L4 pour de nombreuses architectures (Armv6, Armv7, Armv8, x86, x86_64 et RISC-V RV64) [seL22a]. seL4 propose des mécanismes de gestion de la mémoire virtuelle, de gestion des interruptions, de communication inter-processus (*IPC*) qui reposent sur un système gestion des droits par *capacités* [Lev84].

seL4 offre en outre, pour certaines plateformes, une vérification formelle de son implémentation. Cette vérification peut inclure :

- une preuve formelle fonctionnelle du code C (c'est à dire une preuve que le code respecte sa spécification) [Kle+09]
- une preuve formelle de la propagation de la preuve jusqu'au binaire exécutable[SMK13]
- une preuve du maintien de l'*intégrité* et de la *confidentialité* [Sew+11]
- des propriétés temps-réel notamment sur le respect de bornes sur le temps d'exécution (*WCET*)[SKH17]

Pour arriver au code C exécutable, seL4 définit tout d'abord une spécification abstraite, définissant la fonction du code à produire. Ensuite, un prototype en Haskell est implémenté, respectant à priori cette spécification. Ce prototype permet de générer automatiquement une spécification exécutable, définissant comment le code doit remplir sa fonction. L'implémentation réelle en C est réalisée manuellement et doit respecter la spécification exécutable.

La preuve fonctionnelle de code dans le noyau seL4 est donc naturellement découpée en un raffinement en trois couches : la spécification abstraite, la spécification exécutable et l'implémentation réelle. La première étape pour établir une preuve fonctionnelle est de montrer que la spécification exécutable dérivée du prototype Haskell raffine la spécification abstraite. La seconde étape est de montrer que l'implémentation en C raffine la spécification exécutable [Kle+09].

CertiKOS

CertiKOS [Cer22] est un outil d'aide à la conception de noyau de système d'exploitation. CertiKOS propose une conception du noyau par de multiples couches d'abstractions propices au raffinement.

Plus particulièrement, CertiKOS utilise le concept de *raffinement contextuel*. En quelques mots, la méthodologie de CertiKOS consiste à définir des triplets (L_1, M, L_2) représentant chacun une couche d'abstraction définissant une interface. L_2 représente l'interface que l'on souhaite certifier. M représente l'implémentation de l'interface L_2 s'appuyant sur l'interface sous-jacente L_1. L'idée de la méthologie de CertiKOS est que chaque couche d'abstraction L_2 est assez précise pour capturer tous les comportements observables de

l'implémentation M. Ce type de spécification est appelé *spécification profonde*. Ainsi, une fois que la preuve que M implémente l'interface L_2 a été établie, il est possible de raisonner exclusivement sur L_2 sans jamais avoir à raisonner sur M de nouveau [Gu+15].

CertiKOS a certifié mCertiKOS, un noyau de système d'exploitation et hyperviseur, capable de faire tourner Linux et divisé en 40 couches d'abstraction [Gu+11]. Plus récemment, mC2, un noyau de système d'exploitation concurrent [Gu+16 ; Gu+19] a été présenté à la communauté.

Pip

Pip est un noyau de système d'exploitation *minimal* dont le seul but est la gestion de portions isolées de mémoire appelées *partitions*, et du transfert de flot d'exécution entre ces partitions ; le reste des composants du système pouvant être implémenté en dehors du noyau. Pip est muni d'une preuve formelle garantissant que ses appels systèmes ne brisent pas *l'isolation* des partitions. Pip a été initialement conçu pour garantir l'isolation au travers de l'utilisation de mémoire virtuelle configurée par une *MMU*. Cependant, de récents travaux [DGG22] ont fait évoluer Pip pour qu'il garantisse l'isolation de partitions sans utiliser de mémoire virtuelle sur les systèmes plus modestes disposant d'une *MPU*.

Dans Pip, les partitions forment une structure arborescente qui détermine les droits d'accès à la mémoire, comme illustré avec la figure 2.2. Au démarrage du système, une partition responsable de la totalité de la mémoire du système est créée. Pip permet à chaque partition de créer une autre partition en partageant une partie de sa propre mémoire avec la partition nouvellement créée. La nouvelle partition est appelée *partition enfant*, la partition ayant partagé sa propre mémoire est appelée *partition parent*. Cette relation parent/enfant crée la structure arborescente : les enfants pouvant créer à leur tour de nouvelles partitions en partageant leur mémoire.

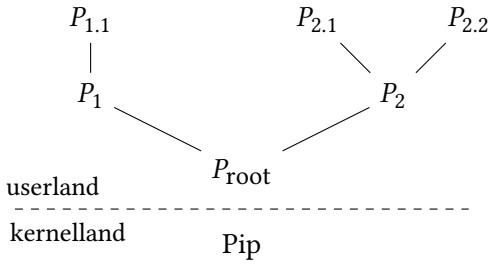

Figure 2.2 – Exemple d'arbre de partition dans Pip

Propriété d'isolation La propriété d'isolation de Pip est divisée en trois sous-propriétés :

— La première, la propriété de **partage vertical**, stipule que la mémoire partagée par une partition parent avec ses enfants reste accessible au parent par conception.

— La seconde, la propriété d'**isolation horizontale**. Dans Pip, chaque partition peut créer plusieurs partitions enfant ; cependant les portions de mémoire partagées avec chacune doivent être strictement disjointe. Autrement dit, une partition ne peut pas partager une même portion de mémoire avec deux partitions enfant simultanément. Ainsi, deux partitions enfant issues d'une même partition parent sont *isolées* : la mémoire accessible dans l'une d'entre elle est nécessairement inaccessible dans l'autre.

— La dernière, la propriété d'**isolation noyau**. À chaque création de partition, Pip réserve une petite portion de mémoire afin d'y stocker les structures nécessaires au contrôle des droits. Ces portions de mémoire deviennent inaccessibles à n'importe quelle partition.

Méthodologie de preuve Pip a pris le contrepied des projets majeurs du domaine en utilisant un *shallow embedding* de C plutôt que d'utiliser un *deep embedding*. Ce *shallow embedding* nécessite l'utilisation d'une monade d'état en Coq (voir 2.3.2) et d'un outil spécifique (voir *∂x* 2.3.2) afin de reconstruire l'AST ou le code source du noyau pour produire un binaire exécutable. Le pari de cette méthodologie est de pouvoir se concentrer sur la sémantique des programmes à prouver afin d'alléger l'effort de preuve général par rapport aux méthodologies utilisant un *deep embedding*. Par ailleurs, les propriétés de préservation de l'isolation de Pip ont été prouvées directement plutôt que par raffinement.

2.4 Relation entre l'état de l'art et les chapitres de contribution

Ceci achève l'état de l'art présenté dans ce document. Les prochains chapitres décrivent les contributions de cette thèse, qui sont liés aux sujets présentés dans ce chapitre.

En particulier, les notions relatives au transfert de flot d'exécution ainsi que le fonctionnement des droits et des élévations de privilèges sur l'architecture Intel x86 sont exploitées dans le chapitre suivant par la première contribution, présentant le service de transfert de flot d'exécution de Pip sur l'architecture Intel x86. Ce chapitre se sert aussi des notions sur la preuve de propriétés formelles sur du code présentées dans la section 2.3, afin de décrire le déroulement de la preuve de préservation de l'isolation sur ce service.

Le second chapitre de contribution, décrivant un ordonnanceur *Earliest Deadline First* dont la fonction d'élection est prouvée correcte, repose sur les notions d'ordonnancement présentées en section 2.2. Ce chapitre se sert aussi de certaines notions décrites dans la section 2.3 pour relater le déroulement de la preuve de correction de la fonction d'élection.

Le dernier chapitre de contribution, décrivant une évolution de l'architecture de Pip séparant la définition des modèles d'isolation du code des services de Pip, repose intégralement sur la section 2.3.

3 Service de transfert de flot d'exécution avec preuve d'isolation

Ce chapitre décrit la première contribution de cette thèse : un service de transfert de flot d'exécution pour Pip. Ce chapitre commencera par exposer les motivations qui ont conduit à ce service de transfert de flot d'exécution.

La seconde section décrira le service tel qu'il a été conçu : en premier lieu, nous exposerons le principe général derrière le service, en explicitant notamment les structures de données et le prototype du service. Cette exposition du service sera suivie d'une illustration de l'utilisation du service sur les trois différents transferts de flot d'exécution au sein d'un système : les appels systèmes entre différents espaces d'adressages, ainsi que les transferts de flot d'exécution suite à une faute ou une interruption. Cette section s'achèvera sur une vue interne du service, décrivant les différents blocs unifiant ces trois différents transferts.

La troisième section expliquera le processus de preuve du service, en commençant par la définition des types nécessaires à l'écriture du service et plus généralement de la conception des ajouts à l'interface avec la monade. Cette section détaillera ensuite les différentes propriétés d'isolation, puis identifiera les points délicats de l'établissement de la preuve en s'appuyant sur les différents blocs détaillés dans la section précédente.

La dernière section de ce chapitre reviendra sur la conception de ce service d'un point de vue pragmatique, en s'intéressant à quelques métriques et en revenant sur la pertinence de la preuve.

3.1 Motivations

Changement d'espace d'adressage

Une première motivation concernant ce nouveau service de transfert de flot d'exécution est de vérifier le changement d'espace d'adressage lorsque le flot d'exécution passe d'une partition à une autre. Le modèle d'isolation de l'état de la machine dans Pip contient une variable globale `currentPartition` indiquant la partition qui s'exécute à l'intérieur des portions de code non privilégiées. Cette variable permet aussi de déterminer quel espace d'adressage est actuellement en vigueur au sein du modèle de Pip.

Lors d'un transfert de flot d'exécution, cette variable doit être mise à jour, ne serait-ce que pour refléter le nouvel espace d'adressage effectif au sein du modèle. Cependant ce comportement ne peut être reflété qu'en modélisant le service de transfert de flot d'exécution. Cette modélisation, ainsi que la preuve que la variable `currentPartition` est correctement mise à jour, viennent compléter la preuve de préservation de l'isolation de Pip.

Vérification de lecture ou d'écriture au titre d'une partition arbitraire

Un service de transfert de flot d'exécution doit pouvoir charger un contexte d'exécution d'une quelconque manière, afin d'exécuter le code ciblé. Similairement, il doit être capable de sauvegarder un contexte d'exécution afin de pouvoir reprendre l'exécution du code appelant au besoin. Dans le cadre de transferts de flot d'exécution entre partitions de Pip, ces contextes d'exécution sont placés dans l'espace d'adressage des partitions ; les partitions indiquent à Pip l'emplacement de sauvegarde ou de lecture de ces contextes. Or, le service de transfert de flot d'exécution est un morceau de code du noyau, et s'exécute en mode *privilégié*. Il a donc les droits suffisants pour lire et écrire dans les pages de l'espace d'adressage réservées au noyau.

De ce fait, ce service doit s'assurer que les lectures et écritures de contextes qu'il entreprend au nom de partitions n'outrepassent pas leurs droits de lecture et d'écriture. De plus, le service doit effectuer ces vérifications en manipulant les espaces d'adressage de partitions qui ne sont pas nécessairement la partition courante. Ces vérifications font de ce service un service particulier, qui est le seul à avoir de telles considérations.

Mise à l'épreuve de la méthodologie de Pip

De nombreux modes de transfert de flot existent au sein d'un système. Un appel système interrompt le flot d'exécution de l'appelant, sauvegarde son contexte, et charge le contexte de la fonction du noyau. Le retour d'un appel système charge le contexte précédemment sauvegardé de l'appelant, en oubliant le contexte d'exécution de la fonction du noyau. Les interruptions matérielles interrompent le flot d'exécution courant, sauvegardant le contexte d'exécution du flot interrompu, et chargeant le contexte de traitement de l'interruption. Les fautes interrompent le flot d'exécution fautif, sauvegardent le contexte d'exécution, et chargent le contexte d'exécution de gestion de la faute.

Ces différents modes de transferts – à priori différents – reposent en réalité sur les mêmes abstractions de sauvegarde et de chargement de contextes d'exécutions et peuvent donc être factorisés. Cette factorisation est la dernière motivation majeure, réduisant l'ensemble des transferts de flot d'exécution à un simple fil d'exécution générique. Elle s'inscrit dans le courant minimaliste de Pip et permet d'éprouver sa méthodologie de co-design – le but final de cette méthodologie étant de minimiser l'effort de preuve sur les services.

3.2 Description du service

Avant toute chose, il faut définir ce qu'on attend du service, sa spécification. En particulier, il s'agit de spécifier les transferts de flot de contrôle valides au sein du système, que ce soit pour les transferts explicites tels que les appels systèmes ou les transferts implicites comme les fautes ou les interruptions. Cette spécification doit pouvoir accomoder tous les cas d'usage de transfert de flot d'exécution au sein d'un noyau tel que Pip, conçu comme une tour de virtualisation.

Pour rappel, Pip définit des partitions de mémoire qui sont responsables de la mémoire qui leur est attribuée. Chaque partition de mémoire a son propre espace d'adressage. Ces partitions peuvent engendrer des sous-partitions, en partageant en partie de leur propre mémoire. Les sous-partitions engendrées de cette manière sont appelées les partitions enfants. La partition ayant partagé sa mémoire avec son enfant est appelée la partition parent. Au démarrage du système, une seule partition est créée par Pip. Cette partition a accès à l'intégralité de la mémoire : c'est la partition racine.

Flôts d'exécution valides au sein d'une tour de virtualisation

Les transferts de flot de contrôle valides entre les différentes partitions reprennent les trois modalités présentées dans le chapitre précédent en section 2.1 en les voyant à travers le prisme d'une tour de virtualisation.

La tour de virtualisation crée un système de délégation des fonctionnalités. L'intégralité des fonctions du système est initialement endossée par la partition racine, qui peut décider de déléguer certaines fonctionnalités à ses enfants. Les partitions enfant peuvent à leur tour déléguer ces fonctionnalités à leurs propres enfants ; la partition racine n'en a cependant pas forcément connaissance. C'est pourquoi les transferts de flot d'exécution explicites ne sont nécessaires qu'entre parent et enfants ; chaque partition connait les fonctionnalités qui lui incombent, et peut donc diriger le flot d'exécution vers une autre partition si nécessaire. **Ainsi, chaque partition offre un certain nombre de services qui définissent son interface.**

Lorsqu'une faute survient, une partition manque à ses responsabilités. La faute remonte la chaîne de responsabilité vers son parent qui peut alors gérer l'incident.

Les interruptions matérielles signalent un évènement extérieur dont la responsabilité peut incomber à n'importe quelle partition, et seule la partition racine connait l'ensemble des chaînes de responsabilité. Ainsi, lorsqu'une interruption matérielle survient, la partition racine récupère le flot d'exécution et peut – si nécessaire – diriger l'interruption vers la partition qui en a la responsabilité. Ceci est semblable à un superviseur muni d'une fonction de multiplexage.

Ainsi, même s'il existe un grand nombre de modalités de transfert de flot d'exécution en pratique, l'architecture du proto-noyau Pip promeut un modèle unifié qui adapte à une tour de virtualisation les trois situations génériques. En réduisant à trois cas distincts l'ensemble des modalités de transferts de flot d'exécution, le travail de preuve de programme nécessaire pour établir une garantie de sécurité est simplifié. Cependant, la sous-section suivante s'attache à démontrer qu'il est possible de résumer ces trois cas distincts en un seul service dont la preuve de bon fonctionnement apporte les garanties de sécurité à l'ensemble des situations de transfert de flot d'exécution possibles au sein de l'architecture x86.

3.2.1 Principe de fonctionnement du service

Structures de données du service

VIDT Les services exposés par les différentes partitions sont définis dans une structure appelée *Virtual Interrupt Descriptor Table* ou *VIDT*. Cette structure reprend les concepts de l'*IDT* classique (voir 2.1.2), appliqués à chaque partition. Elle doit être placée - par convention - au début de la dernière page virtuelle de chaque partition. Cependant, contrairement à l'*IDT* qui contient des *gates* composées de pointeurs de fonctions et de contrôles de droits, la *VIDT* de chaque partition contient des pointeurs vers des *contextes* d'exécution, comme illustré sur la figure 3.1.

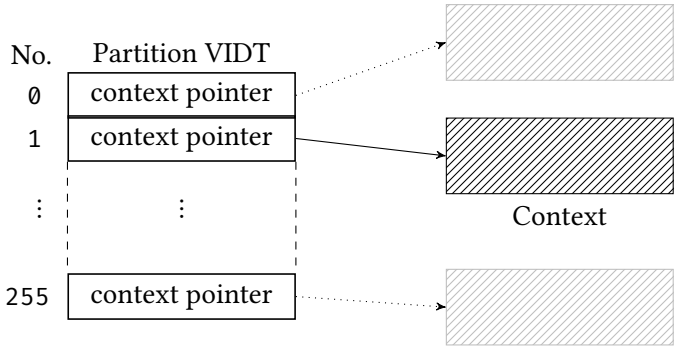

Figure 3.1 – La structure d'une VIDT

Contexte d'exécution Ces *contextes* sont des instantanés de l'état du processeur au moment du transfert du flot d'exécution. Pour l'architecture Intel x86, ils sont partiellement générés par les mécanismes du matériel tels que détaillé dans le chapitre précédent section 2.1.1, puis complétés par du logiciel. Ces contextes d'exécution peuvent aussi être créés ex-nihilo par les partitions afin de définir de nouveaux services.

Plus simplement, les services de chaque partition sont définis dans leur *VIDT* au moyen de pointeurs vers des contextes d'exécution. Il est important de noter que ces contextes d'exécution sont situés dans l'espace d'adressage de chaque partition, et sont donc **accessibles et modifiables** par le code non privilégié.

Principe d'utilisation et prototype du service pour un transfert de flot d'exécution

Lors d'un appel explicite au service de transfert de flot d'exécution dont le protoype est donné par le listing 3.1, la partition appelante doit désigner une autre partition ainsi que le numéro de service désiré. La partition est désignée par l'adresse virtuelle de son descripteur correspondant au paramètre `calleePartDescVAddr`. Dans le cas d'un appel vers la partition parent, l'adresse par défaut est utilisée. Le numéro de service n'est autre

```
yield_checks cg_yieldGlue(vaddr calleePartDescVAddr,
                          userValue userTargetInterrupt,
                          userValue userCallerContextSaveIndex,
                          interruptMask flagsOnYield,
                          interruptMask flagsOnWake);
```

Listing 3.1 – Prototype du point d'entrée du service tel qu'appelée par les partitions

que la position du pointeur vers le contexte d'exécution à restaurer dans la VIDT de la partition ciblée, correspondant au paramètre `userTargetInterrupt`. Ces deux paramètres permettent de déterminer où transférer le flot d'exécution.

De plus, Pip permet à la partition appelante de sauvegarder son contexte d'exécution actuel afin qu'il puisse être restauré et que l'exécution puisse reprendre ultérieurement. La partition appelante doit avoir réservé préalablement de la mémoire pour que Pip puisse y placer un contexte, et renseigné un pointeur vers cet espace dans sa propre VIDT. Pour que Pip préserve le contexte d'exécution, la partition doit fournir l'entier `userContextSaveIndex` qui indique la position du pointeur dans sa VIDT pointant vers l'espace réservé. Si un pointeur nul se trouve à la position indiquée, le contexte n'est pas sauvegardé.

Les deux derniers paramètres, `flagsOnYield` et `flagsOnWake` permettent à la partition de restreindre l'utilisation de certains de ses services. Ce sont en réalité des drapeaux vérifiés par le service de transfert de flot d'exécution de Pip indiquant que certains services de la partition sont temporairement indisponibles, bien qu'ils soient correctement configurés. `flagsOnYield` sont les drapeaux qui seront appliqués immédiatement par Pip à la partition appelante au moment du transfert de flot d'exécution. `flagsOnWake` sont les drapeaux qui seront appliqués au moment de la restauration du contexte d'exécution actuel de la partition.

Enfin, un dernier paramètre contenant un pointeur vers le contexte d'exécution est généré par le code trampoline permettant d'exécuter le code du service écrit en Gallina. Ceci permet au service de sauvegarder le contexte d'exécution comme énoncé précédemment. La figure 3.2 montre le prototype attendu par le code prouvé.

3.2.2 Illustration de mise en place du service sur l'architecture Intel x86 au travers d'un appel explicite

On a pris le parti d'illustrer ce service sur l'architecture Intel, cependant il n'est pas spécifique à l'architecture Intel. Le même service a notamment été intégré sur une implémentation Armv7 muni d'une MMU (Raspberry Pi) ainsi que sur un Arm Cortex muni d'une MPU.

```
Definition checkIntLevelCont (targetPartDescVAddr : vaddr)
                             (userTargetInterrupt : userValue)
                             (userSourceContextSaveIndex : userValue)
                             (flagsOnYield : interruptMask)
                             (flagsOnWake : interruptMask)
                             (sourceInterruptedContext : contextAddr)
                  : LLI yield_checks.
```

Listing 3.2 – Prototype du point d'entrée du service en Gallina

Point d'entrée par `callgate`

Dans l'implémentation de Pip sur l'architecture Intel x86, les services de Pip sont appelables au travers de *callgates* (voir 2.1.3). Ces callgates permettent au code non privilégié des partitions d'appeler les services privilégiés de Pip. Pour ce faire, la partition doit pousser les arguments décrits en section 3.2.1 sur sa pile, puis utiliser un *farcall*. Cet appel est implémenté au sein de la LibPip, la librairie utilisateur facilitant l'utilisation du noyau. Son code est disponible en annexe (voir listing A.1).

Lorsque le processeur exécute l'instruction `lcall` de la partition, le flot d'exécution est transféré vers Pip et le processeur passe en mode privilégié, copie les paramètres et pousse partiellement l'état précédent sur la pile (voir 2.1.3). Dans le cas de l'appel au service de transfert de flot d'exécution, le processeur commence par exécuter une routine qui va sauver sur la pile noyau le contexte d'exécution de la partition encore partiellement présent dans les registres. Accessoirement, cette routine réordonne les éléments de la pile afin de regrouper les différentes parties du contexte et de pouvoir utiliser une structure `gate_ctx_t` pour le représenter. Cette routine étant un peu longue, elle est placée en annexe (voir Fragment de code A.1). La figure 3.2 montre l'état de la pile après l'exécution de la routine.

Harmonisation du contexte d'exécution et appel du service prouvé Avant d'appeler le code prouvé, une dernière transformation est opérée sur le contexte d'exécution. Il est copié en haut de la pile, puis transformé en contexte générique de type `user_ctx_t` afin d'harmoniser les différentes représentations de contexte entre les différents points d'entrées du service. Le code est disponible en annexe (voir Fragment de code A.2).

Introduction générale des étapes du service

Le service écrit en Gallina permettant de transférer le flot d'exécution procède en trois étapes.

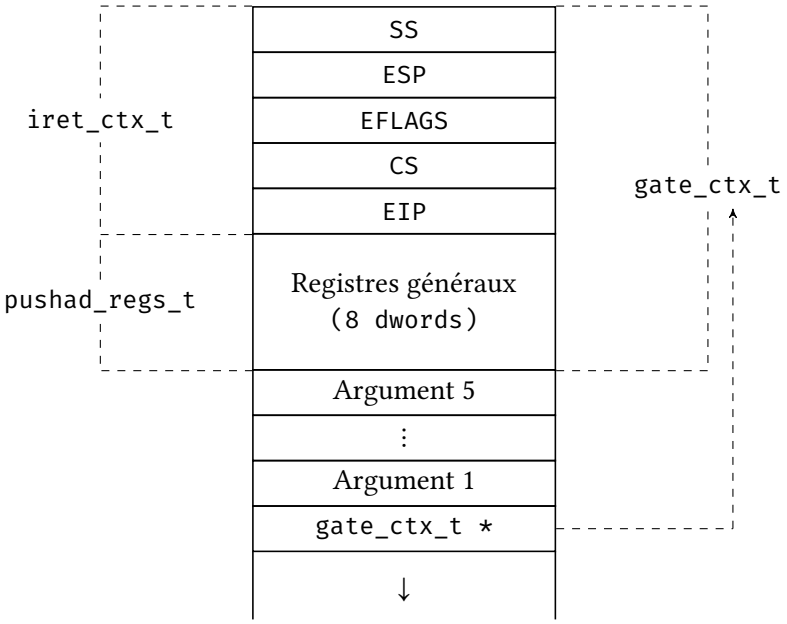

Figure 3.2 – État de la pile du noyau après la routine assembleur exécutée après l'appel du service au travers d'une callgate

Étape préliminaire de validation et de récupération des données Avant toute chose, la première étape du service vérifie la validité des arguments et des structures modifiables en espace utilisateur. En particulier, elle vérifie que l'adresse virtuelle fournie comme la cible du transfert correspond bien à une partition enfant ou parent, et récupère l'adresse réelle à laquelle débute son descripteur. Elle vérifie aussi que les *VIDT* des partitions appelantes et appelées sont accessibles en espace utilisateur, et que les espaces de mémoires ciblés par l'appel sont eux aussi accessibles. Ceci permet par exemple de récupérer le contexte d'exécution de la partition ciblée par l'appel. Cette étape préliminaire permet de s'assurer que le service ne pourra pas rencontrer d'erreur dans les prochaines étapes.

Étape de modification de l'état La seconde partie du service est une étape procédant à la modification de l'état du système. Cette étape regroupe toutes les écritures en mémoire requises par le service. Tout d'abord, le service va procéder à l'écriture du contexte de la partition appelante dans son espace d'adressage (si demandé lors de l'appel). Le contexte est recopié depuis la pile noyau jusqu'à la zone de mémoire pointée par le pointeur dans la *VIDT* de la partition appelante. Ensuite, le service met à jour l'espace d'adressage pour refléter l'espace d'adressage de la partition cible. Enfin, le service procède à la mise à jour des structures de données du noyau afin de préserver les propriétés de cohérence internes de Pip. Cette étape ne nécessite en fait qu'une unique écriture dans une variable globale,

indiquant quelle partition s'exécutera lorsque le flot d'exécution repassera en espace utilisateur.

Étape de transfert de flot d'exécution La troisième et dernière étape du service transfère le flot d'exécution vers la partition appelée au travers du contexte d'exécution récupéré lors de l'étape préliminaire. Cette étape est représentée dans le code prouvé par un appel à une fonction de l'interface.

3.2.3 Décomposition des opérations

Afin de pouvoir accomoder les différents modes de transfert de flot d'exécution au sein d'un même service, le service est composé de plusieurs blocs de code remplissant leur propre fonction et appelant le bloc de code suivant. Ces blocs pourraient être assimilés à des *continuations*. Les blocs composants le service sont illustrés à la fin de la sous-section sur la figure 3.3.

Vérification du numéro de contexte ciblé Le bloc de code `checkIntLevelCont` est le point d'entrée dans le service lors d'un appel explicite, tel que présenté dans le code 3.2. Il se contente de vérifier que le numéro de contexte ciblé soit bien valide et transforme son type pour qu'il soit utilisable par le noyau. Il appelle ensuite le bloc de code `checkCtxSaveIdxCont`.

Vérification du numéro de sauvegarde du contexte de la partition appelante Le bloc de code `checkCtxSaveIdxCont` vérifie que le numéro de sauvegarde du contexte de la partition appelante est valide, et transforme son type pour qu'il soit utilisable par le noyau. Il récupère ensuite le descripteur de partition de la partition appelante, ainsi que son Page Directory (la page mémoire racine de la configuration de son espace d'adressage – voir 2.1.3).

Enfin, ce bloc de code va vérifier si la valeur de l'adresse virtuelle désignant la partition appelée est celle par défaut. Si c'est le cas, le prochain bloc de code sera `getParentPartDescCont`. Si ce n'est pas la valeur par défaut, il va appeler le bloc de code `getChildPartDescCont`.

Récupération de la partition parent Le bloc de code `getParentPartDescCont` est un des deux blocs d'exécution possibles concernant la cible du transfert de flot d'exécution qui traite de l'appel vers le parent. Il vérifie uniquement que la partition appelante n'est pas la partition racine, puisque la partition racine n'a pas de parent. Il récupère ensuite le descripteur de partition de son parent, qu'il passera en tant que descripteur de partition appelée au prochain bloc de code `getSourceVidtCont`. Ce prochain bloc de code est commun aux deux fils d'exécutions alternatifs.

Récupération de la partition enfant Le bloc de code `getChildPartDescCont` est l'autre alternative d'exécution du transfert de flot d'exécution qui traite de l'appel vers un enfant. Il vérifie que l'adresse virtuelle passée comme paramètre correspond au descripteur de partition d'une partition enfant. Pour le vérifier, Pip vérifie tout d'abord que l'adresse virtuelle réside bien dans l'espace d'adressage de la partition appelante, et vérifie ensuite dans ses structures de données internes que l'adresse correspond à un descripteur de partition.

Ensuite, de la même manière que le bloc de code concernant l'appel à un parent, ce bloc de code passe le descripteur de partition au prochain bloc de code `getSourceVidtCont`.

Récupération de la VIDT de la partition appelante Le bloc de code `getSourceVidtCont` vérifie d'abord que la VIDT de la partition appelante est mappée et accessible dans son espace d'adressage. Une fois qu'il a déterminé qu'il était possible de lire dans la VIDT, il va récupérer l'adresse virtuelle de l'espace mémoire où sauvegarder le contexte de la partition appelante. Enfin, la page de mémoire contenant la VIDT, et l'adresse virtuelle de l'espace mémoire de sauvegarde sont passés au bloc de code suivant `getTargetVidtCont`.

Récupération de la VIDT de la partition ciblée Tout d'abord, le bloc de code `getTargetVidtCont` va récupérer le *Page Directory* de la partition appelée à partir de son descripteur de partition. Ensuite, similairement au bloc de code précédent, il vérifie que la VIDT de la partition appelée est mappée et accessible dans son espace d'adressage. Ensuite, la page de mémoire contenant la VIDT ainsi que son *Page Directory* sont passés au bloc de code suivant `getTargetContextCont`.

Récupération du contexte de la partition ciblée Le bloc de code `getTargetContextCont` vérifie que le pointeur vers le contexte d'exécution ciblé pointe bien dans l'espace d'adressage de la partition ciblée, et que cet espace est bien accessible à la partition. En supplément, le bloc va vérifier que l'adresse de fin du contexte ne va pas overflow, et que la dernière adresse du contexte est aussi accessible à la partition afin de s'assurer que l'entièreté du contexte pourra être lu sans déclencher de faute. Ceci introduit l'hypothèse qu'un contexte d'exécution a une taille inférieure à une page mémoire. Enfin, le bloc de code va comparer l'adresse virtuelle de sauvegarde de contexte récupérée au bloc `getSourceVidtCont` (voir 3.2.3) avec l'adresse virtuelle par défaut. Si cette adresse n'est pas celle par défaut, c'est que la partition veut sauvegarder son contexte, et le bloc de code `saveSourceContextCont` est appelé. Sinon, le bloc de code `switchContextCont` est appelé.

Sauvegarde du contexte de la partition appelante Le bloc de code `saveSourceContextCont` va se contenter de vérifier que l'espace mémoire pointé par l'adresse récupérée dans la VIDT de la partition appelante est bien dans l'espace d'adressage de la partition appelante et qu'il est accessible en espace utilisateur. Comme dans le bloc précédent, il va vérifier en supplément que l'adresse de fin de cet espace mémoire n'overflow pas, et

qu'il reste accessible dans son espace d'adressage, afin de s'assurer qu'une écriture dans cette zone mémoire ne déclenchera pas de faute. Une fois les vérifications faites, il écrit le contexte de la partition appelante dans la zone mémoire, puis appelle le dernier bloc de code `switchContextCont`.

Changement d'espace d'adressage et chargement du contexte d'exécution Le bloc de code `switchContextCont` ne procède plus à aucune vérification. Il commence par écrire les drapeaux `flagsOnYield` dans la partition appelante. Ensuite, l'espace d'adressage est changé par celui de la partition appelée – son *Page Directory* ayant été récupéré dans le bloc de code `getTargetVidtCont` (voir 3.2.3). Ensuite, il met à jour la variable de Pip indiquant la partition qui s'exécutera lorsque le processeur repassera en mode utilisateur. Le bloc de code récupère ensuite les drapeaux `flagsOnWake` du contexte d'exécution de la partition à réveiller et les applique à la partition courante, sur le nouvel espace d'adressage. À partir de ces drapeaux, il détermine si la partition appelée peut demander à ne pas être interrompue, puis procède enfin au chargement de son contexte.

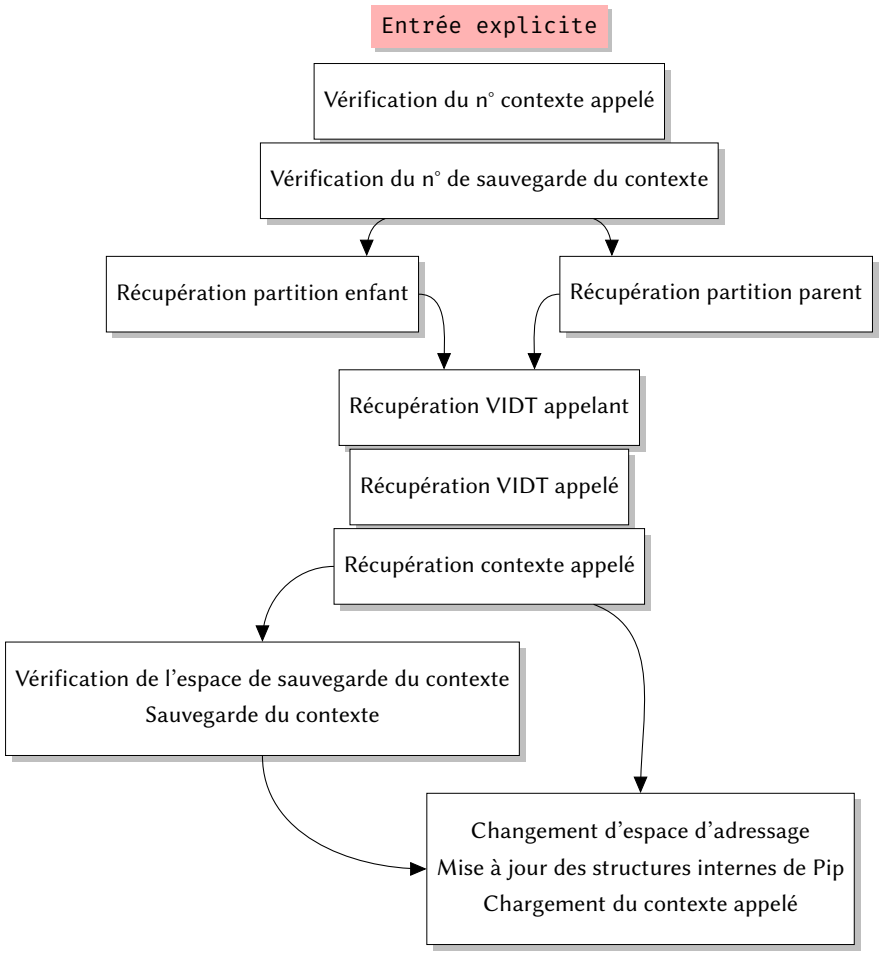

Figure 3.3 – Vue éclatée des blocs constituant le service

3.2.4 Généralisation du service aux fautes et aux interruptions

L'idée principale derrière cette unification est qu'il est possible pour Pip de fixer certains paramètres et de commencer l'exécution à un endroit arbitraire du service. Il est notamment possible de commencer l'exécution directement après la validation des paramètres et de la traduction de l'adresse virtuelle de la partition cible. Ceci permet au système de passer en arguments les adresses réelles de partitions qui seront la cible des différents événements. La figure 3.8 montre comment sont placés les différents points d'entrée du système dans le service.

Cependant, les différents mécanismes décrits dans la section 2.1.3 ne présentent pas d'interface commune ; c'est pourquoi, de petites portions de code C et assembleur sont placées juste avant les différents points d'entrée afin d'harmoniser les différents formats de contexte. Ces morceaux d'assembleur sont placés dans l'*IDT* du système, afin qu'ils soient appelés lors d'une faute ou d'une interruption. Deux niveaux d'interruption sont réservés à la sauvegarde des contextes par Pip lorsqu'une faute ou une interruption survient. Sur l'architecture `x86` ce sont les niveaux 48 et 49 qui sont utilisés (les niveaux d'interruptions 0 à 31 sont réservés par Intel pour les fautes, les niveaux d'interrruptions de 32 à 47 ont été configurés pour correspondre aux interruptions matérielles).

Implémentation des fautes utilisant le service sur l'architecture x86

Pour rappel, les fautes doivent être transmises au parent de la partition fautive. Il est donc naturel de commencer l'exécution du service au bloc de code `getParentPartDescCont`. Le prototype de ce bloc de code est présenté en listing 3.3.

```
Definition getParentPartDescCont (sourcePartDesc : page)
                                 (sourcePageDir : page)
                                 (targetInterrupt : index)
                                 (sourceContextSaveIndex : index)
                                 (nbL : level)
                                 (flagsOnYield : interruptMask)
                                 (flagsOnWake : interruptMask)
                                 (sourceInterruptedContext : contextAddr)
           : LLI yield_checks.
```

Listing 3.3 – Prototype du point d'entrée du service en Gallina

Cependant, l'appel de ce bloc de code n'est pas trivial après une faute, car les paramètres attendus par le bloc de code `getParentPartDescCont` ne sont pas immédiatemment disponibles à Pip. En effet, ces arguments ne sont en aucun cas fournis par la partition fautive,

à contrario des arguments attendus par le service lors d'un appel explicite. La première étape pour Pip est donc de récupérer ces arguments.

Il s'agit d'abord de récupérer le contexte d'exécution de la partition fautive. Comme expliqué précédemment dans la section 2.1.2, lorsqu'une faute survient, le processeur va chercher la *gate* installée dans l'*IDT* dont le numéro correspond au niveau de la faute. Dans Pip, ces *gates* sont toutes des *interrupt gates*, qui s'exécutent en mode privilégié.

De ce fait, le processeur change de pile. Il pousse le segment de pile `SS` ainsi que l'ancien pointeur vers le sommet de la pile `ESP`. Il pousse ensuite l'état des drapeaux du processeur `EFLAGS`, puis pousse le segment de code `CS` et le pointeur d'instruction `EIP` au moment de la faute. Enfin, en fonction de la faute qui a été déclenchée, le processeur peut éventuellement pousser un entier précisant la cause de la faute. La figure 3.4 illustre l'état de la pile noyau après qu'une faute soit survenue.

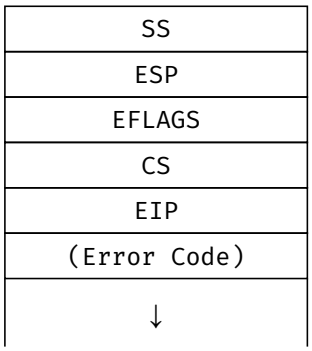

Figure 3.4 – État de la pile noyau après qu'une faute (ou une interruption) soit survenue en espace utilisateur
Reproduction partielle du manuel Intel [Int19d]

Pour unifier la structure des données sur la pile du noyau entre toutes les fautes et interruptions, le bout d'assembleur s'exécutant en sortie de faute va pousser une valeur de bourrage sur la pile si le processeur n'a pas poussé de code d'erreur. Il va pousser le niveau d'interruption de la faute sur la pile à des fins informatives. La routine assembleur va poursuivre en complétant le contexte d'exécution qui avait été partiellement sauvé par le processeur en poussant les registres généraux sur la pile. Ceci complète la structure `int_ctx_t` représentant le contexte d'exécution de la partition fautive. Enfin, le bout d'assembleur va pousser un pointeur vers cette structure. Une description de la pile après

```
void faultInterruptHandler(int_ctx_t *ctx);
```

Listing 3.4 – Prototype de la fonction calculant les arguments du bloc `getParentPartDescCont` du service lors d'une faute

l'exécution de cette routine assembleur est disponible en figure 3.5. Elle s'achève en appelant le code C qui sera chargé de récupérer les arguments pour appeler le bloc de code `getParentPartDescCont`.

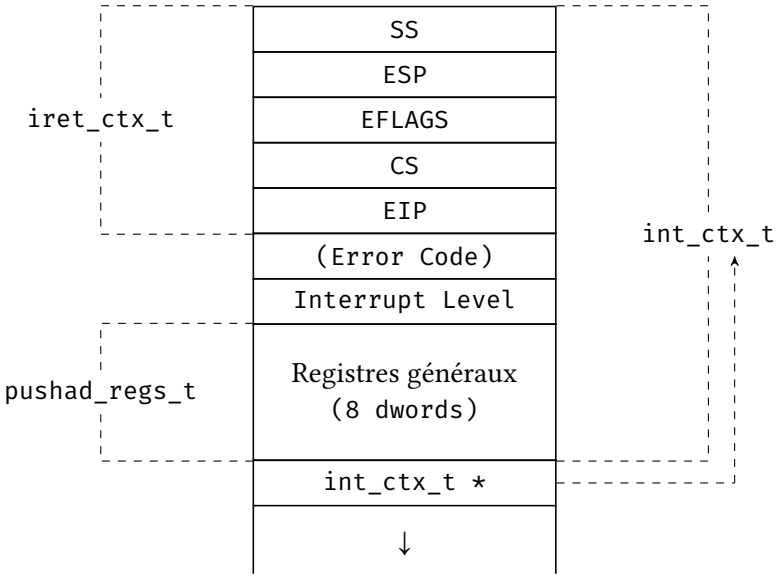

Figure 3.5 – État de la pile noyau après la routine assembleur complétant la structure `int_ctx_t`

Cette fonction s'appelle `faultInterruptHandler` ; son prototype est donné dans le listing 3.4. Son code est trop long pour être inclus dans ce chapitre mais vous pouvez le retrouver en annexe A.3.

Tout comme l'appel du service par la *callgate*, la fonction `faultInterruptHandler` commence par créer un nouveau contexte générique de type `user_ctx_t` à partir du contexte fautif précédemment créé. Ceci permet d'avoir la même représentation du contexte entre les différents points d'entrée du service, comme évoqué dans la section 3.2.2. Cependant, le niveau de faute sera utilisé par la fonction comme argument `targetInterrupt`, pour que le parent soit reveillé avec le contexte lié à la faute. La fonction récupère ensuite le descripteur de partition fautive grâce à la variable globale de Pip indiquant la partition s'exécutant en espace utilisateur, qui deviendra l'argument `sourcePartDesc`. Elle récupère aussi l'état de ses drapeaux dans le descripteur de partition, liés aux arguments `flagsOnYield` et `flagsOnWake`. La fonction va décider où sauvegarder le contexte de la partition fautive en fonction de ces drapeaux. L'implémentation actuelle considère que si le mot mémoire représentant les drapeaux est égal à zéro, la partition ne souhaitait pas être interrompue (on parle de *Virtual CLI*, en référence à l'instruction assembleur désactivant les interruptions). L'état sera alors sauvegardé dans l'espace mémoire pointé à l'index `CLI_SAVE_INDEX`. Si les drapeaux sont différents de zéro, alors l'autre indice réservé, `STI_SAVE_INDEX` sera utilisé pour l'argument `sourceContextSaveIndex`. Enfin, la

fonction récupère le *Page Directory* de la partition à partir de son descripteur qui servira d'argument `sourcePageDir`.

Lorsque tous les arguments ont été récupérés [1], la fonction est prête à appeler le bloc de code `getParentPartDescCont`.

Cas des doubles fautes Il est possible que l'appel au service échoue lors des vérifications pour le transfert de flot d'exécution. Lorsque ce transfert est dû à une faute, la partition l'ayant déclenchée n'est plus en mesure de recevoir le flot d'exécution : elle déclencherait à nouveau la faute. Ainsi, il faut que le service propose une voie de transfert de flot d'exécution dont le succès ne dépend que de la bonne configuration de la partition ayant la responsabilité de la partition fautive – c'est à dire son parent, la partition recevant le flot d'exécution. Dans le cas où le parent ne serait pas non plus en mesure de recevoir le flot d'exécution, le flot d'exécution serait redirigé vers le parent de la partition parent et remonterai la chaîne de responsabilité, jusqu'à ce qu'une partition reçoive le flot d'exécution sans erreur, ou jusqu'à ce que la partition racine elle même échoue à le recevoir.

C'est pourquoi la fonction calculant les arguments appelle à son tour une autre fonction appelée `propagateFault` (cette fonction est disponible en annexe, voir listing A.3). Cette fonction est chargée de faire l'appel au bloc de code `getParentPartDescCont` et de gérer les éventuelles erreurs pouvant survenir après un appel à ce bloc. Les différentes étapes logicielles permettant d'utiliser le service pour une faute sont résumés dans la figure 3.6. Il y a trois cas d'erreurs distincts gérés par la fonction :

— le cas où la fonction ayant réalisé la faute est la partition racine : dans ce cas là, il n'y a plus rien à rattraper, la partition racine étant la base de confiance absolue du système – si elle échoue le système s'écroule. Le service s'arrête alors sur une boucle infinie.

— le cas où le service n'a pas réussi à récupérer la VIDT de la partition fautive, ou l'espace mémoire permettant de sauvegarder le contexte : dans ce cas, la sauvegarde du contexte de la partition fautive est omise, et l'exécution reprend au bloc de code `getTargetVidtCont`.

— dans tous les autres cas d'erreur, la partition parent n'est pas correctement configurée et n'est pas en mesure de rattraper la faute. Dans ce cas, la fonction `propagateFault` fait un appel récursif. La faute est redirigée sur le parent de la cible actuelle, et le niveau d'interruption de la faute est changé au niveau correspondant à la double faute.

1. l'argument `nbL` de l'appel a été omis du texte principal car il est peu intéressant. Cet argument est en fait un paramètre de l'implémentation précisant le nombre d'indirections dans les structures de configurations de la MMU, soit 2 dans l'implémentation Intel x86

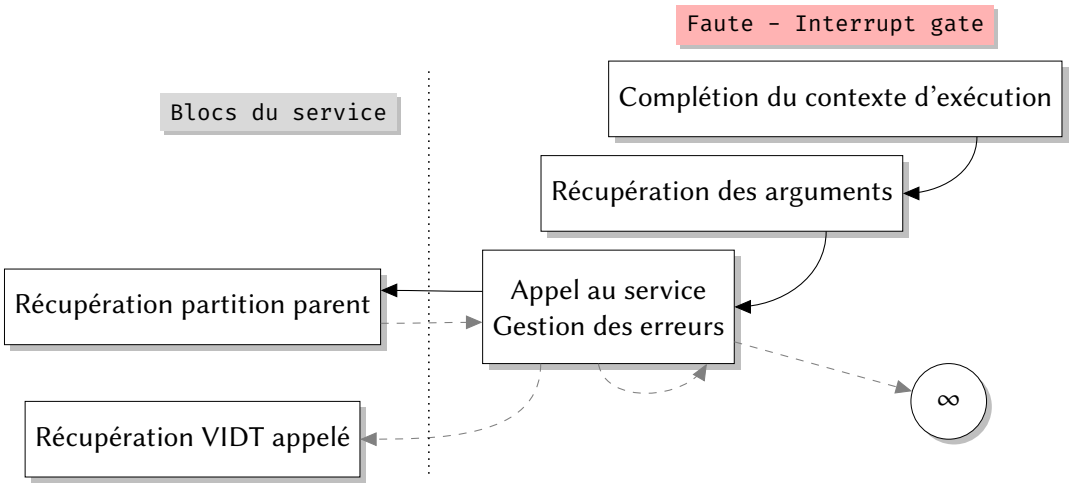

Figure 3.6 – Résumé des différents blocs logiciels permettant d'appeler le service après une faute

Implémentation des interruptions utilisant le service sur l'architecture x86

La même méthode a été employée pour gérer les interruptions matérielles avec le service. Pour rappel, les interruptions matérielles doivent arriver à la partition racine. Il n'y a pas de bloc de code récupérant la partition racine directement, il faut donc commencer à exécuter le service après les blocs de code récupérant le descripteur de partition. Ceci permet d'injecter dans les paramètres le descripteur de la partition racine à la place des partitions initialement prévues par les blocs précédents.

Le bloc idéal pour devoir récupérer le moins d'arguments possibles est le bloc `getSourceVidtCont`. Le prototype du bloc est présenté en listing 3.5.

```
Definition getSourceVidtCont (targetPartDesc : page)
                             (sourcePageDir  : page)
                             (targetInterrupt : index)
                             (sourceContextSaveIndex : index)
                             (nbL                    : level)
                             (flagsOnYield : interruptMask)
                             (flagsOnWake  : interruptMask)
                             (sourceInterruptedContext : contextAddr)
           : LLI yield_checks :=
```

Listing 3.5 – Prototype du bloc de code `getSourceVidtCont`, ciblé par les interruptions matérielles.

Le cheminement jusqu'à l'exécution du bloc du service est extrêmement similaire à celui concernant les fautes. La première étape pour Pip consiste toujours à récupérer les arguments du bloc `getSourceVidtCont`, qui ne sont pas fournis par la partition interrompue. De la même manière que pour une faute, lorsqu'une interruption matérielle arrive, le processeur va récupérer l'*interrupt gate* correspondante dans l'*IDT* (pour rappel, les niveaux d'interruptions correspondants aux interruptions matérielles vont de 32 à 47). Cette similitude continue jusqu'à la récupération des arguments : le processeur change de pile et pousse les mêmes éléments que ceux présentés précédemment en figure 3.5, puis une routine assembleur complète le contexte d'exécution de type `int_ctx_t`, et appelle finalement la fonction de récupération des arguments `hardwareInterrupthandler`. Les arguments à récupérer par la fonction sont les mêmes que pour le bloc `getParentPartDescCont` et sont récupérés de la même manière, mis à part l'argument `targetPartDesc`. Cet argument est l'adresse réelle du descripteur de la partition recevant le flot d'exécution, qui devra donc être le descripteur de la partition racine. La fonction récupère ce descripteur au travers d'une variable globale de Pip.

L'appel du bloc de code `getSourceVidtCont` peut aussi se solder par une erreur : si l'origine de cette erreur est le fait que la VIDT de la partition n'est pas accessible ou que l'emplacement de sauvegarde de son contexte n'est pas accessible, alors la sauvegarde de contexte est omise et la fonction appelle le bloc de code `getTargetVidtCont`. Une autre erreur indiquerait que la partition racine n'est pas en mesure de récupérer le flot d'exécution ; dans ce cas la fonction arrête l'exécution en rentrant dans une boucle infinie.

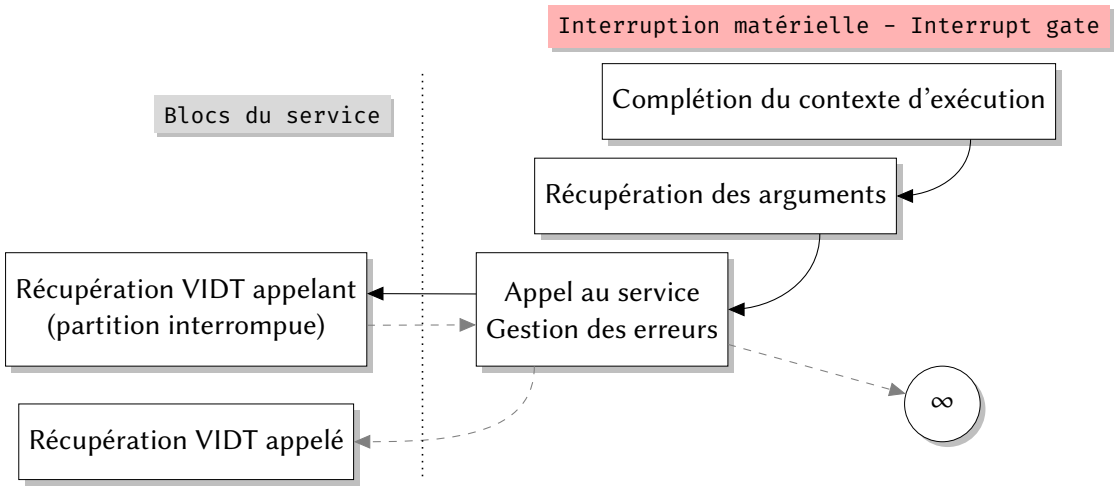

Figure 3.7 – Résumé des différents blocs logiciels permettant d'appeler le service après une interruption matérielle

À l'exception de l'état des piles lors des différents transferts de flot d'exécution, le fonctionnement du service et des routines permettant d'y accéder restent rigoureusement les même entre toutes les architectures.

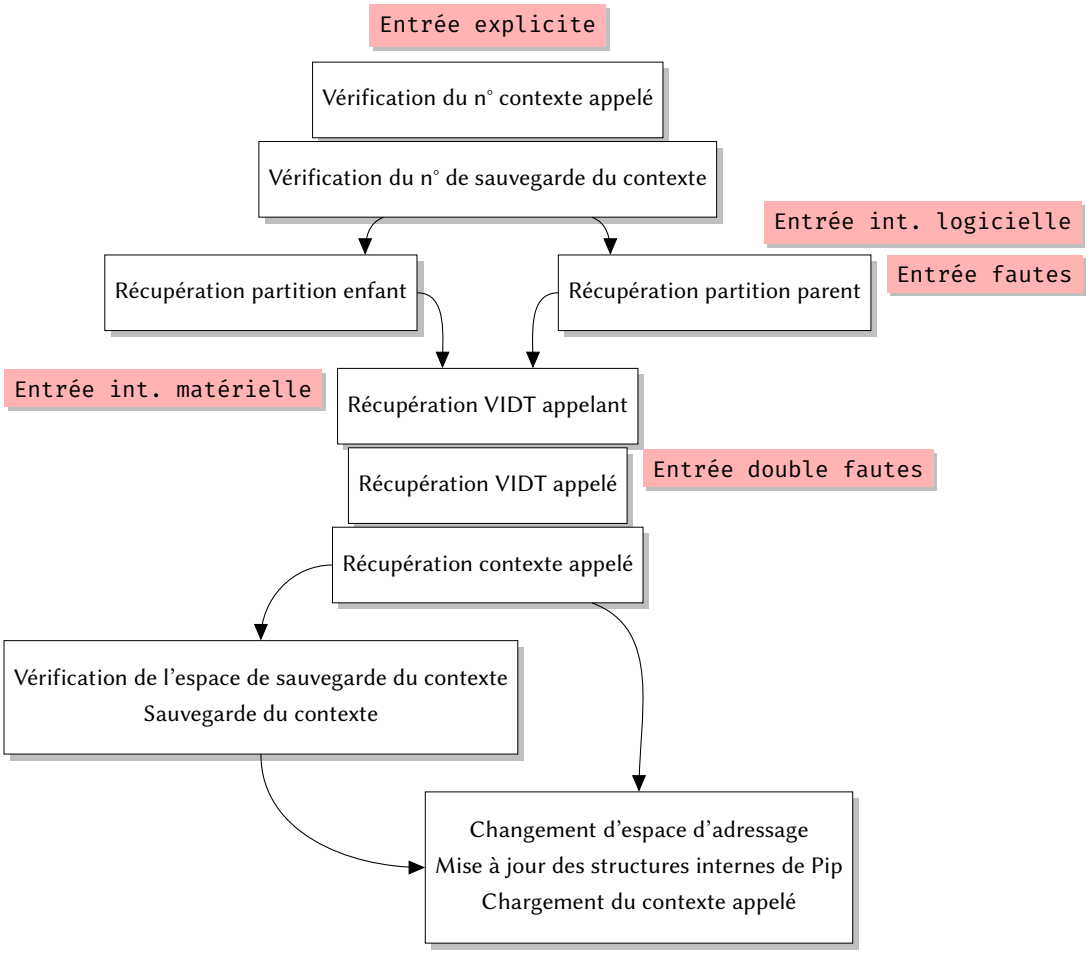

Figure 3.8 – Vue éclatée des blocs constituant le service

3.3 Preuve d'isolation

Cette section a pour but de décrire le processus d'établissement de la preuve d'isolation de Pip sur le service. La première sous-section décrira tout d'abord les nouvelles fonctions introduites dans l'interface de la monade d'état ainsi que la raison de leur inclusion. Ces fonctions sont accompagnées de nouveaux types, qui seront aussi explicités. La seconde sous-section fera un rappel des propriétés d'isolation de Pip. La dernière sous-section décrira le processus d'établissement de la preuve.

Note importante : Dans un souci de transparence et d'intégrité scientifique, je souhaite mentionner que la preuve d'isolation reposant sur les modèles présentés dans cette section n'est pas encore complète. Les travaux initiaux sur ce service ont conduit à une preuve de préservation de l'isolation. Tout au long de ma thèse, j'ai cependant été amené à améliorer le service, notamment en enrichissant certains modèles pour refléter de manière plus précise les comportements de fonctions. Une des fonctions notables est `writeContext`. Ces nouveaux modèles ont rendus caduques la preuve d'isolation établie sur l'ancien modèle ; il me semble néanmoins plus pertinent de présenter les modèles les plus récents, qui sont plus précis et – à mon sens – plus intéressants. Ces nouveaux modèles ne changent en rien l'implémentation réelle des fonctions : ces changements sont cantonnés au monde des mathématiques. Dans cette section, je m'efforcerai donc de vous convaincre que la preuve du maintien de l'isolation reste tout à fait atteignable. Elle n'a pas été établie par manque de temps entre les dernières modifications de modèles et l'écriture de ce document (et par manque de place dans la marge). Nous reviendrons cependant sur l'aspect itératif des modèles dans la section suivante, qui révèle la subtilité de l'établissement de preuves au travers de modèles.

3.3.1 Définition de l'interface avec la monade

Cette sous-section est dédiée à la définition des nouveaux éléments de l'interface avec la monade. Nous discuterons d'abord des nouveaux types et de ce qu'ils représentent, puis nous discuterons des fonctions.

Nouveaux types intégrés à la monade

`userValue` Le type `userValue` est un type *opaque*, c'est à dire un type dont les valeurs n'ont pas vocation à être manipulées dans le modèle. Le type `userValue` indique une valeur arbitraire provenant de l'espace utilisateur. Cette valeur doit subir des tests avant de pouvoir être transformée en un type sain et utilisable par le noyau. Prenons par exemple le type `index`. `index` est un entier naturel dont la valeur ne peut dépasser le nombre de mots mémoire dans une page. Afin de s'assurer qu'un indice passé en paramètre par l'utilisateur respecte bien ces contraintes, le service va le traiter comme une `userValue`. Le service vérifie alors que la contrainte est bien respectée, puis raffine le type de la valeur en un `index`.

Pour certains types de valeurs, il n'est pas nécessaire d'utiliser le type `userValue`. Ceci est vrai pour les types dont l'intégralité des valeurs représentables sont des valeurs valides, comme par exemple les adresses virtuelles.

Le type `userValue` étant un type opaque, son modèle importe peu. Il est représenté par un `nat`.

`interruptMask` Le type `interruptMask` est un type *opaque* représentant les drapeaux servant aux partitions à indiquer si elles souhaitent être interrompues ou non. Ce type est entouré d'une interface permettant de récupérer ces drapeaux depuis un contexte d'exécution, d'appliquer ces drapeaux à une partition, de sauver ces drapeaux dans un contexte d'exécution, ainsi que d'interpréter ces drapeaux. La mise en place d'un tel mécanisme est fortement lié à la plateforme et a été laissée libre à l'implémentation. Le type est représenté comme une liste de booléens dont la longueur est égale au nombre de niveaux d'interruptions, cependant cette information n'est jamais exploitée dans le modèle.

```
Record interruptMask := {
    m  :> list bool;
    Hm :  length m = maxVint+1;
}.
```

Listing 3.6 – Représentation du type `interruptMask` dans le modèle

`contextAddr` Le type `contextAddr` est un type *opaque* représentant l'adresse d'un contexte d'exécution. Il est muni de deux fonctions : une permettant d'écrire ce contexte à une adresse donnée, l'autre l'appliquant au système qui reprendra le flot d'exécution lié. Le type est représenté par un `nat`.

Nouvelles fonctions de l'interface

Certaines de ces nouvelles fonctions font partie de l'interface des nouveaux types présentés dans les paragraphes précédent. Tout d'abord le type `userValue` a engendré deux nouvelles fonctions :

`bool checkIndexPropertyLTB(userValue userIndex);`
Cette fonction retourne vrai si la valeur passée en paramètre respecte la contrainte du type `index` ; la valeur doit être inférieure au nombre de mots mémoire dans une page. Son modèle est disponible en listing 3.7.

Cette fonction de l'interface a cependant été maladroitement choisie, car le choix de la valeur de la comparaison `tableSize` relève de la responsabilité de l'implémentation, alors qu'elle aurait pu rester dans le code du service. La fonction pertinente à ajouter dans

```
Definition checkIndexPropertyLTB (userIndex : userValue) : LLI bool :=
ret (Nat.ltb userIndex tableSize).
```

Listing 3.7 – Modèle de la fonction `checkIndexPropertyLTB`

l'interface au lieu de celle-ci aurait dû être la fonction de comparaison entre une `userValue` et un `index`.

`index userValueToIndex(userValue userIndex);`
Cette fonction transforme la valeur de type `userValue` en fonction de type `index`. Cette fonction n'a pas d'effet particulier sur la valeur. Son modèle est disponible en listing 3.8. Ce modèle met en lumière que la valeur `userIndex` doit être inférieure à la constante `tableSize`, qu'une preuve en soit établie, et que cette preuve soit passée au constructeur du type `index` (ici déterminée automatiquement par Coq au travers du symbole _).

```
Program Definition userValueToIndex (userIndex : userValue) : LLI index :=
    if lt_dec userIndex tableSize
    then
        ret (Build_index userIndex _ )
    else undefined 85.
```

Listing 3.8 – Modèle de la fonction `userValueToIndex`

Le service a ajouté quatre fonctions dans l'interface qui intéragissent avec le type `interruptMask`.

`interruptMask getInterruptMaskFromCtx(contextAddr context)` est la première fonction permettant de récupérer les drapeaux présents dans un contexte d'exécution. La seconde fonction `bool noInterruptRequest(interruptMask flagsOnWake)` permet d'interpréter les drapeaux passés en paramètres pour renvoyer si la partition ne souhaite pas être interrompue. La troisième fonction en rapport avec le type `interruptMask` est la fonction `void setInterruptMask(uint32_t interrupt_state)` qui permet d'appliquer les masques à la partition courante. La dernière fonction ajoutée, `uint32_t get_self_int_state()`, permet de récupérer les drapeaux de la partition courante.

Du point de vue du modèle, ces fonctions sont peu intéressantes. Elles n'ont pas été intégrées car la position des drapeaux n'a pas été définie dans l'interface et peut donc être placée arbitrairement par l'implémentation. L'implémentation Intel x86, place ces drapeaux dans le descripteur de la partition, afin que leur écriture soit atomique. Ainsi, ces fonctions renvoient soit une valeur arbitraire, soit n'ont aucun effet sur le modèle. La fonction `uint32_t get_self_int_state()` n'a pas été modélisée car elle n'apparait pas dans le service, mais dans les fonctions de récupération des arguments discutés en section 3.2.4.

Les fonctions les plus intéressantes de cet ajout à l'interface sont celles qui manipulent le type `contextAddr`.

`contextAddr vaddrToContextAddr(vaddr contextVAddr);`
Cette fonction convertit une adresse virtuelle en pointeur vers un contexte d'exécution. Le modèle de cette fonction renvoie une valeur arbitraire.

`void loadContext(contextAddr ctx, bool enforce_interrupts);`
Cette fonction charge le contexte d'exécution pointé par `ctx`, en s'assurant que le futur flot d'exécution sera interruptible si `enforce_interrupts` est vrai.

Cette fonction est intéressante car elle ne peut être écrite en Gallina, le langage de Coq. En effet, Coq s'assure que tous les programmes écrits en Gallina terminent, ce qui n'est pas le cas de cette fonction. Par exemple, dans l'implémentation Intel x86, cette fonction charge tout les registres généraux puis pousse une petite partie du contexte sur pile, et exécute l'instruction assembleur `iret`. À partir du moment où cette instruction a été exécutée, le flot d'exécution a déjà changé : la pile et le pointeur d'instruction ne sont plus les mêmes. De ce fait cette fonction n'a aucun effet dans le modèle.

```
Definition loadContext (contextToLoad : contextAddr)
                       (enforce_interrupt : bool) : LLI unit :=
    ret tt.
```

Listing 3.9 – Modèle de la fonction `loadContext`

Cependant, il n'est pas *nécessaire* que cette fonction ne retourne pas. En effet, dans l'implémentation actuelle sur Intel x86, la fonction copie le contexte sur le dessus de la pile et le charge immédiatement. Il s'avère que ce code de chargement est aussi disponible dans les portions d'assembleur appelant le service, dans le cas où une erreur surviendrait pendant l'exécution du service. Dans ce cas, le service retourne prématurément pour signaler une erreur et la fonction `loadContext` n'est jamais atteinte. Ces portions de code restaurent le contexte appelant le service, mais il serait possible que la fonction `loadContext` vienne modifier ce contexte initial pour qu'il corresponde au nouveau contexte à charger. Ainsi, le transfert de flot d'exécution ne s'opérerait pas dans la fonction `loadContext`, mais dans la portion d'assembleur de retour au contexte initial. Ainsi, il serait possible d'éviter cette dualité entre le monde du modèle qui retourne et renvoie un succès, et le monde réel qui ne retourne pas et charge directement le contexte dans la fonction. Cette approche est néanmoins beaucoup plus compliquée à mettre en place comparée à une simple copie des données sur le sommet de la pile, et n'apporte pour le moment pas d'autre avantage que l'agréable sentiment d'harmonie dans la création.

`void writeContext(contextAddr ctx, vaddr ctxSaveVAddr, interruptMask flagsOnWake);`
Cette fonction écrit le contexte d'exécution pointé par `ctx` à l'adresse virtuelle `ctxSaveVAddr`, en y écrivant les drapeaux `flagsOnWake`. Cette fonction est différente des autres fonctions d'écriture de l'interface avec la monade car elle effectue plusieurs écritures successives en mémoire. De plus, ces écritures sont faites en espace utilisateur. Son modèle se contente d'appeler une fonction récursive qui va itérer sur la taille d'un contexte, et écrire aux adresses virtuelles successives une valeur arbitraire.

Les écritures multiples de la fonction `writeContext` ont mené à des vérifications supplémentaires qui ont nécessité l'ajout de deux fonctions dans l'interface. La première est `vaddr getNthVAddrFrom(vaddr base, uint32_t size)`. Cette fonction retourne l'adresse virtuelle se trouvant à l'adresse de base plus une certain nombre d'octets `size`. Cette fonction permet notamment de récupérer la dernière adresse d'un contexte d'exécution, afin de vérifier qu'elle est bien valide avant d'entamer l'écriture.
La seconde est `bool firstVAddrGreaterThanSecond(vaddr vaddr1, vaddr vaddr2)`, qui retourne si une adresse virtuelle est plus grande que la seconde. Cela permet au service de vérifier qu'un overflow n'a pas eu lieu lors du calcul de la dernière adresse du contexte.

La fonction `vaddr getVaddrVIDT()` fait aussi partie des nouveaux ajouts à l'interface, et retourne l'adresse fixe de la VIDT. Cette valeur est définie comme la dernière page de l'espace d'adressage virtuel.

Enfin, la fonction `void updateMMURoot(page MMURoot)` a été ajoutée. Cette fonction applique un nouvel espace d'adressage décrit par `MMURoot`. Dans l'implémentation Intel, cette fonctionne écrit le Page Descriptor `MMURoot` – structure racine de la configuration de la MMU – dans le registre `CR3` du processeur. Cette fonction est spéciale car elle aurait

```
Definition writeContext (callingContextAddr : contextAddr)
                        (contextSaveAddr : vaddr)
                        (flagsOnWake : interruptMask) : LLI unit :=
    perform maxIdx := getMaxIndex in
    perform idxContextInPage := ret (List.last contextSaveAddr index_d) in
    writeContextAux contextSaveAddr idxContextInPage maxIdx contextSize.

Fixpoint writeContextAux (contextSaveAddr : vaddr) (currIdx : index)
                         (maxIdx : index) (bound : nat) : LLI unit :=
  match bound with
  | 0 ⇒ ret tt
  | S dec_bound ⇒
    storeVirtual contextSaveAddr currIdx vaddrDefault ;;
    if idxEq currIdx maxIdx then
      writeContextAux (getNextVaddr contextSaveAddr) idx0 maxIdx dec_bound
    else
      perform nextIdx := idxSuccM currIdx in
      writeContextAux (getNextVaddr contextSaveAddr) nextIdx maxIdx dec_bound
  end.
```

Listing 3.10 – Modèle de la fonction `writeContext` et sa fonction auxiliaire récursive

pu être cachée dans l'implémentation, et n'a aucun sens dans le modèle d'isolation de Pip. Elle été ajoutée dans l'optique de produire une preuve fonctionnelle du service.

C'est sur ces différents modèles de fonction que la preuve d'isolation sera établie. Dans la prochaine section nous préciserons les propriétés d'isolation, avant de procéder à l'explication de l'établissement de la preuve.

3.3.2 Rappel des propriétes d'isolation de Pip

Comme exprimé dans la section 2.3.3, la preuve d'isolation de Pip repose sur trois propriétés fondamentales : la propriété d'*isolation horizontale*, la propriété d'*isolation noyau* ainsi que la propriété de *partage verticale*. Ces trois propriétés principales sont accompagnées d'une multitude de propriétés liées à la cohérence des structures du noyau. Certaines de ces propriétés seront mises en avant dans ce document, puisqu'elle seront discutées dans la section suivante traitant de l'établissement de la preuve du service.

Les propriétés présentées dans cette sous-section sont parfois décrites grâce à des fonctions dites « fictives », comme par exemple la fonction `getPartitions`. Ces fonctions permettent entre autres de spécifier des ensembles ou des résultats retournés par des fonctions du code des services. Ces fonctions n'ont pas vocation à être compilées, elles n'existent qu'à des fins de spécifications.

Isolation horizontale

La propriété d'isolation horizontale stipule que deux enfants d'une même partition parent ne peuvent partager de page mémoire au travers de leur espace d'adressage ou de leurs pages de configuration. Les deux ensembles des pages concernant l'une et l'autre des partitions enfants doivent être strictement disjoints. Le listing 3.11 montre comment est exprimée formellement cette propriété dans l'assistant de preuve.

```
Definition partitionsIsolation s : Prop :=

forall parent child1 child2 : page,
  In parent (getPartitions multiplexer s) →
  In child1 (getChildren parent s) →
  In child2 (getChildren parent s) →
  child1 <> child2 →
  disjoint (getUsedPages child1 s)(getUsedPages child2 s).
```

Listing 3.11 – Propriété d'isolation horizontale telle qu'exprimée dans Coq

La fonction `getPartitions` récupère les pages contenant les descripteurs de partitions. La fonction `getChildren` recupère la liste des pages contenant les descripteurs de partitions de tous les enfants de la partition passée en paramètre. La fonction `getUsedPages` récupère l'ensemble des pages mémoires renseignées dans les structures de données de la partition passée en paramètre. Cela inclus les structures internes de Pip ainsi que la description de l'espace d'adressage de la partition.

La propriété peut être lue de la sorte :

▸ **Propriété 3.1.** Pour toutes pages de mémoire `parent`, `child1` et `child2`. Si `parent` est dans la liste des descripteurs de partitions, si `child1` et `child2` sont tous deux dans la liste des partitions enfant de `parent`, et si `child1` et `child2` sont différents, alors l'ensemble des pages utilisées par la partition `child1` est disjoint de l'ensemble des pages utilisées par la partition `child2`. ◂

Isolation noyau

La propriété d'isolation du noyau stipule que les pages de configuration d'une partition (c'est à dire les pages réquisitionnées par Pip pour y loger ses structures, notamment les structures relative à l'espace d'adressage de la partition) sont inaccessibles à *n'importe quelle* partition. Cette propriété est exprimée formellement par le code décrit en listing 3.12.

La fonction `getAccessibleMappedPages` permet de récupérer les pages de mémoire renseignées comme accessibles sans privilège dans l'espace d'adressage de la partition pas-

```
Definition kernelDataIsolation s : Prop :=

forall partition1 partition2,
    In partition1 (getPartitions multiplexer s) →
    In partition2 (getPartitions multiplexer s) →
    disjoint (getAccessibleMappedPages partition1 s)
             (getConfigPages partition2 s).
```

Listing 3.12 – Propriété d'isolation du noyau telle qu'exprimée dans Coq

sée en paramètre. La fonction `getConfigPages` permet de récupérer l'ensemble des pages de mémoire de la partition passée en paramètre contenant les structures internes à Pip.

La propriété peut être lue de la sorte :

▶ **Propriété 3.2.** Pour toutes pages de mémoire `partition1` et `partition2`. Si `partition1` et `partition2` sont toutes deux dans la liste des descripteurs de partitions, alors l'ensemble des pages contenant les structures internes de Pip relatives à la `partition2` est disjoint de l'ensemble des pages accessibles à la `partition1`. ◀

Partage vertical

La dernière propriété fondamentale d'isolation de Pip, appelée propriété de partage vertical, stipule que chaque page relative à une partition enfant fait partie de l'espace d'adressage de son parent. Ces pages concernent aussi bien les pages de configuration de la partition enfant reservées à Pip que les pages disponibles dans son espace d'adressage. Certaines pages ne sont cependant pas accessibles à la partition parent bien qu'elles soient mappées, en particulier les pages de configuration de la partition enfant réquisitionnées par Pip.

```
Definition verticalSharing s : Prop :=

forall parent child : page ,
  In parent (getPartitions pageRootPartition s) →
  In child (getChildren parent s) →
  incl (getUsedPages child s) (getMappedPages parent s).
```

Listing 3.13 – Propriété de partage vertical de la mémoire telle qu'exprimée dans Coq

La fonction `getMappedPages` permet de récupérer toutes les pages de mémoire mappées dans la partition passée en paramètre.

La propriété peut être lue de la manière suivante :

▶ **Propriété 3.3.** Pour toutes pages mémoire `parent` et `child`, si `parent` fait partie de la liste des descripteurs de partitions et que `child` est dans la liste des descripteurs de partitions enfant de `parent`, alors l'ensemble des pages relatives à `child` est inclus dans l'ensemble des pages mappées dans l'espace d'adressage de la partition `parent`. ◀

Propriétés de cohérence des structures du noyau

Dans Pip, il y a 25 propriétés annexes, qui décrivent comment sont organisées les structures de données internes de Pip. Ces propriétés sont appelées les propriétés de cohérence des structures du noyau. Ces propriétés ne seront pas détaillées ici, car seules certaines d'entre elles présentent un intérêt dans la preuve de préservation de l'isolation. Elles seront introduites au fil de l'eau dans les sections suivantes aux endroits pertinents.

3.3.3 Déroulement de la preuve d'isolation

Il pourrait exister une multitude de preuves permettant d'arriver aux propriétés d'isolation. Cette section décrit la stratégie que j'ai utilisé pour conduire cette preuve.

Cette preuve est axée autour de deux points critiques ; la mise à jour de la partition courante, ainsi que l'écriture du contexte de la partition appelante dans son espace d'adressage. Nous commencerons par passer en revue les propriétés souhaitées ainsi que la méthode utilisée pour chacun des deux points critiques de cette preuve, puis nous verrons comment ces propriétés sont récupérées au fur et à mesure de l'exécution des blocs préliminaires du service.

Modification de la partition courante

Comme décrit précédemment, la modification de la partition courante s'effectue dans le bloc `switchContextCont`. La fonction responsable de cette modification est la fonction `updateCurPartition`, dont le modèle est présenté en listing 3.14.

```
Definition updateCurPartition (phyPartition : page) : LLI unit :=
    modify (fun s ⇒ {|
        currentPartition := phyPartition;
        memory := s.(memory)
    |}).
```

Listing 3.14 – Modèle de la fonction `updateCurPartition` modifiant la variable contenant la partition courante

Cette fonction est *monadique*, et a donc accès à l'état au travers de la monade. C'est une fonction d'écriture, elle décrit donc comment est modifié l'état lors de son exécution. Cette fonction prend en argument une page mémoire arbitraire `phyPartition` et remplace

la composante `currentPartition` de l'état par le paramètre `phyPartition`. Ainsi, après l'exécution de `updateCurPartition`, si on nomme s' le nouvel état produit à partir de l'état initial s, alors :

```
s' := {| currentPartition := phyPartition ; memory := (memory s) |}.
```

Seules deux propriétés mentionnent directement le champ `currentPartition` de l'état : `currentPartitionInPartitionsList` et `currentPartitionIsNotDefaultPage`. Ces deux propriétés sont des propriétés de cohérence.

```
Definition currentPartitionInPartitionsList (s : state) :=
    In (currentPartition s) (getPartitions pageRootPartition s).

Definition currentPartitionIsNotDefaultPage (s : state) :=
    (currentPartition s) <> pageDefault.
```

Listing 3.15 – Propriétés de cohérence directement affectées par le changement du champ `currentPartition` de l'état

Ces propriétés assurent respectivement que la variable globale contenant la partition courante contiendra toujours un descripteur de partition provenant de la liste des descripteurs de partitions, et qu'elle ne soit pas égale à la valeur par défaut.

Elles doivent être déduites de propriétés précédemment établies sur la page passée en paramètre. En particulier, on veut donc pouvoir montrer que `targetPartDesc` représente bien un descripteur de partition et qu'il n'est pas nul. Ces deux propriétés, décrites en listing 3.16, doivent avoir été établies *avant* l'appel à la fonction `updateCurPartition`, durant la phase de validation des paramètres.

```
List.In targetPartDesc (StateLib.getPartitions pageRootPartition s) ∧
targetPartDesc <> pageDefault
```

Listing 3.16 – Propriétés supplémentaires nécessaires pour montrer la préservation de deux propriétés de cohérence après exécution de la fonction `updateCurPartition`

Le reste des propriétés d'isolation et de cohérence sur le nouvel état s' peut être déduit des propriétés sur l'état s. Pour les déduire, il faut montrer qu'elles ne portent que sur la portion `memory` de l'état qui reste inchangée entre s et s'. Cela nécessite de déplier la définition de *chaque propriété* (y compris celle des fonctions et propriétés intermédiaires) afin de constater que les termes entre les propriétés sur l'état s et celles qu'on souhaite prouver sur s' sont identiques.

Illustrons par l'exemple avec la preuve de conservation de la propriété d'isolation noyau. On souhaite prouver la propriété suivante (décrite en listing 3.17) :

```
Lemma kernelDataIsolationActivate (partDesc : page) (s : state) :
    let s' := {| currentPartition := partDesc; memory := memory s |} in
    kernelDataIsolation s → kernelDataIsolation s'.
```

Listing 3.17 – Propriété de conservation de l'isolation noyau entre l'état s et s'

Il est nécessaire de déplier la définition de l'isolation noyau pour commencer à établir la preuve. La propriété avec la définition dépliée est disponible en listing 3.18.

```
Lemma kernelDataIsolationActivate (partDesc : page) (s : state) :
    let s' := {| currentPartition := partDesc; memory := memory s |} in

    (forall partition1 partition2 : page,
    In partition1 (getPartitions pageRootPartition s) →
    In partition2 (getPartitions pageRootPartition s) →
    Lib.disjoint (getAccessibleMappedPages partition1 s)
                 (getConfigPages partition2 s))
    →
    forall partition1 partition2 : page,
    In partition1 (getPartitions pageRootPartition s') →
    In partition2 (getPartitions pageRootPartition s') →
    Lib.disjoint (getAccessibleMappedPages partition1 s')
                 (getConfigPages partition2 s')
```

Listing 3.18 – Propriété de conservation de l'isolation noyau entre l'état s et s', où la définition de l'isolation noyau a été dépliée

Cette propriété utilise trois fonctions sur l'état : `getPartitions`, `getAccessibleMappedPages` et `getConfigPages`. La première étape est de montrer que le résultat produit par ces fonctions ne dépend pas du champ `currentPartition` de l'état. Par exemple pour la fonction `getPartitions`, on veut montrer la propriété décrite en listing 3.19.

Une preuve similaire devra être produite pour les fonctions `getAccessibleMappedPages` et `getConfigPages`. Dès lors, en substituant les appels aux fonctions sur s' par des appels sur s grâce aux nouvelles propriétés sur les fonctions, les deux côtés de l'implication de la propriété de préservation de l'isolation du noyau deviennent identiques. Le reste de la preuve de la propriété consiste à prouver $A \implies A$.

```
Lemma partitionTreeRemains (partDesc : page) (s : state) :
    let s' := {| currentPartition := partDesc; memory := memory s |} in
    forall (root : page),
        getPartitions root s = getPartitions root s'.
```

Listing 3.19 – Propriété stipulant que la fonction `getPartitions` retourne le même résultat pour l'état s ou l'état s', et ce quels que soient l'argument `root` ou la valeur du champ `currentPartition` de l'état

La méthode de preuve reste la même pour toutes les autres fonctions et propriétés restantes, que nous ne passerons donc pas en revue. Notez néanmoins que ce processus de preuve est *extrêmement* fastidieux. En effet, pour repartir de notre exemple, la preuve d'équivalence de `getPartitions` repose sur la définition de la fonction `getChildren`. Il faut donc d'abord prouver l'équivalence des résultats de `getChildren` avant de pouvoir réaliser la preuve concernant `getPartitions`. Cependant, `getChildren` repose elle-même sur les fonctions `getMappedPagesAux`, `getPd` et `getPdsVaddr` dont il faudra aussi prouver l'équivalence avant de pouvoir prouver l'équivalence de `getChildren`, et ainsi de suite jusqu'à arriver aux définitions élémentaires qui pourront être prouvées directement. De plus, cette preuve doit être réalisée sur l'ensemble des propriétés d'isolation et les 23 propriétés de cohérence restantes de Pip. Nous donnerons quelques métriques concernant cette preuve dans la section suivante.

Sauvegarde du contexte d'exécution de la partition appelante

La preuve de conservation de l'isolation de la fonction `writeContext` repose sur le fait que les pages de configuration spécifiques à Pip ainsi que celles concernant les espaces d'adressage des partitions restent inchangées. Pour arriver à cette conclusion, on procède de la manière suivante. Tout d'abord, on montre que dans l'état initial s, les pages dans lesquelles la fonction `writeContext` va écrire sont des pages accessibles à *une* partition. Ensuite, on utilise la propriété d'isolation du noyau (démontrable sur l'état initial) pour en déduire que ces pages ne font pas partie de l'ensemble des pages de configuration de Pip. Enfin, on déduit que les pages de configuration restent inchangées entre l'état initial s et le nouvel état s' car s' est le résultat des écritures sur des pages n'appartenant pas aux pages de configuration. De cette manière, on peut montrer que les propriétés prouvables sur l'état s restent prouvables sur l'état suivant s', car la configuration du noyau n'a pas changé.

Cette preuve est détaillée dans la suite de cette sous-section. On pose que s représente l'état (tel que définit dans le modèle) juste avant les écritures de la fonction `writeContext`, et on pose s' comme étant l'état résultant des écritures de `writeContext`.

Démonstration que les pages appartiennent aux pages accessibles à la partition La première étape est de montrer que les pages concernées par les écritures de `writeContext` sont des pages accessibles à l'espace utilisateur. Les deux pages concernées sont `contextPage` et `contextEndPage`.

Ces propriétés proviennent nécessairement des tests effectués sur les paramètres passés à `writeContext` avant l'appel de la fonction, et sont donc des préconditions à cette preuve. Cette précondition pourrait être énoncée formellement de la sorte :

$$\begin{aligned} &\exists p,\ p \in \texttt{getPartitions}\ s, \\ &\texttt{contextPage} \in \texttt{getAccessibleMappedPages}\ p\ s\ \wedge \\ &\texttt{contextEndPage} \in \texttt{getAccessibleMappedPages}\ p\ s \end{aligned}$$

▸ **Propriété 3.4.** Il existe une partition p valide dans l'état s, telle que les pages `contextPage` et `contextEndPage` sont dans l'espace d'adressage de p et accessibles sans privilège, comme calculé par la fonction `getAccessibleMappedPages`. ◂

Démonstration que les pages n'appartiennent pas aux pages de configuration L'étape suivante de cette preuve de préservation de l'isolation est de montrer que les pages `contextPage` et `contextEndPage` ne sont pas dans l'ensemble des pages de configuration de Pip. Cette propriété peut être formulée formellement comme suit :

$$\begin{aligned} &\forall p,\ p \in \texttt{getPartitions}\ s, \\ &\texttt{contextPage} \notin \texttt{getConfigPages}\ p\ s\ \wedge \\ &\texttt{contextEndPage} \notin \texttt{getConfigPages}\ p\ s \end{aligned}$$

▸ **Propriété 3.5.** Pour toute partition p valide dans l'état s, les pages `contextPage` et `contextEndPage` ne sont pas dans l'ensemble des pages de configuration de la partition p, comme calculé par la fonction `getConfigPages`. ◂

Cette propriété peut être déduite de la propriété d'isolation du noyau `kernelDataIsolation` que voici :

$$\begin{aligned} &\forall\ p_1,\ p_1 \in \texttt{getPartitions}\ s, \\ &\forall\ p_2,\ p_2 \in \texttt{getPartitions}\ s, \\ &(\texttt{getAccessibleMappedPages}\ p_1\ s \cap \texttt{getConfigPages}\ p_2\ s) = \emptyset \end{aligned}$$

En prenant $p_1 = p$ dans cette propriété, cela nous donne :

$$\forall\ p_2,\ \texttt{getAccessibleMappedPages}\ p\ s \cap \texttt{getConfigPages}\ p_2\ s = \emptyset$$

Or les préconditions nous donnent $\exists\, p,\ \texttt{contextPage} \in \texttt{getAccessibleMappedPages}\ p\ s$ ainsi que $\exists\, p,\ \texttt{contextEndPage} \in \texttt{getAccessibleMappedPages}\ p\ s$. Ainsi, on en déduit :

$$\forall p_2, \texttt{contextPage} \notin \texttt{getConfigPages}\ p_2\ s \wedge \texttt{contextEndPage} \notin \texttt{getConfigPages}\ p_2\ s$$

Définition alternative de s' Cette étape consiste à exprimer l'état s' en fonction de l'état initial s, afin de pouvoir appliquer à l'état s' des propriétés prouvables sur s. Cette définition consiste à isoler les écritures de la fonction `writeContext` dans un ensemble E afin de faire apparaître l'état de la mémoire original (`memory` s). On souhaite avoir :

$$\texttt{memory}\ s' = (\texttt{memory}\ s) \cup E$$

En particulier, on pourrait montrer la propriété suivante :

$$\begin{aligned}
&\exists\, E : \texttt{list}\,(\texttt{paddr} * \texttt{value}),\\
&\quad s' = \{\, \texttt{currentPartition}\ s\,;\, (\texttt{memory}\ s) \cup E\,\},\\
&\forall\, e : (\texttt{paddr} * \texttt{value}),\\
&\quad e \in E \implies\\
&\exists\, i : \mathit{index},\\
&\quad e = (\texttt{contextPage}\ i) * \texttt{defaultVAddr} \vee\\
&\quad e = (\texttt{contextEndPage}\ i) * \texttt{defaultVAddr}
\end{aligned}$$

▶ **Propriété 3.6.** Il existe un ensemble E de couples d'adresses et de valeurs, tel que la mémoire de l'état s' peut être représentée par l'union de la mémoire de l'état s avec l'ensemble E, et pour toute paire d'adresse et de valeur e appartenant à l'ensemble E, alors il existe un indice i, tel que e est constitué d'une adresse pointant vers la page `contextPage` à l'indice i ou la page `contextEndPage` et l'indice i, et dont la valeur à cette adresse est une valeur par défaut. ◀

Cette propriété se déduit de l'appel à la fonction `writeContext` (voir listing 3.10), et plus particulièrement de la fonction `storeVirtual`. Cette fonction simule l'activité de la MMU en traduisant l'adresse virtuelle passée en paramètre et en écrivant à cette adresse. C'est pourquoi cette preuve requiert des préconditions supplémentaires à celles déjà identifiées. Une première garantit que l'adresse virtuelle de sauvegarde de contexte `sourceContextSaveVAddr` passée à la fonction `writeContext` correspond à la page de mémoire réelle `contextPage`. La seconde garantit que la dernière adresse virtuelle du contexte `sourceContextEndSaveVAddr` correspond bien à la page de mémoire réelle `contextEndPage`, afin de pouvoir fournir la preuve que la page reste accessible même si les écritures de `writeContext` débordent sur la page suivante. On suppose aussi que la taille d'un contexte d'exécution est inférieure à celle d'une page mémoire, cette propriété est ajoutée en tant qu'axiome.

Démonstration que les pages de configuration restent inchangées Cette propriété consiste à montrer que, quelles que soient les pages de configuration du noyau dans s, alors ces pages seront aussi les pages de configuration du noyau dans l'état s'. Cette propriété est la propriété clée de la démonstration de la préservation de l'isolation, permettant de prouver toutes les propriétés restantes. Elle est exprimée de la sorte :

$$\begin{aligned} &\forall\, p,\ p \in \texttt{partitionTree}\ s, \\ &\texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s' \end{aligned}$$

▸ **Propriété 3.7.** Pour toute partition p valide dans l'état s, l'ensemble des pages de configuration de la partition p dans l'état s est égal à l'ensemble des pages de configuration de la partition dans l'état s'. ◂

Cette propriété se déduit de la propriété 3.6 précédente redéfinissant s' en définissant l'ensemble E contenant toutes les écritures de la fonction `writeContext`, et de la propriété 3.5 indiquant que les pages `contextPage` et `contextEndPage` ne font partie des pages de configuration d'aucune partition valide dans l'état s. De cette manière, on pourrait réaliser une preuve par récurrence sur l'ensemble E pour raisonner de manière générique sur chaque élément. L'hypothèse de récurrence est la suivante :

$$\begin{aligned} E &= \bigcup_{i=1}^{\texttt{contextSize}} e_i, \\ &\exists\, n,\ 0 \leq n < \texttt{contextSize}, \\ \text{let } E_n &= \begin{cases} \emptyset & \text{si } n = 0 \\ \bigcup_{i=1}^{n} e_i & \text{si } 0 < n \leq \texttt{contextSize} \end{cases} \\ \text{let } s_n &= \{\, \texttt{currentPartition}\ s\ ;\ (\texttt{memory}\ s) \cup E_n \,\}, \\ &\forall\, p,\ p \in \texttt{getPartitions}\ s \\ &\texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s_n \end{aligned} \tag{3.1a}$$

Le cas de base est trivial. Pour $n = 0$, on a $s_0 = s$ et on doit prouver :

$$\begin{aligned} &\forall\, p,\ \texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s_0 \\ \Longrightarrow\ &\forall\, p,\ \texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s \end{aligned}$$

On suppose que l'hypothèse de récurrence est vraie au rang n (avec $n < \texttt{contextSize}$), et on cherche à prouver que cette hypothèse est vraie au rang $n+1$. On cherche à prouver :

$$\begin{aligned} &\forall\, p,\ p \in \texttt{getPartitions}\ s \\ &\texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s_{n+1} \end{aligned}$$

On peut exprimer s_{n+1} en fonction de s_n et de s. En effet, on a :

$$\begin{aligned} & s_{n+1} = \{\, \texttt{currentPartition}\ s\ ;\ (\texttt{memory}\ s) \ \cup\ E_{n+1} \,\} \\ \Longrightarrow\ & s_{n+1} = \{\, \texttt{currentPartition}\ s\ ;\ (\texttt{memory}\ s) \ \cup\ E_n \ \cup\ e_{n+1} \,\} \\ \Longrightarrow\ & s_{n+1} = \{\, \texttt{currentPartition}\ s\ ;\ (\texttt{memory}\ s_n) \ \cup\ e_{n+1} \,\} \end{aligned} \tag{3.1b}$$

Or, d'après la propriété 3.6, on a :

$$\begin{aligned} e_{n+1} &= (\texttt{contextPage}\ i) \quad * \ \texttt{defaultVAddr} \ \vee \\ e_{n+1} &= (\texttt{contextEndPage}\ i) * \texttt{defaultVAddr} \end{aligned} \tag{3.1c}$$

De plus, d'après la propriété 3.5, on a :

$$\begin{aligned} \texttt{contextPage} \notin \texttt{getConfigPages}\ p\ s\ \wedge \\ \texttt{contextEndPage} \notin \texttt{getConfigPages}\ p\ s \end{aligned} \tag{3.1d}$$

L'hypothèse de récurrence nous permet de déduire que :

$$\begin{aligned} \texttt{contextPage} \notin \texttt{getConfigPages}\ p\ s_n\ \wedge \\ \texttt{contextEndPage} \notin \texttt{getConfigPages}\ p\ s_n \end{aligned} \tag{3.1e}$$

Ainsi, on en déduit que l'écriture de l'élément e_{n+1} dans la mémoire n'a pas d'influence sur la fonction $\texttt{getConfigPages}\ p\ s_n$, et donc que :

$$\begin{aligned} \forall\ p,\ p \in \texttt{getPartitions}\ s, \\ \texttt{getConfigPages}\ p\ s_{n+1} = \texttt{getConfigPages}\ p\ s_n \end{aligned} \tag{3.1f}$$

On applique une nouvelle fois l'hypothèse de récurrence, qui nous permet enfin de conclure que :

$$\begin{aligned} \forall\ p,\ p \in \texttt{getPartitions}\ s, \\ \texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s_{n+1} \end{aligned} \tag{3.1g}$$

Cela reste vrai pour $n < \texttt{contextSize}$. En particulier, si $n = \texttt{contextSize} - 1$, on a $s_{n+1} = s'$. On en déduit alors la propriété 3.7 :

$$\begin{aligned} \forall\ p,\ p \in \texttt{getPartitions}\ s, \\ \texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s' \end{aligned}$$

Exemple de la preuve de la propriété d'isolation du noyau À partir de la propriété 3.7, il est possible de déduire le reste des propriétés nécessaires à la preuve de préservation de l'isolation. Ce paragraphe montre à titre d'exemple comment arriver à la propriété d'isolation pour l'état s'. De la même manière que la preuve de la propriété d'isolation noyau a été conduite sur l'état changeant la partition courante dans la section 3.3.3, il s'agit de déplier chaque fonction pour prouver que le résultat des fonctions est égal entre un appel sur l'état s et un appel sur l'état s'.

$$\texttt{kernelDataIsolation}\ s \implies \texttt{kernelDataIsolation}\ s'$$

En dépliant la définition de `kernelDataIsolation`, on constate qu'il faut prouver l'égalité des appels sur les fonctions `getConfigPages` et `getAccessibleMappedPages`. La propriété 3.7 nous donne l'égalité sur la fonction `getConfigPages`. On souhaite prouver l'égalité des appels de la fonction `getAccessibleMappedPages`.

`getAccessibleMappedPages` $p\ s$ parcourt les pages de configurations de la MMU de la partition p dans l'état s pour récupérer les pages marquées comme présentes et accessibles dans l'espace utilisateur. Or la propriété 3.7 nous donne que les pages de configuration restent les mêmes et la propriété 3.6 nous indique que les écritures n'ont pas lieu dans ces pages de configuration. Ainsi, les appels à `getAccessibleMappedPages` restent égaux entre l'état s et l'état s'. On a donc :

$$\begin{aligned} &\forall\ p,\ p \in \texttt{getPartitions}\ s, \\ &\texttt{getConfigPages}\ p\ s = \texttt{getConfigPages}\ p\ s' \ \wedge \\ &\texttt{getAccessibleMappedPages}\ p\ s = \texttt{getAccessibleMappedPages}\ p\ s' \end{aligned}$$

Ce qui nous permet de déduire que la propriété d'isolation du noyau est prouvable sur s'. L'ensemble des proprétés restantes sur s' sont prouvables par la même méthode.

Validation des paramètres

Pour parvenir à prouver la préservation des propriétés d'isolation et de cohérence lors des différentes instructions de modification de l'état du service, il est nécessaire d'amener des propriétés sur les paramètres fournis par l'utilisateur ainsi que sur les valeurs intermédiaires calculées pendant l'exécution du service.

Préconditions requises par la preuve du changement de partition courante Comme indiqué dans la section 3.3.3, certaines propriétés nécessaires à l'établissement de la preuve doivent être fournies avant l'appel de la fonction `updateCurPartition`.

$$\begin{aligned} &\texttt{In targetPartDesc (getPartitions pageRootPartition}\ s) \ \wedge \\ &\texttt{targetPartDesc} \neq \texttt{pageDefault} \end{aligned}$$

Ces propriétés sont récupérées lors des vérifications préalables sur le descripteur de partition qui sont effectuées soit dans le bloc de code `getChildPartDescCont`, soit dans le bloc `getParentPartDescCont`.

Le bloc `getChildPartDescCont` vérifie d'abord que l'adresse virtuelle passée en argument est effectivement dans l'espace d'adressage de la partition, puis appelle la fonction `checkChild` sur cette adresse virtuelle pour vérifier que cette adresse correspond bien au descripteur d'une partition enfant. Le bloc traduit ensuite l'adresse virtuelle pour récupérer la page réelle du descripteur de partition.

La propriété `childInPartTree`, exprimant que la fonction `checkChild` renvoie `true` seulement si l'adresse virtuelle correspond à un descripteur de partition valide, a déjà été prouvée. Elle est décrite dans le listing 3.20.

```
Lemma childInPartTree (childPartDesc parentPartDesc : page)
                      (parentPageDir childLastMMUPage : page)
                      (s : state)
                      (nbL : level)
                      (childPartDescVAddr : vaddr)
                      (idxChildPartDesc : index) :
consistency s
→ Some nbL = getNbLevel
→ nextEntryIsPP parentPartDesc idxPageDir parentPageDir s
→ currentPartition s = parentPartDesc
→ isPE childLastMMUPage (getIndexOfAddr childPartDescVAddr levelMin) s
→ getTableAddrRoot childLastMMUPage idxPageDir
                   parentPartDesc childPartDescVAddr s
→ pageDefault <> childLastMMUPage
→ getIndexOfAddr childPartDescVAddr levelMin = idxChildPartDesc
→ isEntryPage childLastMMUPage idxChildPartDesc childPartDesc s
→ entryPresentFlag childLastMMUPage idxChildPartDesc true s
→ true = checkChild parentPartDesc nbL s childPartDescVAddr
→ In childPartDesc (getPartitions pageRootPartition s).
```

Listing 3.20 – Propriété `childInPartTree` permettant d'affirmer que le descripteur de partition d'une partition enfant est un descripteur de partition valide

Cette propriété à l'allure intimidante n'est en fait que l'accumulation des propriétés récupérées lors d'une traduction d'adresse virtuelle et du parcours des tables de la MMU, suivi d'un appel à `checkChild`. De plus amples détails sur des préconditions similaires seront donnés dans la sous-section suivante 3.3.3. En posant `childPartDesc = targetPartDesc`, elle permet de récupérer la propriété `In targetPartDesc (getPartitions pageRootPartition s)`.

La seconde propriété à récupérer, `targetPartDesc ≠ defaultPage` peut être déduite de la propriété de cohérence `isPresentNotDefaultIff`, décrite en listing 3.21.

```
Definition isPresentNotDefaultIff s :=
forall table idx ,
    readPresent  table idx (memory s) = Some false ⟷
    readPhyEntry table idx (memory s) = Some pageDefault.
```

Listing 3.21 – Propriété de cohérence `isPresentNotDefaultIff`

Cette propriété de cohérence indique que la fonction `readPresent` retourne `false` si et seulement si la valeur lue dans la `table` à l'indice `idx` est égale à `pageDefault`.
Or, la traduction de l'adresse virtuelle du descripteur de partition de la partition enfant nous donne :

$$\texttt{readPresent childMMUTable idxChildPD (memory s) = Some true} \implies$$
$$\texttt{readPhyEntry childMMUTable idxChildPD (memory s) ≠ Some pageDefault}$$

De plus, la lecture du descripteur de partition nous donne :

$$\texttt{readPhyEntry childMMUTable idxChildPD (memory s) = Some targetPartDesc}$$

Ainsi, on en déduit que `targetPartDesc ≠ defaultPage`, complétant les préconditions requises par la preuve concernant les partitions enfants.

On considère à présent ce qu'il se passe dans le bloc `getParentPartDescCont` concernant l'appel à la partition parent. Il vérifie que la partition appelante n'est pas la partition racine, puis récupère le descripteur de partition du parent au travers d'une entrée du descripteur de partition de l'enfant. La définition de cette lecture nous donne la propriété suivante :

$$\texttt{nextEntryIsPP sourcePartDesc idxParentDesc targetPartDesc s}$$

Cette propriété permet notamment d'utiliser la propriété de consistence `parentInPartitionList` (présenté en listing 3.22) pour récupérer la propriété sur le fait que la partition parent est une partition valide.
Cette propriété de cohérence stipule que pour tout descripteur de partition valide, si on lit à l'indice `idxParentDesc` du descripteur la valeur `parent`, alors `parent` est un descripteur de partition valide. Si on prend `parent = targetPartDesc`, on déduit de cette propriété que `In targetPartDesc (getPartitions pageRootPartition s)`.
La seconde propriété à récupérer, `targetPartDesc ≠ defaultPage`, se déduit de la propriété de cohérence `partitionDescriptorEntry`, présenté en listing 3.23.

```
Definition parentInPartitionList (s : state) :=
forall partition, In partition (getPartitions pageRootPartition s) →
forall parent, nextEntryIsPP partition idxParentDesc parent s →
In parent (getPartitions pageRootPartition s).
```

Listing 3.22 – Propriété de cohérence `parentInPartitionList`

```
Definition partitionDescriptorEntry s :=
forall (partition : page),
In partition (getPartitions pageRootPartition s) →
forall (idxroot : index),
( (* disjonction de cas sur la valeur de idxroot *)
    (*[ ... ]*) ∨
    idxroot = idxParentDesc
) →
(*[ ... ]*) ∧
exists entry,
    nextEntryIsPP partition idxroot entry s ∧
    entry <> pageDefault.
```

Listing 3.23 – Propriété de cohérence `partitionDescriptorEntry`

Cette propriété de cohérence stipule que pour tout descripteur de partition valide, il existe une valeur `entry` à l'indice `idxParentDesc` du descripteur, telle que `entry` ≠ `defaultPage`. Or la propriété issue de la lecture nous donne que `targetPartDesc` est le candidat idéal pour `entry`, ce qui permet de déduire que `targetPartDesc` ≠ `defaultPage`.

Préconditions requises par la preuve d'écriture des contextes Les propriétés requises pour pouvoir prouver la préservation des propriétés d'isolation lors de l'écriture des contextes d'exécution sont relatives aux adresses virtuelles de contexte `contextSaveAddr` et `contextEndSaveAddr`. Deux propriétés sont attendues pour chaque adresse virtuelle : la première atteste que l'adresse virtuelle correspond à une page mémoire dans l'espace d'adressage de la partition, la seconde atteste que cette page mémoire est une page mémoire accessible en espace utilisateur. Ces deux préconditions sont récupérées dans le bloc `saveSourceContextCont` avant l'appel à la fonction `writeContext`.

La première propriété est semblable aux premières préconditions requises pour le lemme `childInPartTree` disponible en listing 3.20. Les propriétés requises pour `contextSaveAddr` sont décrites en listing 3.24. Ces propriétés sont récupérées lors de la traduction de

l'adresse virtuelle. Des préconditions similaires sont requises pour l'adresse `contextEndSaveAddr`.

```
getTableAddrRoot ctxMMUPage idxPageDir sourcePartDesc contextSaveAddr s
∧ ctxMMUPage <> pageDefault
∧ getIndexOfAddr contextSaveAddr levelMin = idxCtxMMUPage
∧ isPE ctxMMUPage idxCtxMMUPage s
∧ entryPresentFlag ctxMMUPage idxCtxMMUPage true s
∧ entryUserFlag ctxMMUPage idxCtxMMUPage true s
```

Listing 3.24 – Préconditions requises sur `contextSaveAddr` pour la preuve de conservation d'isolation de la fonction `writeContext`

Chaque propriété est issue d'une étape spécifique de la traduction d'adresse :

```
getTableAddrRoot ctxMMUPage idxPageDir sourcePartDesc contextSaveAddr s
    ∧ ctxMMUPage <> pageDefault
```

Ces deux premières propriétés indiquent qu'il existe une page `ctxMMUPage`, récupérée par la fonction `getTableAddrRoot`, telle que `ctxMMUPage` n'est pas la page par défaut et qu'elle est la dernière page de configuration de la MMU contenant l'entrée vers la page mémoire vers laquelle pointe l'adresse `contextSaveAddr`.

```
getIndexOfAddr contextSaveAddr levelMin = idxCtxMMUPage
```

Cette propriété indique que la page pointée par `contextSaveAddr` se situe à l'indice `idxCtxMMUPage` dans la page `ctxMMUPage`. Cet indice est récupéré par la fonction `getIndexOfAddr`.

```
isPE ctxMMUPage idxCtxMMUPage s
```

Cette propriété découle aussi de l'appel à la fonction `getTableAddr`. `getTableAddr` garantit que si la page retournée (ici en l'occurence `ctxMMUPage`) n'est pas la page par défaut, alors chaque valeur dans cette page sera du type attendu. Ici, la fonction `getTableAddr` a été appelée sur les structures de configuration de la MMU (au travers de l'indice `idxPageDir`), ce qui garantit que les entrées seront de type `physicalEntry`.

```
entryPresentFlag ctxMMUPage idxCtxMMUPage true s
```

Cette propriété est ramenée par l'appel à la fonction `readPresent` qui détermine si la valeur à l'indice `idxCtxMMUPage` de la page `ctxMMUPage` qui représente une page de

mémoire est déclarée comme faisant partie l'espace d'adressage de la partition. Sur l'architecture x86, cette fonction vérifie un bit qui détermine si oui ou non cette entrée est considérée comme valide dans l'espace d'adressage.

```
entryUserFlag ctxMMUPage idxCtxMMUPage true s
```

Cette propriété est ramenée par l'appel à la fonction `readAccessible` qui détermine si la valeur à l'indice `idxCtxMMUPage` de la page `ctxMMUPage` qui représente une page de mémoire est déclarée comme étant accessible sans privilèges particuliers. Sur l'architecture x86, cette fonction vérifie un bit qui détermine si cette entrée peut être utilisée en espace utilisateur.

Comme écrit précédemment, des préconditions similaires sont requises pour l'adresse `contextEndSaveAddr`. L'ensemble des préconditions pour la fonction `writeContext` sont décrites dans le listing 3.25.

```
(* Paramètres *)
(sourcePartDesc, ctxMMUPage, ctxEndMMUPage : page)
(contextSaveAddr, contextEndSaveAddr : vaddr)
(idxCtxMMUPage, idxCtxEndMMUPage : index)
(s : state)

  (* Propriété d'isolation *)
  partitionsIsolation s
∧ kernelDataIsolation s
∧ verticalSharing s
∧ consistency s

  (* Propriété indiquant que la partition est valide *)
∧ In sourcePartDesc (getPartitions pageRootPartition s)

  (* Propriété indiquant que contextSaveAddr est présent *)
  (* et accessible dans l'espace d'adressage             *)
∧ getTableAddrRoot ctxMMUPage idxPageDir sourcePartDesc
                                         contextSaveAddr s
∧ ctxMMUPage <> pageDefault
∧ getIndexOfAddr contextSaveAddr levelMin = idxCtxMMUPage
∧ isPE ctxMMUPage idxCtxMMUPage s
∧ entryPresentFlag ctxMMUPage idxCtxMMUPage true s
∧ entryUserFlag ctxMMUPage idxCtxMMUPage true s

  (* Propriété indiquant que contextSaveEndAddr est présent *)
  (* et accessible dans l'espace d'adressage                *)
∧ getTableAddrRoot ctxEndMMUPage idxPageDir sourcePartDesc
                                            contextEndSaveAddr s
∧ ctxEndMMUPage <> pageDefault
∧ getIndexOfAddr contextEndSaveAddr levelMin = idxCtxEndMMUPage
∧ isPE ctxEndMMUPage idxCtxEndMMUPage s
∧ entryPresentFlag ctxEndMMUPage idxCtxEndMMUPage true s
∧ entryUserFlag ctxEndMMUPage idxCtxEndMMUPage true s
```

Listing 3.25 – Préconditions de la fonction `writeContext` et ses paramètres

3.4 Retour d'expérience

Les travaux exposés dans cette section sont disponibles gratuitement et en accès libre sur Github. Ils ont conduits à l'écriture d'un court papier disponible gratuitement et en accès libre sur HAL, qui ne détaille cependant pas la preuve décrite dans ce document.

3.4.1 Métriques

Le nouveau service compte 417 lignes de code Coq. Ce code est très peu dense, et a été écrit pour maximiser la lisibilité. Par exemple, certains appels de fonction se retrouvent sur plusieurs lignes et des commentaires parsèment le code. À titre de comparaison, le code C généré par Digger (l'outil permettant de compiler notre code shallow-embedded vers du C, voir section 2.3.2) comprend 471 lignes ; ce code comprend un appel de fonction, un argument, ou une accolade par ligne, sans commentaire.

Le nombre de lignes de code permettant d'appeler le code du service représente environ 1000 lignes de code C et assembleur sur l'architecture Intel x86 (en incluant les commentaires). Néanmoins, ce code est *particulièrement peu dense* de par sa complexité ; en moyenne, pour chaque ligne de code ou instruction assembleur, il y a environ deux lignes de commentaires. Pour rappel, ce service permet de gérer tous les transferts de flot d'exécution au sein d'un système.

La preuve permettant de prouver le maintien des propriétés d'isolation sur l'état après exécution du service représente actuellement un peu plus de 4000 lignes de preuve en comptant les définitions de modèles et les scripts d'établissement de la preuve. Ce chiffre est dérisoire comparé aux plus de 80 000 lignes relatives à la preuve déjà écrites pour Pip. De plus, l'établissement de la preuve d'isolation de ce service n'a duré "que" quelques semaines, alors que je n'avais aucune expérience dans ce domaine. Ces chiffres semblent montrer que l'effort de preuve diminue avec la proportion de preuve déjà existante, indiquant une forte réutilisation des preuves déjà établies. De plus, le temps *relativement* restreint passé sur la preuve initiale du service semble indiquer que la méthodologie de preuve proposée par Pip est efficace et permet aux novices de conduire efficacement un raisonnement sur du code.

3.4.2 Modélisation

Tous les modèles utilisés pour représenter l'objet étudié nous permettent de mieux comprendre l'objet réel. Le raisonnement est guidé par la représentation que nous avons de l'objet, qui évolue avec notre compréhension de l'objet. Ainsi, lorsque l'étude de l'objet avance, les modèles utilisés pour l'étude peuvent finalement représenter une vision partielle ou même erronnée de l'objet étudié : cette vision ayant été améliorée et élargie lors de son étude.

En ce sens, les modèles nous incitent à toujours apporter des modifications reflétant l'état actuel de notre compréhension de l'objet, puis dévoilants leurs nouvelles limites au

fur et à mesure de l'étude de l'objet qu'ils représentent. Cette course sans fin est à l'origine des nouveaux modèles des fonctions et de la preuve exposée dans la section 3.3.

3.4.3 Prise de recul sur la nature de la preuve

Le but ultime de l'établissement de cette preuve est de convaincre le plus rapidement et le plus robustement possible l'assistant de preuve que la preuve est valide. Cette preuve est donc la preuve la plus rapide et non la preuve la plus didactique, à contrario de ce qui est habituellement recherché dans une preuve mathématique.

La preuve de maintien de l'isolation pour ce service ne présentait que peu d'intérêt en dehors du contexte dans lequel elle a été réalisée : les écritures étant faites en espace utilisateur, il était évident que la preuve ne présenterait aucun obstacle majeur. Cette preuve me semblait presque être une tautologie des propriétés d'isolation : si Pip écrit dans l'espace d'adressage d'une partition cela brise-t'il les propriétés d'isolation ? Évidemment non, puisque les partitions, isolées et dénuées de privilèges, sont elles mêmes capables d'écrire dans leur propre espace d'adressage.

Cette preuve d'isolation n'étant pas pertinente, on pourrait alors se demander quelles seraient les propriétés intéressantes à prouver sur ce service. Une idée prometteuse serait de réaliser une preuve *fonctionnelle* du service représentant sa spécification formelle – à contrario des preuves existantes de Pip qui ne s'en soucient aucunement. Cette idée se retrouva rapidement confrontée à un obstacle majeur. En effet, le modèle de Pip est intrinséquement lié aux propriétés d'isolation : de nombreuses fonctions – dont la fonction changeant effectivement l'espace d'adressage – n'ont simplement pas de modèle car le modèle de Pip a été conçu pour représenter de manière minimale la partie de l'état de la machine nécessaire à prouver l'isolation. Remplacer l'espace d'adressage par celui de la nouvelle partition courante est pourtant une des fonctions principales de ce service. Ne pas prouver une telle propriété diminuerait fortement l'intérêt de cette preuve fonctionnelle.

Dans l'état actuel de Pip, c'est une impasse : le modèle ne permet pas de formuler cette propriété intéressante. Seule une réécriture complète du code avec un nouveau modèle, conçu cette fois-ci pour intégrer les propriétés intéressantes pour ce nouveau service, permettrait d'y arriver. Cette approche nécessiterait aussi de réécrire les preuves déjà établies sur le modèle actuel. C'est pourquoi cette solution n'est pas satisfaisante : imaginons qu'après avoir implémenté ce nouveau modèle, on souhaite inclure de nouvelles propriétés. De la même manière que la propriété que je souhaite inclure dans le modèle actuel, il est probable que la nouvelle propriété ne soit pas formulable ou prouvable dans le nouveau modèle. D'autres solutions pourraient être envisagées, mais il est peu raisonnable d'espérer créer un modèle « universel » : un tel modèle est en contradiction avec l'essence même de ce qu'est un modèle.

Une alternative à cette démarche serait de réécrire des modèles en limitant le plus possible l'effort d'implémentation et le portage des anciennes preuves sur celui-ci. En particulier, si les propriétés à prouver (qui ne relèvent que du monde des mathématiques) diffèrent et sont amenées à évoluer, le code et l'interface de la monade restent constant dans tous ces

modèles. Il serait alors intéressant de pouvoir garder intactes les preuves d'anciennes propriétés sur d'anciens modèles, tout en proposant de nouvelles propriétés sur de nouvelles implémentations du modèle. Cette piste de travail a mené à la troisième contribution de cette thèse.

3.5 Synthèse du travail effectué et des contributions de ce chapitre

Cette brève section vise à résumer les contributions des travaux présentés dans ce chapitre. Ce chapitre a présenté un service de transfert de flot d'exécution unifié au sein du noyau Pip dont la propriété formelle classique de Pip sur la préservation de l'isolation a été établie. Le service permet de gérer de manière unifiée les transferts explicites de flot d'exécution, les interruptions logicielles, les fautes, ainsi que les interruptions matérielles au travers de points d'entrées placés de manière opportune le long d'un fil d'exécution unique. Ce service de transfert de flot d'exécution, présenté ici pour l'architecture Intel x86, n'est cependant pas spécifique à cette architecture, et a notamment été implémenté sur l'architecture Armv7. Les modèles ayant permis la preuve de préservation de l'isolation initiale ont été enrichis, décrivant de manière plus précise les comportements du service ; ce document présente une ébauche détaillée de preuve de préservation de l'isolation sur ces modèles.

De plus, ces travaux viennent contribuer à la complétion de la preuve de préservation de l'isolation sur l'ensemble des services de Pip. Ils corroborent les résultats de la méthodologie de co-design de Pip, visant à fournir les services strictement nécessaires à un système fonctionnel tout en réduisant l'effort de preuve requis.

4 Politique d'ordonnancement prouvée

Le chapitre précédent a porté sur un service de transfert de flot d'exécution au sein de Pip. Ce service permet de gérer les transferts explicites de flot d'exécution, ainsi que les interruptions. De cette manière, Pip fournit les outils nécessaires à la préemption de flots d'exécution. La préemption est nécessaire à la mise en place d'ordonnancement au sein d'un système ; elle n'est cependant pas suffisante pour implémenter un ordonnanceur. Celui-ci doit être muni d'une politique d'ordonnancement, lui permettant de choisir le prochain flot d'exécution.

Ce chapitre porte sur l'implémentation d'un ordonnanceur Earliest Deadline First (abrégé EDF) s'exécutant en espace utilisateur. L'ordonnanceur est conçu pour fonctionner dans une partition de Pip, et est muni d'une preuve formelle que sa fonction d'élection respecte la politique d'ordonnancement EDF. Ces travaux ont été menés conjointement avec Vlad Rusu, avec qui j'ai conçu les modèles sur lesquels reposent la preuve et qui a établit les preuves formelles sur cet ordonnanceur. Dans ce travail commun, ma contribution a été de concevoir les composants de l'ordonnanceur, leur interface ainsi qu'une partie de leur modèle, ainsi que d'écrire l'implémentation pour que cet ordonnanceur puisse être compilé et s'exécuter dans une partition de Pip. [2].

Ces travaux ont fait l'objet d'une publication et d'une présentation à RTAS2022, dont ce chapitre s'inspire fortement. Le papier est disponible à l'adresse suivante : https://hal.archives-ouvertes.fr/hal-03671598. La vidéo de la présentation est disponible à l'adresse suivante : https://pip.univ-lille.fr/recordings/RTAS.mp4. Le dépot contenant le code ainsi que les instructions pour exécuter l'ordonnanceur est hébergé sur Github : https://github.com/2xs/pip_edf_scheduler.

4.1 Motivations

La motivation première de cette contribution est de proposer des garanties supplémentaires aux garanties d'isolation classiques de Pip. Cette motivation nous a mené à nous intéresser aux problématiques des systèmes temps réel, et notamment sur le respect des échéances, car nous pensons que la vision de Pip peut être pertinente. La preuve formelle du respect de la politique d'ordonnancement est un premier pas vers des garanties temps réel dans l'écosystème de Pip.

2. David Nowak, Gilles Grimaud mais aussi Samuel Hym ont aussi contribué à ce projet de manière plus modeste. David a participé à l'élaboration des modèles, Gilles a participé à l'établissement des interfaces, et Samuel a participé à l'écriture de l'implémentation

Cette contribution est aussi un moyen d'éprouver la méthode de développement des logiciels et preuves de Pip, en l'appliquant à un autre objet d'étude. On peut noter deux différences fondamentales entre Pip et cet ordonnanceur. La première est que Pip s'exécute en mode privilégié, contrairement à l'ordonnanceur qui s'exécute en espace utilisateur. Cette différence a des répercussions sur la conception des interfaces ; la dépendance à du code non prouvé est moins critique en espace utilisateur. Cette différence a aussi un impact implicite sur les preuves : la logique de Hoare n'est valable que lorsque l'état ne change pas entre deux instructions – il faut donc s'assurer que les interruptions ne perturbent pas l'ordonnanceur. La seconde réside dans le fait que les propriétés prouvées sur l'ordonnanceur relèvent de l'ordre des propriétés fonctionnelles, c'est à dire qui décrivent le comportement de l'ordonnanceur. Les propriétés d'isolation de Pip traditionnellement prouvées au travers de cette méthode ne sont pas des propriétés fonctionnelles. Aussi, nous ne chercherons pas à établir des propriétés non-fonctionnelles sur la préemption telles que des propriétés sur le partage du temps d'exécution du processeur. Ces deux différences font de cet ordonnanceur un objet d'étude intéressant du point de vue du développement logiciel.

De plus, cette contribution utilise tous les aspects du service de transfert de flot d'exécution décrit dans le chapitre précédent. Cette contribution permet d'exhiber par l'expérience que l'utilisation de ce service est pertinente dans un composant commun de système d'exploitation.

Enfin, cette contribution apporte une preuve inédite à la communauté des systèmes temps réels. En effet, les travaux montrant de manière formelle des propriétés sur les algorithmes d'ordonnancement sont fréquents dans la communauté. Néanmoins, peu de travaux s'intéressent à la preuve de *l'implémentation* de l'algorithme, qui permettent par exemple de propager les preuves jusqu'au code compilé, et qui tiennent compte de détails supplémentaires (tels que les structures de données par exemple). Cette contribution fournit une preuve formelle que l'implémentation de la fonction d'élection respecte la politique d'ordonnancement EDF pour des *jobs* arbitraires, ce qui n'avait été prouvé sur aucun autre ordonnanceur EDF auparavant. Des travaux indépendants et concomitants sur le projet CertiKOS ont montré une propriété similaire sur un ordonnanceur. Cependant leur ordonnanceur est limité à l'ordonnancement de *jobs* provenant de tâches périodiques.

4.2 Description structurelle

Cette section décrit la structure générale de l'ordonnanceur. Elle commencera par sa partie phare, la fonction d'élection. Cette section en donnera une description fonctionnelle, ainsi que son prototype. Cette section décrira ensuite la place de l'ordonnancement dans Pip. Elle discutera du choix de placer l'ordonnanceur dans une partition, et des différences principales avec un système d'exploitation classique tel que Linux. Enfin, cette section détaillera les composants de l'ordonnanceur, leur fonction ainsi que leurs interactions.

4.2.1 Définition prototype et oracle de la fonction d'élection

La fonction d'élection est le morceau de logiciel que l'ordonnanceur doit appeler à chaque préemption, afin d'élire le prochain flot d'exécution du système. La fonction d'élection fait ce choix en accord avec une politique d'ordonnancement qui lui permet de discriminer les flots d'exécution selon des critères spécifiques qui sont propres à chaque politique. En particulier, la fonction d'élection de notre contribution suit la politique d'ordonnancement *Earliest Deadline First*, qui est une politique temps réel dont la priorité est d'exécuter les *jobs* dont l'échéance est la plus proche.

La fonction d'élection de notre ordonnanceur agit comme un oracle, ne prennant aucun argument et retournant un type composite. Ce type composite contient un booléen indiquant à l'ordonnanceur qu'il doit attendre la prochaine préemption car il n'y a aucun *job* à exécuter. Ce type contient l'identifiant du *job* à exécuter dans le cas contraire, ainsi qu'un autre booléen indiquant si le *job* a dépassé son échéance. Ce dernier booléen sert à informer l'ordonnanceur que l'ensemble des *jobs* n'est pas ordonnançable.

```
Definition CNat := nat.
Definition CBool := bool.
Definition CRet : Type := (option CNat) * CBool.

Definition scheduler : RT CRet.
```

Listing 4.1 – Prototype de la fonction d'élection et définition de son type de retour

b

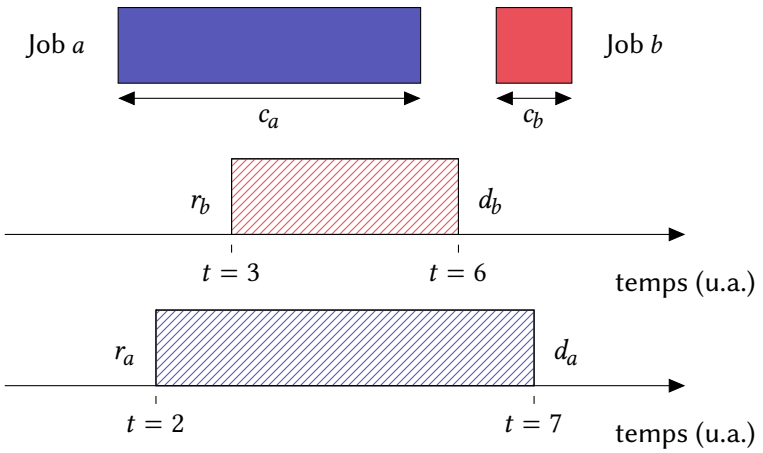

Figure 4.1 – Exemple de jobs pour l'illustration du fonctionnement de la fonction d'élection

Exemple d'appel à la fonction d'élection

Pour rappel, chaque *job* est muni au minimum des informations suivantes : une *release date*, notée r, indiquant l'instant à partir duquel il est possible d'exécuter le *job*, une échéance, notée d, pour laquelle le *job* doit avoir terminé son exécution, et un budget d'exécution, noté c, indiquant le nombre de périodes d'exécution allouées pour ce *job*.

Dans cet exemple, on considère l'ensemble de *jobs* constitué des *jobs a* et *b* représenté sur la Figure 4.1. Le *job a* représenté par un rectangle bleu sur la Figure 4.1 a pour *release date* $r_a = 2$, pour échéance $d_a = 7$, et pour budget d'exécution $c_a = 4$ (indiqué par la longueur du rectangle bleu le représentant). Le *job b* représenté par le rectangle rouge a pour *release date* $r_b = 3$, pour échéance $d_b = 6$, et pour budget d'exécution $c_b = 1$ (indiqué par la longueur du rectangle rouge). La période d'exécution potentielle de chaque *job*, s'étalant de leur *release date* jusqu'à leur échéance, et représentée par les rectangles hachurés de la Figure 4.1.

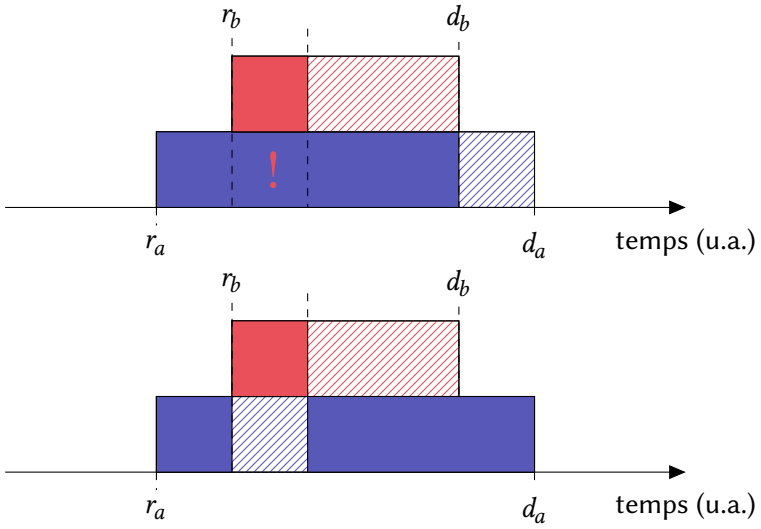

Figure 4.2 – Ordonnancement des jobs selon la politique d'ordonnancement *Earliest Deadline First*

La figure 4.2 présente comment la politique va ordonnancer les *jobs a* et *b*. La partie supérieure de la figure présente un plan d'ordonnancement erroné des deux *jobs* : les *jobs* sont exécutés dès l'arrivée de leur *release date*. Cet ordonnancement n'est pas correct, car il est prévu que les deux *jobs* s'exécutent simultanément sur la période suivant la *release date* du *job b* (ce conflit est indiqué par un point d'exclamation rouge). Ainsi, la politique doit choisir entre l'exécution du *job a* et du *job b* sur cette période. La politique *Earliest Deadline First* privilégie alors le *job* ayant l'échéance la plus proche, dans notre exemple le *job b*. L'exécution du *job a* est alors suspendue jusqu'à la terminaison du *job b*, qui survient avant la période d'exécution suivante puisque son budget d'exécution c_b n'est que d'une

seule période d'exécution. L'exécution du *job a* reprend alors, comme indiqué dans la partie inférieure de la figure 4.2.

La figure 4.3 montre les résultats retournés par la fonction d'élection lors d'appels à des instants t différents. À $t = 1$, ni le *job a*, ni le *job b* ne sont prêts à être exécutés ; la fonction d'élection retourne alors une valeur nulle. À $t = 2$, la *release date* du *job a* est atteinte, ainsi il est prêt à être exécuté ; l'appel à la fonction d'élection retourne le *job a*. À $t = 3$, la *release date* du *job b* est atteinte, ainsi le *job a* et le *job b* sont prêts à être exécutés. C'est alors que le rôle de la fonction d'élection prend son sens et choisi le *job b*, conformément à la politique *Earliest Deadline First*. À $t = 4$, le *job b* a terminé son exécution : il a épuisé son budget d'exécution c_b. Ainsi, la fonction d'élection retourne le *job a*, seul *job* pouvant encore être exécuté.

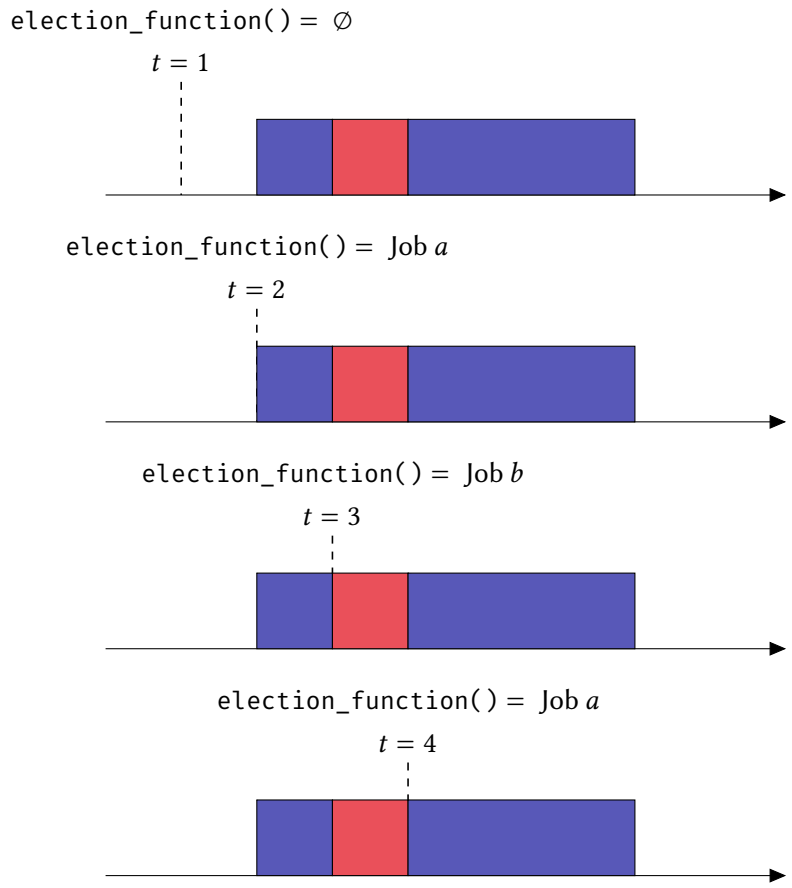

Figure 4.3 – Illustration d'un appel à la fonction d'élection sur les *jobs a* et *b*

4.2.2 Place de l'ordonnancement dans Pip

Cet ordonnanceur ne s'exécute pas en espace privilégié, comme c'est le cas dans les systèmes d'exploitation traditionnels tels que Linux ou même dans des systèmes d'exploitation de niche tels que seL4.

La première raison en faveur d'une implémentation de l'ordonnanceur dans une partition de Pip concerne la méthode de conception des logiciels autour de Pip. En effet, placer l'ordonnanceur en espace utilisateur lui fait implicitement profiter des propriétés d'isolation des partitions de Pip. Ce choix est en adéquation avec la vision de Pip du système, conçu comme une tour de virtualisation. Une tour de virtualisation dont chaque étage devient partie prenante de la base de confiance (TCB) des étages supérieurs. Dans le modèle de Pip, les différents logiciels se construisent les uns au dessus des autres, profitant des propriétés prouvées sur le logiciel sous-jacent.

La seconde raison est une conséquence directe de ce choix, relatif à l'effort de preuve. Si l'ordonnanceur avait été implémenté dans Pip, alors il aurait fallu montrer que l'ordonnanceur respecte les propriétés d'isolation de Pip. À titre de comparaison, la preuve de préservation de l'isolation initialement écrite sur le service décrit dans le chapitre précédent a nécessité plus de 2500 lignes de preuves, alors même que le service ne modifie pas les structures du noyau, et n'introduit pas de nouvelle propriété de consistence. Produire une telle preuve sur l'ordonnanceur aurait sans doute requis un effort d'un tout autre ordre de grandeur, simplement pour prouver la propriété d'isolation. De plus, l'ordonnanceur développé dans cet contribution a été conçu autour de la politique d'ordonnancement EDF. Ce choix est arbitraire, et il serait tout à fait pertinent de le munir d'une autre politique d'ordonnancement. Ainsi, cet effort aurait été requis pour chaque autre variante de l'ordonnanceur engendrée par chaque politique souhaitée – encore une fois uniquement pour garantir l'isolation. Cette propriété est *donnée* dès lors que l'ordonnanceur est sorti du code du noyau.

4.2.3 Décomposition des éléments de l'ordonnanceur

Vue générale

La figure 4.4 donne une vue générale des composants de l'ordonnanceur et de leurs interactions. Le *back-end* est le point d'entrée de l'ordonnanceur. Il est appelé par le noyau lorsqu'une interruption survient, appelle la fonction d'élection puis exécute le *job* choisi par la fonction d'élection. Le *back-end* doit aussi mettre à jour une partie de l'état sur laquelle repose la fonction d'élection, que nous appelons l'environnement. L'environnement est la partie de l'état disponible en lecture seule à la fonction d'élection, que la fonction d'élection peut interroger au travers des *oracles*. La seconde partie est la partie mutable de l'état, que la fonction d'élection ne peut manipuler qu'au travers de l'*interface*, principalement composée de types de données et de fonction pour les manipuler.

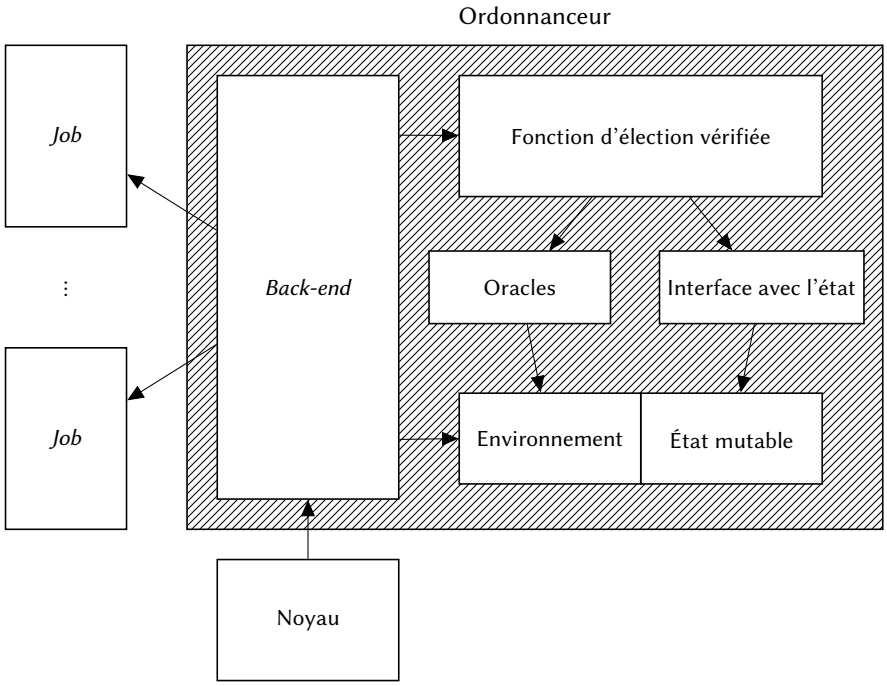

Figure 4.4 – Vue générale de l'ensemble des composants de l'ordonnanceur et de leurs interactions

État interne de l'ordonnanceur

L'état interne de l'ordonnanceur est chargé de stocker les informations nécessaires à la fonction d'élection pour calculer le prochain *job* à élire. Il existe deux représentations de cet état : une implémentation exécutable, et un modèle utilisé pour la preuve.

L'état est – de sucroit – divisé en deux parties, qui se distinguent par leur fonction par rapport à la fonction d'élection. La première partie de l'état est maintenue par la fonction d'élection, qui peut y lire et y écrire des données à sa guise en utilisant l'interface de l'état prévue à cet effet. Cette partie de l'état est appelée l'état mutable. La seconde partie est la partie de l'état que la fonction d'élection ne peut qu'observer, qui est appelé l'environnement. L'environnement contient les informations relatives aux évènements extérieurs essentiels au bon fonctionnement de l'ordonnanceur, tels que l'arrivée de nouveaux *jobs* à ordonnancer, ou la complétion d'un *job* précédemment exécuté. Cette partie de l'état est donc accessible en lecture seule à la fonction d'élection. La fonction d'élection peut consulter l'environnement aux travers d'oracles, qui servent d'interface avec l'environnement.

Interface avec l'état – Oracles, Types opaques et fonctions associées

Comme indiqué dans la sous-section précédente, la fonction d'élection effectue des modifications sur l'état ou observe l'environnement grâce à l'interface et aux oracles. Cette interface est *nécessaire* car elle permet de définir les opérations qui doivent être supportées par l'état, et donc par ses deux représentations (le code exécutable et le modèle). Sans cette interface, certaines opérations pourraient être possible dans une des deux représentations sans qu'elle soit possible dans l'autre. L'interface ainsi que les oracles sont composés de types opaques, implémentés de manière indépendante dans les modèles ou dans le code exécutable. Ces types sont munis de fonctions elles aussi pourvues d'une implémentation exécutable ainsi que d'un modèle.

Les oracles sont une partie de l'interface spécifique. Les oracles sont la partie de l'interface qui permet à la fonction d'élection d'observer les phénomènes extérieurs, plus précisément la terminaison du *job* précédemment exécuté et la liste des nouveaux *jobs* à ordonnancer. Bien qu'ils disposent aussi de deux représentations, leur modèle ne décrit pas réellement les résultats que l'oracle doit produire, mais plutôt des contraintes sur les résultats produits. Le modèle va se contenter de décrire l'ensemble des résultats possibles, plutôt que d'en désigner un en particulier. Par exemple, une des contraintes sur la liste des nouveaux jobs est que leur date d'arrivée ne peut pas être supérieure à la date courante.

Back-ends

Nous avons vu que l'état est composé en partie d'une partie non mutable, appelée l'environnement. Les informations contenues dans l'environnement sont mises à jour par le *back-end*. Le *back-end* est aussi chargé d'appeler la fonction d'élection et d'exécuter le *job* qui aura été élu.

Cette contribution est munie de deux *back-ends* à choisir au moment de la compilation de l'ordonnanceur.

— Le premier back-end est conçu comme une simulation permettant d'exécuter la fonction d'élection dans un simple processus Linux, afin de fournir des informations sur l'état interne de l'ordonnanceur. Ce back-end appelle la fonction d'élection, affiche le *job* choisi pour exécution et autres informations d'intérêt sur l'état interne de la fonction d'élection tels que la liste des *jobs* en attente, etc ;

— Le second back-end fourni est l'implémentation d'un ordonnanceur Earliest Dealine First dans une partition de Pip. Cette implémentation exécute la fonction d'élection et transfère le flot d'exécution vers le *job* élu par la fonction d'élection. Chaque job se trouve dans sa propre partition, avec un espace d'adressage qui lui est propre. Le flot d'exécution revient au back-end soit lorsqu'une interruption d'horloge survient, interrompant le *job* élu, soit lorsque le *job* lui signale sa terminaison. Le back-end attend alors la prochaine interruption pour réélire un nouveau *job*.

Les back-ends ne font pas partie du modèle mathématique de l'ordonnanceur, si ce

n'est au travers des *oracles*. Ils ne font pas partie de la politique d'ordonnancement ; ils appliquent simplement ses arbitrages et lui fournissent les informations dont elle a besoin.

Fonction d'élection

La fonction d'élection est divisée en deux parties principales. La première partie maintient la liste des *jobs* en attente et sélection le prochain *job* à exécuter, la seconde maintient la cohérence de son état interne.

Dans la première partie, la fonction d'élection invoque un oracle, qui lui retourne la liste des nouveaux *jobs* à ordonnancer. Ces jobs sont ajoutés à la liste des jobs à ordonnancer. Ensuite, elle invoque un autre oracle qui lui renvoie l'état du dernier job exécuté, afin de vérifier s'il a complété son exécution ou s'il a dépassé son échéance. S'il remplit l'une de ces deux conditions, il est enlevé de la liste des jobs à ordonnancer. Si il a excédé son échéance, l'erreur sera remontée lors du retour de la fonction. Enfin, la fonction récupère l'identifiant du premier *job* de la liste (si la liste n'est pas vide) et le retourne avec le drapeau indiquant si le dernier *job* exécuté a dépassé son échéance. Une valeur par défaut est retournée si aucun *job* n'est disponible.

La seconde partie de la fonction d'élection maintient la cohérence de l'état interne de la fonction d'élection pour les futurs appels. Elle commence par décrémenter le budget d'exécution du *job* élu. Elle décrémente ensuite l'échéance relative des *jobs* en attente, pour tenir compte du temps qui sera passé à exécuter le *job* élu. Finalement, la fonction d'élection incrémente le compteur de temps, qui garde le compte de périodes de temps s'étant écoulées depuis le premier appel à la fonction d'élection.

4.3 Conduite de la preuve

Cette section met en avant la preuve fonctionnelle de la fonction d'élection de l'ordonnanceur EDF, écrit sous la forme d'un programme monadique dans l'assistant de preuve Coq.

Cette section commencera par décrire la méthodologie de preuve, qui utilise un raffinement en trois niveaux d'abstraction. La section décrira ensuite les différents niveaux d'abstraction et le raffinement qui les relie, en détaillant particulièrement pourquoi le raffinement préserve les propriétés de correction du niveaux d'abstraction supérieur. Enfin, la section conclura sur quelques détails d'élaboration de la preuve en Coq.

4.3.1 Méthodologie de preuve

Cette preuve est conduite par raffinement. Cette méthode permet d'appliquer la stratégie "diviser pour régner" pour gérer la complexité de la preuve. Cette preuve commence par une définition abstraite de la politique d'ordonnancement Earliest Deadline First, et montre que la politique est *correcte*. Cette étape signifie que, sous réserve de quelques hypothèses, n'importe quel ensemble de *jobs* ordonnançable ordonnancé selon cette politique

sera ordonnancé tel que chaque *job* complètera son exécution sans excéder son échéance. La preuve continue en définissant un algorithme de fonction d'élection idéalisé, et montre que cet algorithme implémente la politique d'ordonnancement définie précédemment. Il suit que l'algorithme idéalisé est aussi *correct*. L'étape finale de la preuve consiste à écrire la fonction d'élection qui sera finalement compilée en code source C. Il faut ensuite montrer que la fonction d'élection qui sera traduite en C exhibe les mêmes comportements que l'algorithme idéalisé de la fonction d'élection (en particulier les effets de bord et les valeurs de retour). La preuve est alors complète, la preuve de correction a été propagée jusqu'au code C.

Modèle de *job*

Les *jobs* sont modélisés comme des tuples $j = (i_j, r_j, d_j, c_j, \delta_j)$ d'entiers naturels, où i_j est l'identifiant du *job*, r_j représente l'instant à partir duquel il est possible de l'ordonnancer (*release date*), d_j représente son échéance (*deadline*), et c_j représente son budget temporel, qui est une borne supérieure sur sa durée d'exécution. δ_j représente sa durée d'exécution maximum réelle (Worst Case Execution Time – *WCET*).

Les *jobs* doivent respecter certaines contraintes de construction :

- $r_i + c_i \leq d_i$, l'échéance est assez tardive pour que le *job* puisse terminer son exécution sans la dépasser s'il est le seul à s'exécuter sur le processeur ;
- $0 < \delta_j \leq c_j$, le budget temporel alloué est bien une borne supérieure au temps d'exécution maximum réel (WCET), lui-même strictement positif ;
- l'identifiant de chaque *job* doit être unique ;
- il n'existe qu'une seule occurrence de chaque *job*.

Hypothèse d'ordonnançabilité

Soient deux instants quelconques t et t' tels que $t < t'$, soit $\Gamma_{t,t'}$ l'ensemble des *jobs* à ordonnancer dans l'intervalle $[t, t']$ (c'est à dire les *jobs* j ayant une release date r_j supérieure à t et une échéance d_j inférieure à t'). L'hypothèse d'ordonnançabilité stipule que la somme des durées δ_j des *jobs* à ordonnancer dans l'intervalle $[t, t']$ est inférieure à la durée $t' - t$ de l'intervalle.

$$\forall t, t' . \, t < t' \implies \sum_{j \in \Gamma_{t,t'}} \delta_j \leq t' - t \tag{4.1}$$

Le fait qu'un ensemble de jobs ordonnançable puisse être ordonnancé par un ordonnanceur Earliest Deadline First sur un unique processeur est un résultat classique de la théorie de l'ordonnancement ([Sta+12], pp. 33-34). Les conditions d'ordonnançabilité d'un ensemble peuvent parfois être représentées plus simplement, par exemple lorsque les *jobs* sont des

instances de tâches périodiques. Cependant cette contribution porte sur un ordonnanceur de *jobs arbitraires*, ce qui ne nous permet pas d'utiliser les hypothèses d'ordonnançabilité de cas spécifiques.

4.3.2 Couches d'abstraction et étapes de raffinement

Politique d'ordonnancement Earliest Deadline First

La première couche d'abstraction, la plus abstraite de toutes, est celle de la politique d'ordonnancement. Elle est définie de la manière suivante : pour tout *job* j, et à n'importe quel instant t, si le *job* j s'exécute à l'instant t, alors pour chaque autre *job* j' qui aurait pu être exécuté à la place, on a $d_j \leq d_{j'}$.

De manière formelle,

$$\forall j, \forall t, [...]\ \texttt{run}\ j\ t \implies \forall j',\ \texttt{waiting}\ j'\ t \implies d_j \leq d_{j'}. \tag{4.2}$$

avec :

j, j', deux identifiants de job,
t, un instant arbitraire,
$\texttt{run}\ i\ t$, prouvable si le *job* j s'exécute à l'instant t,
waiting $x\ t$, prouvable si le *job* j' est prêt à s'exécuter à l'instant t

La *propriété de correction* pour la politique d'ordonnancement EDF est décrite de la manière suivante. Étant donné un ensemble ordonnançable de *jobs* (c'est à dire satisfaisant l'hypothèse d'ordonnancement 4.1), si la politique d'ordonnancement EDF est appliquée, alors aucun *job* de l'ensemble ne dépassera son échéance.

$$\begin{gathered} \textit{ordonnançable} \implies \\ \forall j, \forall t.\quad \text{EdfPolicyUpTo}\ t \implies \\ \neg\text{overdue}\ j\ t. \end{gathered} \tag{4.3}$$

avec :

EdfPolicyUpTo t signifie que la politique a été appliquée jusqu'à l'instant t
overdue $j\ t$, prouvable si le *job* j a dépassé son échéance à l'instant t

Fonction d'élection idéalisée implémentant la politique d'ordonnancement

La première étape du raffinement est de montrer que la fonction d'élection idéalisée `functional_scheduler_star` implémente la politique d'ordonnancement EDF. La propriété suivante établie que si la fonction d'élection idéalisée est exécutée jusqu'à un certain

instant t, alors la politique EDF a été appliquée jusqu'à cet instant.

$$\forall t, \forall o, \forall s.\ \ \texttt{functional\_scheduler\_star}\ (t) = (o, s) \implies \\ \text{EdfPolicyUpTo (now } s\text{).} \tag{4.4}$$

avec :

s, l'état du programme après avoir exécuté la fonction d'élection idéalisée après t unités de temps,
now s, extrait le compteur d'unité de temps de l'état s
o , l'identifiant du *job* élu à l'instant t

De cela, on peut déduire la propriété de correction de la fonction d'élection idéalisée : étant donné un ensemble de *job* ordonnançable, alors pour tout *job* j de cet ensemble, ce *job* ne dépassera pas son échéance à l'instant t si la fonction a été appelée à chaque instant jusqu'à l'instant t

$$\begin{gathered} \textit{ordonnançable} \implies \\ \forall t, \forall o, \forall s.\ \ \texttt{functional\_scheduler\_star}\ (t) = (o, s) \implies \\ \forall i.\ \ \neg\text{overdue } i \text{ (now } s\text{).} \end{gathered} \tag{4.5}$$

Fonction d'élection monadique raffinant la fonction d'élection idéalisée

La prochaine étape consiste à prouver que la fonction d'élection monadique raffine la fonction d'élection idéalisée. Cette propriété contient des triplets de Hoare. Les préconditions de la propriété sont paramétrés par un environnement observable `env` et par un état mutable s. Les postconditions sont paramétrées par la valeur de retour o du programme x, et un état s', le résultat de l'exécution du programme c sur l'état s.

Cette étape de raffinement peut être décrite de la manière suivante : pour tout instant t, si (o, s') sont les valeurs retournées par la fonction d'élection monadique `scheduler_star` sur l'environnement E et l'état initial `init`, alors (o, s') sont aussi les valeurs retournées par la fonction d'élection idéalisée `functional_scheduler_star` à l'instant t. Le triplet de Hoare correspondant est le suivant :

$$\begin{gathered} \forall t. \\ \{\ \lambda\ \texttt{env}\ s.\ \ \texttt{env} = E\ \ \wedge\ \ s = \texttt{init}\ \} \\ \texttt{scheduler\_star}\ (t) \\ \{\ \lambda\ o\ s'.\ \ \texttt{functional\_scheduler\_star}\ (t) = (o, s')\ \} \end{gathered} \tag{4.6}$$

À partir de ce triplet, on peut déduire le triplet de Hoare exprimant la propriété de correction de la fonction d'élection monadique.

$$\begin{gathered} \mathit{ordonnançable} \implies \\ \forall t. \\ \{\, \lambda\, \mathtt{env}\ s.\ \ \mathtt{env} = E \ \wedge\ s = \mathtt{init} \,\} \\ \mathtt{scheduler\_star}\ (t) \\ \{\, \lambda\ \_\ s'.\ \ \forall i.\ \ \neg \mathrm{overdue}\ i\ (\mathrm{now}\ s') \,\} \end{gathered} \tag{4.7}$$

4.3.3 Hypothèses et déroulement de la preuve en Coq

Les détails de la preuve sont disponibles sur le dépot, et nous avons décrit la structure principale de la preuve. Nous allons ici développer les faits les plus caractéristiques de cette preuve, donnant une vision plus précise de son déroulement.

Les preuves Coq suivent de près l'approche par raffinement décrite dans la sous-section précédente. Tout d'abord, le modèle choisi pour les jobs, décrit sous forme mathématiquement dans la section 4.3.1 est implémenté dans l'assistant de preuve. Les hypothèses listées dans la même section 4.3.1 sont des hypothèses globales, qui s'appliquent donc à l'ensemble de la preuve. Il est facile d'argumenter que ces hypothèses sont des hypothèses raisonnables sur l'ensemble étudié, c'est à dire qui ne restreignent pas les ensembles considérés, ou sans lesquelles l'ordonnancement d'un tel ensemble est impossible.

La première couche d'abstraction modélisant la politique d'ordonnancement nécessite des hypothèses supplémentaires. Ces hypothèses – appelées hypothèses *locales* – ne sont utilisées que pour établir la preuve de correction de la politique. Elles disparaissent dès le premier raffinement vers la fonction d'élection idéalisée ; ces hypothèses deviennent alors des définitions et des lemmes Coq.

On suppose localement que deux fonctions sont fournies avec la politique : `run` qui permet de déterminer quel *job* sera exécuté à n'importe quel instant, et `rem`, qui garde le compte du temps d'exécution restant de chaque *job* à n'importe quel moment. Ces fonctions ont besoin elles aussi d'hypothèses locales supplémentaires, qui représentent des propriétés générales sur l'ordonnancement mono-processeur :

- à chaque instant, au maximum un *job* est exécuté ;
- le temps d'exécution restant d'un *job* est égal à sa durée au moins jusqu'à sa *release date* ;
- le temps d'exécution restant d'un *job* diminue lorsqu'il est exécuté, et reste constant lorsqu'il ne l'est pas ;
- il y a un *job* qui s'exécute à chaque instant où au moins un *job* est prêt à être exécuté.

Le premier raffinement représente la plus grosse portion de la preuve. Ce raffinement consiste à définir les fonctions abstraites `run` et `rem` supposées fournies dans la couche

d'abstraction de la politique, puis à prouver les hypothèses localement supposées sur ces fonctions. Une fois ces hypothèses prouvées, la propriété de correction de la fonction d'élection idéalisée est une simple conjonction entre la propriété de correction de la politique d'ordonnancement et la propriété de raffinement. Il est plus intéressant de prouver une relation d'implémentation plutôt que de prouver directement la propriété de correction de la fonction d'élection idéalisée, car les obligations de preuve de la preuve par raffinement sont plus concrètes. La preuve par raffinement nécessitant l'établissement d'abstractions avant son écriture, elle nous propose des termes de preuves plus directs que ceux qui auraient été proposés par la preuve directe.

Toutes les propriétés à prouver à ce niveau de raffinement sont des *invariants* – des prédicats qui doivent être vrais sur tous les états atteignables depuis l'état initial, obtenus en exécutant un nombre arbitraire de fois la fonction d'élection idéalisée. La preuve de certains de ces invariants a cependant nécessité la conception d'invariants *inductifs* supplémentaires, vrais sur l'état initial, et préservés par chaque exécution subséquente de la fonction d'élection idéalisée. Ces invariants ont été conçus de telle manière à ce qu'ensembles, ils impliquent l'invariant initial à prouver.

La dernière étape de raffinement permet de montrer que la fonction d'élection exécutable raffine la fonction d'élection idéalisée. Elle consiste à montrer que l'ensemble des états atteignables par la fonction d'élection exécutable est inclus dans l'ensemble des états atteignables par la fonction d'élection idéalisée. La différence notable entre les deux fonctions d'élection est que la fonction d'élection idéalisée calcule en une grosse itération ce que la fonction d'élection exécutable calcule en plusieurs petites itérations. La fonction d'élection idéalisée n'est pas assujettie à de telles contraintes et peut de ce fait factoriser ces longues séquences de petites itérations en une seule grosse itération. Ce faisant, la fonction d'élection idéalisée nous avait épargné la preuve d'invariants supplémentaires à propos des états intermédiaires générés par la fonction d'élection exécutable, qu'il a fallu prouver dans cette étape de raffinement.

La conclusion globale indiquant que la fonction d'élection exécutable est correcte découle de la propriété de correction de la politique d'ordonnancement et des preuves des deux raffinements successifs.

Ainsi s'achève la section sur la conduite de la preuve de correction de la fonction d'élection de notre ordonnanceur. La prochaine section décrira l'implémentation de l'interface et de l'état interne de l'ordonnanceur.

4.4 Implémentation et modèle de l'état et de son interface

Cette section commence par rappeler le rôle de la monade d'état dans le monde mathématique et de la nécessité de son interface. Elle illustre ensuite le problème de *dualité* des modèles et de l'objet réel modélisé avec un exemple s'étant présenté pendant l'implémentation. Dans une seconde partie, cette section présente notre implémentation et les modèles

relatifs à l'état et aux fonctions de l'interface. Elle décrit l'implémentation de l'interface avec la monade d'état, qui permet à la fonction d'élection d'intéragir avec l'état interne du programme.

4.4.1 Dualité implémentation/modélisation

La monade d'état représente l'état du programme dans le monde fonctionnel du modèle qui en est autrement dépourvu. Les fonctions qui permettent d'interagir avec la monade d'état constituent son interface. Lors de la compilation du code Gallina vers du code C, la monade d'état disparaît, pour laisser place à l'état intrinsèque du monde impératif. Il existe donc deux penchants de l'interface : le modèle, intéragissant avec la monade d'état, et son implémentation, intéragissant directement avec l'état intrinsèque du programme. Cette interface est supposée correcte : on suppose que l'appel à une fonction de l'interface a les mêmes effets dans le monde mathématique et dans le monde réel. La méthode usuelle pour minimiser l'impact de cette supposition est de créer une interface constituée de fonctions dont les effets sur l'état sont les plus simples possibles, idéalement de l'ordre d'une simple instruction assembleur. Malheureusement, cette méthode est souvent en tension avec l'effort de preuve, qui augmente de manière exponentielle lorsque les modèles se rapprochent de plus en plus du matériel. Ainsi, chaque modèle trouve son propre point d'équilibre entre coût de la preuve et risque de distorsion entre le modèle et la réalité.

Pour illustrer ce propos, voici un exemple relatif à l'établissement de la preuve décrite dans la section précédente. Afin de simplifier grandement la preuve, notre monade d'état contient des listes. Le modèle de l'interface repose de ce fait sur les listes fonctionnelles du langage, qui créent des copies à chaque opération et qui ne manquent jamais de mémoire. Ce type de liste *idéalisé* ne peut être implémenté qu'avec des logiciels ad-hoc lourds comme, par exemple, les ramasse-miettes. Incorporer de tels logiciels dans la base de confiance est en nette contradiction avec notre méthodologie. Nous avons de ce fait décidé d'opter pour une interface utilisant des listes avec modification en place, de telle sorte à fournir une interface qui soit pratique à utiliser dans le code monadique, dont le modèle fonctionnel est simple du point de vue de la preuve, et qui soit facile à implémenter en C. Ce choix nous contraints néanmoins à ne jamais laisser l'utilisateur manipuler les objets de listes directement, sous peine de créer une distorsion entre la réalité et le modèle.

Pour illustrer cette distorsion, prenons par exemple la séquence d'instructions idéalisée suivante :

```
do liste_initiale <- get_liste_initiale;
do liste_modifiee <- modifie_liste liste_initiale;
do queue_liste_initiale <- get_tail liste_initiale;
```

Dans le monde fonctionnel, l'interface utilise les listes *idéalisées* qui retournent des *copies* (éventuellement modifiées) de la liste initiale. La variable `queue_liste_initiale` contient effectivement une référence vers une copie de la queue de la liste initiale. Cependant, l'implémentation réelle de l'interface utilise une implémentation de liste sans copie,

avec une modification des listes en place. La variable `queue_liste_initiale` contient une référence vers la queue de la seule liste existante, c'est à dire celle de la liste modifiée.

Afin d'éviter de s'exposer à ce problème de distortion, notre interface ne fournit donc pas d'objet de liste, mais des fonctions pour modifier les listes contenues dans l'état. Cette approche a cependant le désavantage de créer des primitives non-triviales, comme par exemple l'insertion triée dans une liste. Nous reviendrons sur ce choix d'inclusion de fonctions non-triviales dans l'interface (et donc dans la base de confiance) à la fin de ce chapitre, dans la section 4.5.2.

4.4.2 Implémentation vue comme un cas particulier de l'interface abstraite

Cette sous-section détaille comment l'interface du code vérifié avec l'état a été conçue, décrivant à la fois les modèles et leur implémentation. Pour rappel, les différentes parties de l'interface et leurs interactions sont représentées dans la section 4.2.3, dans la figure 4.4.

Cette section décrira d'abord l'état interne de la fonction d'élection, composé de deux parties : une partie en lecture seule appelée l'environnement, et une partie mutable. Ensuite, la section expose comment la fonction d'élection utilise les *oracles* pour récupèrer les informations de l'environnement, et comment la fonction d'élection utilise l'interface pour manipuler la partie mutable de l'état.

État de l'ordonnanceur

Le modèle de l'état est divisé en deux parties : l'environnement, contenant les valeurs calculées par le back-end sur lesquelles la fonction d'élection n'a aucun contrôle, et la partie de l'état modifiable par la fonction d'élection.

Le modèle de l'environnement `Env` est défini comme une fonction qui, en fonction de l'instant passé en paramètre, retourne la liste des *jobs* à ajouter à la liste des *jobs* prêts à être ordonnancés.

```
Record Job  :=
mk_Job {
    jobid : CNat ;
    arrival : CNat ;
    duration : CNat ;
    budget : CNat ;
    deadline : CNat
}.

Definition Env : Type := CNat → list Job.
```

Listing 4.2 – Définitions des modèles de la structure `Job` et de l'environnement

Le modèle de la partie mutable `State` est constitué d'une structure avec deux champs. Le premier est un simple compteur de temps, gardant le compte des interruptions d'horloge ; l'autre champ est une liste d'`entries`. Les *entries* sont des structures qui associent chaque *job* à son budget d'exécution restant et à son échéance relative. Le modèle de cet état mutable est donc défini de la manière suivante :

```
Record Entry :=
mk_Entry {
    id : CNat ;
    cnt : CNat ;
    del : CNat
}.
```

```
Record State :=
mk_State {
    now : CNat ;
    active : list Entry ;
}.
```

Listing 4.3 – Définitions des modèles des structures Entry et de la partie mutable de l'état

La monade d'état, composé de l'environnement et de la partie mutable, a le type suivant :

```
Definition RT (A : Type) : Type :=
    Env → State → A * State.
```

Listing 4.4 – Définition du type de la monade d'état et d'environnement

Dans notre modèle, chaque fonction monadique a un type de retour `RT A`, et prend donc en paramètre l'environnement et l'état mutable actuel et renvoie une valeur de type arbitraire `A` "enrobée" avec la nouvelle partie mutable de l'état.

Les paragraphes suivants décrivent comment l'état est implémenté. L'implémentation profite du fait que la fonction d'élection crée une nouvelle structure `entry` pour chaque *job* à ordonnancer. L'implémentation réserve de l'espace mémoire supplémentaire dans la structure de chaque *job*, afin que la fonction d'élection puisse y écrire la structure `entry` associée.

Notre implémentation fournit un ensemble jouet de job à ordonnancer sous la forme d'un tableau de `coq_N` éléments (où `coq_N` est une borne supérieure au nombre de *jobs* à ordonnancer connue à la compilation). Les jobs sont discriminés par leur position au sein de ce tableau. Chaque élément de ce tableau correspond donc à un *job* de l'ensemble à ordonnancer. Il contient les informations immuables qui le caractérisent (par exemple sa *release date*, son échéance, son budget temporel d'exécution), ainsi qu'une portion de mémoire non initialisée qui sera utilisée par la fonction d'élection pour y écrire la structure `entry` qui lui est associée. De plus, chaque élément contient deux mots mémoire supplémentaires. Ils sont utilisés pour maintenir deux listes nécessaires au fonctionnement de l'ordonnanceur. La première est la liste des nouveaux *jobs* à ordonnancer, exposée par l'environnement et maintenue par le *back-end*. La seconde est la liste des *jobs* prêt à être exécutés, maintenue par la fonction d'élection.

Chaque élément est représenté en C avec le type décrit en listing 4.5.

```
typedef struct __internal_s__ {
    struct __internal_job__   job;
    struct __internal_entry__ entry;
    int jobset_next_job_index;
    int active_next_entry_index;
} internal_t;
```

Listing 4.5 – Implémentation du type des éléments du tableau contenant les *jobs* à ordonnancer

```
internal_t INTERNAL_ARRAY[coq_N] = EXAMPLE_JOB_SET;
int JOBS_ARRIVING_HEAD_INDEX = -1;
int ACTIVE_ENTRIES_HEAD_INDEX = -1;
unsigned int CLOCK = 0;
coq_CBool JOB_DONE = false;
```

Listing 4.6 – Implémentation de l'environnement et de la partie mutable de l'état de l'ordonnanceur

L'implémentation de l'état, donnée en listing 4.6, est donc constituée :

- du tableau contenant les informations sur les *jobs* et l'espace supplémentaire requis par la fonction d'élection pour y stocker les structures `entry`
- d'une référence vers la liste des nouveaux *jobs* à ordonnancer
- d'une référence vers la liste des *jobs* prêts à être ordonnancés
- de la variable gardant le compte des interruptions d'horloge
- de la variable indiquant si le dernier *job* exécuté a terminé son exécution

Oracles

Il existe deux fonctions qui retournent des valeurs de l'environnement calculées par le back-end. La première, `jobs_arriving`, récupère les nouveaux *jobs* à ordonnancer ; la seconde, `job_terminating`, retourne si le dernier *job* exécuté a terminé son exécution (si un tel *job* existe). Nous appelons ces fonctions les *oracles* de l'ordonnanceur, car leur modèle ne décrit que les contraintes imposées sur les résultats qu'elles produisent, et non pas comment ces résultats sont fournis.

La fonction `jobs_arriving`, qui récupère les identifiants des nouveaux jobs à ordonnancer, a un modèle n'imposant qu'une seule contrainte à l'implémentation : dans la liste des identifiants de *jobs* retournés par l'oracle, aucun identifiant ne doit être plus grand que N, où N est une borne supérieure définie à la compilation. Cette borne est la même que la borne `coq_N` présentée dans la section précédente sur l'implémentation de l'état. La liste

renvoyée par l'oracle est cachée derrière le type opaque `JobSet`.
Le modèle de cette fonction est exposé dans le listing 4.7.

```
Definition jobs_arriving (N : nat) : RT JobSet :=
  fun env s ⇒
    let f :=  List.filter
      (fun j ⇒  Nat.ltb j N)
      (map jobid (env s.(now)))
  in (f, s).
```

Listing 4.7 – Modèle de l'oracle `jobs_arriving`

Rappelons que ce modèle n'indique aucunement comment calculer une telle liste de *jobs*. Ce choix est délibéré, afin de s'assurer que la fonction d'élection n'est pas en capacité de prévoir l'arrivée de nouveaux *jobs*, guarantissant que n'importe quel ensemble arbitraire de *jobs* puisse être utilisé avec cet ordonnanceur.

L'implémentation de la fonction retournant la prédiction de l'oracle est particulièrement simple : elle retourne simplement la variable de l'état contenant la référence vers la liste des nouveaux *jobs* à ordonnancer. Plus précisement, cette variable contient l'indice de l'élément du tableau étant en tête de cette liste. La liste a été construite par le back-end avant l'exécution de la fonction d'élection qui fait appel à l'oracle. La construction de cette liste est donc totalement déléguée à l'implémentation du back-end, sans aucune répercussion sur le monde mathématique.

```
static inline coq_JobSet
Primitives_jobs_arriving(coq_CNat n) {
     return JOBS_ARRIVING_HEAD_INDEX;
}
```

Listing 4.8 – Implémentation de l'oracle `jobs_arriving`

L'autre oracle est une fonction qui retourne si le *job* exécuté durant la précédente période d'exécution a terminé son exécution (si un tel *job* existe). Le modèle de cet oracle est plus strict ; il force l'implémentation de l'oracle à retourner `True` si le nombre de périodes passées à exécuter le *job* excède son *WCET*. Son modèle est présenté en figure 4.9.

Ce modèle décrémente le compteur `e.(cnt)` à partir d'une valeur initiale de `j.(budget)` à chaque fois que le *job* est élu pour exécution. Tout comme l'implémentation de l'oracle précédent, l'implémentation de cet oracle est particulièrement simple, comme démontré en listing 4.10. Elle ne fait que retourner la variable `JOB_DONE` de l'état interne ; cette variable est mise à jour par le *back-end* lorsqu'un *job* termine son exécution, et remis à `False` après chaque appel à la fonction d'élection.

```
Definition job_terminating : RT CBool :=
fun _ s ⇒ ((
    match head s.(active) with
    | None ⇒ false
    | Some e ⇒
        let j := Jobs (e.(id)) in
        Nat.leb e.(cnt) (j.(budget) - j.(duration))
    end
), s).
```

Listing 4.9 – Modèle de l'oracle `job_terminating`

```
static inline coq_CBool Primitives_job_terminating(void) {
    return JOB_DONE;
}
```

Listing 4.10 – Implémentation de l'oracle `job_terminating`

La dernière portion de cette section détaillera l'interface de la fonction d'élection avec la partie mutable de l'état.

Types opaque et interface avec l'état mutable

Tous les types utilisés par la fonction d'élection sont des types opaques, c'est à dire que des types agnostiques de leur représentation et uniquement définis au travers des fonctions qui permettent de les manipuler, qu'on appelle leurs *primitives*.

Ces travaux comportent des types opaques pour des valeurs booléennes, pour des entiers, pour les structures en lecture seule telles que celles contenant les informations initiales sur les *jobs* ou la liste des nouveaux *jobs* à ordonnancer, ainsi que pour les structures contenant des données mutables telles que les structures `entry`. La plupart des primitives sur ces types sont extrêmement simples ; par exemple, le type `CBool` permettant de représenter les booléens est muni de trois primitives : `not`, `and`, et `or`. Le listing 4.11 montre la modélisation de ce type avec sa primitive `or`. Son implémentation est décrite dans le listing 4.12.

```
Definition CBool := bool.

Definition or (b1 b2 : CBool) : RT CBool :=
  ret (orb b1 b2).
```

Listing 4.11 – Modèle du type opaque `CBool` et de sa primitive `or`

```
typedef int coq_CBool;

static inline coq_CBool CBool_or(coq_CBool b1, coq_CBool b2) {
    return b1 || b2;
};
```

Listing 4.12 – Implémentation du type opaque `CBool` et de sa primitive `or`

Les structures mutables, telles que le type `entry`, sont munies de primitives permettant de les créer, ainsi que de lire et modifier leurs champs. Dans le même esprit, les structures de données en lecture seule ne sont munies que de primitives permettant de lire les informations qu'elles renferment. Par exemple, le type opaque `JobSet`, retourné par l'oracle `jobs_arriving` n'est muni que de trois primitives : une vérifiant si l'ensemble est vide, une autre récupérant le premier élément de l'ensemble, et une dernière retournant un plus petit `JobSet` contenant les *jobs* restants. Le type opaque `Job` quand à lui ne possède que des primitives permettant de récupérer individuellement chaque champ de la structure.

Enfin, il existe huit autres primitives conçues pour intéragir directement avec l'état mutable de la fonction d'élection. La fonction d'élection a notamment besoin de gérer le compteur d'interruption d'horloge ; pour cela il existe deux primitives, un *getter* et un *setter* pour la variable de compteur. Ces fonctions sont extrêmement simples à modéliser et implémenter, comme en attestent les listings 4.13 et 4.14.

```
Definition get_time_counter : RT CNat :=
fun _ s ⇒ ((now s), s).

Definition set_time_counter (counter : nat) : RT unit :=
fun _ s ⇒ (tt,
    {|
        now    := counter ;
        active := (active s) ;
    |}).
```

Listing 4.13 – Modèle des fonctions de récupération et de modification de la variable de compteur de l'état

La fonction d'élection doit aussi maintenir la liste des jobs prêts à être ordonnancés. Elle a à sa disposition six primitives :

- une primitive permettant de vérifier si cette liste est vide
- une primitive retournant le *job* à la tête de la liste
- une primitive ajoutant un *job* à la liste de manière triée selon une fonction de comparaison fournie en paramètre

```
static inline coq_CNat State_get_time_counter(){
    return CLOCK;
};

static inline void State_set_time_counter(coq_CNat counter){
    CLOCK = counter;
};
```

Listing 4.14 – Implémentation des fonctions de récupération et de modification de la variable de compteur de l'état

— une primitive retirant le premier job de la liste

— une primitive mettant à jour la structure `entry` liée au premier *job* de la liste

— une primitive mettant à jour la structure `entry` de tous les *job* de la liste

La plupart de ces primitives sont aussi simples à modéliser qu'à implémenter ; cependant deux d'entre elles dérogent à cette règle. La première est la fonction `insert_new_active_entry`, qui insère une structure `entry` (contenant les valeurs manipulables d'un job par la fonction d'élection) de manière triée selon une fonction de comparaison. Son modèle est donné dans la moitié supérieure du listing 4.15.

Étant donné une structure `entry` et une fonction de comparaison, ce modèle extrait d'abord la liste du modèle de l'état de l'ordonnanceur. Le modèle passe cette liste à une fonction auxiliaire purement fonctionnelle qui retourne la liste contenant la structure `entry` à insérer. Cette fonction auxiliaire crée en réalité une nouvelle liste, en comparant chaque structure `entry` de la liste initiale avec la structure `entry` à insérer grâce à la fonction de comparaison passée en paramètre, et en ajoutant au fur et à mesure des copies des éléments à cette liste.

Pour des raisons de performances et de minimisation de la base de confiance, l'implémentation ne crée pas de nouvelle liste à chaque nouvelle insertion, mais modifie en place la liste actuelle. L'interface choisie empêche le problème de distortion exposé dans la section précédente 4.4.1, où le modèle crée des copies et où l'implémentation modifie la liste en place.

Le code de l'implémentation, disponible dans la motié inférieure du listing 4.15, est similaire à la fonction auxiliaire récursive du modèle, mais utilise une boucle et met à jour le lien entre les éléments quand la nouvelle struture `entry` est insérée, plutôt que de créer des copies des éléments de la liste en les réordonnant correctement.

La fonction `update_active_entries`, qui applique une modification arbitraire à chaque structure `entry` de la liste, a subit la même méthode de modélisation et d'implémentation. L'implémentation modifie les éléments de la liste en place plutôt que de créer une nouvelle liste avec des copies modifiés des éléments originaux.

Nous défendons l'argument que les primitives présentées dans cette section, définis-

```
Definition insert_new_active_entry (entry : Entry)
           (comp_func : Entry → Entry → CBool) : RT unit :=
  fun _ s ⇒ (tt, {|
    now := now s ;
    active := (insert_Entry_aux entry (active s) comp_func);
  |}
).

Fixpoint insert_Entry_aux (entry : Entry) (entry_list : list Entry)
         (comp_func : Entry → Entry → CBool) : list Entry :=
  match entry_list with
  | nil ⇒ cons entry nil
  | cons head tail ⇒
      match comp_func entry head with
      | true ⇒ cons entry (cons head tail)
      | false ⇒ cons head (insert_Entry_aux entry tail comp_func)
      end
  end.
```

```
void State_insert_new_active_entry
     (coq_Entry entry, entry_cmp_func_type entry_comp_func) {
  int *entry_index_ptr = &(ACTIVE_ENTRIES_HEAD_INDEX);
  int next_index = -1;
  while (*entry_index_ptr ≠ -1) {
    if (entry_comp_func(entry, &(INTERNAL_ARRAY[*entry_index_ptr].entry))) {
      next_index = *entry_index_ptr;
      break;
    }
    entry_index_ptr = &(INTERNAL_ARRAY[*entry_index_ptr].active_next_entry_index);
  }
  *entry_index_ptr = entry→id;
  INTERNAL_ARRAY[entry→id].active_next_entry_index = next_index;
}
```

Listing 4.15 – Comparaison entre le modèle de la primitive d'insertion triée dans la liste et son implémentation

sant l'interface de la fonction d'élection avec l'implémentation réelle de l'ordonnanceur et de son état interne, sont assez simples pour se convaincre que les effets de leur implémentation correspond bien au effet décrits par leur modèle. Néanmoins, les fonctions `insert_new_active_entry` et `update_active_entries` présentées à la fin de cette section sont objectivement plus complexes que les autres primitives présentées. Cet argument est débatable sur ces deux fonctions ; aussi la section 4.5.2 présentera des arguments supplémentaires en faveur de ce choix de primitives.

4.5 Discussion sur la méthodologie suivie

Cette section sert de retour d'expérience sur la contribution, en prenant du recul sur le travail fourni. Elle commence par présenter différentes métriques concernant l'ordonnanceur, discutant des proportions de code prouvé, de lignes de preuves et de lignes d'implémentation, et du temps investi. Dans une seconde partie, cette section discutera des choix fait pour ces travaux, et plus particulièrement du choix des primitives et de l'impact de ce choix sur la confiance apportée par la preuve.

4.5.1 Métriques

La partie technique de cette contribution est le fruit de 9 mois de travail de deux personnes à temps plein, aidés ponctuellement par d'autres membres de nos équipes de recherche respectives. La présentation de l'ordonnanceur en conférence et la publication du papier ont eu lieu un an et demi après le premier commit sur le dépôt de code du projet.

Implémentation

La fonction d'élection, écrite en Gallina, compte à elle seule environ 150 lignes de code Gallina ; sans surprise, le nombre de lignes de code C produit lors de la compilation est équivalent. L'implémentation de l'état, des types opaques et des fonctions formant l'interface avec l'état compte environ 350 lignes de code C, composé majoritairement de fichiers *headers* particulièrement verbeux.

Concernant le back-end de *simulation* de l'ordonnanceur, celui-ci a requis environ 100 lignes de code C. L'implémentation de la partition de Pip contenant l'ordonnanceur préemptif temps réel, sans compter la fonction d'élection ni l'implémentation de l'état et de l'interface, compte pour environ 1150 lignes de code. Sont pris en compte, les fichiers de code C et leur *headers*, les fichiers de code assembleur, mais aussi les scripts d'édition de liens. En revanche, ce nombre de lignes ne contient pas les lignes de la librairie utilisateur de Pip. Sur ces 1150 lignes, environ 900 lignes sont requises par Pip pour créer les partitions de l'ordonnanceur et des *jobs*, ainsi que pour mapper leur code dans leur espace d'adressage respectifs. Les 250 lignes restantes sont dédiées au fonctionnement de l'ordonnanceur en lui-même.

Modèles et preuves

Les modèles de l'état, des types opaques, et des fonctions permettant d'intéragir avec l'état représentent en tout environ 650 lignes de code.

Pour rappel, la preuve se découpe en trois parties majeures : la définition de la politique d'ordonnancement *Earliest Deadline First* et l'établissement de sa propriété de correction, l'implémentation d'une fonction d'élection idéalisée et de sa preuve de raffinement de la politique d'ordonnancement, puis d'un raffinement vers la fonction qui sera traduite en C, compilée et exécutée.

La première partie de cette preuve concernant la correction de la politique d'ordonnancement est relativement courte. L'écriture de la politique d'ordonnancement, avec sa preuve de correction représente environ 1100 lignes de code et de preuve. Ce nombre de lignes relativement restreint est dû au haut niveau d'abstraction, profitant à l'établissement de la preuve : cela permet de se concentrer sur l'essentiel sans s'encombrer de détails d'implémentation.

Comme énoncé dans la section 4.3.3, le premier raffinement, allant de la politique d'ordonnancement vers la fonction d'élection idéalisée, représente la majeure partie de la preuve, et qui a necéssité de redoubler d'ingéniosité. Cela s'explique par le fossé d'abstraction entre la politique d'ordonnancement (composée de termes algébriques relativement simples), et la couche d'abstraction de la fonction d'élection idéalisée qui décrit la politique en fonction d'opération sur des structures de données, en particulier des listes chainées. Cette partie de la preuve représente environ 1600 lignes de Coq (comptabilisant aussi les lignes de code de la fonction d'élection idéalisée).

Le second raffinement est relativement simple comparé au premier raffinement, car les fonctions d'élection idéalisées et exécutables sont à des niveaux d'abstractions relativement proches ; en particulier, elles utilisent les mêmes structures de données. À titre de comparaison, cette dernière étape de raffinement est courte, environ 500 lignes de preuve.

Ainsi, l'établissement de la preuve sur la fonction d'élection longue de 150 lignes a nécessité au total 3250 lignes de Coq, quand le nombre total de ligne nécessaires pour implémenter l'ordonnanceur avoisinne les 1400 lignes de code. Le code prouvé représente donc environ 10% du code total de l'ordonnanceur.

4.5.2 Choix des primitives et discussion sur la base de confiance

Les preuves formelles transfèrent la confiance accordées aux hypothèses vers la conclusion d'une démonstration. Lorsque ces hypothèses sont relatives à la sécurité informatique, on parle alors de base de confiance (*Trusted Computing Base* ou *TCB*). On peut distinguer trois catégories d'hypothèses dans les bases de confiance. La première catégorie, renfermant les hypothèses les plus fondamentales, sont les hypothèses sur la correction du matériel et des outils utilisés pour vérifier la preuve. La seconde catégorie englobe les hypothèses sur l'ensemble des logiciels non vérifiés s'exécutant avec le code prouvé. La troisième et dernière catégorie regroupe les hypothèses concernant le processus de com-

pilation, et plus particulièrement sur la sémantique associée aux opérations utilisées par le code source (sur laquelle la preuve repose) et la sémantique utilisée par le code compilé (qui est réellement exécuté). Cette sous-section questionnera la base de confiance de l'ordonnanceur présenté dans ce chapitre, en la comparant avec celle de travaux connexes.

Barrières fondamentales

Il existe des hypothèses universelles, présentes dans la base de confiance de l'intégralité de la communauté de vérification formelle de code.

Les premières de ces hypothèses sont celles de l'outil utilisé pour vérifier la preuve formelle : dans notre cas, l'assistant de preuve Coq. Sa partie critique est circonscrite par son noyau – la partie minimiale de l'assistant qui vérifie si une tentative de preuve est valide au sens formel. On pourrait accroitre notre confiance dans le noyau en le vérifiant formellement ; c'est ce que le projet MetaCoq [Soz+20] vise à atteindre, en formalisant le noyau de Coq dans Coq.

Néanmoins, il existe, dans la base de confiance commune, d'autres hypothèses plus fortes par plusieurs ordres de grandeur : la spécification du matériel exécutant le code vérifié. On peut découper cet argument en deux parties :

— Tout d'abord, la (quasi ?) totalité des fabriquants ne fournissent pas de spécification formelle du fonctionnement de leur matériel. Cela implique que pour raisonner sur ce matériel, il faut *interpréter* la documentation informelle pour produire la spécification formelle. Ce travail est fastidieux et délicat, ce qui le rend particulièrement sujet à l'erreur. Par effet de bord, il est difficile de vérifier du logiciel qui s'appuierait directement sur des primitives fournies par le hardware. Néanmoins, de nouveaux projets visent à améliorer les choses, avec notamment la démocratisation du développement de matériel open-source et l'apparition de spécifications formelles pour certains processeurs [Rei17] ;

— Par ailleurs, si on considérait qu'il existait une spécification formelle du matériel sur lequel le code s'exécuterait, il n'existe pour l'instant aucune méthode permettant de s'assurer que le matériel se comporte exactement comme la spécification le décrit.

Ainsi, en dépit de la complexité grandissante des divers composants matériels, exécuter du code sur du matériel spécifique inclus le bon fonctionnement de ce matériel dans la base de confiance, peu importe les efforts fournis.

Inclusion de librairies et primitives dans la base de confiance

Dans la section 4.4.2, nous avons soutenu l'argument que les fonctions de l'interface utilisées par la fonction d'élection étaient assez simples pour se convaincre *sans l'ombre d'un doute* que leur implémentation correspondait à leur spécification. Cependant deux fonctions de cette interface sont trop complexes pour pouvoir l'affirmer : une fonction

exécutant une insertion triée dans une liste, et une fonction appliquant une modification à chaque élément d'une liste.

La raison principale derrière ce choix d'interface est que nous pensons que le coeur de notre contribution est la preuve de correction de la fonction d'élection. Nous avons estimé que l'aspect de gestion de la mémoire requise avec l'utilisation de liste dépassait ce cadre, et que cette preuve était orthogonale aux propriétés que nous avons prouvé. Étant donné leurs tailles raisonnables (respectivement 13 et 7 lignes de code C), nous avons décidé des les considérer comme du logiciel de confiance – comme les oracles et le back-end – et que nous n'allions pas les détailler dans le modèle.

Néanmoins, dans le cadre d'une preuve où chaque élément logiciel *doit* être prouvé (par exemple dans le cadre d'une preuve reposant seulement sur des primitives du matériel), on pourrait imaginer appliquer la même méthodologie pour se débarasser de ces primitives. La première étape serait de concevoir une nouvelle interface qui décompose les deux primitives, puis d'écrire la fonction d'élection en utilisant uniquement les nouvelles primitives, puis en montrant que la nouvelle fonction d'élection raffine la fonction d'élection originale.

Par ailleurs, nous souhaitons revenir sur l'existence même de cette discussion à propos de nos choix d'interface. Nous pensons que notre méthodologie de preuve, et particulièrement le *shallow embedding*, met en lumière les hypothèses introduites dans la base de confiance, et nous pousse à les discuter. Plus spécifiquement, puisque la preuve et l'implémentation sont exprimées dans le même langage, ils partagent la même définition des hypothèses de la base de confiance, ce que nous pensons particulièrement bénéfique au processus. À titre de comparaison, il nous semble peu discutable que les preuves conduites avec un *deep embedding*, et donc conduites sur la représentation de l'*AST* du code du langage représenté, ajoute une couche supplémentaire d'obfuscation aux hypothèses introduites dans la base de confiance.

Raisonnement sur le code Gallina et confiance dans le code compilé

Une chaîne d'outils qui propage les propriétés jusqu'au code compilé est le modèle d'excellence que chaque projet muni de preuve formelle sur du code devrait chercher à atteindre. Nous nous efforçons d'atteindre ce but. Comme mentionné dans l'état de l'art, la chaîne de compilation de Pip utilise soit Digger, soit ∂x pour compiler le code Gallina *shallow embedded* vers du code C classique. Aucun de ces deux outils n'est pour l'instant muni de preuve de préservation de la sémantique ; ∂x a cependant été développé pour poursuivre ce but.

De nombreux travaux existent déjà à ce sujet. Œuf [Mul+18] permet de compiler un grand sous-ensemble de code Gallina idiomatique vers du code C, avec la garantie que :

> *des appels valides aux fonctions compilées par Œuf auront un comportement équivalent à l'implémentation originale en Gallina* – [Mul+18], Section 2.1

Le programme Gallina est compilé vers la représentation `Cminor` de CompCert, et peut

donc être compilé vers du code assembleur avec CompCert. D'une manière assez similaire, le projet CertiCoq [Ana+17] vise à vérifier la compilation de *n'importe quel* programme Gallina en `Clight`, qui peut donc être compilé à nouveau vers du code assembleur par n'importe quel compilateur C – y compris CompCert.

Ces deux contributions de la communauté ont un point commun : elles ajoutent un ramasse-miette au code C compilé. Un ramasse-miette est un logiciel complexe permettant au développeur de ne pas se préoccuper de libérer la mémoire allouée dynamiquement lorsqu'il n'en a plus besoin. Ce logiciel fait parti de l'environnement d'exécution de Gallina, mais n'est pas nativement inclus avec le langage C ou disponible sur du matériel. L'utilisation d'un ramasse-miette a généralement un impact négatif sur les performances, et peu engendrer une surconsommation de mémoire. De plus, l'inclusion d'un ramasse-miette dans le code compilé ajoute un bout de logiciel complexe (et non-prouvé) dans la base de confiance. Le compromis apporté par l'inclusion d'un tel logiciel contraste avec les contraintes typiques des systèmes embarqués, qui s'accomodent plutôt de logiciels légers munis d'un minimum de dépendances.

D'autres travaux ont abordé le problème avec une méthodologie opposée, partant du code C pour le vérifier formellement. RefinedC [Sam+21], la contribution la plus récente du projet RustBelt, utilise des annotations de code C pour créer des spécifications et automatiser la plupart des preuves de programmes. Le code C est compilé vers un langage appelé Caesium, qui est représenté sous la forme d'un *deep-embedding* en Gallina, et sur lequel la preuve est vérifiée. Le code source initial étant du C, il peut être compilé directement sans inclure de ramasse-miette. Néanmoins, tout comme notre propre méthodologie, cette méthodologie suppose que leur outil de compilation de C vers Gallina est correct et que la sémantique de Caesium est correcte.

Le projet *VST* (pour *Verified Software Toolchain* a suivi la même approche. Leur contribution la plus récente est VST-Floyd [Cao+18]. VST-Floyd est un écosystème de lemmes et de tactiques conçues pour faciliter l'utilisation de *Verifiable C*, une logique de séparation d'ordre supérieur. *Verifiable C* est prouvé correct selon la sémantique opérationnelle de `Clight`, formalisé dans Coq. Cela permet de raisonner sur n'importe quel programme `Clight`, procurant outils et méthodes pour faciliter la vérification de programmes.

4.6 Synthèse des travaux et contributions de ce chapitre

Ce chapitre a présenté les travaux qui ont mené à une implémentation d'un ordonnanceur *Earliest Deadline First* pour des séquences arbitraires de *jobs*, dont la fonction d'élection a été prouvée correcte formellement. Plus spécifiquement, il a été montré que l'implémentation exécutable de la fonction d'élection respecte la politique d'ordonnancement *Earliest Deadline First*. L'ordonnanceur présenté s'exécute en espace utilisateur dans une partition de mémoire sur un système s'exécutant avec Pip. Ce chapitre a présenté les étapes de la preuve de correction de la fonction d'élection, puis a présenté l'interface du code vérifié avec le code faisant partie de la TCB. Nous avons ensuite discuté de la TCB de

nos travaux en la comparant avec la TCB de travaux connexes. À notre connaissance, nos travaux sont les premiers qui proposent une implémentation prouvée d'un ordonnanceur EDF pouvant ordonnancer une séquence arbitraire de *jobs*.

De plus, l'ordonnanceur implémenté utilise le service de transfert de flot d'exécution décrit dans le chapitre précédent, montrant par l'exemple que le service est suffisant pour implémenter efficacement des composants systèmes tel qu'un ordonnanceur. La preuve de correction de l'ordonnanceur a été menée selon la méthode de co-design usuelle de Pip, mais a cependant été conduite par raffinement, contrairement aux preuves d'isolation de Pip qui ont été prouvée par méthode directe. Ces travaux montrent que cette méthodologie est efficace pour mener des preuves sur d'autres propriétés que les propriétés classiques de Pip et qu'elle s'accomode d'autres méthodes de preuve, comme par exemple le raffinement.

5 Méthode d'ajout de nouvelles propriétés sur le code des services

Ce chapitre présente une preuve de concept affranchissant le code des services de Pip de toutes ses dépendances au modèle actuel d'isolation. Ces travaux n'ont pas été publiés, mais sont néanmoins consultables sur la branche `state_abstraction` du projet Pip, disponible sur le dépôt accessible à l'adresse suivante : https://gitlab.univ-lille.fr/2xs/pip/pipcore. Cette branche contient une implémentation imparfaite des idées exposées dans ce chapitre, et restreinte à la fonction `switchContextCont`.

Par ailleurs, ce chapitre aborde de manière très superficielle certains objets issus de la théorie des catégories, qui pourraient attiser votre curiosité. Je souhaite mentionner ici [Mil19], un livre (gratuit !) qui aborde selon le point de vue d'un programmeur des concepts de la théorie des catégories. Cette lecture vous permettra certainement de mieux cerner – entre autres – le concept de foncteur ou encore de monade.

5.1 Motivations

La section de discussion du premier chapitre a mis en avant le fait que Pip était conçu autour des propriétés d'isolation formellement prouvées. Le modèle des fonctions de l'interface et de l'état, jusqu'à la monade intégrée au code, sont liés aux propriétés d'isolation. Cette forte proximité est une conséquence de la philosophie de conception minimaliste de Pip, qui a incité à ne définir que les éléments strictement nécessaires à l'établissement de la preuve de préservation de l'isolation. Cette approche a permis de minimiser l'effort de preuve permettant de garantir la propriété d'isolation, mais présente un désavantage majeur : le code des services de Pip n'est pas indépendant des modèles sur lesquels il repose.

Ainsi, il n'existe qu'un modèle unique dans Pip qui ne peut évoluer que de manière itérative. Chaque évolution rend caduques les propriétés établies sur l'ancien modèle, et implique de produire une nouvelle preuve des mêmes propriétés avec le nouveau modèle. Le moindre ajout de chaque itération rendant de plus en plus difficile l'établissement la preuve à produire.

Ceci est un frein considérable à la vérification de nouvelles propriétés sur le code de Pip, telle que la preuve fonctionnelle du service évoquée dans le second chapitre. Si les nouvelles propriétés impliquent des changements trop importants sur le modèle, l'effort de preuve à fournir deviendrait inatteignable après seulement quelques itérations.

De plus, il n'est pas possible d'utiliser la composition verticale des propriétés telle que proposée dans le chapitre précédent. Ici, il n'est pas question d'apporter des garanties formelles à un nouvel élément logiciel ; on souhaite apporter de nouvelles garanties formelles à du logiciel existant. On pourrait par exemple reprendre le constat du premier chapitre

sur la pertinence de la preuve de préservation de l'isolation sur le code du service de transfert de flôt d'exécution (3.4.3). On souhaiterait par exemple pouvoir prouver des propriétés relatives à la correction du service en plus des propriétés d'isolation.

Ce constat a motivé la preuve de concept développée dans ce chapitre, qui propose une solution au problème de la croissance exponentielle de l'effort de preuve.

5.2 Architecture monolithique

Cette section décrira de manière synthétique les composants actuels de Pip, en essayant de décrire leurs dépendances d'un point de vue logiciel. Elle commencera par donner brièvement une vue d'ensemble du projet. Puis, dans une première partie, elle décrira les dépendances du code des services sur les modèles décrits dans Pip. Elle dépliera les définitions pour mettre en lumière les dépendances qui existent entre les modèles des différents composants. Ensuite, dans une seconde partie, la section se penchera sur la méthode de preuve nécessaire à l'établissement de la preuve d'isolation. La section se concluera sur le processus de compilation du code, compilant le code des services de Gallina vers du code C.

5.2.1 Vue générale

L'architecture actuelle de Pip est décrite dans la figure 5.1. Les modèles nécessaires à la spécification des propriétés d'isolation sont placés dans la partie gauche de la figure. Ces modèles sont inter-dépendants : le modèle de l'état dépend du modèle des types, la monade d'état dépend du modèle de l'état, et le modèle des fonctions de l'interface dépend de la monade d'état, et du modèle des types. De plus, le code des services est dépendant de tout ces modèles. Par ailleurs, les fonctions "fictives" permettant de décrire les propriétés d'isolation, les triplets de Hoare et propriétés d'isolation en elles-même sont dépendants du code des services et des modèles précédents.

La partie exécutable du noyau est représentée dans la partie droite de la figure 5.1. Le code des services écrit en Gallina est compilé en code C grâce à Digger ou ∂x. Le code des services compilé en C repose sur les implémentations exécutables des fonctions intéragissant avec l'état et les types réels. Le modèle de l'état et la monade d'état disparaissent pour laisser place à l'état et la séquentialité intrinsèque du code exécutable.

Le listing 5.1 représente la portion du code des services prise pour exemple dans ce chapitre. Les prochaines sections vont commenter les dépendances de cette portion de code avec les modèles d'isolation, puis montrer comment elle peut être affranchie de ces dépendances graĉe à cette contribution.

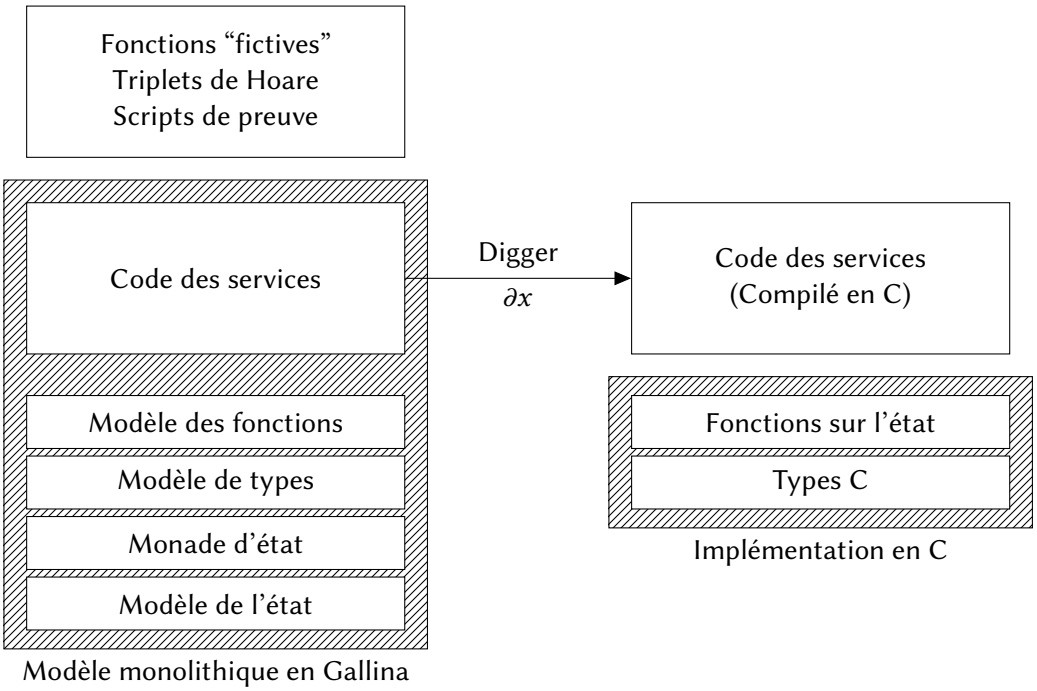

Figure 5.1 – Architecture actuelle de Pip et dépendances des composants

5.2.2 Dépendance du code au modèle d’isolation

Monade dépendante du modèle de l’état

Dans l’architecture actuelle de Pip, le code des services repose directement sur les définitions de la monade, en utilisant le type monadique `LLI`, et les fonctions `bind` et `ret` pour représenter la mise en séquence des instructions des services. Dans l’exemple présenté en listing 5.1, la fonction retourne un type monadique `LLI yield_checks`, utilise la fonction `bind` au travers du sucre syntaxique `perform [ ... ] := [ ... ] in`, et indique sa valeur de retour grâce à la fonction `ret`. Malheureusement, le type monadique `LLI`, décrit en listing 5.2, dépend de l’état `state`, décrit en listing 5.3. `state` est le modèle de l’état conçu pour la preuve de préservation de l’isolation. Ceci est une première dépendance du code au modèle d’isolation, dont le code devra se passer pour devenir indépendants des modèles construits pour Pip.

Code dépendant des modèles de types

D’autres dépendances du code aux modèles d’isolation passent par la représentation des types. Le code dépend des types utilisés pour représenter ses propres arguments, et valeur de retour enrobée par le type monadique `LLI`, mais aussi les arguments et valeurs de

```
Definition switchContextCont (targetPartDesc : page)
                             (targetPageDir  : page)
                             (flagsOnYield   : interruptMask)
                             (targetContext  : contextAddr)
                             : LLI yield_checks :=
  setInterruptMask flagsOnYield ;;
  updateMMURoot targetPageDir ;;
  updateCurPartition targetPartDesc ;;
  perform flagsOnWake := getInterruptMaskFromCtx targetContext in
  setInterruptMask flagsOnWake ;;
  (* allow root partition to prevent Pip from enforcing interrupts *)
  perform rootPartition := getPageRootPartition in
  perform targetReqNoInterrupt := noInterruptRequest flagsOnWake in
  (
  if (targetIsRoot && targetReqNoInterrupt)
  then
    loadContext targetContext false
  else
    loadContext targetContext true
  ) ;;
  (* Actually never reached *)
  ret SUCCESS.
```

Listing 5.1 – Code du bloc de continuation `switchContextCont` du service de transfert de flot d'exécution

```
Definition LLI (A :Type) : Type := state → result (A * state).
```

Listing 5.2 – Définition du type de la monade d'état `LLI` dans le modèle actuel de Pip

```
Record state : Type := {
    currentPartition : page;
    memory : list (paddr * value)
}.
```

Listing 5.3 – Définition de l'état `state` dans le modèle actuel de Pip

retour des fonctions de l'interface. Par exemple, la fonction `switchContextCont` décrite dans le listing 5.1, dépend du modèle des types `page`, `interruptMask`, `contextAddr` et `yield_checks`. Les modèles de ces types sont représentés dans le listing 5.4. Cette dépendance renforce les liens entre le modèle d'isolation de Pip et le code de ses services, et doit donc disparaître.

```
Record page := {
  p :> nat;
  Hp : p < nbPage }.

Record interruptMask := {
 m :> list bool;
 Hm : length m = maxVint+1;
}.

Definition contextAddr := nat.

Inductive yield_checks : Type :=
(* Autres éléments de l'enum omis
| [ ... ] *)
| SUCCESS_Cons.

Definition SUCCESS := SUCCESS_Cons.
```

Listing 5.4 – Définition des types nécessaires à la fonction `switchContextCont` dans le modèle actuel de Pip

Code dépendant des modèles des fonctions intéragissant avec l'état

Enfin, la dernière dépendance du code aux modèles est par le biais des fonctions intéragissant avec l'état. Le code des services fait directement appel *aux modèles* de ces fonctions. Ainsi, la fonction `switchContextCont` présentée dans le listing 5.1, est dépendant des modèles des fonctions `setInterruptMask`, `updateMMURoot`, `updateCurPartition`, `getInterruptMaskFromCtx`, `getPageRootPartition`, `noInterruptRequest` et `loadContext`. Cette dépendance n'a pas lieu d'être, et doit être supprimée pour atteindre un code des services agnostique des modèles.

Extraction de l'*AST* dépendant des modèles

Ce dernier paragraphe est dédié au fichier source Coq extrayant l'*AST* du code des services de Pip. Ce fichier, une fois évalué par Coq, produit le fichier attendu en entrée par

```
Definition setInterruptMask (mask : interruptMask) : LLI unit :=
  ret tt.

Definition updateMMURoot(pageDir : page) : LLI unit :=
  ret tt.

Definition updateCurPartition (phyPartition : page) : LLI unit :=
  modify (fun s ⇒
    {| currentPartition := phyPartition ; memory := s.(memory) |}
  ).

Definition getInterruptMaskFromCtx (context : contextAddr)
           : LLI interruptMask :=
  ret int_mask_d.

Definition getPageRootPartition : LLI page :=
  ret pageRootPartition.

Definition page_eqb (p1 : page) (p2 : page) : SAM bool :=
  ret (p1 =? p2).

Definition noInterruptRequest (flags : interruptMask) : LLI bool :=
  ret true.

Definition loadContext (contextToLoad : contextAddr)
                       (enforce_interrupt : bool) : LLI unit :=
  ret tt.
```

Listing 5.5 – Définition des fonctions de l'interface avec l'état nécessaire à la fonction `switchContextCont` dans le modèle actuel de Pip

Digger, un des outils de compilation du code Gallina *shallow embedded* vers du code C utilisé dans le projet Pip. Ce fichier a pour dépendances l'ensemble des modèles d'isolation sur lesquels reposent le code. Ainsi, dans l'état actuel du projet, l'extraction de l'*AST* du code des services n'est possible que si l'intégralité des modèles d'isolation peut être évalué par Pip. Coq ne permet pas d'extraire un fragment d'AST si chaque fonction et chaque type utilisés par ce fragment ne sont pas implémentés ou axiomatisés. Cette dépendance doit donc être supprimée pour que le code des services puisse être compilé en C sans avoir recours aux modèles.

5.2.3 Processus de preuve sur le code dépendant du modèle

Cette sous-section est dédiée à la structure actuelle de la preuve d'isolation sur le code des services, mettant en avant les dépendances des différents groupes de fichiers nécessaires à chaques preuves.

Définition des propriétés d'isolation et des fonctions "fictives"

Les premiers fichiers nécessaires à l'établissement de la preuve sont ceux contenant les fonctions "fictives", nécessaires à l'expression des propriétés d'isolation sur le noyau. Pour rappel, les fonctions "fictives" ont vocation à spécifier ce qui doit être prouvé et non à être exécutées en tant que tel. Elles servent de fondation à l'expression des triplets de Hoare qu'on souhaite prouver pour garantir de manière formelle les propriétés souhaitées sur le code, comme les propriétés d'isolation. Ces fonctions peuvent permettre entre autres de définir des ensembles nécessaires à certaines propriétés d'isolation. C'est le cas de la fonction `getAccessibleMappedPages` qui récupère les pages mappées et accessibles dans l'espace d'adressage d'une partition, nécessaire à la propriété d'isolation noyau `kernelDataIsolation`. Ces fonctions peuvent aussi être des miroirs purement fonctionnels de code monadique présent dans les services de Pip, telle que la fonction `readPhysical` permettant de lire l'adresse d'une page mémoire.

Une fois les définitions de ces fonctions établies, il est possible de définir les propriétés qui justifient leur existence, c'est à dire ici les propriétés d'isolation. Les fonctions "fictives" et propriétés définies de cette manière dépendent – **à juste titre** – des modèles d'isolation, que ce soit le modèle de types, de l'état, ou des fonctions intéragissant avec l'état. En effet, ces dépendances sont nécessaires à l'établissement d'une preuve formelle ; d'une certaine manière, ces modèles projettent les dépendances de l'objet étudié (le code des services de Pip) vers le monde mathématique et permettent de les représenter.

Définition des triplets de Hoare, des lemmes intermédiaires et des scripts de preuve

Une fois que ces définitions établies, il est possible d'exprimer les triplets de Hoare sur le code des services. À titre d'exemple, le listing 5.6 décrit le triplet de Hoare de la fonction `switchContextCont`. Sous chaque triplet (et chaque lemme intermédiaire) se trouve un script de preuve, décrivant les règles d'inférence (ou tactiques) à appliquer successivement pour faire progresser Coq vers la conclusion. Les triplets de Hoare dépendent du code des services, des fonctions fictives utiles à la définition des propriétés, et dépendent donc à fortiori des modèles d'isolation, encore une fois, à juste titre.

```
Lemma switchContextCont (targetPartDesc : page)
                        (targetPageDir  : page)
                        (flagsOnYield   : interruptMask)
                        (targetContext  : contextAddr) :
{{ (* Preconditions *)
  fun s ⇒
  partitionsIsolation s ∧
  kernelDataIsolation s ∧
  verticalSharing s ∧
  consistency s ∧
  List.In targetPartDesc (StateLib.getPartitions pageRootPartition s) ∧
  targetPartDesc <> pageDefault
}}

switchContextCont targetPartDesc targetPageDir flagsOnYield targetContext

{{ (* Postconditions *)
  fun _ s  ⇒
  partitionsIsolation s ∧
  kernelDataIsolation s ∧
  verticalSharing s ∧
  consistency s
}}.
```

Listing 5.6 – Définition du triplet de Hoare de la fonction `switchContextCont` pour la preuve de préservation de l'isolation de Pip

5.3 Abstraction des modèles dans le code des services de Pip

Cette section détaille l'objet de la preuve de concept mise à l'honneur dans ce chapitre : la modularisation des modèles et preuves des services de Pip, ainsi que l'autonomie du code des services vis à vis des modèles. Elle commence par donner une vue globale de la nouvelle architecture du projet, indiquant les nouvelles relations entre les différents composants du projet. Dans un second temps, elle détaille les interfaces créées, en illustrant les changements apportés à Pip du point de vue de la fonction `switchContextCont`, utilisée comme exemple dans la section précédente. Cette section met en évidence les différences avec l'implémentation précédente dépendantes des modèles. Cette seconde partie décrit aussi le processus d'extraction de l'*AST* du code des services. Dans une dernière partie, cette section décrit la nouvelle structure des fichiers de preuve, en illustrant les différences (plus conceptuelles) avec l'architecture précédente.

5.3.1 Vue générale

La principale contribution de cette preuve de concept est l'ajout d'*interfaces* décrivant les dépendances fondamentales du code des services aux autres composants logiciels évoqués dans la section précédente 5.2.3. Le code des services de Pip repose sur cette interface, qui ne décrit que les opérations ou types à fournir au code. L'implémentation réelle (et exécutable) de cette interface est réalisée en C, et s'exécutera conjointement avec le code des services compilé par Digger ou ∂x. Du coté du monde de la preuve formelle, de *multiples* modèles peuvent décrire cette interface et ses effets. La figure 5.2 décrit l'architecture de Pip selon cette preuve de concept. La colonne du milieu représente les interfaces nouvellement créées, sur lesquelles le code des services repose. La colonne de gauche représente les modèles décrivant les interfaces, et les preuves reposant sur ces interfaces. La colonne de droite représente l'implémentation réelle de l'interface en C sur laquelle repose le code des services compilé par Digger ou ∂x. Les flèches continues dans cette figure indiquent des composants dérivés automatiquement d'autres composants, par exemple le code des services compilé en C est dérivé automatiquement du code des services écrit en Gallina. Les flèches discontinues traduisent une relation d'implémentation. Par exemple, le modèle des fonctions implémente l'interface des fonctions.

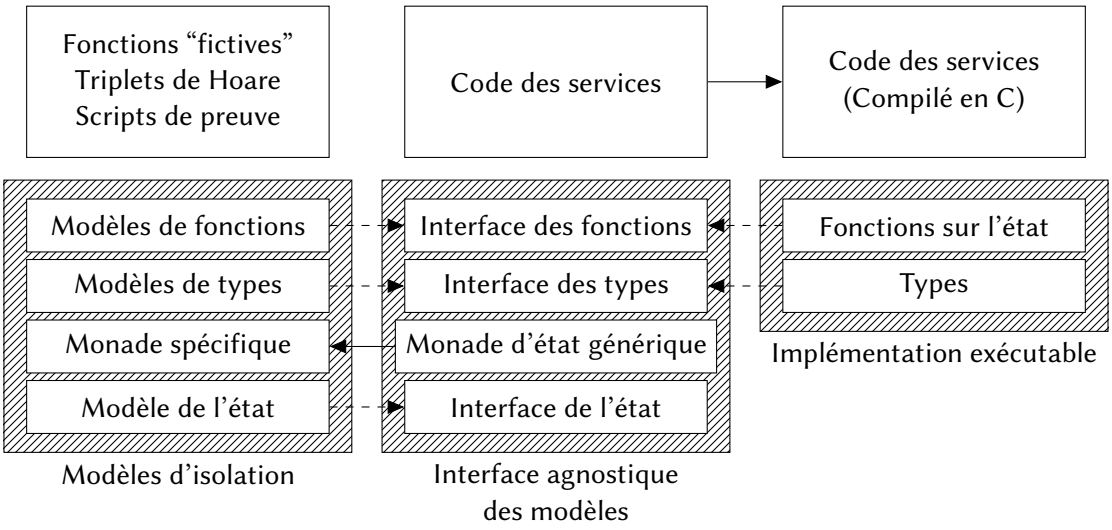

Figure 5.2 – Nouvelle architecture de Pip et dépendances des composants apportées par la preuve de concept

Chaque interface est décrite en Coq sous la forme d'un *Module Type*. Les implémentations de *Modules Type* sont décrit en Coq par des *Modules*, qui doivent nécessairement donner une définition à chaque élément déclaré dans le *Module Type*.

5.3.2 Définition de code générique indépendant des modèles

Abstraction du modèle de l'état

Le premier composant abstrait par cette preuve de concept est la définition de l'état. Cette abstraction donne naissance au premier *Module Type* de cette contribution, le *Module Type* de l'état. Ce *Module Type* est extrêmement simple : il ne contient qu'une unique déclaration, le type `state`. La définition du *Module Type* est présentée en listing 5.7. Ainsi, chaque modèle devra fournir son propre *Module* contenant la définition du type de l'état.

```
Module Type StateModel.

  Parameter state : Type.

End StateModel.
```

Listing 5.7 – Définition de l'interface de l'état

La définition du *Module* implémentant le modèle d'isolation reprend les définitions présentées dans la section précédente, et est disponible en annexe en listing B.1.

Définition d'une monade d'état agnostique du modèle de l'état

La seconde modification apportée par cette contribution est la création d'une monade agnostique du modèle de l'état. Bien que la monade d'état ait besoin de faire référence à l'état, elle n'a pas besoin de sa définition. Ainsi, il est possible de créer un *Module* paramétré par le *Module Type* de l'état. Le *Module* ainsi créé est de ce fait un foncteur (*functor* en anglais), une fonction des *Modules* vers les *Modules*. En particulier, le foncteur de la monade d'état prend en paramètre un *Module* implémentant le *Module Type* `StateModel`, et renvoie un *Module* décrivant la monade d'état associée. Le *Module* implémentant la monade d'état agnostique au modèle de l'état est disponible en annexe, B.2.

Abstraction du modèle de types utilisés par Pip

La seconde interface créée est celle des types utilisés par le code des services et les fonctions sur l'état. Tout comme l'interface de l'état, cette interface est créée grâce à un *Module Type* dont les déclarations pourront être utilisées par le code des services et les fonctions sur l'état. De la même manière, les objets déclarés dans ce *Module Type* doivent être implémenté par chaque modèle sous la forme d'un *Module*. Ces déclarations doivent aussi être implémentées en C ; cette implémentation sera utilisée par le code des services traduit en C par Digger ou ∂x. Le code du *Module Type* déclarant les définitions nécessaires à la fonction `switchContextCont` est disponible en listing 5.8.

```
Module Type TypesModel.

  Parameter page : Type.
  Parameter contextAddr : Type.
  Parameter interruptMask : Type.

  Inductive yield_checks : Type :=
  (* [ ... ] *)
  | SUCCESS.

End TypesModel.
```

Listing 5.8 – Définition de l'interface des types nécessaires à la fonction `switchContextCont`

Abstraction du modèle des fonctions de l'interface avec l'état

```
From Pip.Model.Meta Require Import TypesModel StateModel StateAgnosticMonad.

Module Type InterfaceModel (Export Types : TypesModel)
                           (Export State : StateModel).

  Module SAMM := StateAgnosticMonad State.
  Export SAMM.
  (* Monad Interface *)

  Parameter setInterruptMask : interruptMask → SAM unit.
  Parameter updateMMURoot : page → SAM unit.
  Parameter updateCurPartition : page → SAM unit.
  Parameter getInterruptMaskFromCtx : contextAddr → SAM interruptMask.
  Parameter getMultiplexer : SAM page.
  Parameter page_eqb : page → page → SAM bool.
  Parameter noInterruptRequest : interruptMask → SAM bool.
  Parameter loadContext : contextAddr → bool → SAM unit.

End InterfaceModel.
```

Listing 5.9 – Définition de l'interface des fonctions utilisées par le code des services (restreinte aux définitions nécessaires à la fonction `switchContextCont`)

La dernière interface créée est celle déclarant les fonctions intéragissant avec l'état. Cette interface a besoin de faire référence à la fois aux types utilisés par les fonctions mais aussi à la monade, puisque les fonctions décrites sont monadiques. Ainsi, cette interface est définie par un *Module Type* paramétré par le *Module Type* déclarant les types nécessaires aux fonctions et par le *Module Type* de l'état. Ce *Module Type* représente donc le type d'un foncteur allant d'un *Module* décrivant les types et d'un *Module* décrivant l'état vers un *Module* décrivant les fonctions monadiques d'interaction avec l'état utilisant les types passés en paramètre.

L'implémentation de ce *Module Type* (restreint aux définitions nécessaires à la fonction `switchContextCont`) est disponible en listing 5.9. Cette implémentation utilise le foncteur `StateAgnosticMonad` sur le *Module* d'état `State` passé en paramètre pour instancier le *Module* de la monade d'état `SAMM` qui lui est nécessaire. Cette instanciation permet ensuite de déclarer les prototypes de fonction utilisant le type monadique défini dans ce *Module*. Les fonctions dont les prototypes ont été déclarés dans ce *Module Type*, ainsi que la monade d'état utilisée devront être définis par le *Module* implémentant le *Module Type*.

Code agnostique des modèles

La création de toutes ces interfaces permet de définir le code des services affranchit de toute dépendance aux modèles. Le code des services (restreint à la seule fonction `switchContextCont` dans cette preuve de concept) est disponible en listing 5.12. Le code est défini dans un *Module* dépendant de l'implémentation de trois *Modules* : un *Module* décrivant l'état, un *Module* décrivant les types, et un *Module* décrivant les fonctions monadiques. Ainsi, le code de Pip est devenu un foncteur allant de ces *Modules* vers le *Module* décrivant le code **spécifique** aux *Modules* dont il dépend.

Deux différences minimes se dégagent de cette nouvelle manière de représenter le code. La première est la déclaration des modèles paramètres du *Module* du code des services, mis en évidence dans le listing 5.10.

```
Module ModelAgnosticCode (Export Types : TypesModel)
                         (Export State : StateModel)
                         (Export Interface : InterfaceModel Types State).
```

Listing 5.10 – Déclarations des *Modules* paramètres du *Module* du code des services

La seconde différence de cette nouvelle représentation du code est l'utilisation de la monade agnostique de l'état `SAM`, à la place de la monade dépendante du modèle de l'état d'isolation `LLI`, comme décrit dans le listing 5.11.

Une représentation complète de la fonction `switchContextCont` est disponible en listing 5.12. Seules ces deux différences permettent de distinguer le code des services écrit comme un foncteur dans ce listing du code dépendant directement des modèles décrit en listing 5.1.

```
Definition switchContextCont (* [ ... ] *) : SAM yield_checks :=
```

Listing 5.11 – Monade instanciée `SAM`, type de retour de la fonction `switchContextCont` dans la preuve de concept

5.3.3 Extraction de l'*AST* du code des services

Pour que la preuve de concept présentée dans ce chapitre soit pertinente, il faut que le code des services puisse être compilé vers du code C. Comme évoqué dans les chapitres précédents, Pip est conçu pour s'interfacer avec deux outils permettant la compilation de code Gallina *shallow-embedded* vers du code C : Digger ou *∂x*.

Cette sous-section discute du fichier source qui, une fois evalué par Coq, produit une représentation de l'*AST* du code des services nécessaire à Digger pour produire le code C. La section précédente 5.2.2 discutait du fait que le fichier source importait tous les modèles d'isolation, liant de ce fait le code des services à ces modèles. Cependant, dans cette preuve de concept, le code n'est plus qu'un foncteur des modèles vers une instance spécifique du code. D'une certaine manière, l'ancien code des services est une instance particulière du code paramétrique de cette preuve de concept : le cas où le code aurait été instancié avec les *Modules* des modèles d'isolation.

Malheureusement, il n'est pas possible pour Coq de produire l'*AST* d'un foncteur au moment où j'écris ces lignes. Néanmoins, Coq accepte de produire l'*AST* d'une fonction – et ce même sans connaître la définition des objets sur lesquels cette fonction repose. Ainsi, il est possible de définir des *Modules* "creux", qui se contentent de déclarer qu'ils implémentent les *Modules Type* attendus par le code des services agnostique des modèles, sans préciser d'implémentation. Dès lors, il est possible d'instancier le code avec ces modèles creux, ce qui transforme le code des services en fonctions dont Coq peut extraire l'*AST*. De cette manière, il est possible d'extraire l'*AST* des services de Pip sous la forme proposée par cette preuve de concept sans dépendre d'une implémentation particulière des modèles.

Les listings 5.13 et 5.14 présentent la même portion de code extraits avec l'architecture actuelle de Pip ou dans la preuve de concept en instanciant le code de Pip avec les modèles creux. La ligne de code concernée est la suivante : `updateCurPartition targetPartDesc ;;`, provenant de la fonction `switchContextCont` décrit en listing 5.1. Dans le listing 5.13, la fonction `bind` utilisée provient du fichier `Hardware` où est définie la monade reposant sur le modèle de l'état d'isolation. La fonction `updateCurPartition` provient du fichier `MAL`, qui définit une partie des modèles des fonctions intéragissant avec l'état selon le modèle d'isolation.

Dans le listing 5.14, la fonction `bind` provient de la monade générique `SAMM` instanciée avec le modèle creux de l'état. La fonction `updateCurPartition` provient de la l'implémentation creuse de l'interface des fonctions intéragissant avec l'état.

```
From Pip.Model.Meta Require Import TypesModel StateModel InterfaceModel.
Require Import Bool.

Module ModelAgnosticCode (Export Types : TypesModel)
                         (Export State : StateModel)
                         (Export Interface : InterfaceModel Types State).


Definition switchContextCont (targetPartDesc : page)
                             (targetPageDir  : page)
                             (flagsOnYield   : interruptMask)
                             (targetContext  : contextAddr)
                             : SAM yield_checks :=

  setInterruptMask flagsOnYield ;;
  updateMMURoot targetPageDir ;;
  updateCurPartition targetPartDesc ;;
  perform flagsOnWake := getInterruptMaskFromCtx targetContext in
  setInterruptMask flagsOnWake ;;
  (* allow root partition to prevent Pip from enforcing interrupts *)
  perform rootPartition := getMultiplexer in
  perform targetIsRoot := page_eqb rootPartition targetPartDesc in
  perform targetReqNoInterrupt := noInterruptRequest flagsOnWake in
  (
  if (targetIsRoot && targetReqNoInterrupt)
  then
    loadContext targetContext false
  else
    loadContext targetContext true
  ) ;;
  (* Actually never reached *)
  ret SUCCESS.

End ModelAgnosticCode.
```

Listing 5.12 – Définition du code affranchi de toute dépendance aux modèles

```
    (* [ ... ] *)
Hardware.bind (MAL.updateCurPartition targetPartDesc) (\_ →
    (* [ ... ] *)
```

Listing 5.13 – Fragment de l'*AST* du code de Pip actuel représenté sous forme de code Haskell

```
    (* [ ... ] *)
HollowModel._HollowInterface__SAMM__bind
  (HollowModel._HollowInterface__updateCurPartition targetPartDesc)
    (\_ →
    (* [ ... ] *)
```

Listing 5.14 – Fragment de l'*AST* du code de Pip instancié sur les modèles creux représenté sous forme de code Haskell

5.3.4 Méthode de preuve sur des foncteurs

Cette preuve de concept ne serait pas utile s'il n'était pas possible de raisonner formellement sur le code des services. Cette brève sous-section discute de la manière de structurer les nouveaux modèles et les nouvelles preuves afin de pouvoir prouver des propriétés formelles. L'établissement de modèles et la méthode de preuve restent *identiques* aux méthodes utilisées jusqu'à présent.

Tout d'abord, il est important de souligner que les *Modules* implémentant les *Modules Types* sont libres des choix de leur implémentation : ils peuvent dépendre de n'importe quel modèle et décrire ce que bon leur semble. Pour rappel, ces *Modules* sont une projection de l'implémentation exécutable de l'interface dans le monde des mathématiques. La seule règle que doivent respecter ces *Modules* est qu'ils doivent définir l'ensemble des éléments déclarés dans le *Module Type* qu'ils implémentent.

Dans cette preuve de concept, le code des services a été séparé des modèles. Afin de pouvoir raisonner formellement sur le code des services, il est désormais nécessaire de le spécialiser avec des modèles. Plus précisemment, il est nécessaire d'instancier un *Module* contenant le code des services spécialisé avec les *Modules* décrivant les modèles sur lesquels on souhaite raisonner. Par exemple, afin de pouvoir exprimer le triplet de Hoare de la fonction `switchContextCont` sur les modèles et propriétés d'isolation, le code des services doit tout d'abord être instancié avec les modèles d'isolation. La définition du triplet de Hoare de cette fonction est illustrée en annexe B.5.

Ce triplet de Hoare, lié cette preuve de concept, est en tout point similaire à la définition actuelle du triplet de Hoare décrit en listing 5.6 et pour cause : cette définition fait appel *aux mêmes objets* que la définition actuelle du triplet. Pour y parvenir, la preuve de concept **instancie** le code des services (et donc de la fonction `switchContextCont`) avec les mo-

dèles d'isolation. Le code de la fonction passe alors de l'état de foncteur à l'état de fonction, tel qu'exprimé actuellement dans les preuves de préservation de l'isolation de Pip.

La ligne d'intérêt dans l'annexe B.5, créant une instance du code des services spécialisés avec les modèles d'isolation, est isolée dans le listing 5.15. Cette ligne crée un nouveau *Module* en appliquant le foncteur du code des services aux *Modules* de type, de l'état et des fonctions d'isolation.

```
Module IsolationSpecificCode :=
  ModelAgnosticCode IsolationTypes IsolationState IsolationInterface.
```

Listing 5.15 – Instanciation du code des services sur les modèles d'isolation

Ainsi, la seule différence notable relative à la méthodologie de preuve apportée par cette preuve de concept est la nécessité de créer les modèles au travers de *Modules*.

5.4 Perspectives de recherche et discussion

Cette preuve de concept, bien qu'incomplète, montre qu'il est possible d'adapter le code des services de Pip et le modèle d'isolation actuel afin que le code des services devienne indépendant des modèles d'isolation. Cette indépendance permettrait à Pip d'intégrer de nouveaux modèles conçus pour raisonner sur d'autres aspects du code que ceux abordés par le modèle actuel. Cette section commencera par évoquer des perspectives de recherche à court terme pouvant émerger de cette preuve de concept, puis discutera des limites apportées par les preuves de propriétés sur des modèles parallèles.

5.4.1 Perspectives de recherche liées au développement d'un modèle alternatif

Comme évoqué dans les motivations de cette contribution en début de chapitre, l'établissement d'un modèle du code dédié à une preuve fonctionnelle des services est une des principales perspectives de cette contribution. On pourrait s'intéresser particulièrement aux propriétés de correction du service de transfert de flot d'exécution, pour lequel les propriétés d'isolation se sont révélées être peu intéressantes.

De plus, l'établissement de propriétés de bon fonctionnement sur le service de tranfert de flot d'exécution permettrait d'étudier à quel point il est possible de réutiliser les propriétés prouvées formellement sur la base de confiance (*TCB*) afin d'établir de nouvelles propriétés sur un nouvel objet.

Lien entre la preuve de bon fonctionnement et le back end de l'ordonnanceur

Cette perspective de recherche pourrait se concrétiser sur l'établissement d'une preuve de correction du *back-end* de l'ordonnanceur présenté dans le chapitre précédent reposant

sur la preuve de correction du service de transfert de flot d'exécution. Cette preuve se servirait des propriétés sur la sauvegarde et la restauration de contextes du service pour transférer le flot d'exécution aux *jobs* élus par la fonction d'élection, et montrerait que les *jobs* élus par la fonction d'élection sont effectivement exécutés après leur élection.

Cette perspective de recherche mettrait en évidence une composition verticale de propriétés formellement prouvées. Pour rappel, la composition verticale consiste à faire profiter aux logiciels ayant des dépendances des propriétés prouvées sur leurs dépendances. La particularité intéressante de cette composition verticale est que le lien entre les morceaux de logiciels munis de propriétés formelles serait direct : c'est à dire sans l'intermédiaire de code inclus dans la base de confiance.

5.4.2 Discussion sur la coexistence de preuve sur des modèles distincts

La preuve de concept présentée dans ce chapitre permet de faire coexister plusieurs modèles décrivant le code des services dans le but de conduire des preuves formelles sur ce code. Une question se pose alors : quelle est la relation de deux preuves formelles conduites sur des modèles différents et que disent-elles du code qui s'exécutera finalement sur la machine ?

La réponse est simple : les preuves formelles menées sur des modèles différents *ne sont aucunement reliées*, et ne *garantissent rien* sur le code exécutable. Au risque d'enfoncer une porte déjà ouverte dans les chapitres précédents, les modèles décrivant un objet du monde réel ne sont pas intrinsèquement équivalents à l'objet modélisé. Chaque preuve formelle propose une vérité absolument indéniable dans un monde féérique décrit par des modèles mathématiques ; transposer ces modèles au monde réel transfère inévitablement aux conclusions de la preuve formelle l'incertitude que les modèles décrivent correctement l'objet étudié. Ce fait n'enlève rien à l'utilité des modèles qui permettent d'étudier l'objet décrit. Il est cependant important de garder à l'esprit que les modèles ne renferment pas une vérité universelle ; vérité universelle dont l'existence même est une question philosophique.

Cet argument s'applique aussi aux modèles entre eux : des modèles différents décrivent des univers mathématiques différents. Ainsi, prétendre que les propriétés prouvées sur un modèle le sont aussi sur un autre introduit le même type d'incertitude. C'est pourquoi cette méthodologie n'est pas strictement équivalente à la méthodogie de preuve de propriétés n'impliquant qu'un unique modèle. Dans le cas d'un unique modèle, les propriétés doivent être prouvées *conjointement* sur le modèle unique. Une seule incertitude s'applique donc dans ce cas : il faut que le modèle unique décrive correctement l'objet étudié. Dans le cas de modèles parallèles, les propriétés sont prouvées indépendamment les unes des autres sur des modèles différents. Dans ce cas, il existe autant d'incertitude concernant la correction des modèles qu'il existe de modèles. Comme les raisonnements sur des modèles disjoints sont eux ausi disjoints, si les modèles (hypothèses) de chacun de ces raisonnements sont contradictoires les uns avec les autres, les raisonnements qui s'appuient sur chacuns d'eux n'ont aucune chance d'exhiber ces contradictions.

5.4.3 Preuve générique applicable à différents modèles

Un des rêves initiaux de cette contribution était de proposer une preuve générique, pouvant s'appliquer à tous les modèles. Cette preuve aurait été conçue comme une "preuve à trous", dont les morceaux manquants difficiles, et spécifiques à chaque modèles auraient été la partie manuelle de la preuve ; les autres aspects de la preuve ayant été couverts par la preuve générique.

L'architecture aurait été la suivante : chaque fonction de l'interface aurait été munie d'un invariant paramétrique, à définir et prouver comme des fonctions d'un *Module*. Ces invariants, disponibles pour chacunes des opérations du code, auraient été appliqués successivement par la preuve pour arriver à ses conclusions finales. J'espérais initialement que cette preuve générique allègerait le processus de preuve en faisant disparaître l'aspect répétitif de la preuve de code actuel, où la vaste majorité du raisonnement consiste à récupérer les nouvelles propriétés apportées par l'exécution des fonctions de l'interface, puis à les propager comme des propriétés supplémentaires disponibles lors de l'exécution des instructions suivantes.

Cette preuve générique est cependant beaucoup moins intéressante qu'initialement espéré. Tout d'abord, cette preuve générique ne simplifie aucunement le processus de preuve de propriétés actuelle. En effet, toute la difficulté de la preuve formelle réside dans la recherche et la preuve d'invariants, ce que cette preuve générique ne nous épargne pas. D'un point de vue subjectif, faire disparaître l'application successive des invariants sur les instructions retire une grosse portion du sentiment de progression dans l'établissement de la preuve formelle. De plus, il est parfois nécessaire d'exécuter plusieurs instructions avant de pouvoir arriver à un invariant simple, notamment dans le cas de modification de structures. Une preuve générique, s'appliquant donc à tous les modèles, s'accomoderait difficilement de ces écarts spécifiques à chaque preuve. Cette perspective de recherche semble donc finalement peu prometteuse.

5.5 Synthèse des travaux du chapitre et des contributions

Ce chapitre a présenté une preuve de concept séparant les modèles d'isolation de Pip du code des services. Cette séparation permet de définir des modèles alternatifs permettant de raisonner sur d'autres propriétés que les propriétés d'isolation classiques de Pip. Cette séparation a été rendue possible par la définition de *Modules* rendant explicite l'interface du code des services avec le reste de la TCB (telle que les types, l'état, et les primitives d'interaction avec l'état). Ce chapitre a aussi montré que cette séparation ne remet pas en question les fonctionnalités traditionnelles de Pip, telle que l'extraction de l'AST du code permettant de compiler Pip vers du C ou la preuve formelle de propriétés sur le code.

En outre, cette contribution permet d'ouvrir les perspectives de recherche sur Pip, notamment celles concernant la preuve de propriétés alternatives sur le code des services. Le premier chapitre de contribution sur le service de transfert de flot d'exécution avait

mentionné que les propriétés d'isolation n'étaient pas particulièrement pertinentes sur ce service. Ces travaux proposent une solution permettant d'établir des propriétés alternatives pertinentes, comme la preuve de bon fonctionnement du service.

6 Conclusion

Les principales contibutions de la thèse ont été exprimées dans les chapitres précédents. Ce dernier chapitre vise à fournir une synthèse des contributions développées dans ce document et à les remettre en relation, puis concluera ce document en proposant des perspectives de recherche à long terme liées aux travaux de cette thèse.

6.1 Résumé des contributions

Le premier chapitre de contribution a présenté un service de transfert de flot d'exécution unifié au sein du noyau Pip. Ce service est muni d'une preuve formelle de la propriété classique de Pip sur la préservation de l'isolation. Le service permet de gérer de manière unifiée les transferts explicites de flot d'exécution, les interruptions logicielles, les fautes, ainsi que les interruptions matérielles au travers de points d'entrées placés de manière opportune le long d'un fil d'exécution unique. Ce service de transfert de flot d'exécution, présenté dans ce chapitre pour l'architecture Intel x86, n'est cependant pas spécifique à cette architecture, et a notamment été implémenté sur l'architecture Armv7. Les modèles ayant permis la preuve de préservation de l'isolation initiale ont été enrichis tout au long de la thèse, décrivant de manière plus précise les comportements du service ; le chapitre décrit une ébauche détaillée de preuve de préservation de l'isolation sur les derniers modèles sur lesquels la preuve d'isolation n'a pas encore été établie.

De plus, ces travaux viennent contribuer à la complétion de la preuve de préservation de l'isolation sur l'ensemble des services de Pip. Ils corroborent les résultats de la méthodologie de co-design de Pip, visant à fournir les services strictement nécessaires à un système fonctionnel tout en réduisant l'effort de preuve requis. L'utilisabilité du service est montré dans le second chapitre de contribution de ce document : l'ordonnanceur temps-réel utilise le service de transfert de flot d'exécution pour exécuter les *jobs* élus par la fonction d'élection, et est réveillé par les interruptions d'horloge, aussi gérées par ce service.

Le second chapitre de contribution a présenté les travaux qui ont mené l'implémentation d'un ordonnanceur *Earliest Deadline First* pour des séquences arbitraires de *jobs*, dont la fonction d'élection a été prouvée correcte formellement. Plus spécifiquement, il a été montré que l'implémentation exécutable de la fonction d'élection respecte la politique d'ordonnancement *Earliest Deadline First*. L'ordonnanceur présenté s'exécute en espace utilisateur dans une partition de mémoire sur un système s'exécutant avec Pip. Ce chapitre a présenté les étapes de la preuve de correction de la fonction d'élection, puis a présenté l'interface du code vérifié avec le logiciel inclus dans la base de confiance de notre ordonnanceur. Le chapitre propose ensuite une discussion sur cette base de confiance en la comparant à

celle de travaux connexes. À notre connaissance, cet ordonnanceur *Earliest Deadline First* est le premier de la littérature dont l'implémentation est munie d'une preuve formelle de correction de sa fonction d'élection et pouvant ordonnancer une séquence arbitraire de *jobs*.

La preuve de correction de l'ordonnanceur a été menée selon la méthode de co-design usuelle de Pip, mais a cependant été conduite par raffinement, contrairement aux preuves précédentes de Pip qui ont été prouvée par la méthode directe. Ces travaux montrent que cette méthodologie est efficace pour mener des preuves sur d'autres propriétés que les propriétés classiques de Pip et qu'elle s'accomode d'autres méthodes de preuve, comme par exemple le raffinement. Ces travaux montrent aussi qu'il est possible de vérifier formellement du code s'exécutant en espace utilisateur avec la méthodologie de Pip.

Le dernier chapitre de contribution a présenté une preuve de concept séparant les modèles d'isolation de Pip du code des services. Cette séparation permet de définir des modèles alternatifs permettant de raisonner sur d'autres propriétés que les propriétés d'isolation classiques de Pip. Cette séparation a été rendue possible par la définition de *Modules* rendant explicite l'interface du code des services avec le reste de la TCB (telle que les types, l'état, et les primitives d'interaction avec l'état). Ce chapitre a aussi montré que cette séparation ne remet pas en question les fonctionnalités traditionnelles de Pip, telle que l'extraction de l'AST du code permettant de compiler Pip vers du C ou la preuve formelle de propriétés sur le code.

Cette contribution permet d'élargir significativement les perspectives de recherche relatives à Pip, notamment celles concernant la preuve de propriétés alternatives sur le code des services. Le premier chapitre de contribution sur le service de transfert de flot d'exécution avait mentionné que les propriétés d'isolation n'étaient pas particulièrement pertinentes pour ce service. Ces travaux proposent une solution permettant d'établir des propriétés alternatives pertinentes, comme la preuve de bon fonctionnement du service.

6.2 Perspectives de recherche à long terme

6.2.1 Ordonnancement Earliest Deadline First régi par les évènements

Une des pistes de recherche qui pourraient mener à une contribution conséquente serait une amélioration de la contribution présentée en chapitre 4 à propos de l'ordonnanceur. En effet, dans ce chapitre, nous décrivons un ordonnanceur préemptif temps réel muni d'une politique d'ordonnancement *Earliest Deadline First*. La préemption est effectuée à chaque période de temps dont la durée est prédéfinie par le système et mise en application au travers de la programmation de l'horloge du système, qui déclenche une interruption à chaque signal de l'horloge. Cette manière de concevoir un ordonnanceur s'appelle le *quantum-driven scheduling* : chaque *job* est ordonnancé à un moment multiple de la période décrite par l'horloge du système.

Cependant, cette méthode d'ordonnancement n'est pas idéale. En effet, le *job* exécuté

sera interrompu même si aucune nouvelle information n'est disponible pour l'ordonnanceur. Ainsi, dans ce cas, le système interromp le *job* élu à la période précédente, recalcule – avec les mêmes informations – le nouveau *job* à exécuter (qui sera donc le même job qu'à la période précédente), puis renvoie son résultat pour reprendre l'exécution du *job* précédent. Par ailleurs, l'ordonnanceur n'est exécuté qu'en chaque début des périodes décrites par l'horloge du système. Ainsi, si de nouvelles informations arrivent en plein milieu de période, comme par exemple la terminaison du *job* exécuté, alors cette information n'est traitée qu'au début de la prochaine période. Ceci engendre un délai de traitement des informations mises à disposition de l'ordonnanceur, gaspillant de précieux cycles d'exécution pour les *jobs*.

Une amélioration de l'ordonnancement consiste à ne pas interrompre le *job* en train d'être exécuté si aucune nouvelle information n'est disponible pour l'ordonnanceur. En particulier, les interruptions liées à l'ordonnancement ne sont plus levées à chaque période d'horloge, mais sont levées lorsque de nouveaux *jobs* sont prêts à être ordonnancés ou quand le *job* exécuté a terminé son exécution. Cette méthode d'ordonnancement est nommée *event-driven scheduling* et permet d'améliorer les performances de l'ordonnanceur [BA09].

Une telle amélioration est envisageable pour l'ordonnanceur présenté en chapitre 4, mais nécessite une réécriture de l'algorithme de la fonction d'élection ainsi qu'une adaptation de la preuve de correction, en particulier sur les propriétés relatives au temps. Pour ces dernières, le compteur de temps n'augmentait que d'une période à chaque nouvelle exécution de la fonction d'élection, ce qui ne sera plus vrai avec l'*event-driven scheduling*.

De plus, ces travaux présentent une opportunité d'étudier les performances de notre ordonnanceur. Ces résultats, couplés à la preuve formelle de correction de la fonction d'élection, sont susceptibles – selon moi – d'intéresser la communauté temps réel, au prix d'un effort de réécriture et de comparaison des résultats conséquents.

6.2.2 Logique de séparation

Une perspective de recherche qui pourrait porter ses fruits à long terme serait de remplacer la logique de Hoare classique dans les preuves de Pip par la logique de séparation. La logique de séparation est un système formel dérivé de la logique de Hoare permettant de raisonner de manière plus efficace sur les programmes utilisant des structures de données et manipulant des pointeurs [OHe19 ; Rey02]. En particulier, les propriétés définies avec la logique de séparation sont liées à un *store* et une *heap*. Le *store* est représenté par une fonction associant les variables du programme à leur valeur, la *heap* est représentée par une fonction partielle associant des adresses à des valeurs. Ainsi, les propriétés de la logique de séparation se présente de la manière suivante : $s, h \vDash P$ où P représente une propriété quelconque, s représente le *store* lié à cette propriété et où h représente la *heap* associée à cette propriété.

Les propriétés exprimées avec la logique de séparation sont munis des opérateurs clas-

siques de la logique booléenne tels que $\neg, \vee, \wedge, \implies, \exists, \forall$, etc. mais sont aussi munis de nouveaux opérateurs spécifiques à la logique de séparation.

- L'opérateur *emp* permet de définir que la *heap* est vide, comme par exemple dans la propriété $s, h \vDash emp$;
- L'opérateur $\mapsto$ permet de définir une unique association dans la *heap*, comme par exemple dans la propriété $s, h \vDash$ adresse $\mapsto$ valeur ;
- L'opérateur de conjonction de séparation $*$, comme par exemple dans la propriété $s, h \vDash P * Q$, permet de définir que la *heap* peut être séparée en deux parties distinctes telle que l'une vérifie la propriété Q en partie droite de l'opérateur et que l'autre vérifie la propriété P en partie gauche de l'opérateur ;
- L'opérateur d'implication de séparation $-\!\!*$ (aussi appelé opérateur "baguette magique"), comme par exemple dans la propriété suivante $s, h \vDash P -\!\!* Q$ permet de définir que pour toute *heap* h' disjointe de la *heap* originale h, si h' satisfait la propriété P en partie gauche de l'opérateur alors l'union des deux *heaps* $h \bigcup h'$ satisfait la propriété Q en partie droite de l'opérateur.

La logique de séparation est munie des règles d'inférence de la logique de Hoare classique, mais ajoute néanmoins une règle d'inférence qui fait de cette logique un outil très puissant : la règle du cadre (*frame rule dans la littérature*).

$$\{ P \}\, c \,\{ Q \} \vdash \{ P * R \}\, c \,\{ Q * R \}$$

Cette règle permet de raisonner localement sur le code c. La règle peut être décrite de la manière suivante : si le code c peut s'exécuter dans un état restreint satisfaisant les propriétés P pour produire un état satisfaisant les propriétés Q, alors le code c peut s'exécuter dans un état plus grand que P auquel on aurait rajouté une portion d'état arbitraire R, et le code c transforme l'état pour qu'il satisfasse les propriétés Q sans altérer la portion d'état arbitraire R rajoutée.

La logique de séparation me semble propice à la preuve des propriétés d'isolation de Pip. Néanmoins, cette pensée n'est qu'une intuition et l'utilisation de la logique de séparation pour reprouver les propriétés d'isolation ne semble pas particulièrement valorisable. La preuve de propriétés supplémentaires grâce à la logique de séparation présenterait – à mon humble avis – un intérêt pour la communauté scientifique, en particulier grâce à la comparaison de l'effort de preuve relatif à l'usage de ces deux logiques. Cette comparaison apparaitrait significativement biaisée sur la preuve d'isolation, à cause de la preuve déjà établie au travers d'années de travaux. On pourrait argumenter que cette expérience a considérablement facilité l'établissement de la preuve d'isolation au travers de la logique de séparation, et que notre comparaison avec la logique de Hoare n'est pas recevable. C'est pourquoi l'utilisation de la logique de séparation me semble être une perspective de recherche à long terme, où le temps investi dans la compréhension et l'inclusion de cette logique dans Pip ne pourrait montrer ses fruits qu'au bout d'une longue période de recherche, comparée aux perspectives développées dans les chapitres de contribution.

6.3 Retour d'expérience sur la preuve formelle de logiciel

En guise d'épilogue à ce document, je voudrais partager l'évolution de ma vision sur la méthode formelle appliquée à du logiciel au cours de mes travaux de thèse. À mon sens, cette vision a évolué avec ma compréhension du sujet et cette compréhension est celle qui a le plus progressé avec mes travaux, c'est pourquoi je souhaite la partager.

Je pensais initialement que munir du logiciel d'une preuve formelle garantissait qu'il était infaillible ; que cette preuve était semblable aux preuves *mathématiques*, implacables et indéniables, dispensées lors de cours sur les mathématiques que j'ai eu la chance de suivre pendant ma scolarité. Cette vision s'est révélée être partiellement vraie : ces preuves formelles sur du logiciel sont aussi implacables et indéniables que les mathématiques qu'elles emploient – le raisonnement est indiscutablement valide.

La principale réalisation qui m'est venue est que ces preuves formelles ne garantissent pas pour autant l'infaillibilité du logiciel. Elles garantissent son infaillibilité *dans un monde idéalisé* qui n'est cependant pas le monde dans lequel ce logiciel s'exécute. Ce monde idéalisé est créé par les *modèles* sur lesquels reposent les preuves. Ces modèles sont *nécessaires* : sans les modèles, il n'y aurait pas de base au raisonnement et il serait impossible d'établir quoi que ce soit. Ces modèles introduisent néanmoins – en dépit de tout effort et avec une certaine ironie cruelle – l'incertitude dans la méthode d'étude dont nous cherchions à nous débarrasser.

> *« Celui qui s'appuie sur la régression à l'infini est celui dans lequel nous disons que ce qui est fourni en vue d'emporter la conviction sur la chose proposée à l'examen a besoin d'une autre garantie, et cela à l'infini, de sorte que, n'ayant rien à partir de quoi nous pourrons commencer d'établir quelque chose, la suspension de l'assentiment s'ensuit. »* – Sextus Empiricus, Esquisses Pyrrhoniennes

Il n'en reste pas moins que la méthode formelle est celle permettant d'atteindre les plus fortes garanties possibles sur le logiciel vérifié. La méthode formelle permet de comprendre en détail l'objet étudié, bien qu'elle ne permette pas de lever toutes les incertitudes. En ce sens, il n'est pas vain à mes yeux d'utiliser cette méthode.

> *« Nous avons une impuissance à prouver, invincible à tout le dogmatisme. Nous avons une idée de la vérité, invincible à tout le pyrrhonisme. Nous souhaitons la vérité, et ne trouvons en nous qu'incertitude. »* – Blaise Pascal, Pensées

Bibliographie

[Ana+17] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau et Matthew Weaver. **CertiCoq : A verified compiler for Coq**. In : *The third international workshop on Coq for programming languages (CoqPL)*. 2017 (cf. p. 106).

[ASI17] Ittai Anati, Oren Ben Simhon et Intel. *Control Flow Enforcement Technology (CET). Slides of a talk given at CATC 2017*. 2017. url : https://www.intel.com/content/dam/develop/external/us/en/documents/catc17-introduction-intel-cet-844137.pdf (visité le 27/09/2022) (cf. p. 17).

[Aud+90] Neil C Audsley, Alan Burns, MF Richardson et AJ Wellings. **Deadline monotonic scheduling** (1990) (cf. p. 23).

[AY08] Ahmad AlSa'deh et Adnan H Yahya. **Shortest remaining response time scheduling for improved web server performance**. In : *International Conference on Web Information Systems and Technologies*. Springer. 2008, 80-92 (cf. p. 21).

[BA09] Björn B Brandenburg et James H Anderson. **On the implementation of global real-time schedulers**. In : *2009 30th IEEE Real-Time Systems Symposium*. IEEE. 2009, 214-224 (cf. p. 131).

[Bal+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K Rajamani et Abdullah Ustuner. **Thorough static analysis of device drivers**. *ACM SIGOPS Operating Systems Review* 40 :4 (2006), 73-85 (cf. p. 18).

[Bal+18] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu et Irene Finocchi. **A survey of symbolic execution techniques**. *ACM Computing Surveys (CSUR)* 51 :3 (2018), 1-39 (cf. p. 18).

[BDN09] Ana Bove, Peter Dybjer et Ulf Norell. **A brief overview of Agda–a functional language with dependent types**. In : *International Conference on Theorem Proving in Higher Order Logics*. Springer. 2009, 73-78 (cf. p. 25).

[But11] Giorgio C Buttazzo. **Hard real-time computing systems : predictable scheduling algorithms and applications**. T. 24. Springer Science & Business Media, 2011 (cf. p. 23).

[Cao+18] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds et Andrew W Appel. **VST-Floyd : A separation logic tool to verify correctness of C programs**. *Journal of Automated Reasoning* 61 :1 (2018), 367-422 (cf. p. 106).

[Cer22] CertiKOS. *CertiKOS. Certified Kit Operating System*. 2022. url : https://flint.cs.yale.edu/certikos/index.html (visité le 23/09/2022) (cf. p. 32).

[Coq+] Thierry Coquand, Gérard Huet, Christine Paulin et al.. *The Coq Proof Assistant*. url : https://coq.inria.fr/ (visité le 22/10/2022) (cf. p. 25).

[Cur+58] Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley et Jonathan P Seldin. **Combinatory logic**. T. 1. North-Holland Amsterdam, 1958 (cf. p. 25).

[Cur34] Haskell B Curry. **Functionality in combinatory logic**. *Proceedings of the National Academy of Sciences* 20 :11 (1934), 584-590 (cf. p. 25).

[DEB98] Willem-Paul De Roever, Kai Engelhardt et Karl-Heinz Buth. **Data refinement : model-oriented proof methods and their comparison**. 47. Cambridge University Press, 1998 (cf. p. 26).

[DGG22] Nicolas Dejon, Chrystel Gaber et Gilles Grimaud. **From MMU to MPU : adaptation of the Pip kernel to constrained devices**. In : *3rd International Conference on Internet of Things & Embedded Systems (IoTE 2022)*. 2022 (cf. p. 33).

[Dun+15] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen et Enrico Tassi. **ELPI : Fast, Embeddable, λProlog Interpreter**. In : *Logic for Programming, Artificial Intelligence, and Reasoning*. Sous la dir. de Martin Davis, Ansgar Fehnker, Annabelle McIver et Andrei Voronkov. Berlin, Heidelberg : Springer Berlin Heidelberg, 2015, 460-468. isbn : 978-3-662-48899-7 (cf. p. 31).

[EA03] Dawson Engler et Ken Ashcraft. **RacerX : Effective, static detection of race conditions and deadlocks**. *ACM SIGOPS operating systems review* 37 :5 (2003), 237-252 (cf. p. 20).

[Gro18] Samuel saelo Groß. *Exploit for a WebKit JIT optimization bug used during Pwn2Own 2018. CVE-2018-4233*. 2018. url : https://github.com/saelo/cve-2018-4233 (visité le 27/09/2022) (cf. p. 16).

[Gru+16] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp et Stefan Mangard. **Prefetch side-channel attacks : Bypassing SMAP and kernel ASLR**. In : *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 2016, 368-379 (cf. p. 17).

[Gu+11] Liang Gu, Alexander Vaynberg, Bryan Ford, Zhong Shao et David Costanzo. **CertiKOS : A certified kernel for secure cloud computing**. In : *Proceedings of the Second Asia-Pacific Workshop on Systems*. 2011, 1-5 (cf. p. 33).

[Gu+15] Ronghui Gu, Jérémie Koenig, Tahina Ramananandro, Zhong Shao, Xiongnan (Newman) Wu, Shu-Chun Weng, Haozhong Zhang et Yu Guo. **Deep Specifications and Certified Abstraction Layers**. In : *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India : Association for Computing Machinery, 2015, 595-608. isbn : 9781450333009. doi : 10.1145/2676726.2676975. url : https://doi.org/10.1145/2676726.2676975 (cf. p. 33).

[Gu+16] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Newman Wu, Jieung Kim, Vilhelm Sjöberg et David Costanzo. **CertiKOS : An Extensible Architecture for Building Certified Concurrent OS Kernels**. In : *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, 653-669 (cf. p. 33).

[Gu+19] Ronghui Gu, Zhong Shao, Hao Chen, Jieung Kim, Jérémie Koenig, Xiongnan (Newman) Wu, Vilhelm Sjöberg et David Costanzo. **Building Certified Concurrent OS Kernels**. *Commun. ACM* 62 :10 (sept. 2019), 89201399. issn : 0001-0782. doi : 10.1145/3356903. url : https://doi.org/10.1145/3356903 (cf. p. 33).

[HO] Samuel Hym et Veïs Oudjail. *Digger. A tool to convert C-style Gallina into the corresponding C code.* url : https://gitlab.univ-lille.fr/2xs/digger (visité le 21/09/2022) (cf. p. 31).

[How80] William A Howard. **The formulae-as-types notion of construction**. *To HB Curry : essays on combinatory logic, lambda calculus and formalism* 44 (1980), 479-490 (cf. p. 25).

[Hym] Samuel Hym. *∂x. A tool to derive (hence the name) C code from a monadic Gallina code.* url : https://gitlab.univ-lille.fr/samuel.hym/dx (visité le 23/10/2022) (cf. p. 31).

[Int19a] Intel. *Intel IA-32 Architecture Software Developer's Manual Vol. 3A*. Section 5.8.5 - Stack Switching. Intel. 2019 (cf. p. 12).

[Int19b] Intel. *Intel IA-32 Architecture Software Developer's Manual Vol. 3A*. Section 6.11 - IDT Descriptors. Intel. 2019 (cf. p. 12).

[Int19c] Intel. *Intel IA-32 Architecture Software Developer's Manual Vol. 3A*. Section 4.3 - 32-Bit Paging - Figure 4.2. Intel. 2019 (cf. p. 14).

[Int19d] Intel. *Intel IA-32 Architecture Software Developer's Manual Vol. 3A*. Section 6.13 - Figure 6.4. Intel. 2019 (cf. p. 47).

[Jac88] Jeremy Jacob. **Security specifications**. In : *Proceedings. 1988 IEEE Symposium on Security and Privacy*. IEEE Computer Society. 1988, 14-14 (cf. p. 27).

[Jen22] Seth Jenkis. *Bringing back the stack attack. CVE-2022-42703.* 2022. url : https://googleprojectzero.blogspot.com/2022/12/exploiting-CVE-2022-42703-bringing-back-the-stack-attack.html (visité le 11/12/2022) (cf. p. 17).

[Jun+17] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers et Derek Dreyer. **RustBelt : Securing the foundations of the Rust programming language**. *Proceedings of the ACM on Programming Languages* 2 :POPL (2017), 1-34 (cf. p. 18).

[KL88] Judy Kay et Piers Lauder. **A fair share scheduler**. *Communications of the ACM* 31 :1 (1988), 44-55 (cf. p. 21).

[Kle+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish et al. **seL4 : Formal verification of an OS kernel**. In : *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 2009, 207-220 (cf. p. 32).

[Koc+20] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher et al. **Spectre attacks : Exploiting speculative execution**. *Communications of the ACM* 63 :7 (2020), 93-101 (cf. p. 18).

[Ler09] Xavier Leroy. **A formally verified compiler back-end**. *Journal of Automated Reasoning* 43 :4 (2009), 363-446. url : http://xavierleroy.org/publi/compcert-backend.pdf (cf. p. 31).

[Lev84] Henry M. Levy. **Capability-Based Computer Systems**. USA : Butterworth-Heinemann, 1984. isbn : 0932376223 (cf. p. 32).

[Lip+20] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom et al. **Meltdown : Reading kernel memory from user space**. *Communications of the ACM* 63 :6 (2020), 46-56 (cf. p. 18).

[LL73] Chung Laung Liu et James W Layland. **Scheduling algorithms for multiprogramming in a hard-real-time environment**. *Journal of the ACM (JACM)* 20 :1 (1973), 46-61 (cf. p. 23).

[LW82] Joseph Y-T Leung et Jennifer Whitehead. **On the complexity of fixed-priority scheduling of periodic, real-time tasks**. *Performance evaluation* 2 :4 (1982), 237-250 (cf. p. 23).

[Mac71] Saunders Mac Lane. **Categories for the working mathematician**. T. 5. Springer Science & Business Media, 1971 (cf. p. 30).

[Mil19] Bartosz Milewski. **Category theory for programmers**. Bartosz Milewski, 2019. URL : https://github.com/hmemcpy/milewski-ctfp-pdf (cf. p. 109).

[MK14] Nicholas D Matsakis et Felix S Klock. **The rust language**. *ACM SIGAda Ada Letters* 34 :3 (2014), 103-104 (cf. p. 18).

[Mor06] Carroll Morgan. **The shadow knows : Refinement of ignorance in sequential programs**. In : *International Conference on Mathematics of Program Construction*. Springer. 2006, 359-378 (cf. p. 27).

[Mou+15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn et Jakob von Raumer. **The Lean theorem prover (system description)**. In : *International Conference on Automated Deduction*. Springer. 2015, 378-388 (cf. p. 25).

[Moz] The Rust Foundation (formerly Mozilla). *Rust Programming Language. A language empowering everyone to build reliable and efficient software.* URL : https://www.rust-lang.org/ (visité le 15/12/2022) (cf. p. 18).

[Muj21] Alan Mujumdar. *Armv8.1-M Pointer Authentication and Branch Target Identification Extension*. 2021. URL : https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/armv8-1-m-pointer-authentication-and-branch-target-identification-extension (visité le 27/09/2022) (cf. p. 17).

[Mul+18] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock et Dan Grossman. **Oeuf : Minimizing the Coq Extraction TCB**. In : *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*. CPP 2018. Los Angeles, CA, USA : Association for Computing Machinery, 2018, 172-185. ISBN : 9781450355865. DOI : 10.1145/3167089 (cf. p. 105).

[Ngu20] Andy TheFlow Nguyen. *Use-After-Free In* `IPV6_2292PKTOPTIONS` *leading To Arbitrary Kernel R/W Primitives. CVE-2020-7457*. 2020. URL : https://hackerone.com/reports/826026 (visité le 27/09/2022) (cf. p. 19).

[Ngu21] Andy TheFlow Nguyen. *Turning* `\x00\x00` *into 10000$. CVE-2021-22555*. 2021. URL : https://google.github.io/security-research/pocs/linux/cve-2021-22555/writeup.html#bypassing-kaslrsmep (visité le 27/09/2022) (cf. p. 17).

[Ngu22] Andy TheFlow Nguyen. *Use-After-Free In* `IPV6_2292PKTOPTIONS` *leading To Arbitrary Kernel R/W Primitives. CVE-2020-7457*. 2022. URL : https://hackerone.com/reports/1441103 (visité le 27/09/2022) (cf. p. 19).

[NM92] Robert HB Netzer et Barton P Miller. **What are race conditions ? Some issues and formalizations**. *ACM Letters on Programming Languages and Systems (LOPLAS)* 1 :1 (1992), 74-88 (cf. p. 19).

[NW08] Misja Nuyens et Adam Wierman. **The foreground–background queue : a survey**. *Performance evaluation* 65 :3-4 (2008), 286-307 (cf. p. 21).

[NWP02] Tobias Nipkow, Markus Wenzel et Lawrence C Paulson. **Isabelle/HOL : a proof assistant for higher-order logic**. Springer, 2002 (cf. p. 25).

[OCa+17] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll et Nimrod Partush. *Engineering Record And Replay For Deployability : Extended Technical Report*. 2017. doi : 10.48550/ARXIV.1705.05937. url : https://arxiv.org/abs/1705.05937 (cf. p. 19, 20).

[OHe19] Peter O'Hearn. **Separation Logic**. *Commun. ACM* 62 :2 (jan. 2019), 86-95. issn : 0001-0782. doi : 10.1145/3211968. url : https://doi.org/10.1145/3211968 (cf. p. 131).

[PL] CompCert Project et Xavier Leroy. *Context and motivations. Formal Verification of Compilers*. url : https://compcert.org/motivations.html (visité le 16/09/2022) (cf. p. 31).

[plu22] plutoo. *A one-in-a-million bug in Switch kernel*. 2022. url : https://gist.githubusercontent.com/plutooo/2aadbd4a718e269df474079dd2e584fb/raw/7b3af77b5202366c8934c88ef251f1e905967040/gistfile1.txt (visité le 27/09/2022) (cf. p. 19).

[Rei17] Alastair Reid. **Who guards the guards ? Formal validation of the Arm v8-M architecture specification**. *Proceedings of the ACM on Programming Languages* 1 :OOPSLA (2017), 1-24 (cf. p. 104).

[Rey02] John C Reynolds. **Separation logic : A logic for shared mutable data structures**. In : *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE. 2002, 55-74 (cf. p. 20, 131).

[RI] Microsoft Research et Inria. *F* : a Proof-Oriented Programming Language*. url : https://fstar-lang.org/ (visité le 22/10/2022) (cf. p. 25).

[Sam+21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer et Deepak Garg. **RefinedC : Automating the Foundational Verification of C Code with Refined Ownership Types**. In : *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada : Association for Computing Machinery, 2021, 158-174. isbn : 9781450383912. doi : 10.1145/3453483.3454036. url : https://doi.org/10.1145/3453483.3454036 (cf. p. 106).

[Sch+17] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel et Thorsten Holz. **{kAFL} :{Hardware-Assisted} Feedback Fuzzing for {OS} Kernels**. In : *26th USENIX Security Symposium (USENIX Security 17)*. 2017, 167-182 (cf. p. 18).

[seL22a] seL4. *seL4 - Hardware support. seL4 supported platforms and verification status*. 2022. url : https://docs.sel4.systems/Hardware (visité le 23/09/2022) (cf. p. 32).

[seL22b] seL4. *seL4 Project website. The seL4® Microkernel - Security is no excuse for bad performance*. 2022. url : https://sel4.systems/ (visité le 23/09/2022) (cf. p. 32).

[Sew+11] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick et Gerwin Klein. **seL4 enforces integrity**. In : *International Conference on Interactive Theorem Proving*. Springer. 2011, 325-340 (cf. p. 32).

[SKH17] Thomas Sewell, Felix Kam et Gernot Heiser. **High-assurance timing analysis for a high-assurance real-time operating system**. *Real-Time Systems* 53 :5 (2017), 812-853 (cf. p. 32).

[SMK13] Thomas Arthur Leck Sewell, Magnus O Myreen et Gerwin Klein. **Translation validation for a verified OS kernel**. In : *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 2013, 471-482 (cf. p. 32).

[Soz+20] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau et Théo Winterhalter. **The metacoq project**. *Journal of automated reasoning* 64 :5 (2020), 947-999 (cf. p. 104).

[Sta+12] John A Stankovic, Marco Spuri, Krithi Ramamritham et Giorgio C Buttazzo. **Deadline scheduling for real-time systems : EDF and related algorithms**. T. 460. Springer Science & Business Media, 2012 (cf. p. 88).

[Tic+14] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano et Geoff Pike. **Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM**. In : *23rd USENIX security symposium (USENIX security 14)*. 2014, 941-955 (cf. p. 17).

[Wan+17] Yu Wang, Linzhang Wang, Tingting Yu, Jianhua Zhao et Xuandong Li. **Automatic detection and validation of race conditions in interrupt-driven embedded software**. In : *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2017, 113-124 (cf. p. 20).

[YDS95] Frances Yao, Alan Demers et Scott Shenker. **A scheduling model for reduced CPU energy**. In : *Proceedings of IEEE 36th annual foundations of computer science*. IEEE. 1995, 374-382 (cf. p. 21).

[Yua+22] Shenghao Yuan, Frédéric Besson, Jean-Pierre Talpin, Samuel Hym, Koen Zandberg et Emmanuel Baccelli. **End-to-end mechanized proof of an ebpf virtual machine for micro-controllers**. In : *International Conference on Computer Aided Verification*. Springer. 2022, 293-316 (cf. p. 31).

Deuxième partie

Annexes

A Annexes de la première contribution

A.1 Implémentation de l'appel au service de Pip au sein de la LibPip, sa librairie utilisateur

```
uint32_t yield(uint32_t a, uint32_t b, uint32_t c,
               uint32_t d, uint32_t e)
{
    register uint32_t r asm("eax");

    asm volatile (
        "push  %5;"
        "push  %4;"
        "push  %3;"
        "push  %2;"
        "push  %1;"
        "lcall $0×c0,$0×0;"
        "add   $0×14,%%esp;"
        : "=r" (r)
        :  "r" (a), "r" (b), "r" (c), "r" (d), "r" (e)
        :
    );

    return r;
}
```

Listing A.1 – Implémentation de l'appel vers le service en espace utilisateur

A.2 Implémentation de la routine de sauvegarde du contexte et d'harmonisation de la pile

```
cg_yieldGlue:
    ; interrupts are not cleared upon call gate entry.
    ; this might create a situation where an interrupt
    ; occurs in kernelland
    ;
    ; Fortunately, this situation can be avoided on return
    ; (even if we use retf which does not restore eflags)
    ; since sti is only effective upon execution of the next
    ; instruction.
    ;
    ; The reason why we are not using a retf is to unify the
    ; different ways of calling a control flow transfer
    cli

    ; pop eip in eax
    pop eax
    ; pop cs in edx
    pop edx

    ; first, copy the arguments higher on the stack

    ; set esp where the args should be copied in order to save esi, edi
    ; we need 11 dwords free (eflags + cs + eip + pusha 8 dwords )
    ; stack top is currently at ss + esp + 5
    ; so we set esp to esp + 11 * 4
    sub esp, 11 * 4

    ; we are going to modify esi, edi and eflags
    ; those are not scratch registers so we need to
    ; save them first
    push esi
    push edi
    pushfd

    ; clear direction flag so esi and edi are incremented with movsd
    cld
    ; set destination before our pushes on the stack
```

```
lea edi, [esp + 3 * 4]
; set source 11 dwords higher
lea esi, [edi + 11 * 4]
; repeat for 5 args
mov ecx, 5
; copy
rep movsd

; restore previously saved registers
popfd
pop edi
pop esi

; go down the stack to replace the args we copied
; hopefully it doesn't mess up eflags
add esp, (11 + 5) * 4
; push EFLAGS, replacing the first argument
pushf
; push cs
push edx
; push eip
push eax
; clean cloberred registers
; in case we trigger a context change
xor eax, eax
xor ecx, ecx
xor edx, edx
; push general purpose registers (8 dwords)
pushad

; save the context pointer into EAX
mov eax, esp

; go back to the stack top
sub esp,  5 * 4

; push the context pointer
push eax

; enforce interrupts if needed
call fix_eflags_gate_ctx
```

```
; call C handler (gate_ctx_t *, arg1, ... , arg%2)
call yieldGlue

; skip pointer to the context and args
; and jump to the general purpose registers to save
; called function return values
add esp, (5 + 1 + 8) * 4
; we save the return values on top of the previous values
; explicitly push EAX ECX EDX (as per linux calling conventions)
push eax
push ecx
push edx
; jump to the top of the general purpose registers
; (skip EBX ESP EBP ESI EDI)
sub esp, 5 * 4
; restore general purpose registers
popad
; we are left with the iretable structure
iret
```

Fragment de code A.1 – Implémentation de la routine de sauvegarde du contexte et d'harmonisation de la pile

A.3 Création du contexte générique et appel vers le code prouvé

```
yield_checks yieldGlue(gate_ctx_t *gate_ctx,
                       vaddr calleePartDescVAddr,
                       userValue userTargetInterrupt,
                       userValue userCallerContextSaveIndex,
                       interruptMask flagsOnYield,
                       interruptMask flagsOnWake)
{
    user_ctx_t user_ctx;
    user_ctx.regs = gate_ctx→regs;
    user_ctx.regs.esp = gate_ctx→useresp;
    user_ctx.eip = gate_ctx→eip;
    user_ctx.eflags = gate_ctx→eflags;
    user_ctx.valid = 1;

    return checkIntLevelCont(calleePartDescVAddr, userTargetInterrupt,
                             userCallerContextSaveIndex, flagsOnYield,
                             flagsOnWake, &user_ctx);
}
```

Fragment de code A.2 – Création du contexte générique et appel vers le code prouvé

A.4 Fonction de récupération des arguments après une faute

```
void faultInterruptHandler(int_ctx_t *ctx)
{
    DEBUG(TRACE, "Received fault int n°%d\n", ctx->int_no);
    DEBUG(TRACE, "Error code is : %x\n", ctx->err_code);

    user_ctx_t uctx;
    uctx.eip = ctx->eip;
    uctx.regs = ctx->regs;
    uctx.eflags = ctx->eflags;
    uctx.regs.esp = ctx->useresp;
    uctx.valid = 1;

    // TODO The below code could be written in Coq

    page currentPartDesc = getCurPartition();

    interruptMask currentPartitionIntState = get_self_int_state();
    unsigned saveIndex;
    if (currentPartitionIntState == 0) {
        saveIndex = CLI_SAVE_INDEX;
    }
    else {
        saveIndex = STI_SAVE_INDEX;
    }
    page currentPageDir  = getPd(currentPartDesc);
    propagateFault(currentPartDesc, currentPageDir, ctx->int_no,
                   saveIndex, getNbLevel(), currentPartitionIntState,
                   currentPartitionIntState, &uctx);
}

void propagateFault(page callerPartDesc,
                    page callerPageDir,
                    unsigned targetInterrupt,
                    unsigned callerContextSaveIndex,
                    unsigned nbL,
                    interruptMask flagsOnYield,
                    interruptMask flagsOnWake,
                    user_ctx_t *callerInterruptedContext)
```

```
{
    yield_checks rc = getParentPartDescCont(callerPartDesc,
                                            callerPageDir,
                                            targetInterrupt,
                                            callerContextSaveIndex,
                                            nbL,
                                            flagsOnYield,
                                            flagsOnWake,
                                            callerInterruptedContext);
    switch(rc) {
        case coq_FAIL_UNAVAILABLE_CALLER_VIDT:
        case coq_FAIL_CALLER_CONTEXT_SAVE:
            getTargetVidtCont(getParent(callerPartDesc),
                              callerPageDir,
                              getVaddrVIDT(),
                              0,
                              targetInterrupt,
                              nbL,
                              getIndexOfAddr(getVaddrVIDT(), levelMin),
                              flagsOnYield,
                              flagsOnWake,
                              0);
            break;
        case coq_FAIL_ROOT_CALLER:
            // Fault inside the root partition
            // Guru meditation
            for(;;);
            break;
        default:
            // Be sure to handle the root case differently,
            // as it has no parent
            page parentPartDesc = getParent(callerPartDesc);
            // We are still trying to save the faulting
            // partition's context, even though it is very
            // unlikely the partition will ever wake up again
            // TODO is it worth a try ?
            propagateFault(parentPartDesc,
                           callerPageDir,
                           DOUBLE_FAULT_LEVEL,
                           callerContextSaveIndex,
                           nbL,
```

```
                                flagsOnYield,
                                flagsOnWake,
                                callerInterruptedContext);
                    break;
            }
        }
```

Fragment de code A.3 – Fonction de récupération des arguments après une faute

B Annexes de la dernière contribution

B.1 Implémentation du modèle d'isolation

```
From Pip.Model.Meta Require Import StateModel.
From Pip.Model.Isolation Require Import IsolationTypes.

Module IsolationState <: StateModel.
Export IsolationTypes.

  Record Pentry : Type:=
  {
    read    : bool;
    write   : bool;
    exec    : bool;
    present : bool;
    user    : bool;
    pa      : page
  }.

  Record Ventry : Type:=
  {
    pd : bool;
    va : vaddr
  }.

  Inductive value : Type :=
  |PE : Pentry → value
  |VE : Ventry → value
  |PP : page → value
  |VA : vaddr → value
  |I  : index → value
  |U  : userValue → value.

  Record IsolationState : Type := {
    currentPartition : page;
```

```
    memory : list (paddr * value)
  }.

  Definition state := IsolationState.

End IsolationState.
```

Fragment de code B.1 – Définition du modèle d'isolation de l'état selon la nouvelle architecture

```
Require Import String.
From Pip.Model.Meta Require Import StateModel.

Module StateAgnosticMonad (Export State : StateModel).

  Inductive result (A : Type) : Type :=
  | val : A → result A
  | undef : string → state → result A.

  Definition SAM (A : Type) : Type :=
    state → result (A * state).

  Definition ret {A : Type} (a : A) : SAM A :=
    fun s ⇒ val (a , s).

  Definition bind {A B : Type} (m : SAM A)
                               (f : A → SAM B)
                               : SAM B :=
    fun s ⇒ match m s with
    | val (a, s') ⇒ f a s'
    | undef msg s' ⇒ undef msg s'
    end.

  Notation "'perform' x ':=' m 'in' e" := (bind m (fun x ⇒ e))
  (* [ ... ] *)
  Notation "m1 ;; m2" := (bind m1 (fun _ ⇒ m2))
  (* [ ... ] *)

  Definition undefined {A : Type} (msg : string) : SAM A :=
    fun s ⇒ undef msg s.
```

```
  Definition hoareTriple {A : Type} (P : state → Prop)
                                    (m : SAM A)
                                    (Q : A → state → Prop)
                                    : Prop :=
    forall s, P s → match m s with
    | val (a, s') ⇒ Q a s'
    | undef _ _⇒ False
    end.

  Notation "{{ P }} m {{ Q }}" := (hoareTriple P m Q)
  (* [ ... ] *)

End StateAgnosticMonad.
```

Fragment de code B.2 – Définition de la monade d'état

```
Require Import List Arith.
Require Import Coq.Program.Tactics.

From Pip.Model.Meta Require Import TypesModel.

Module IsolationTypes <: TypesModel.

  Record page_s := {
    p :> nat;
    Hp : p < nbPage
  }.

  Definition contextAddr := nat.

  Record interruptMask_s := {
    m :> list bool;
    Hm : length m = maxVint+1;
  }.

  Definition interruptMask := interruptMask_s.

  Inductive yield_checks : Type :=
  (* [ ... ] *)
  | SUCCESS.
```

```
End IsolationTypes.
```

Fragment de code B.3 – Définition du modèle d'isolation des types selon la nouvelle architecture

```
From Pip.Model.Meta Require Import TypesModel StateModel StateAgnosticMonad Interfa
From Pip.Model.Isolation Require Import IsolationTypes IsolationState.

Require Import Coq.Strings.String Lia NPeano.
Require Import Arith Bool List.
Import List.ListNotations.

Module IsolationInterface <: InterfaceModel IsolationTypes IsolationState.

  Module SAMM := StateAgnosticMonad IsolationState.
  Export SAMM.

  Definition setInterruptMask (mask : interruptMask) : SAM unit :=
    ret tt.

  Definition updateMMURoot(pageDir : page) : SAM unit :=
    ret tt.

  Definition updateCurPartition (p : page) : SAM unit :=
    modify (fun s ⇒ {| currentPartition := p; memory := s.(memory) |}).

  Definition getInterruptMaskFromCtx (context : contextAddr) : SAM interruptMask :=
  ret int_mask_d.

  Definition getPageRootPartition : LLI page :=
    ret pageRootPartition.

  Definition page_eqb (p1 : page) (p2 : page) : SAM bool :=
    ret (p1 =? p2).

  Definition noInterruptRequest (flags : interruptMask) : SAM bool :=
    ret true.

  Definition loadContext (contextToLoad : contextAddr)
```

```
                          (enforce_interrupt : bool) : SAM unit :=
    ret tt.

End IsolationInterface.
```

Fragment de code B.4 – Définition des modèles d'isolation des fonctions selon la nouvelle architecture

```
From Pip.Model.Isolation Require Import IsolationTypes IsolationState
  IsolationInterface.
From Pip.Core Require Import ModelAgnosticCode.

From Pip.Proof.Isolation Require Import ModelExclusiveFunctions
  IsolationProperties ConsistencyProperties.

Import IsolationTypes.
Import IsolationInterface.SAMM.

Module IsolationSpecificCode :=
  ModelAgnosticCode IsolationTypes IsolationState IsolationInterface.
Import IsolationSpecificCode.

Lemma switchContextCont (targetPartDesc : page)
                        (targetPageDir  : page)
                        (flagsOnYield   : interruptMask)
                        (targetContext  : contextAddr) :

{{ (* Preconditions *)
  fun s ⇒
  partitionsIsolation s ∧
  kernelDataIsolation s ∧
  verticalSharing s ∧
  consistency s ∧
  List.In targetPartDesc (getPartitions pageRootPartition s) ∧
  targetPartDesc <> pageDefault
}}

switchContextCont targetPartDesc targetPageDir flagsOnYield targetContext

{{ (* Postconditions *)
```

```
  fun _ s  ⇒
  partitionsIsolation s ∧
  kernelDataIsolation s ∧
  verticalSharing s ∧
  consistency s
}}.
Admitted.
```

Fragment de code B.5 – Définition du triplet de Hoare de la fonction `switchContextCont` instanciant le code générique avec les modèles d'isolation