Plates-formes et mises à jour dynamiques configurables

Sébastien Martinez

UNIVERSITE
BRETAGNE
LOIRE

THÈSE / Télécom Bretagne
sous le sceau de l'Université Bretagne Loire
pour obtenir le grade de Docteur de Télécom Bretagne
En accréditation conjointe avec l'Ecole Doctorale Matisse
Mention : Informatique

présentée par
Sébastien Martinez
préparée dans le département Informatique
Laboratoire Irisa

Plates-formes et mises à jour dynamiques configurables

Thèse soutenue le 14 mars 2016
Devant le jury composé de :

Erven Rohou
Directeur de Recherche, Inria – Rennes / président

Fabienne Boyer
Maître de conférences (HDR), Université de Grenoble / rapporteur

Lionel Seinturier
Professeur, Université de Lille 1 / rapporteur

Sara Bouchenak
Professeur, Insa - Lyon / examinateur

Fabien Dagnat
Maître de conférences, Télécom Bretagne / Directeur de thèse

Jérémy Buisson
Maître de conférences, Écoles de Saint-Cyr Coëtquidan / co-encadrant

N° d'ordre : 2016telb0395

Sous le sceau de l'Université Bretagne Loire

Télécom Bretagne

En accréditation conjointe avec l'Ecole Doctorale Matisse

Ecole Doctorale - MATISSE

Plates-formes et mises à jour dynamiques configurables

Thèse de Doctorat

Mention : Informatique

Présentée par **Sébastien Martinez**

Département : Informatique

Laboratoire : IRISA

Directeur de thèse : Fabien Dagnat

Jury :

Mme Fabienne Boyer, Maître de conférences, Université de Grenoble (Rapporteur)
M. Lionel Seinturier, Professeur, Université de Lille 1 (Rapporteur)
Mme Sara Bouchenak, Professeur, Insa Lyon (Examinateur)
M. Erven Rohou, Directeur de Recherche, INRIA (Examinateur)
M. Fabien Dagnat, Maître de conférences, Télécom Bretagne (Directeur de thèse)
M. Jérémy Buisson, Maître de conférences, Écoles de Saint-Cyr Coëtquidan (co-encadrant)

Plates-formes et mises à jour dynamiques configurables

Sébastien Martinez

Thèse de doctorat

Travaux effectués sous la direction de Fabien Dagnat et le co-encadrement de Jérémy Buisson

«Paradox is a pointer telling you to look beyond it. If paradoxes bother you, that betrays your deep desire for absolutes. The relativist treats a paradox merely as interesting, perhaps amusing or even, dreadful thought, educational.»

– Frank Herbert, God Emperor of Dune

Remerciements

Je remercie Fabien Dagnat et Jérémy Buisson, mes encadrants, grâce à qui j'ai vécu trois années passionnantes, découvert un domaine à la fois stimulant intellectuellement et original. Leurs conseils et leur soutien m'ont permis d'arriver au bout de cette aventure et d'apprendre beaucoup, tant sur le plan scientifique que humain.

Je souhaite remercier les élèves qui, au travers de projets étudiants et de stages, ont aidé mes recherches en explorant de nouvelles idées : Hugo Hervoche, Jérémy Cloarec, Anthony Abou Naoum, Nicolas Szlifierski et Rachid Ayoubi. Je tiens particulièrement à remercier Vincent Deprez, Arnaud Saric et Dorian Gilly pour leur aide sur Cmoult. Merci également à Alexandre Manoury pour sa participation aux dernières versions de Pymoult.

Merci également aux membres du département informatique de Télécom Bretgane avec qui il est toujours un plaisir d'échanger connaissances, expériences, et bonne humeur. Merci également à Serge Guelton qui m'a donné le goût de la recherche et à Siegfried Rouvrais qui m'a initié à l'enseignement.

Je tiens également à remercier ma famille et mes amis pour leur soutient et leurs encouragements. Mes parents, toujours à mes côtés, en m'encourageant à progresser et en stimulant ma curiosité, m'ont permis d'atteindre ce niveau et d'écrire ces lignes. Pour tout cela, et bien plus, je les remercie.

Et merci à la région Bretagne qui m'a financé pendant ces trois années dans le cadre du projet IMAJD.

Table des matières

Chapitre 1

Introduction

Mettre à jour un logiciel nécessite usuellement son redémarrage : il faut sauvegarder son état, l'arrêter, modifier son code, le démarrer à nouveau et enfin, recharger son état. Si cette tâche s'avère déjà ennuyeuse dans le cas d'un logiciel de bureautique ou d'un système d'exploitation personnel, elle devient coûteuse ou impossible dans certains domaines particuliers. En effet, mettre à jour un service informatique de manière traditionnelle le rend indisponible, générant des manques à gagner pour les services commerciaux ou causant des problèmes de sécurité pour des services critiques (par exemple, les systèmes d'une tour de contrôle aérien). Différentes solutions sont adoptées pour palier à cette indisponibilité. L'une d'entre elles consistant en la distribution des services sur de nombreux serveurs dont les logiciels sont mis à jour à tour de rôle, assurant ainsi un service minimum en permanence. Ces solutions viennent avec leurs propres avantages et inconvénients qui ne seront pas explicités ici. Il est seulement à noter qu'elles ne s'adressent qu'à un problème : maintenir la disponibilité de services lors de leurs redémarrages.

Le domaine de la mise à jour dynamique (DSU pour *Dynamic Software Updating*) s'adresse, à un autre problème : mettre à jour les logiciels sans que leur redémarrage ne soit nécessaire. Dans cette optique, de nombreuses plates-formes ont été développées. Ces plates-formes sont combinées avec les programmes, qui constituent les logiciels, avant leur démarrage. Chaque fois qu'une mise à jour dynamique est invoquée, la plate-forme utilise des mécanismes pour appliquer les modifications au programme. Ce dernier continue son exécution, sans interruption de service, et paraît n'avoir jamais été mis à jour : il semblera, pour un observateur extérieur, que le programme a été lancé dans sa version la plus récente depuis le départ.

D'une manière générale, il est voulu que la plate-forme soit la plus transparente possible. C'est-à-dire qu'à chaque mise à jour, les développeurs doivent fournir le moins d'informations possible. Dans ce but, seul le code de la nouvelle version du programme, parfois associé à quelques instructions supplémentaires, est demandé aux développeurs et aux personnes appliquant la mise à jour. Les mécanismes employés pour appliquer la mise à jour sont toujours les mêmes durant toute la vie du programme. En conséquence, les capacités des plates-formes en terme d'application de mises à jour sont limitées. En effet, chaque mécanisme de mise à jour a ses avantages et inconvénients et peut, en conséquence, ne pas être compatible avec l'application de certaines modifications. En fixant une combinaison de mécanismes employée pour chaque mise à jour, les plates-formes se spécialisent quant au type de modifications qu'elles peuvent supporter. Ainsi, la majorité des plates-formes se spécialise dans l'application de correctifs de bugs ou de failles de sécurité qui ne changent pas la sémantique du programme.

Par exemple, la plate-forme OPUS [2] permet de mettre à jour des programmes C dynamiquement à la condition que les modifications se limitent aux fonctions et n'en changent pas la sémantique. C'est-à-dire que les mises à jour ne peuvent ni changer la signature des fonctions, ni changer leurs effets de bord (les opérations d'écriture sur la mémoire partagée avec le reste du programme ne doivent pas changer lors d'une mise à jour). De telles restrictions proviennent du choix des mécanismes employés, guidé par l'objectif de la plate-forme : permettre d'appliquer des mises à jour corrigeant des bugs ou des failles de sécurité. En effet, il est peu fréquent qu'une mise à jour corrective change la sémantique des fonctions d'un programme ou qu'elle affecte les données traitées par le programme (types, variables).

Si cette approche des mises à jour dynamiques change progressivement pour chercher à supporter le plus de modifications possible, la politique habituelle consistant à utiliser une même combinaison de mécanismes pour appliquer toutes les mises à jour est toujours la règle. Par exemple, la plate-forme Kitsune [1] [44], par sa combinaison de mécanismes peut appliquer dynamiquement davantage de modifications. Utiliser Kitsune nécessite de modifier le code du programme avant de le lancer. Il est demandé

1. Dont une description plus détaillée est donnée chapitre 4, section 4.4.

de placer des *points de mise à jour* aux endroits du code où le programme est *stable*. Pour un programme de type serveur (le cœur de cible de Kitsune), composé d'une boucle principale traitant des requêtes au fil de leurs arrivées, ces points peuvent être placés au début et à la fin de cette boucle. Il est également demandé de modifier le code du programme en plaçant des instructions de branchement permettant de sauter les phases d'initialisation du programme. Une fois le programme lancé, les mises à jour sont appliquées dynamiquement lorsque l'exécution du programme atteint un point de mise à jour. Le nouveau code est alors chargé, les données sont mises à jour et le programme est redémarré en conservant ses données. Les instructions de branchement permettent alors de sauter les phases d'initialisation et d'atteindre rapidement le point qui a déclenché la mise à jour. En réduisant la transparence pour les utilisateurs de la plate-forme (développeurs, applicateurs de mise à jour), Kitsune peut appliquer dynamiquement un plus grand nombre de modifications. Certaines restrictions persistent toutefois : par exemple, la structure du code du programme peut rendre le placement des points de mise à jour difficile. De plus, la moindre modification nécessite le redémarrage de tout le programme.

Utiliser la même combinaison de mécanismes pour chaque mise à jour, est restrictive. Elle peut être inadaptée à une modification donnée, voir être incompatible. Le choix de mécanismes peut également imposer des restrictions sur les programmes à mettre à jour. S'il est possible de choisir une plate-forme de mise à jour dynamique offrant la meilleure compatibilité avec un programme donné, il est difficile de prévoir les modifications qui seront apportées par les futures mises à jour. Au mieux, il est possible de choisir une plate-forme dont les mécanismes offrent la meilleure compatibilité avec le programme ainsi qu'avec la majorité des mises à jour prévisibles (c'est-à-dire les mises à jour qu'il est possible d'anticiper au moment de lancer le programme).

De plus, il est impossible de combiner les mécanismes de plusieurs plates-formes à la fois. Il n'est, par exemple, pas possible de combiner OPUS et Kitsune pour appliquer des modifications mineures selon la méthode de OPUS et des mises à jour importantes selon la méthode de Kitsune. Quelque soit la combinaison de mécanismes choisie, c'est cette combinaison qui sera utilisée à chaque mise à jour. Pour pallier à cela, certaines plates-formes embarquent plus de mécanismes que nécessaire et appliquent chaque mise à jour au moyen d'un sous-ensemble de ces mécanismes, utilisant les plus appropriés à chaque fois. Ainsi, la plate-forme pour Java, Jvolve [72] utilise deux mécanismes différents pour mettre à jour les méthodes des classes en fonction des modifications à apporter : la méthode est intégralement redéfinie si son code source est changé, son *bytecode* est recompilé si seulement son code binaire est changé (par exemple, si des références statiques sont changées par effets de bord d'une mise à jour). Cette approche tend à réduire les restrictions imposées par le choix des mécanismes qu'embarque une plate-forme. La plate-forme Rubah [63] permet, pour chaque mise à jour, de choisir quelle stratégie adopter pour mettre à jour les données du programme entre stratégie *pressée* (toutes les données sont mises à jour immédiatement) ou *paresseuse* (les données sont mises à jour au moment où elles sont utilisées dans le programme).

Choisir quels mécanismes employer à chaque mise à jour en fonction des modifications qu'elle apporte permet d'assurer une compatibilité de la plate-forme avec un grand nombre de mises à jour. Il devient possible de choisir les mécanismes optimaux pour appliquer une mise à jour donnée tout en assurant une compatibilité avec le programme à modifier. **Ainsi, pour offrir une meilleure couverture des modifications possibles, le développement des mises à jour dynamiques doit suivre un principe similaire au développement d'un logiciel original. Pour chaque mise à jour, une phase de conception doit s'assurer du respect des contraintes du logiciel modifié et des besoins de la mise à jour, en choisissant les mécanismes les plus adaptés. L'écriture de son code suit alors ces choix de conception.** Un nouveau rôle intervient alors dans le développement des programmes : le *développeur de mises à jour* dont la tâche est d'identifier les mécanismes à employer pour chaque mise à jour. Les plates-formes de mise à jour ont alors pour but, non plus d'appliquer des mises à jour dynamiquement de manière transparente, mais de fournir les mécanismes et les outils nécessaires aux développeurs de mises à jour pour concevoir chaque mise à jour dynamique.

Cette nouvelle approche du domaine de la mise à jour dynamique est la thèse défendue par le présent manuscrit dont les prochains chapitres répondent aux problématiques qu'elle pose et démontrent sa faisabilité, tout en présentant les intérêts. Laisser aux développeurs de mises à jour le soin de concevoir chaque mise à jour en sélectionnant les mécanismes à employer nécessite d'établir des modèles utilisables dans cette tâche. Il devient important d'identifier les principaux composants d'une plate-forme de mise à jour dynamique, d'étudier les mécanismes de l'état de l'art pour identifier leurs apports ainsi que leurs restrictions, et de proposer des méthodes de définition d'une mise à jour dynamique. Répondre à ces questions permet d'adopter un nouveau regard sur les mécanismes de mises à jour dynamique, les voyant comme les éléments de base combinables et configurables des mises à jour.

Dans un premier temps, une analyse des plates-formes du domaine permet d'effectuer un inventaire des techniques permettant de mettre à jour les programmes, d'identifier les synergies que ces techniques peuvent avoir et de regrouper les plates-formes par similarité. Cette analyse conduit à l'établissement de modèles pour la conception de plates-formes suivant la nouvelle approche défendue dans ce manuscrit. Ces modèles s'appuyant sur l'idée d'une conception centrée autour des mécanismes qu'une plate-forme doit fournir, une étude des mécanismes permet d'identifier leurs besoins, ainsi que leur capacité à être combinés avec d'autres mécanismes. Ces résultats, combinés aux modèles précédemment définis permettent alors de concevoir et développer des plates-formes, conformes à l'approche défendue dans ce manuscrit. Ces plates-formes configurables permettent aux développeurs des mises à jour de choisir quels mécanismes doivent être utilisés à chaque fois.

Les trois principales contributions présentées ici sont détaillées dans trois parties distinctes de ce manuscrit. La première partie page 15 décrit comment trois tables d'analyse des plates-formes de l'état de l'art ont été établies, en présente des analyses statistiques ainsi que les interprétations de ces statistiques. Cette partie présente également des modèles génériques permettant la comparaison des différentes plates-formes du domaine. La deuxième partie page 57 présente la sémantique opérationnelle de λ_{DSU}, une version du λ-calcul permettant aux programmes d'être mis à jour dynamiquement. Cette sémantique est utilisée pour analyser les mécanismes courants. La troisième partie page 77 présente deux plates-formes suivant la nouvelle approche : Pymoult pour le langage Python et Cmoult pour le langage C. Leur conception est détaillée et des exemples d'utilisation de Pymoult sont également donnés.

La première partie effectue une analyse empirique de l'état de l'art et étudie la constitution des plates-formes, ce qui permet également d'identifier les mécanismes les plus fréquents. La deuxième partie utilise le formalisme d'une sémantique opérationnelle pour étudier les exigences des mécanismes de mise à jour. La troisième partie combine les résultats des deux précédentes parties et s'appuie sur le développement de Pymoult et Cmoult pour discuter l'implémentation de plates-formes configurables.

L'approche des travaux décrits dans ce manuscrit a été pragmatique et orientée vers le développement de solutions logicielles. Les principales validations de la thèse défendue ici sont par conséquent les plates-formes Pymoult et Cmoult dont le développement a autant été alimenté par les résultats des deux premières parties qu'il a lui même participé à l'orientation des travaux présentés. Pymoult fait l'objet de deux publications [53, 54] qui en décrivent l'implémentation et d'une présentation à la conférence PyConFR 2015 qui réunit les utilisateurs francophones du langage Python. Pymoult est également utilisé par Pycots, une plate-forme de reconfiguration pour programmes orientés composant qui fait également l'objet de deux publications [18, 21]. Le développement de Pymoult et Cmoult suit une approche logicielle libre : depuis le début du développement, le code source ainsi qu'une documentation sont disponibles sur un dépôt publique [11]. Des tutoriaux ainsi que des exemples démonstratifs sont également disponibles sur ce dépôt.

L'état de l'art répertorie de nombreuses plates-formes de mise à jour dynamique et toutes les détailler ou toutes les mentionner serait trop long pour le présent manuscrit. Seul un sous-ensemble de ces plates-formes choisies est utilisé dans les prochains chapitres. Les propos qu'ils contiennent restent cependant généraux et s'adressent à l'ensemble des plates-formes de l'état de l'art. Les plates-formes décrites ont été choisies pour des raisons didactiques ou parce qu'elles sont représentatives d'une catégorie de plates-formes sur laquelle porte le discours.

1.1 Conventions de nommage et modèles

Le présent manuscrit, utilise certains termes pour identifier les éléments en jeu lors d'une mise à jour dynamique et présentera les logiciels selon un modèle présenté dans cette section. Chaque fois que ces usages ou que ce modèle sera invalidé, comme dans la partie III, cela sera précisé.

Dans les prochains chapitres, un logiciel est un ensemble de programmes coopérant dans un objectif commun. Une plate-forme se combine à un programme et il est imaginable d'utiliser plusieurs plates-formes différentes pour mettre à jour les différents programmes d'un même logiciel. Dans la suite du manuscrit, seule la mise à jour d'un unique programme est considérée, qu'il s'agisse de mettre à jour l'unique programme d'un logiciel ou de mettre à jour partiellement un plus gros logiciel en modifiant un de ses programmes. S'il est possible de voir la mise à jour d'un logiciel complet comme plusieurs mises à jour simultanées des différents programmes qui le composent pour des cas simples, des cas complexes peuvent invalider cette généralisation. Toutefois, les logiciels constitués d'un unique programme sont suffisamment nombreux pour que la contribution faite dans ce manuscrit soit intéressante.

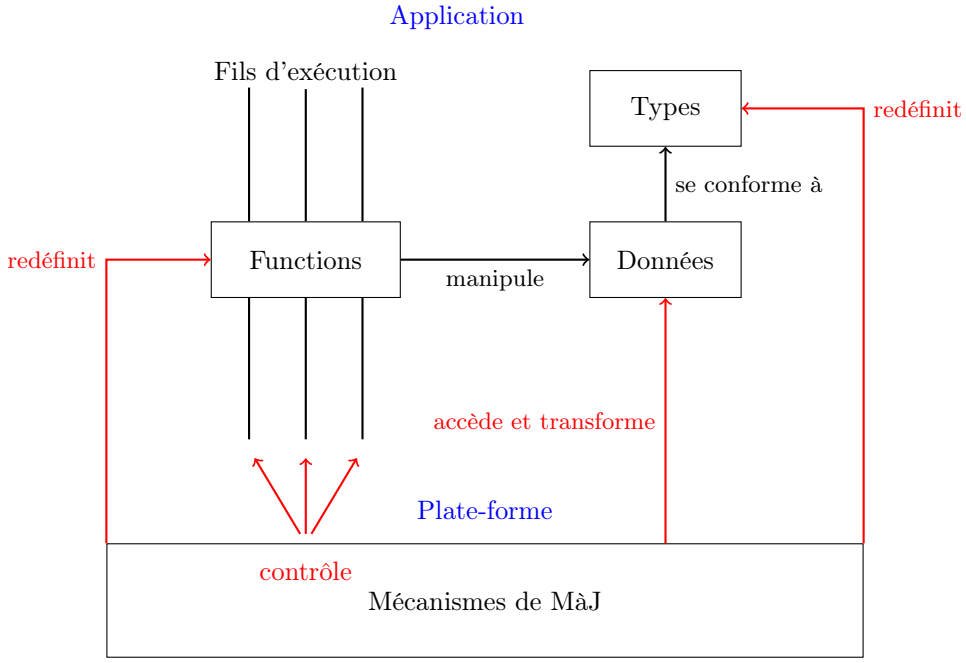


FIGURE 1.1 – Carte d'un programme

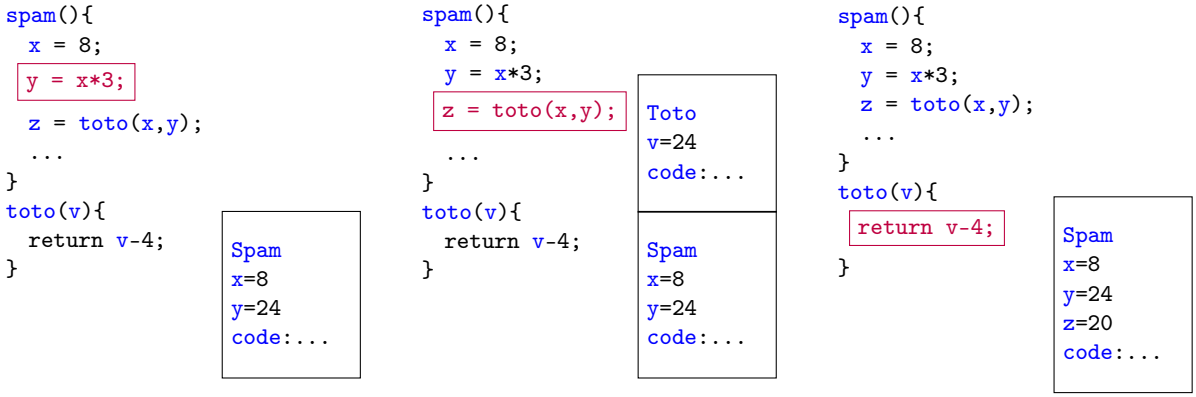


FIGURE 1.2 – Exécution d'un fil d'exécution

1.1.1 Structure d'un programme

Afin de pouvoir parler des programmes en général, la modélisation suivante (représentée sur la figure 1.1) est adoptée dans ce manuscrit : chaque programme est composé d'un ou plusieurs fils d'exécution qui exécutent des fonctions et utilisent des données. Ces données peuvent ensuite se diviser en deux catégories : les données liées à l'exécution du programme (pointeur d'instruction courante, pile d'exécution, ...) et les données liées à la *logique* du programme (variables, objets, ...). Les données de la seconde catégorie sont souvent caractérisées par des types définis par le programme ou son système d'exécution. Les données logiques sont manipulées directement par les développeurs du programme et des mises à jour. Le développeur du programme définit les types et créé les variables, le développeur de mise à jour indique comment mettre à jour ces types et ces variables. Les données d'exécution sont généralement manipulées par les mécanismes de mise à jour dynamique dans le but de maintenir un état cohérent du programme (par exemple en modifiant la pile d'exécution pour changer toutes les références à du code obsolète et les remplacer par des références vers la nouvelle version). Dans ce manuscrit, le terme *données* désigne les données logiques d'un programme, sauf mention du contraire.

Lorsqu'un programme s'exécute, tous ses fils s'exécutent en parallèle en appelant les fonctions du programme. La figure 1.2 montre un exemple d'exécution d'un fil. Lorsqu'un fil appelle une fonction, il crée un espace mémoire nommé *frame* dans lequel sont enregistrées les variables locales à la fonction

```
/* Nouveau code */
fonction: foo_v2(arg1,arg2){...}
type: Spam_v2 {champ1, champ2}

/* Script de mise à jour */
fonction: transformSpam(var){...};

Quand foo est inactive{
  Redéfinir foo en foo_v2
  Redéfinir Spam en Spam_v2
  Accéder immédiatement aux variables de type Spam:
    Leur appliquer transformSpam
}
```

FIGURE 1.3 – Exemple de mise à jour (pseudo code)

ainsi que son code. Cette *frame* est ajoutée à la *pile d'exécution* du fil. Le fil exécute alors le code de cette *frame*. Les variables locales sont enregistrées dans la *frame* et les variables globales sont enregistrées dans la mémoire globale, partagée entre tous les fils. Si l'exécution du code appelle une fonction, une nouvelle *frame* est crée et c'est le code de cette dernière qui est exécutée. Lorsqu'un appel de fonction est terminé (c'est-à-dire lorsque la fonction retourne), la *frame* est supprimée et le fil continue l'exécution de la *frame* précédente. Lorsqu'un fil termine d'exécuter la première *frame*, il s'arrête : il a terminé son exécution. Le programme termine quand tous ses fils d'exécution ont terminé.

Ce modèle permet d'identifier les principaux éléments des programmes affectés par les mises à jour dynamiques quelque soit le langage de programmation utilisé. Si le programme est écrit dans un langage objet comme Java par exemple, les types sont les classes, les données logiques sont les objets du tas, et les fonctions sont les méthodes des classes. Si le programme utilise des composants les types sont les types des composants, et les données sont l'état de ces composants et les fonctions sont ces composants eux mêmes.

1.1.2 Structure d'une mise à jour

Dans ce manuscrit, une distinction est faite entre développeur du programme et développeur de mise à jour, le premier fournissant le logiciel original et le second fournissant les mises à jour (qui peuvent inclure plus que le code de la nouvelle version des programmes). Même si en pratique, ce sont souvent les mêmes personnes physiques qui remplissent ces deux rôles, cette distinction permettra de discerner ces deux rôles qui agissent à deux étapes différentes de la vie d'un logiciel en utilisant des méthodes et des outils éventuellement différents.

Ce que le développeur de mise à jour doit fournir pour chaque mise à jour dynamique dépend de la ou des plates-formes utilisées. Ces entrées sont généralement transformées en un tout communément appelé *patch dynamique* dans le domaine. Ce manuscrit identifie deux composants élémentaires d'un patch dynamique : le *nouveau code* et le *script de mise à jour*. Le premier peut être le code de la nouvelle version du programme, le code des éléments modifiés du programme ou encore un *diff* des codes originaux et modifiés (code source, *bytecode* ou encore binaire). Le second est une série d'instructions indiquant quelles tâches doivent être accomplies pour mettre à jour le programme. Selon les plates-formes, le script de mise à jour peut contenir plus ou moins d'instructions en fonction de la marge de manœuvre laissée au développeur des mises à jour. Par exemple, dans le cas d'une plate-forme fixant de manière permanente tous les mécanismes qu'elle utilise comme OPUS [2], le script de mise à jour est vide. Il est à noter que dans un tel cas, le script de mise à jour est considéré comme fixé par la plate-forme, et ce pour toutes les mises à jour. Le nouveau code est donc intégralement fourni dans le patch dynamique tandis que le script de mise à jour peut être en partie fixé par la plate-forme. La suite de ce manuscrit montrera que dans une majorité de cas, la plate-forme fixe l'intégralité du script de mise à jour et que le patch dynamique ne contient que le nouveau code.

La figure 1.3 montre un exemple théorique de patch dynamique. Le nouveau code définit la nouvelle version d'une fonction `foo` et d'un type `Spam`. Le script de mise à jour indique quand et comment mettre à jour ces deux éléments. Lorsque la fonction `foo` est inactive (c'est-à-dire quand aucun appel à `foo` n'est en cours), les deux éléments sont redéfinis et toutes les variables de type `Spam` sont accédées puis mises à jour par la fonction `transformSpam`. Ces tâches et les concepts qu'elles emploient sont présentés dans la prochaine section et détaillés dans le chapitre 3.

1.2 Principes de base de la mise à jour dynamique

Pour pouvoir mettre à jour dynamiquement un programme, il faut habituellement le combiner avec une plate-forme avant de le démarrer. Suivant la plate-forme utilisée, cela peut nécessiter de modifier le code du programme (par exemple, Kitsune [44] demande de placer des points de mise à jour dans le code du programme).

Lorsqu'une mise à jour est requise par un développeur, un patch dynamique est généré. Il précise quels éléments du programme vont être modifiés et quelles sont ces modifications. Le patch est ensuite soumis à la plate-forme qui agira, soit elle-même, soit par l'intermédiaire d'un gestionnaire de mise à jour. La première étape est d'abord de charger le nouveau code dans le programme. Il faut ensuite attendre un moment propice à la mise à jour. En effet, si le programme n'est pas dans un état *stable*, appliquer une mise à jour peut provoquer des erreurs, ou le *crash* du programme. Par exemple, une fonction fraîchement mise à jour peut tenter d'accéder à de vieilles données qui ne sont plus compatibles. Généralement, dans cet état stable, les éléments modifiés ou les éléments qui leurs sont liés sont *quiescents*. C'est-à-dire que ces éléments ne sont pas utilisés et ne le seront pas dans un futur proche. Par extension un peu abusive, cet état stable est communément appelé *quiescence du programme*. Cependant, il existe aussi d'autres conditions de stabilité plus ou moins restrictives que la quiescence. Pour éviter toute confusion, la suite de ce manuscrit appellera *altérabilité* tout état stable propice à la mise à jour. Une définition plus précise de l'altérabilité est donnée dans le chapitre 3 page 17.

Lorsque le programme est altérable (dans un état d'altérabilité), son exécution est suspendue et les modifications peuvent être appliquées : il faut remplacer les fonctions obsolètes par leur nouvelle version, mettre à jour les types puis accéder aux données et les transformer pour qu'elles adoptent la nouvelle version de leur type. Pour chaque type de modification, il existe plusieurs mécanismes dans la littérature, chacun utilisant une méthode différente avec ses propres avantages, exigences et désavantages.

Lorsque toutes les modifications ont été appliquées, l'exécution du programme est reprise. Ainsi, le programme continuera son exécution dans sa nouvelle version, comme s'il avait été démarré dans sa dernière version depuis le début.

Si les fonctions et les types sont identifiables dans le programme, les données peuvent être statiques (variables globales, constantes, ...) ou dynamiques (objets crées par l'exécution du programme, fils d'exécution, ...). Les éléments statiques étant définis explicitement par le développeur du programme sont faciles d'accès lors de la mise à jour : le développeur de la mise à jour peut explicitement faire référence à un élément statique donné. Les éléments dynamiques sont plus difficiles à modifier car le développeur de la mise à jour peut ignorer leur existence (par exemple, il ne connaît pas a priori l'ensemble des objets créés par l'exécution du programme). Il doit alors faire référence à ces éléments à partir des éléments statiques (toutes les variables d'un type donné, tous les fils d'exécution exécutant une fonction donnée, ...). Généralement, les éléments dynamiques qu'un développeur de mise à jour souhaite accéder font partie de l'ensemble des données du programme. C'est pourquoi les plates-formes fournissent des mécanismes permettant l'accès aux données dynamiques : d'où le nom de cette tâche : *accès aux données*.

On identifie donc les cinq principales tâches d'une mise jour dynamique :

1. détecter l'altérabilité du programme ;
2. modifier les fonctions ;
3. modifier les types ;
4. accéder aux données ;
5. transformer les données.

Comme le montre le prochain chapitre qui présente un état de l'art du domaine, pour chacune de ces tâches il existe plusieurs mécanismes permettant de l'accomplir de manière différente. Le prochain chapitre montre comment les six plates-formes OPUS [2], Ginseng [58], Jvolve [72], ActiveContext [76], Ksplice [4] et K42 [8] remplissent ces tâches et à l'aide de quels mécanismes.

Chaque plate-forme doit proposer des mécanismes permettant d'accomplir ces cinq tâches. L'approche usuelle de la mise à jour dynamique veut que chaque plate-forme utilise un seul mécanisme pour chaque tâche et l'emploie de la même façon à chaque mise à jour. La nouvelle approche défendue dans ce manuscrit veut que chaque plate-forme offre plusieurs mécanismes pour chacune de ces tâches et que le développeur de la mise à jour puisse choisir celui qui convient le mieux au cas par cas.

Chapitre 2

Etat de l'art

Chaque plate-forme de mise à jour dynamique embarque un petit ensemble de mécanismes, généralement un mécanisme par tâche à accomplir. Par conséquent, chaque mise à jour est appliquée suivant le même scénario et par les mêmes mécanismes. C'est un effet volontaire permettant de rendre les plates-formes transparentes, offrant ainsi une plus grande simplicité d'utilisation mais réduisant leur flexibilité. Comme effet secondaire, le redémarrage devient nécessaire pour certaines mises à jour trop complexes ou impossibles à appliquer avec les mécanismes de la plate-forme. C'est le cas du système d'exploitation K42 [8] qui, malgré une conception spécialement prévue pour la mise à jour dynamique, supporte mal les changements d'interface de programmation (API).

En effet, il est communément admis dans le domaine que les mises à jour effectuées dynamiquement corrigent des erreurs du code ou des failles de sécurité. Les mises à jour changeant la sémantique du programme de manière significative sont souvent considérées comme hors du champ d'application des mises à jour dynamiques. Cela permet à certaines plates-formes de se focaliser sur certaines modifications. Par exemple, OPUS [2] ne permet pas de changer la signature des fonctions.

2.1 Précédents états de l'art

Deux articles [69, 55] répertorient des plates-formes de l'état de l'art. *A survey of dynamic software updating* par Seifzadeh et al. propose une évaluation des plates-formes selon une large gamme de métriques quantifiant de nombreux aspects comme par exemple le niveau d'abstraction adressé (code source ou modèle), la capacité des plates-formes à assurer la consistance de l'application des mises à jour (la plate-forme est alors dite consistante, consistante sous réserve, ou inconsistante en fonction des garanties qu'elle apporte), ou encore les mécanismes qu'elles adoptent, par exemple, pour accéder aux données et les transformer.

A survey about dynamic software updating par Miedes et al. répertorie plusieurs publications liées à la mise à jour dynamique et établit une synthèse des propriétés nécessaires aux plates-formes de mise à jour dynamique. Par exemple, «les modifications doivent êtres indiquées de manière déclarative et non de manière opérationnelle» (il ne faut donc pas écrire de nouveau programme pour décrire une mise à jour) ou encore «les modifications doivent amener le programme dans un état consistant». L'article détaille également les mécanismes qu'emploient certaines plates-formes.

Ces deux articles tentent d'établir une synthèse ainsi qu'une classification du domaine, le premier se concentrant sur la classification des plates-formes en tant que tout indissociable, le second s'orientant vers une énumération des principaux concepts et mécanismes liés à la mise à jour dynamique. Ils permettent tous les deux d'identifier les principales tâches d'une mise à jour listées dans la section 1.2 ainsi que les concepts clés du domaine comme la quiescence, ou encore l'importance de la consistance des mises à jour et de leur application.

Les limitations de ces états de l'art sont détaillées dans la partie I qui établit une classification des plates-formes basée sur les mécanismes qu'elles emploient pour mettre à jour les programmes. La suite du présent chapitre est consacrée à la description de certaines plates-formes et des mécanismes qu'elles emploient. Il ne s'agit pas, ici, d'établir une description exhaustive de l'état de l'art mais de présenter les principales contributions du domaine qui ont amené à établir la thèse défendue dans ce manuscrit. Le chapitre 4 de la première partie présente une analyse plus en profondeur de certaines plates-formes de l'état de l'art et l'annexe A offre un inventaire presque exhaustif des plates-formes tout en récapitulant les propriétés de ces plates-formes.

2.2 Plates-formes pour langages compilés

2.2.1 OPUS

OPUS [2] est une plate-forme pour programmes écrits en langage C. Elle permet de modifier les fonctions autres que la fonction *main* à condition que leur signature ne change pas. Les fonctions ne doivent pas non plus changer de comportement extérieur : les modifications ne doivent pas changer de valeur de retour ni ajouter d'opérations d'écriture dans des données non locales à la fonction. Une analyse statique vérifie que ces conditions sont respectées lors de la génération du patch dynamique à partir du code source de la nouvelle version du programme.

Les mises à jour sont appliquées par un installateur qui est un processus externe au programme. L'installateur s'attache au processus du programme pour en prendre le contrôle, ce qui a aussi pour effet de le suspendre. L'installateur parcours alors la pile des fils d'exécution pour s'assurer que les fonctions modifiées n'y sont pas. Si une fonction modifiée n'est présente dans aucune pile, elle est immédiatement mise à jour par indirection : une instruction *jump* vers la nouvelle version de la fonction est écrite par-dessus sa première instruction. Si la fonction est encore présente dans la pile, l'installateur met en place des points d'arrêt (ou *trap*) pour intercepter l'entrée et la sortie de la vieille version de la fonction. Puis l'exécution est reprise. Lorsqu'un fil d'exécution déclenche un de ces *trap*, il est suspendu. Lorsque tous les fils sont suspendus de cette manière, la fonction est mise à jour. L'altérabilité pour les mises à jour appliquées par OPUS est sous la condition que les fonctions modifiées ne soient présentes dans aucune pile. En effet, dans le cas contraire, les futurs appels à ces fonctions utiliseraient la nouvelle version tandis que les anciens appels (qui ont été repérés dans une pile) utiliseraient l'ancienne version. Le programme deviendrait inconsistant et cela pourrait entraîner des erreurs.

OPUS combine deux mécanismes classiques de mise à jour : l'indirection de fonction et le parcours des piles pour détecter l'altérabilité. Le placement de *trap* est un mécanisme supplémentaire ajouté par les plates-formes pour le langage C. Cela permet de gagner en performance par rapport à l'alternative qui consiste à attendre et réessayer plus tard si une fonction modifiée est présente dans la pile.

2.2.2 Ginseng

Ginseng [58] est une plate-forme pour programmes écrits en C utilisant une chaîne de compilation spéciale pour adapter le programme d'origine à ses besoins et préparer les mises à jour à partir du nouveau code.

Lors de la compilation du programme d'origine, le développeur du programme est invité à placer des points de mise à jour aux endroits du code où, selon son jugement, le programme est globalement stable (c'est-à-dire altérable au vu de la majorité des mises à jour). Les points de mise à jour sont ensuite annotés lors de la compilation pour indiquer quels éléments du programme peuvent être modifiés sans porter atteinte à la stabilité du programme.

Lorsqu'une mise à jour est requise, l'exécution du programme continue jusqu'à ce qu'un point de mise à jour soit rencontré. Lorsqu'un point de mise à jour est rencontré, Ginseng vérifie que le point est compatible avec la mise à jour en se servant des annotations placées lors de la compilation du programme original. Si le point est compatible, le nouveau code (qui contient les nouvelles versions des éléments modifiés) est chargé sous forme d'une bibliothèque dynamique. Le programme est ensuite mis à jour alors que ses fils d'exécution sont suspendus. Si le point n'est pas compatible, l'exécution du programme continue jusqu'à ce qu'un point compatible soit atteint.

Les fonctions sont mises à jour au moyen d'indirection : lors de la compilation du programme original, un pointeur est introduit pour chaque fonction et chaque appel de fonction est remplacé par un appel au travers de ce pointeur. Quand une fonction est mise à jour, son pointeur est redirigé vers la nouvelle version de la fonction.

Lorsqu'un type T est mis à jour, sa nouvelle version est chargée comme un type différent T_{n+1} (n étant le numéro de la version précédente). Il est toutefois mémorisé qu'il s'agit de la nouvelle version de T. Les données sont ensuite mises à jour de manière progressive. Lors de la compilation du programme original, un appel à une fonction vérifiant la validité des variables est placé avant chaque utilisation de chaque variable. Cette fonction vérifie que la variable de type T_n sur le point d'être utilisée est bien du type T_{n+i} le plus récent. Si ce n'est pas le cas, la variable est mise à jour pour être conforme au type le plus récent. La mise à jour des données se produit donc à mesure que le programme s'exécute.

Ginseng est une des plates-formes nécessitant le plus de préparation pour le programme d'origine. Il faut placer les points de mise à jour et, lors de la compilation, l'indirection des fonctions est installée et les appels aux fonctions de vérification pour les données sont placés. En revanche, les mises à jour ne

demandent pas de travail spécifique de la part du développeur de mise à jour : le patch dynamique est construit uniquement à partir du code source de la nouvelle version du programme.

Ginseng combine plusieurs mécanismes classiques : l'altérabilité est détectée au moyen de points de mise à jour placés par le développeur du programme, les fonctions sont mises à jour au moyen d'indirection (bien que sous une forme différente de celle vue pour OPUS) et les données sont accédées de manière progressive.

Remarque : stratégie d'accès

Si dans la littérature on parle de stratégie *paresseuse* ou *pressée* pour la mise à jour des données, ce manuscrit fait une distinction entre stratégie d'accès (*progressive* ou *immédiate*) et moment de transformation (*instantané* ou *retardé*). Ainsi la stratégie de mise à jour paresseuse (respectivement pressée) est une combinaison des politiques progressive et instantanée (respectivement, immédiate et instantanée). Cette distinction permet une description plus précise des mécanismes de mise à jour. Par exemple, une politique de mise à jour accédant immédiatement aux données pour y attacher un gestionnaire qui les modifiera plus tard (transformation retardée) ne correspond exactement ni à la stratégie paresseuse, ni à la stratégie pressée.

2.3 Plates-formes pour langages à machine virtuelle

2.3.1 Jvolve

Jvolve [72] est une plate-forme pour programmes Java, utilisant la machine virtuelle *Jikes RVM*. Elle génère ses patchs dynamiques à partir du code source de la nouvelle version du programme, sous la forme de *transformers* : des fonctions dont l'exécution met à jour des données ou des types. Le développeur de la mise à jour peut également modifier les *transformers*.

Les classes peuvent être mises à jour dans leur intégralité (c'est-à-dire à la fois les champs statiques, les attributs, et les méthodes). Selon l'étendue des modifications, différents mécanismes sont utilisés.

Si seulement les méthodes sont changées, elles sont modifiées sans changer la classe en utilisant l'indirection native de la machine virtuelle (la classe est, de base, reliée avec ses méthodes via des pointeurs). Cette opération a lieu quand les méthodes modifiées sont quiescentes (c'est-à-dire qu'elles ne sont dans aucune pile). Cela est vérifié en inspectant les piles lorsque la machine virtuelle atteint un point *sûr* (moment de l'exécution où la machine virtuelle peut suspendre le programme pour, par exemple, appeler le ramasse-miettes). Lorsqu'un point sûr où une méthode est quiescente est atteint, la méthode est mise à jour. En même temps, toutes les méthodes non modifiées dans le code source mais dont le *bytecode* doit changer (par exemple, à cause d'optimisation de l'adresse de la méthode modifiée) sont remplacées sur la pile : le *bytecode* de ces méthodes est invalidé et le compilateur juste à temps de la machine virtuelle se charge de le remplacer par du *bytecode* valide.

Si les modifications appliquées à une classe changent plus que les méthodes, la classe est intégralement rechargée et les instances de cette classe sont transformées au prochain passage du ramasse-miettes.

Jvolve combine points de mise à jour (les points *sûrs* de la machine virtuelle) et parcours des piles pour détecter l'altérabilité. Elle utilise l'indirection et le remplacement sur pile pour mettre à jour les fonctions (méthodes) et elle adopte une politique d'accès aux données *immédiate* et de transformation *instantanée* (tous les objets sont accédés et transformés dès le passage du ramasse-miettes).

2.3.2 ActiveContext

ActiveContext [76] est une plate-forme pour programmes Smalltalk. Elle utilise des contextes pour permettre à plusieurs versions de coexister. Chaque fil d'exécution est associé au contexte qui était le plus récent au moment de son lancement. Chaque contexte correspond à une version du programme. Il contient une copie des données ainsi que des classes. Si plusieurs fils d'exécution utilisent deux copies d'une même donnée dans deux contextes différents, les copies sont synchronisées entre les contextes.

Les mises à jour sont donc appliquées en ajoutant un nouveau contexte. Les vieux fils d'exécution s'exécutent toujours dans de plus vieux contextes et les prochains fils s'exécuteront dans ce nouveau contexte. Quand tous les vieux fils seront terminés, le programme est intégralement dans sa nouvelle version. Les vieux contextes sont alors supprimés par le ramasse-miettes.

Un patch dynamique est composé de deux *state transformers* spécifiant les changements depuis le contexte précédent : un *state transformer* précise comment migrer les données et les classes du précédent contexte au nouveau et l'autre précise comment effectuer la migration inverse : du nouveau contexte au précédent contexte.

ActiveContexte utilise des mécanismes peu fréquents en dehors des plates-formes pour Smalltalk : la coexistence de plusieurs versions avec synchronisation des données.

2.4 Systèmes d'exploitation

2.4.1 Ksplice

Ksplice [4] est une plate-forme permettant de mettre à jour le système d'exploitation Linux. Elle permet de mettre à jour les fonctions au niveau binaire. Le patch dynamique est généré à partir des binaires du programme original et de la nouvelle version du programme.

Lorsqu'une mise à jour est invoquée, Ksplice parcours les piles pour vérifier la quiescence des fonctions mises à jour. Si aucune fonction n'est quiescente, la mise à jour est reportée d'un court laps de temps. Les fonctions quiescentes sont mises à jour par indirection en écrasant la première instruction de l'ancienne version par une instruction *jump*, comme OPUS.

Ksplice ne prévoit pas de mécanisme pour mettre à jour les types ou les données. Elle permet toutefois au développeur de la mise à jour d'écrire du code qui sera exécuté pendant la mise à jour. Ce code pourra alors se charger de mettre à jour les types et les données.

2.4.2 K42

K42 [8] est un système d'exploitation spécialement conçu pour être mis à jour dynamiquement. Il a notamment la particularité que ses fils d'exécution ont tous une durée de vie courte. Son architecture est orientée composants. Des fabriques permettent de produire les composants et gardent une référence sur chacun des composants qu'elles créent.

Les mises à jour consistent en le remplacement d'un ou plusieurs composants par leur nouvelle version. Lorsqu'une mise à jour est invoquée, les fabriques sont d'abord remplacées par leur nouvelle version. Les références vers les composants sont transférées de l'ancienne fabrique vers la nouvelle. Puis, un *mediator* est attaché à chaque composant référencé par la fabrique. Ces *mediators* bloquent tous les appels entrant dans les composants à condition qu'ils ne soient pas issus d'appels récursifs. Ainsi, tous les *jeunes* fils d'exécution entrant dans de vieux composants sont suspendus. Quand tous les *vieux* fils sont terminés, les composants sont alors quiescents et sont remplacés par leur nouvelle version. Les *mediators* débloquent ensuite les fils d'exécution avant d'être détachés des composants.

Tous les remplacements de composants ou de fabriques sont effectués en utilisant de l'indirection à base de pointeurs, comme dans Ginseng et Jvolve. Pour accéder aux composants et les mettre à jour, K42 utilise une stratégie d'accès immédiate pour les mettre à jour de manière retardée : les *médiators* sont attachés immédiatement mais le remplacement des composants n'a lieu que lorsque ceux-ci sont quiescents.

2.5 Travaux similaires ou connexes

D'autres travaux extérieurs au domaine abordent des thématiques similaires à celles de la mise à jour dynamique. Des techniques communes à la mise à jour dynamique sont en effet utilisées par des outils comme Padrone [66] ou encore LAM/MPI [68].

Padrone est un outil d'optimisation pour binaires en cours d'exécution. L'outil se connecte à un processus pour analyser dynamiquement le programme qu'il exécute et détecter les fonctions critiques qui gagneraient à être optimisées (par exemple, des fonctions qui sont appelées un grand nombre de fois). Padrone peut alors redéfinir les fonctions en une version optimisée et donc plus efficace. S'il utilise un mécanisme courant de mise à jour dynamique, Padrone n'est pas conçu pour appliquer des mises à jour à des programmes. La nouvelle version des fonctions est générée automatiquement sans impliquer de développeur de mise à jour. La plupart des plates-formes de mise à jour dynamique requièrent une instrumentation du code du programme et supportent mal les optimisations qu'un compilateur peut apporter au code des programmes. Padrone est conçu pour être utilisé sur des programmes potentiellement optimisés ou dont les symboles ont été supprimés. La perte de ces informations est compatible avec

l'objectif de Padrone : optimiser le code d'un programme en exécution, mais rend difficile l'association entre code binaire et code source qui est nécessaire pour définir le script de mise à jour. Il devient par exemple difficile d'exprimer quelles parties du code correspondent à une fonction redéfinie par une mise à jour.

LAM/MPI utilise un mécanisme de sauvegarde et de retour en arrière pour récupérer d'un état d'erreur ou encore déplacer un processus d'un noeud à un autre dans une grille de calcul. Le *framework* LAM/MPI permet à des sauvegardes de l'état du programme d'être effectuées sur commande. En cas de besoin (état d'erreur, migration, ...), les processus peuvent être redémarrés et l'état sauvegardé est rechargé. Ce *framework* utilise des mécanismes semblables aux points de mise à jour de Ginseng (bien que les points de sauvegarde soient évalués dynamiquement par LAM/MPI) et au redémarrage des fils d'exécution de Kitsune [44].

2.6 Des mécanismes divers mais transversaux

Si au travers des domaines d'application des plates-formes on observe des mécanismes originaux, on constate également que certains mécanismes sont présents dans tous les domaines. Il est en effet évident que l'utilisation du ramasse-miettes pour mettre à jour les données est propre aux langages qui disposent d'un tel mécanisme. Mais cela n'empêche pas la stratégie d'accès *immédiate* d'apparaître sous une autre implémentation dans K42. Au sein d'un même langage, différentes implémentations d'un mécanisme donné peuvent exister. On a vu que la mise à jour de fonctions par indirection peut se faire à l'aide de pointeurs comme dans Ginseng ou Jvolve, mais également à l'aide d'instructions *jump* comme dans OPUS ou Ksplice.

Comme le montrera la première partie, si certains mécanismes sont spécifiques à un type donné de plates-formes (par exemple, les contextes de ActiveContext qui existent peu en dehors de plates-formes pour Smalltalk), la majorité d'entre eux existent dans différents types de plates-formes (par exemple, le parcours des piles se retrouve dans les plates-formes pour langages compilés, à machines virtuelles et pour les systèmes d'exploitation). On peut alors supposer que le choix des mécanismes dépend plus des propriétés du programme ou des mises à jour que du langage ou du système d'exécution employé, même si ces derniers peuvent avoir une influence sur la difficulté d'implémentation de certains mécanismes. En effet, dans Ginseng, le programme doit subir de nombreuses modifications (insertion des points de mise à jour, ajout des appels aux fonctions de vérification pour les données, introduction de l'indirection à base de pointeurs) pour pouvoir utiliser les mécanismes employés par la plate-forme de mise à jour.

Première partie

Analyse des plates-formes

Analyse des plates-formes

Quels sont les composants essentiels d'une plate-forme de mise à jour dynamique ? Quels sont les entités permettant la modification dynamique d'un programme ? Cette partie répond à ces questions grâce à une étude empirique de l'état de l'art. Il s'agit ici de comprendre comment les plates-formes appliquent les mises à jour en analysant leur structure et leur fonctionnement.

Cette partie se découpe en trois chapitres. Le chapitre 3 page 17 propose des modèles généraux permettant de décrire une plate-forme quelque soit son langage de programmation et quelque soit le type de programme qu'elle cible. Ces modèles permettent d'analyser les plates-formes de la littérature et de comparer leurs structures et les mécanismes qu'elles emploient. Le chapitre 4 page 23 présente les résultats de cette analyse au travers de tables répertoriant les composants des plates-formes, comment ils sont configurés ainsi que les mécanismes qu'emploie chaque plate-forme. Ces deux chapitres apportent les réponses aux deux questions posées ci-dessus : les modèles du premier chapitre donnent une description générale des plates-formes de mise à jour et les tables d'analyse identifient les composants de chaque plate-forme et comment ils permettent de mettre à jour les programmes. Les résultats de ces deux chapitres soulèvent deux nouvelles questions auxquelles répond le chapitre 5 page 37 : Y a-t-il des synergies entre les différents choix de conceptions d'une plate-forme ? et y a-t-il des groupes de plates-formes similaires ?

Le chapitre 5 applique des analyses statistiques aux résultats des deux précédents chapitres et observe les combinaisons fréquentes parmi les mécanismes et choix de conception des plates-formes. Certaines de ces combinaisons indiquent des synergies qui sont listées dans le chapitre. Enfin, des familles de plates-formes aux propriétés similaires sont identifiées.

Chapitre 3

Une étude centrée sur les mécanismes

Si on peut compter quelques travaux répertoriant les différents types de mise à jour comme ceux de Neamtiu et al. [78], Buckley et al. [16] et Giuffrida et al. [38], l'étude des mécanismes a reçu beaucoup moins d'attention. Or l'étude de ces mécanismes, l'analyse de leurs exigences, de leur capacité à être combinés, configurés et inter-changés est une étape clé vers la conception de plates-formes suivant la nouvelle approche défendue dans ce manuscrit.

Dans l'objectif d'identifier les composants permettant aux plates-formes de mettre à jour dynamiquement les programmes, ce chapitre détaille un modèle générique qui est utilisé dans les prochains chapitres pour décrire les plates-formes de façon uniformes. Il devient alors possible de comparer des plates-formes ciblant des programmes ou des langages différents, et d'étudier les liens entre composants et mécanismes.

3.1 Limitations des taxonomies

L'état de l'art de Miedes et al. [55] résume les contributions et les propos d'autres articles du domaine. Cela amène dans quelques cas à détailler l'implémentation des mécanismes de certaines plates-formes. C'est une bonne piste pour étudier leurs limitations ou leurs exigences, mais cela ne suffit pas pour établir une cartographie des mécanismes. L'article identifie des thèmes et des techniques couramment employés comme la réécriture de code binaire ou l'utilisation de *proxy*. En dehors de cela, aucune synthèse sur le rôle des mécanismes n'est effectué. Les mécanismes détaillés ne font pas l'objet d'une classification et la façon dont ils sont combinés au sein de chaque plate-forme n'est pas étudiée. Ces résultats auraient permis de lister les mécanismes usuels de mise à jour dynamique tout en les regroupant par fonctionnalité (les modifications qu'ils permettent) ou propriétés similaires (exigences, limitations, ...).

L'état de l'art de Seifzadeh et al [69] cherche à comparer tous les travaux du domaine de la mise à jour dynamique sur de nombreux aspects. Certains de ces aspects sont liés à des mécanismes de mise à jour comme par exemple la stratégie de mise à jour des données (paresseuse ou pressée). Mais la plupart de ces aspects sont orientés pour la quantification ou la qualification de propriétés de la plate-forme en tant qu'entité entière. Par exemple, l'aspect *unité de mise à jour* peut avoir comme valeur *remplacement du programme en entier*, *mise à jour d'un module* ou *unité de mise à jour*, pour exprimer la granularité des mises à jour permises par une plate-forme. La dernière valeur indique que la plate-forme définit sa propre unité de mise à jour, c'est-à-dire qu'elle définit cette granularité. Ces qualifications ne sont pas suffisamment précises pour une analyse des mécanismes. Une analyse des plates-formes concentrée sur les mécanismes qu'elles emploient permettrait, en réduisant la variété des aspects à considérer, d'étudier plus finement comment chaque tâche des mises à jour dynamiques est effectuée par chaque plate-forme, listant les mécanismes utilisés et remarquant la façon dont ils sont combinés dans chaque plate-forme. Une telle analyse permettrait, en plus de répertorier les différents mécanismes de l'état de l'art, d'identifier les différents choix de conception effectués dans chaque plate-forme et constituerait une première étape vers l'étude de l'impact de ces choix sur les modifications qu'une plate-forme permet. De plus, pour permettre une analyse plus fine des plates-formes, il serait nécessaire d'identifier quels sont les éléments de base qui constituent ces plates-formes. Un tel résultat est particulièrement utile pour la conception de plates-formes adoptant la nouvelle approche car elles doivent pouvoir reproduire le comportement de plusieurs plates-formes habituelles.

3.2 Un modèle générique

Afin de pouvoir comparer des plates-formes parfois très différentes, il est nécessaire d'établir des modèles et des terminologies génériques pouvant s'appliquer au plus grand nombre possible de ces plates-formes. Il s'agit par exemple de comparer un système d'exploitation modifiable dynamiquement comme Ksplice [4] avec une plate-forme pour applications Java telle que Jvolve [72]. À première vue, ces plates-formes sont trop différentes pour être comparées. Mais les mécanismes qu'elles emploient sont similaires. Par exemple, elles mettent toutes les deux à jour les fonctions quand elles sont inactives (c'est-à-dire lorsqu'elles ne sont dans aucune pile d'exécution).

Il est possible de comparer des plates-formes très différentes à condition d'adopter un niveau d'abstraction adéquat et de *projeter* chaque plate-forme sur un modèle standardisé. Ce modèle standardisé doit pouvoir représenter les spécificités de chaque plate-forme tout en regroupant celles qui utilisent des mécanismes similaires. Par exemple, Ksplice et Jvolve utilisent tous les deux de l'indirection pour redéfinir les fonctions. Ksplice insère des instructions *jump* tandis que Jvolve utilise l'indirection native de la JVM. Si l'implémentation du mécanisme de mise à jour des fonctions diffère entre les deux plates-formes, adopter un niveau d'abstraction regroupant ces deux implémentations sous l'étiquette *redéfinition par indirection* permet de noter la similarité entre ces deux mécanismes.

Transposer les programmes ciblés par chaque plate-forme sur le modèle présenté dans le premier chapitre à la page 3 est une première étape vers leur comparaison. Il devient alors possible d'abstraire les implémentations des mécanismes employés et donc d'analyser les variations possibles de ces mécanismes au travers des différentes plates-formes.

L'architecture des plates-formes peut suivre un procédé similaire. Quelque soit la plate-forme, certains rôles remplis par des éléments de cette dernière peuvent être identifiés. Parmi ces rôles figurent la surveillance de l'altérabilité, le contrôle de l'application des modifications ou encore la génération des patchs dynamiques. Adopter une abstraction adaptée pour décrire les choix de conception effectués par chaque plate-forme permet, ici aussi, de comparer les plates-formes entre elles.

Le cycle de vie des mises à jour appliquées par une plate-forme (appelé *cycle de vie de la plate-forme* par la suite) est l'ensemble des étapes que suit la plate-forme lors de l'application d'une mise à jour. Lors de l'analyse des plates-formes, il a été constaté que le cycle de vie de chaque plate-forme est transposable sur un cycle de vie générique qui est présenté dans ce chapitre.

La suite de ce chapitre définit la terminologie employée pour l'analyse des plates-formes et dans la suite du manuscrit. Un cycle de vie générique capable d'exprimer le cycle de vie de chaque plate-forme étudiée est également détaillé. Les résultats de ce chapitre établissent le modèle générique sur lequel chaque plate-forme est projetée, permettant de la comparer aux autres plates-formes.

3.3 Terminologie

3.3.1 Altérabilité

Dans ce manuscrit, le terme *altérabilité* est utilisé pour décrire un état stable du programme vis-à-vis d'une mise à jour. C'est-à-dire que la mise à jour peut être effectuée sans causer de problèmes (*crash* du programme, comportement incohérent, ...). Des *critères d'altérabilité* permettent d'exprimer des conditions nécessaires et suffisantes pour qu'un programme soit altérable vis-à-vis d'une mise à jour.

Définition 1. ***Altérabilité***

Un programme P est ***altérable*** *vis-à-vis d'une mise à jour U si appliquer U à P ne provoquera pas d'erreur et ne pourra pas rendre P inconsistant (au sens définit ci-dessous).*

Un ensemble de conditions $C_0, ..., C_n$ sur l'état de P est l'ensemble des ***critères d'altérabilité*** *pour U si la vérification de toutes les conditions $C_0, ..., C_n$ est nécessaire et suffisante pour que P soit* ***altérable*** *vis-à-vis de U. On dit alors pour tout $i \leq n$ que C_i est un* ***critère d'altérabilité*** *de P pour U.*

Définition 2. ***Inconsistance d'un programme***

Un programme P est dit ***inconsistant*** *(au sens de la mise à jour dynamique) si au moins deux versions différentes v_1 et v_2 coexistent dans P, et si il existe e_1 et e_2, deux éléments de P tels que :*

1. *e_1 est dans v_1 mais pas dans v_2.*
2. *e_2 est dans v_2 mais pas dans v_1.*
3. *e_1 peut utiliser e_2.*

Les critères d'altérabilités peuvent être divers. Le plus fréquent étant la *quiescence* des éléments modifiés. Cet état défini par Kramer et al. [47] pour un composant au sein d'un système nécessite que le composant ne soit impliqué dans aucune transaction en cours, qu'il n'initie aucune transaction ni ne soit impliqué dans aucune transaction qui serait initiée dans le futur. Comme montré par Gupta et al. [42], cette propriété peut se transposer hors du contexte de la programmation orientée composant. Une fonction est donc quiescente si elle n'est présente dans aucune pile d'exécution. En effet, si la fonction n'est pas dans la pile, elle n'est impliquée dans aucune transaction (appel) en cours et, à condition que le programme soit suspendu, elle ne sera pas appelée dans le futur, et ne fera pas non plus d'appel.

Il existe d'autres états dérivés de la quiescence permettant d'assurer la stabilité d'un composant pour une mise à jour. Par exemple, la *tranquillité* [73] est un état imposant moins de contraintes au composant. Au lieu de nécessiter qu'aucun composant n'entame de transaction sollicitant le composant cible, elle impose seulement que les composants adjacents [1] au composant ciblé ne soient pas engagés dans de telles transactions au moment considéré. Il est donc possible que le composant ciblé soit sollicité dans un futur lointain. La *sérénité* [34] est un autre exemple d'état de stabilité, dérivé de la tranquillité. Il demande que le composant cible soit *tranquille*, que les modifications à appliquer ne visent ni à supprimer le composant cible ni à le détacher et qu'aucun changement ne soit apporté à la sémantique des composants impliqués dans des transactions actives. La tranquillité ou la sérénité d'un élément sont d'autres exemples de critères d'altérabilité.

Pour la plupart des plates-formes, les critères d'altérabilités restent similaires au fil des mises à jours. Par exemple, quelque soit la mise à jour à appliquer, les critères d'altérabilité de Ksplice sont la quiescence de toutes les fonctions mises à jour. Dans ces conditions et pour simplifier le discours, on peut parler de critères d'altérabilité de la plate-forme, même s'il s'agit en vérité des critères d'altérabilité pour chaque mise à jour appliquée par la plate-forme.

3.3.2 Gestionnaire de mise à jour

Chaque plate-forme a un agent responsable de l'application des modifications. Parfois, il s'agit d'un processus externe qui se connecte au programme en cours d'exécution comme dans OPUS [2]. Dans d'autres cas, c'est une partie de la machine virtuelle sur laquelle s'exécute le programme comme dans Jvolve [72]. Cet agent aux formes diverses est nommé *gestionnaire de mise à jour* (ou *gestionnaire*) dans la suite de ce manuscrit.

Dans certains cas, le gestionnaire remplit d'autres rôles comme la scrutation de l'altérabilité ou le guidage de l'exécution du programme. La nature du gestionnaire, sa portée (les éléments qu'il peut modifier directement) ou encore sa durée de vie (durée pendant laquelle le gestionnaire existe [2]) peuvent être des choix déterminants pour une plate-forme, comme le montre la suite de cette partie.

3.3.3 Stratégie de mise à jour des données

Dans la littérature, la stratégie employée pour mettre à jour les données est décrite comme étant soit *paresseuse* si les données sont mises à jour progressivement à mesure que le programme s'exécute, ou *pressée* si les données sont mises à jour immédiatement, dès que cela est demandé. Si ces deux termes suffisent à décrire une majorité des stratégies de mise à jour, ils sont insuffisants dans certains cas. Par exemple, pour remplacer ses composants, K42 attache un *mediator* à chacun d'entre eux pour les guider vers leur quiescence. Lorsque le composant est quiescent, il est alors remplacé. Cette stratégie de mise à jour n'est ni pressée, ni paresseuse : les *mediators* sont attachés immédiatement mais les composants sont remplacés au fil de l'exécution du programme, lorsqu'ils sont quiescents. Dans ce manuscrit, une distinction est donc faite entre la stratégie d'accès aux données qui peut être *immédiate* ou *progressive* et la transformation qui peut être *instantanée* ou *retardée*. Une stratégie de mise à jour paresseuse correspond à une combinaison d'un accès progressif et d'une transformation instantanée. Les *mediators* de K42 sont donc attachés immédiatement et les composants sont remplacés de manière retardée.

Définition 3. ***Stratégie d'accès***

La stratégie d'accès à une donnée est dite ***immédiate*** *si, du point de vue de l'exécution du programme, la donnée est accédée de manière instantanée. C'est-à-dire que le programme n'exécute aucune instruction entre le moment où l'accès à la donnée est demandé et le moment où la donnée est accédée.*

La stratégie d'accès à cette donnée est dite ***progressive*** *si, au contraire, le programme peut exécuter au moins une instruction entre le moment où l'accès est demandé et le moment où la donnée est accédée.*

1. C'est-à-dire les composants qui solliciteront le composant ciblé.
2. Le gestionnaire de OPUS a donc une durée de vie limitée tandis que le gestionnaire de Jvolve est permanent.

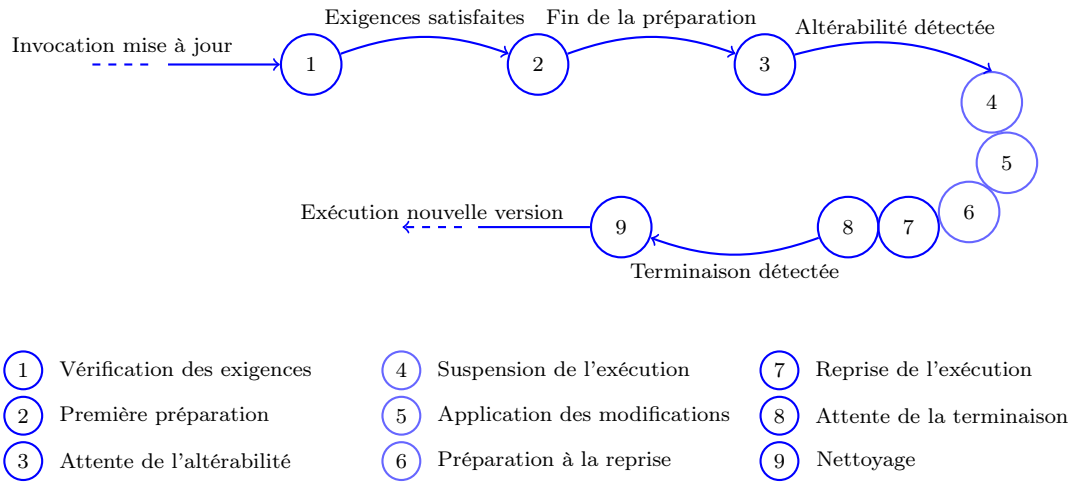


FIGURE 3.1 – Cycle de vie générique des mises à jour

Définition 4. ***Moment de transformation***

Le moment de transformation d'une donnée est dit ***instantané*** *si la donnée est transformée sans que le programme n'exécute d'instruction entre le moment où la donnée est accédée et le moment où elle est transformée.*

Le moment de transformation de cette donnée est dit ***retardé*** *si, au contraire, le programme peut exécuter au moins une instruction entre le moment où la donnée est accédée et le moment où elle est transformée.*

L'accès aux données est requis par la plate-forme au moment d'appliquer les modifications. Certaines plates-formes mettent en place des outils pour faciliter l'accès aux données lorsque ces dernières sont créées, d'autres utilisent des outils déjà présents dans le système d'exécution du programme comme par exemple, un ramasse-miettes.

3.4 Cycle de vie générique

Le cycle de vie de chaque plate-forme suit un schéma généralement similaire. Quand l'altérabilité est détectée, le programme est suspendu (totalement ou en partie) et les modifications sont appliquées. L'exécution du programme est alors reprise et le programme s'exécute dans sa nouvelle version. Certaines plates-formes ajoutent des étapes à ce cycle de vie, par exemple pour attacher un gestionnaire externe au programme avant de détecter l'altérabilité et pour le détacher une fois l'exécution reprise.

Cette section détaille un cycle de vie générique sur lequel le cycle de vie de toutes les plates-formes peut s'aligner. La figure 3.1 présente l'enchaînement des étapes du cycle de vie générique. Certaines plates-formes suivent plusieurs instances du cycle de vie lors de l'application d'une mise à jour. C'est le cas de K42 pour qui la mise à jour de chaque composant est traitée en parallèle de l'attente de l'altérabilité jusqu'à l'étape de nettoyage. D'autres plates-formes peuvent commencer à appliquer une mise à jour alors que la mise à jour précédente n'est pas totalement terminée. C'est le cas de Ginseng qui peut appliquer une nouvelle mise à jour alors que la totalité des données n'est pas encore transformée.

Le cycle de vie générique est composé d'étapes et de transitions. D'une manière générale, la plate-forme agit lors des étapes et attend une transition avant d'effectuer les actions de l'étape suivante. La plate-forme a donc un comportement actif pendant les étapes et un comportement passif lors des transitions. Par exemple, lors de l'étape d'attente de l'altérabilité, Ksplice [4] scrute les piles d'exécution pour détecter la quiescence des fonctions mises à jour. Si elles le sont, la transition *détection de l'altérabilité* est déclenchée et Ksplice passe à l'étape suivante. L'énumération suivante détaille les étapes et les transitions du cycle de vie générique. Les transitions sont bloquantes. C'est-à-dire que la plate-forme ne peut passer à l'étape suivante tant que la transition qui la précède n'a pas été déclenchée. Dans certains cas, des transitions peuvent être déclenchées immédiatement. Par exemple, ActiveContext [76] n'attend pas d'état d'altérabilité (elle considère le programme comme toujours altérable). La transition *détection de l'altérabilité* est donc automatiquement déclenchée. Il est à noter qu'aucune transition ne sépare les étapes 4 à 8 : elles sont enchaînées sans interruption par la plate-forme.

$\longrightarrow$ (1) **: Invocation de mise à jour** Une mise à jour est invoquée par son développeur, par l'intermédiaire d'une commande spécifique ou de toute autre forme de communication avec la plate-forme.

(1) **: Vérification des exigences** Une mise à jour peut nécessiter que le programme respecte certaines exigences (différentes des critères d'altérabilité) ou dépendre d'une précédente mise à jour. C'est pendant cette étape que la plate-forme vérifie ces exigences.

(1) $\longrightarrow$ (2) **: Satisfaction des exigences** Cette transition est déclenchée lorsque les exigences sont satisfaites.

(2) **Première préparation (avant mise à jour)** Lors de cette étape, la plate-forme peut effectuer des actions de préparation nécessaires au bon déroulement des prochaines étapes. Par exemple, c'est lors de cette étape que les *mediators* de K42 sont attachés aux composants à remplacer.

(2) $\longrightarrow$ (3) **: Fin de la première préparation** Cette transition est déclenchée lorsque la première préparation est terminée. D'une façon générale, cette transition est très courte, les tâches de préparation étant instantanées.

(3) **: Attente de l'altérabilité** Lors de cette étape, l'altérabilité du programme est attendue. Certaines plates-formes effectuent des actions pour, par exemple, guider l'exécution du programme vers l'altérabilité ou encore scruter les piles d'exécution.

(3) $\longrightarrow$ (4) **: Détection de l'altérabilité** Cette transition est déclenchée lorsque tous les critères d'altérabilité sont vérifiés.

(4) **: Suspension de l'exécution** C'est là que le programme est suspendu, intégralement ou en partie.

(5) **: Application des modifications** C'est lors de cette étape que les modifications sont appliquées au programme.

(6) **: Préparation à la reprise** Certaines plates-formes ont besoin d'effectuer des actions en préparation à la reprise de l'exécution du programme. Par exemple, pour vider des caches mémoires obsolètes ou pour préparer une tâche qui s'exécutera en parallèle du programme.

(7) **: Reprise de l'exécution** Si l'exécution du programme a été suspendue pendant l'étape de suspension, elle est reprise à cette étape.

(8) **: Attente de la terminaison de la mise à jour** Parfois, une mise à jour met du temps à être complètement appliquée. Il existe alors une étape pendant laquelle le programme a repris son exécution mais n'est pas encore complètement dans sa nouvelle version. Lors de cette étape, la plate-forme peut entreprendre des actions, comme par exemple, piéger l'écriture dans certaines variables pour y accéder progressivement.

(8) $\longrightarrow$ (9) **: Détection de la terminaison** Cette transition est déclenchée lorsque la mise à jour est complètement appliquée.

(9) **: Nettoyage** Lors de cette étape, la plate-forme supprime des éléments temporaires qui auraient été installés lors des étapes de préparation à la mise à jour ou à la reprise. C'est, par exemple, lors de cette étape que K42 détache les *mediators* des composants qui ont été remplacés.

(9) $\longrightarrow$ **: Nouvelle version** Le programme est à jour et s'exécute dans sa nouvelle version jusqu'à ce qu'une nouvelle mise à jour soit invoquée. Pendant ce temps, certaines tâches de fond peuvent toujours exister. C'est le cas de Ginseng où la version des variables utilisées est vérifiée en permanence, ce qui peut parfois provoquer une mise à jour des données.

Les termes ainsi que le cycle de vie générique définis dans ce chapitre permettent de décrire de manière similaire des plates-formes différentes. Par exemple, les *mediators* de K42 sont chacun un gestionnaire indépendant et les extensions de la JVM utilisées par Jvolve peuvent être considérées comme faisant partie d'un gestionnaire inclus dans la machine virtuelle. Cela est possible par construction : le cycle de vie et les termes précédemment définis tiennent compte des similarités entre les plates-formes et alignent des notions jusqu'alors peu clairement définies.

Le modèle générique présenté dans ce chapitre est utilisé dans les prochains chapitres pour décrire les plates-formes de l'état de l'art et permettre leur analyse.

Chapitre 4

Analyse des plates-formes

En adoptant une abstraction appropriée et en *projetant* les plates-formes étudiées sur un modèle générique comme décrit dans le chapitre précédent, il est possible de comparer des plates-formes très différentes en apparence. Ce chapitre analyse les plates-formes de l'état de l'art selon trois aspects : le cycle de vie, l'architecture et les mécanismes employés par ces plates-formes. De cette analyse résultent trois tables d'analyse récapitulant les choix de conception, les mécanismes et le cycle de vie suivi par chaque plate-forme.

Pour recenser les différentes plates-formes de l'état de l'art, plus de mille articles ont été collectés dans les actes de conférences ayant déjà publié des articles liés à la mise à jour dynamique [1] ainsi que sur les plates-formes de recherche de publication scientifique *ACM digital library*, *IEEE Xplore* et *Google Scholar*. Ces articles ont été triés d'abord automatiquement par occurrence de mots clés et par nombre de papiers de référence cités, puis manuellement selon leur intérêt. Les papiers décrivant des plates-formes ou des techniques de mise à jour dynamique sont conservés pour former un total de 206 papiers.

Les tables fournies en annexe A présentent l'analyse de 42 plates-formes décrites dans les 206 papiers résultant du tri précédemment détaillé. Les tables présentées dans ce chapitre ne montrent que les analyses de neuf plates-formes : Kitsune [44], ProteOS [36], Ksplice [4], OPUS [2], K42 [8], Ginseng [58], Jvolve [72], Rubah [63] et ActiveContext [76]. Ces plates-formes ont été choisies soit car elles constituent des cas d'école, soit parce qu'elles sont représentatives d'une famille de plates-formes aux propriétés spécifiques.

Les prochaines sections détaillent les critères d'analyse employés pour étudier les trois aspects de chaque plate-forme : son cycle de vie, son architecture et les mécanismes qu'elle emploie. Des tables récapitulent les résultats obtenus pour chacune des neuf plates-formes précédemment citées.

Les analyses des deux plates-formes Kitsune et ProteOS sont détaillées en fin de chapitre afin d'illustrer la méthode employée pour établir les tables présentées dans ce chapitre.

4.1 Cycle de vie

Comme expliqué dans le chapitre précédent, le cycle de vie d'une plate-forme est l'ensemble des étapes qu'elle suit lors de l'application d'une mise à jour. Le cycle de vie de chaque plate-forme analysée est transposé sur le cycle de vie générique décrit dans le chapitre précédent. Pour chaque étape du cycle de vie, les tables 4.1 et 4.2 récapitulent les opérations entreprises par les plates-formes ou les conditions qui permettent de déclencher une transition. Par exemple, la colonne *Détection de l'altérabilité* récapitule les critères d'altérabilité de chaque plate-forme.

4.1.1 Observations

Il est possible de distinguer plusieurs variations du cycle de vie standard. D'une manière générale, chaque plate-forme attend que le programme soit altérable avant de suspendre l'intégralité du programme, d'appliquer les modifications, puis de reprendre l'exécution du programme. Dans les tables 4.1 et 4.2, seuls ProteOS [36] et K42 [8] ne suspendent que les parties du programme touchées par les modifications à appliquer tandis que ActiveContext [76] ne suspend pas le programme lors des mises à jour.

Une autre variation notoire est la gestion de l'altérabilité : alors que la plupart des plates-formes attendent que le programme devienne altérable naturellement, certaines plate-forme comme K42 ou

1. Usenix ATC, Eurosys, Hotswup, ICSME et RAMSE

Plate-forme	Préparation (1)	Attente Altérabilité	Détection Altérabilité	Suspension	Modification	Préparation (2)
Kitsune			Point de MàJ	Tous fils suspendus	MàJ tas	
ProteOS	Envoi des *state filters* aux processus		Début boucle + *state filters*	Proc. suspendu	Créa. nouveau proc. transfert état	Vérif. cohérence état
Ksplice			Quiescence fonction	suspension totale	Redef. fonctions, MàJ données	
OPUS	Attach. gest.		Quiescence fonction	Tous fils suspendus	Redef. fonctions	
K42	Rempl. *factories*, Attach. *mediators*	Blocage des fils entrant	Plus de fils dans le composant		Rempl. composants	
Ginseng			Point de MàJ	Tous fils suspendus	Redef. fonctions, début MàJ données	
Jvolve			Point de MàJ, Quiescence fonction	Tous fils suspendus	Redef. classes et méthodes, MàJ tas	
Rubah			Point de MàJ	Tous fils suspendus	MàJ tas (ou début)	
ActiveContext					Changement contexte principal	

FIGURE 4.1 – Cycles de vie des plates-formes étudiées (partie 1)

encore Argus [13] (dans la table en annexe A.1) guident l'exécution du programme vers l'altérabilité en bloquant une partie des requêtes entrant dans les sections modifiées du programme.

Il est à noter que lorsque la plate-forme met à jour des éléments cloisonnés et compartimentés du programme (processus pour ProteOS, composant pour K42), il est possible que seules des parties du programme soient suspendues. Dans les autres cas, c'est tout le programme qui est suspendu. On remarque également la même distinction entre les plates-formes qui guident l'exécution du programme vers l'altérabilité. Lorsque le programme suit une architecture à base de composant, il est plus facile de bloquer l'accès à certains composants pour obtenir leur quiescence (ou celle d'un autre composant), comme décrit par Kramer et al. [47].

Assez peu de plates-formes accomplissent des tâches une fois l'exécution reprise. Il s'agit généralement de la migration progressive des données comme pour Ginseng [58], du guidage de l'exécution après un redémarrage complet comme pour Kitsune [44] ou encore Rubah [63].

On peut donc constater un lien entre l'architecture des plates-formes et les premières étapes de leur cycle de vie tandis que les dernières étapes sont plutôt liées aux mécanismes qu'elles emploient. Une analyse plus détaillée des synergies entre les différentes colonnes des tables fait l'objet du prochain chapitre de ce manuscrit.

4.2 Architecture

L'architecture d'une plate-forme regroupe les choix de conception qui la caractérisent dans sa capacité à mettre à jour les programmes dynamiquement. Par exemple, définir une unité de mise à jour impacte directement les modifications qui peuvent être effectuées. Si l'unité de mise à jour est *un composant entier*, cela implique que la moindre modification sur un composant obligera à le mettre à jour dans son intégralité (généralement, en le remplaçant par sa nouvelle version).

Les tables 4.3 et 4.4 récapitulent les choix de conception des neufs plates-formes. Comme pour le cycle de vie, les choix de conceptions font l'objet d'une abstraction adaptée à la comparaison des plates-formes.

Plate-forme	Reprise	Attente terminaison	Détection terminaison	Nettoyage	Nouvelle version
Kitsune	Redémarrage fils	Guidage exéc.	Passage au point MàJ		
ProteOS	Arrêt vieux proc, Démarrage nouveau proc				
Ksplice	Reprise				
OPUS	Reprise fils			Détach. gest.	
K42	Déblocage fils			Détach. *mediators*	
Ginseng	Reprise fils				Transf. progressive des données
Jvolve	Reprise fils				
Rubah	Redémarrage fils	Guidage exéc.	Passage au point MàJ		(Transf. progressive données)
ActiveContext		Synchro. des données	Plus de fils dans l'ancien contexte	Suppression du vieux contexte	

FIGURE 4.2 – Cycles de vie des plates-formes étudiées (partie 2)

4.2.1 Choix de conception

Altérabilité

Qui définit les critères ? Dans la majorité des cas, les critères d'altérabilité sont définis par la plate-forme à partir des mises à jour typiques. Une plate-forme n'autorisant que la modification de fonctions comme OPUS définira ses critères d'altérabilité comme *la quiescence des fonctions modifiées*. Dans d'autres cas, c'est le développeur du programme qui les définit en plaçant des points de mise à jour.

Qui surveille ? Il s'agit ici de définir quelle entité est responsable de la surveillance de l'altérabilité. Cela peut être le gestionnaire de mise à jour ou encore la machine virtuelle sur laquelle s'exécute le programme.

Évaluation des critères Les critères d'altérabilité peuvent être évalués dynamiquement, pendant le processus de mise à jour ou statiquement, avant que le programme ne commence son exécution. Parcourir les piles d'exécution pour évaluer la quiescence d'une fonction est une évaluation dynamique d'un critère d'altérabilité. Lancer une mise à jour quand le programme a atteint un point de mise à jour est une évaluation statique du critère *atteindre un point de mise à jour*.

Contrôle du programme et de la mise à jour

Nature du gestionnaire de mise à jour Il s'agit ici de définir quelle entité remplit le rôle de gestionnaire de mise à jour. Il est possible pour une plate-forme d'avoir plusieurs gestionnaires pour des tâches différentes. Par exemple, K42 utilise plusieurs *mediators* pour remplacer ses composants.

Portée du (des) gestionnaire(s) La portée d'un gestionnaire est l'ensemble des éléments qu'il peut contrôler ou modifier. Dans la plupart des plates-formes, le gestionnaire a une portée globale (il peut contrôler tous les éléments). Dans le cas de K42, par exemple, chaque *mediator* ne peut contrôler que le composant auquel il est attaché et les fils d'exécution qui entrent dans ce composant.

		Altérabilité			Patch dynamique		
Plate-forme	Langage ciblé	Définit	Surveille	Évaluation	Script de contrôle	Nouveau code	Format fourni
Kitsune	C	Dév. Programme	-	Statique	Plate-forme + Dév. MàJ	Dév. MàJ	Nouveau code + traverseur de tas
ProteOS	OS (Minix)	Plate-forme + Dév. MàJ	Introspection dynamique	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Nouveau code + *transformers* + *state filters*
Ksplice	OS	Plate-forme	Gestionnaire	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Nouveau code + code supplémentaire
OPUS	C	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Nouveau code
K42	OS	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Nouveaux composants
Ginseng	C	Dév. Programme	Gestionnaire	Statique + Dynamique	Plate-forme	Dév. MàJ	Nouveau code
Jvolve	Java	Plate-forme	Gestionnaire	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Nouveau code + *transformers* + Critères d'alt.
Rubah	Java	Dév. Programme	-	Statique	Plate-forme + Dév. MàJ	Dév. MàJ	Nouveau code + Classe de mise à jour
ActiveContext	Smalltalk	-	-	-	Plate-forme	Dév. MàJ	Nouveau code + fonctions de transfert

FIGURE 4.3 – Architecture des plates-formes étudiées (partie 1)

Durée de vie du (des) gestionnaire(s) Le gestionnaire peut avoir une durée de vie illimitée : il est exécuté en parallèle du programme et n'est jamais arrêté. Il peut également être actif sur une courte période uniquement. C'est le cas d'OPUS qui attache son gestionnaire (un processus extérieur) au programme pour appliquer les mises à jour avant de le détacher.

Peut-on modifier le(s) gestionnaire(s) ? D'une manière générale, il n'est pas prévu de modifier le gestionnaire car il utilise les mêmes mécanismes tout au long de la vie du programme. Dans certains cas, le gestionnaire peut être modifié pour le mettre à jour.

Granularité des mises à jour

Unité de mise à jour L'unité de mise à jour est le plus petit élément du programme qui peut être mis à jour. Elle dépend souvent du langage ciblé par la plate-forme : par exemple, les langages orientés objets auront *une classe* comme unité de mise à jour.

Plusieurs versions peuvent-elles coexister ? Certaines plates-formes comme ActiveContext permettent à plusieurs versions de coexister dans le programme. C'est un choix de conception impactant les mécanismes employés par la plate-forme : il faut prévoir des mécanismes permettant aux versions de coexister de manière cohérente (comme la synchronisation des données dans ActiveContext).

Cloisonnement des versions Lorsque plusieurs versions coexistent, elles sont cloisonnées à un environnement donné : un composant, un fil d'exécution (c'est le cas dans ActiveContext). Ce choix est effectué par la plate-forme.

	Granularité			Gestionnaire			
Plate-forme	Unité de MàJ	Multi. Vers	Cloisonnement des vers.	Nature	Portée	Durée de vie	Modifiable
Kitsune	Tout le code	Non	-	Driver	Globale	Permanent	Non
ProteOS	Processus	Non	-	Processus spécial	Globale	Permanent	Oui (MàJ dynamique)
Ksplice	Fonction	Non	-	Module noyeau	Globale	Permanente	Non
OPUS	Fonction	Non	-	Proc. externe	Globale	Une MàJ	Non
K42	Composant	Oui	Composant	*mediator* (composant)	Un composant	Une MàJ	Non
Ginseng	Fonction, Type	Oui (mais pas utilisables)	Variable	Bibliothèque chargée dans le prog.	Globale	Permanente	Non
Jvolve	Classe, Méthode	Non	-	Machine Virtuelle	Globale	Permanente	Non
Rubah	Tout le code	Oui (mais pas utilisables)	Objet	Driver	Globale	Permanente	Non
ActiveContext	Contexte	Oui	Fil d'exécution	Machine Virtuelle	Globale	Permanente	Non

FIGURE 4.4 – Architecture des plates-formes étudiées (partie 2)

Patch dynamique

Qui définit le script de contrôle ? Le script de contrôle indique quels mécanismes doivent être utilisés sur quels éléments du programme. Il est généralement défini par la plate-forme. Parfois, le développeur de la mise à jour peut en définir une partie en indiquant des critères d'altérabilité supplémentaires par exemple.

Qui fournit le nouveau code ? Le nouveau code est fourni par le développeur de la mise à jour. Dans quelques rares cas, il peut être déjà présent dans le programme. Ce peut être le cas pour un programme utilisant la mise à jour dynamique pour adapter son comportement selon un algorithme de contrôle.

Que fournit le développeur de la mise à jour ? Il est rare que seul le nouveau code soit demandé au développeur de la mise à jour pour construire un patch dynamique. Il doit souvent fournir des morceaux de code comme par exemple des *transformers* : fonction dont l'exécution sur une donnée met à jour cette donnée.

4.2.2 Observations

Un premier constat est la dominance de certaines propriétés : pour la grande majorité des plates-formes, le ou les gestionnaires ont une portée globale, une durée de vie permanente et ne sont pas modifiables. Cela s'explique par l'approche habituelle de la mise à jour dynamique qui veut que chaque mise à jour soit appliquée par les mêmes mécanismes et de la façon la plus transparente possible. Un gestionnaire présent en permanence et pré-configuré pour utiliser systématiquement les mêmes mécanismes est donc suffisant. Avoir une portée globale permet d'accéder et de modifier plus facilement les éléments

du programme. C'est pour la même raison qu'une majeure partie des plates-formes fixe elle-même le script de mise à jour, parfois en tenant compte de quelques instructions du développeur de mise à jour, et que seul le nouveau code accompagné de *transformers* est attendu de ce dernier.

La nature du gestionnaire est corrélée avec le langage ciblé par les plates-formes : dans le cas de langages reposant sur une machine virtuelle (Java, Caml), le gestionnaire est généralement une partie de cette machine virtuelle. Dans le cas de systèmes d'exploitation, le gestionnaire est généralement une partie du programme et dans le cas du langage C, le gestionnaire est soit une bibliothèque chargée dans le programme, soit un processus externe.

On remarque également qu'il n'est pas impossible pour une plate-forme ciblant un langage particulier d'adopter une architecture habituellement caractéristique des plates-formes ciblant un autre langage. Par exemple, Rubah [63] utilise des points de mise à jour placés par le développeur de mise à jour (ce qui implique une évaluation statique de l'altérabilité), ce qui est assez inhabituel pour le langage Java et plutôt typique du C.

4.3 Mécanismes de mise à jour

Pour chaque tâche de mise à jour (modification des fonctions, accès aux données, ...), les plates-formes utilisent un ou plusieurs mécanismes. Les tables 4.5 et 4.6 récapitulent les mécanismes employés pour chaque tâche de mise à jour par les plates-formes. Comme précisé dans le chapitre précédent, les mécanismes sont abstraits afin de permettre la comparaison d'implémentations différentes ayant des propriétés équivalentes.

	Altérabilité	Accès et Transformation des données		Patch dynamique		Gestion des Erreurs	
Plate-forme	Scrutation	Accès	Transformation	Chargement	Construction	Cohérence des Types	Récupération et Erreurs
Kitsune	-	Immédiat	Instantanée	Bibliothèque dynamique	Chaîne de compil. spéciale	-	-
ProteOS	Introspection dynamique	Immédiat	Instantanée	Mécanisme de chargement de processus	Compil. spécifique (LLVM)	-	Erreurs en MàJ cause retour arrière
Ksplice	Parcours de la pile	-	-	Chargement de module noyau	Options imposées (compil. libre)	-	-
OPUS	Parcours de la pile	-	-	Bibliothèque dynamique	Chaîne de compil. spéciale	Vérification des effets de bord	-
K42	Comptage des fils	Immédiat	*factory* : instant. *compo.* : Ret.	Chargeur de module spécial	Compilation vers un format spécial	-	-
Ginseng	-	Progressif	Instantané	Bibliothèque dynamique	Chaîne de compil. spéciale	Fonctions de vérif. des variables	-
Jvolve	Inspection pile aux points sûr MV.	Immédiat	Instantanée	Chargement dynamique des classes	Chaîne de compil. spéciale	-	-
Rubah	-	Progressif ou Immédiat	Instantanée	Chargement dynamique des classes	Chaîne de compil. spéciale	-	-
ActiveContext	-	Progressif	Instantanée	Chargement dynamique	Écriture de code mettant à jour	-	-

FIGURE 4.5 – Mécanismes employés par les plates-formes étudiées (partie 1)

	Flot d'exécution				Modification des données		Gestionnaire
Plate-forme	Multi. Vers.	Synchro. des Vers.	MàJ des fonctions	Redémarrage	MàJ des Types	Transf. données	Contrôle exec.
Kitsune	-	-	Chargement du code	Tous les fils redémarrés	Chargement du code	Script Dév. MàJ	Oui (guidage après redémarrage)
ProteOS	-	-	Chargement du code	Démarrage d'un nouveau processus	chargement du code	*transformers* appliqué à une copie	Non
Ksplice	-	-	Indirrection (jump)	Non	Script Dév. MàJ	Script Dév. MàJ	Non
OPUS	-	-	Indirrection (jump)	Non	-	-	Non
K42	MàJ parallèle des composants	-	-	Non	Remplacement des *factory*	Remplacement par indirection	Oui (blocage de fils)
Ginseng	MàJ progressive des données	-	Indirection (pointeurs)	Non	Chargement du code	*transformers* générés	Non
Jvolve	-	-	Indirection ou Remplacement sur la pile	Non	Rechargement de classe ou modif. champs	*transformer* appliqué à une copie	Non
Rubah	MàJ progressive des données	-	Chargement du code	Tous les fils redémarrés	Chargement du code	*transformers*	Oui (guidage après redémarrage)
ActiveContext	Contextes	fonction de transfert	Chargement du code	Non	Chargement du code	Fonction de transfert	Non

FIGURE 4.6 – Mécanismes employés par les plates-formes étudiées (partie 2)

4.3.1 Tâches de mise à jour

Altérabilité

Mécanisme de scrutation Plusieurs mécanismes permettent d'évaluer les critères d'altérabilité. Ils sont choisis en fonction des critères d'altérabilité définis. Par exemple, lorsqu'il s'agit de surveiller la quiescence de fonction, il est fréquent que le mécanisme employé soit l'inspection des piles d'exécution.

Accès et mise à jour des données

Stratégie d'accès La stratégie d'accès aux données peut être soit immédiate si les données sont accédées au moment d'appliquer les modifications, soit progressive si les données sont accédées plus tard, lorsque le programme a repris son exécution.

Moment de transformation Le moment de transformation peut être soit instantané si les données sont transformées dès qu'elles sont accédées, soit retardé si les données sont transformées plus tard.

Gestionnaire de mise à jour

Contrôle de l'exécution du programme Le gestionnaire de mise à jour peut contrôler l'exécution du programme pour le guider vers un état facilitant la mise à jour. C'est le cas dans K42 où les *mediators* guident l'exécution du programme vers la quiescence des composants à remplacer.

Génération et chargement du patch dynamique

Chargement du nouveau code Le mécanisme employé pour charger le nouveau code dans la mémoire du programme varie en fonction du langage du programme. Quand le programme est compilé en C (ou C++), le nouveau code est souvent chargé sous forme d'une bibliothèque dynamique.

Construction du patch dynamique Le patch dynamique est construit à partir du nouveau code et d'un script de mise à jour, parfois accompagné de suppléments fournis par le développeur de la mise à jour. Il s'agit ici d'identifier par quel procédé le patch dynamique est construit.

Contrôle et modification du flot d'exécution

Coexistence de plusieurs versions Il s'agit ici d'identifier les mécanismes permettant la coexistence de plusieurs versions. Le système de contexte d'ActiveContext est un exemple de mécanisme remplissant cette fonction.

Synchronisation des versions Lorsque plusieurs versions coexistent, elles partagent généralement des données qu'il faut synchroniser. Il existe plusieurs mécanismes permettant d'assurer cela.

Mise à jour des fonctions Il s'agit ici d'identifier les mécanismes employés pour mettre à jour les fonctions.

Redémarrage du programme Certaines plates-formes utilisent le redémarrage du programme (ou d'une partie du programme) pour réinitialiser les piles d'exécution du programme ou pour recharger le code de la dernière version du programme. Cela s'accompagne généralement de méthodes pour sérialiser l'état du programme et pour en accélérer le redémarrage.

Modification des données

Mise à jour des types Il s'agit ici d'identifier les mécanismes employés pour mettre à jour les types.

Transformation des données La transformation des données est généralement assurée par un *transformer* fourni par le développeur de la mise à jour ou généré par une analyse statique du code de la nouvelle version. Il existe cependant d'autres méthodes.

Gestion des erreurs

Vérification de la cohérence des types Certaines plates-formes vérifient la cohérence des types à chaque mise à jour pour s'assurer qu'une mise à jour ne rendra pas le typage du programme inconsistant. C'est le cas de Ginseng qui utilise des fonctions de vérification pour assurer que les variables aient toujours le type le plus récent.

Retours en arrière et récupération des erreurs Certains mécanismes permettent d'inverser une mise à jour, soit pour récupérer d'une erreur ou d'un état inconsistant, soit pour annuler une mise à jour temporaire. Ces mécanismes vont souvent de pair avec des mécanismes vérifiant la cohérence des types.

4.3.2 Observations

Une première observation est que, si certains mécanismes sont plus répandus dans les plates-formes ciblant un type donné de langage, il n'y a pas de lien fort entre mécanismes et langage ciblé. En effet, si leurs implémentations diffèrent, les mêmes mécanismes peuvent être utilisés pour modifier des programmes dans différents langages. Jvolve [72] et Ksplice [4] parcourent toutes les deux les piles d'exécution pour vérifier la quiescence de fonctions, elles utilisent également de l'indirection pour mettre à jour les fonctions.

Il est à noter que la plupart des plates-formes utilisent une chaîne de compilation spécifique pour générer les patchs dynamiques à partir du nouveau code et des instructions fournis par le développeur de la mise à jour. Cela s'explique par la volonté de rendre les mises à jour aussi transparentes que possible. La plate-forme a donc besoin d'analyser le nouveau code pour identifier quelles modifications sont à appliquer. De même que pour les architectures et cycles de vie des plates-formes, les synergies entre mécanismes et l'analyse des propriétés des plates-formes fait l'objet du prochain chapitre.

4.4 Quelques cas d'étude

Pour illustrer le procédé d'analyse des plates-formes qui a permis d'obtenir les tables 4.1 à 4.6, cette section détaille l'analyse des plates-formes Kitsune et ProteOS, présentes dans les tables de ce chapitre.

4.4.1 Kitsune

```
void main(){
  if (! kitsune_updating()){
    setup_server();
  }
  while (true){
    if (!kitsune_updating_from("B")){
      kitsune_update("A");
      handle_command();
    }
    kitsune_update("B");
  }
}
```

FIGURE 4.7 – Exemple (simplifié) de programme utilisant Kitsune

Kitsune [44] est une plate-forme ciblant les programmes écrits en C, en particulier les serveurs constitués d'une boucle principale appelant différentes fonctions au fil des requêtes qu'ils reçoivent.

La particularité de Kitsune est qu'elle redémarre le programme à chaque mise à jour. Après avoir chargé le nouveau code, Kitsune transforme les données puis redémarre tous les fils d'exécution. Pour que ce redémarrage soit le plus rapide possible, il est demandé au développeur du programme de placer des instructions de guidage du programme permettant de passer les étapes d'initialisation du programme. La figure 4.7 montre ces instructions. Il s'agit de disjonctions *if* qui permettent, lors d'un redémarrage, d'atteindre le point qui a déclenché la mise à jour. Sur la figure 4.7, deux points `A` et `B` sont placés avant et après un appel à la fonction `handle_command`. Lors d'un redémarrage après une mise à jour déclenchée par le point `A`, `kitsune_updating()` retourne `true` et `kitsune_updating_from("B")` retourne `false`. L'exécution du programme est donc guidée jusqu'au point `A` par les disjonctions placées dans le code.

Cycle de vie

②, ③ : (1re Préparation - Attente altérabilité)	-
③ → ④ : (Détection altérabilité)	Point de MàJ
④ : (Suspension)	Tous fils suspendus
⑤ : (Modification)	MàJ du tas
⑥ : (2e Préparation)	-
⑦ : (Reprise)	Redémarrage de tous les fils
⑧ : (Attente terminaison)	Guidage des fils
⑧ → ⑨ : (Détection terminaison)	Passage au point de MàJ
⑨, ⑨ → : (Nettoyage - Nouvelle version)	-

FIGURE 4.8 – Cycle de vie dans Kitsune

Avant de lancer le programme, il est demandé à son développeur de placer des points de mise à jour aux endroits où le programme est globalement quiescent. Le cœur de cible de Kitsune étant des programmes constitués d'une boucle principale appelant différentes fonctions, ces points de mise à jour sont généralement placés entre chaque appel de fonction, dans la boucle principale. Il est également demandé au développeur du programme de placer des instructions permettant de guider l'exécution du programme, lors de son redémarrage, vers le point de mise à jour dont l'atteinte a déclenché la mise à jour du programme. Ces instructions sont des conditions utilisant des fonctions de Kitsune indiquant si le programme redémarre suite à une mise à jour et quel point de mise à jour était atteint lorsque les fils d'exécution ont été redémarrés.

Lorsqu'une mise à jour est invoquée et qu'un point de mise à jour est atteint, le nouveau code est chargé dans la mémoire du programme, tous les fils d'exécution sont suspendus et le tas du programme est mis à jour en exécutant un programme de mise à jour des données fourni par le développeur de la mise à jour. Quand toutes les données ont été mises à jour, tous les fils d'exécutions sont redémarrés. La plate-forme guide alors l'exécution du programme jusqu'à ce qu'il atteigne le point de mise à jour où il a été redémarré. Le programme étant entièrement redémarré lorsqu'il atteint un point de mise à jour, il est nécessaire que le développeur place ces points aux moments où le programme peut être interrompu sans causer d'erreur.

Ce cycle de vie apparaît dans la table 4.8. Kitsune n'effectue aucune action lors de la première étape de préparation ou lors de l'attente de l'altérabilité. L'altérabilité est détectée par l'atteinte d'un point de mise à jour. Pendant l'étape de suspension, tous les fils sont suspendus avant de mettre à jour les données pendant l'étape de modification. Aucune préparation n'est effectuée avant de redémarrer tous les fils pendant l'étape de reprise. Pendant l'étape d'attente de terminaison, l'exécution des fils est guidée vers le point de mise à jour de départ. Son atteinte marque la terminaison de la mise à jour. Kitsune n'effectue aucune action pendant l'étape de nettoyage ou lorsque l'exécution continue dans la nouvelle version.

Architecture

Langage ciblé	**C**		
Altérabilité		*Granularité*	
Définit	**Dév. programme**	Unité de MàJ.	**Tout le code**
Surveille	**-**	Multi-version	**Non**
Évaluation	**Statique**		
Gestionnaire		*Patch Dynamique*	
Nature	**Driver**	Script contrôle	**Plate-forme + Dév. MàJ..**
Portée	**Globale**	Nouveau code	**Dév. MàJ.**
Durée de vie	**Permanente**	Format fourni	**Nouveau code + Traverseur de tas**
Modifiable	**Non**		

FIGURE 4.9 – Architecture de Kitsune

Le critère d'altérabilité pour chaque mise à jour est l'atteinte d'un point de mise à jour. En plaçant ces points de mise à jour, le développeur du programme définit le critère d'altérabilité des futures mises à jour (dans la figure 4.7, il s'agit des points nommés A et B). Les points de mise à jour sont des appels à une fonction spécifique de Kitsune déclenchant une mise à jour. Aucune entité ne surveille l'altérabilité, c'est l'appel à cette fonction qui constitue la détection de l'altérabilité. L'altérabilité est donc définie statiquement.

Le programme est lancé au travers d'un programme externe nommé *driver* qui charge le code du programme et lance son exécution. Ensuite, il remplit le rôle de gestionnaire : il charge le nouveau code, met à jour les données et redémarre le programme à chaque mise à jour. Mettant à jour l'intégralité des données et redémarrant tous les fils d'exécution, le gestionnaire a une portée globale. Il n'est jamais arrêté, sa durée de vie est donc permanente. Comme pour la majorité des plates-formes, le gestionnaire n'est pas modifiable.

À chaque mise à jour, Kitsune redémarre tous les fils d'exécution. Le programme de mise à jour des données, donné par son développeur, parcourt le tas pour accéder aux données et les transformer. Chaque modification implique une mise à jour complète du programme, la granularité de Kitsune est donc l'intégralité du code.

Comme une majorité de plates-formes, Kitsune n'a pas de mécanisme permettant la coexistence de plusieurs versions. À chaque mise à jour, le programme est redémarré dans sa version la plus récente.

Le script de mise à jour est établi par la plate-forme qui décide des mécanismes à employer et par le développeur de la mise à jour qui donne un programme de mise à jour des données écrit dans un langage spécifique. Il s'agit d'un *traverseur de tas* dont le rôle est de parcourir le tas pour accéder aux données et les mettre à jour. Le nouveau code est fourni par le développeur de la mise à jour. Il est donc attendu du développeur des mises à jour, le code de la nouvelle version du programme ainsi que le programme de mise à jour des données.

La figure 4.10 présente un aperçu de l'architecture de Kitsune. Elle montre comment les patchs dynamiques sont compilés à partir du nouveau code et du traverseur de tas fourni par le développeur de

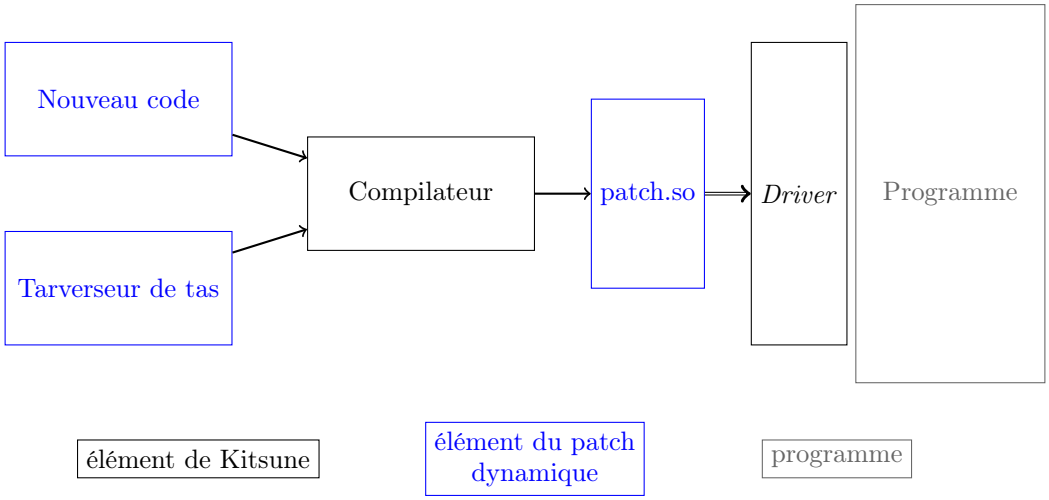


FIGURE 4.10 – Compilation et soumission d'un patch dynamique dans Kitsune

Altérabilité	
Scrutation	**-**
Accès et Transf. données	
Accès	**Immédiat**
Transformation	**Instantanée**
Gestionnaire	
Contrôle d'exécution	**Guidage après redémarrage**
Patch dynamique	
Chargement	**Bib. dynamique**
Construction	**Compil. spéciale**
Flot d'exécution	
Multi-version	**-**
Sync. versions	**-**
MàJ. des fonctions	**Chargement de code**
Redémarrage	**Tous les fils**
Modification des données	
MàJ des types	**Chargement de code**
Tranf. des données	**Script Dév. MàJ.**
Gestion des erreurs	
Cohérence des types	**-**
Récupération et erreurs	**-**

FIGURE 4.11 – Mécanismes de Kitsune

mise à jour avant d'êtres envoyés au *driver*.

Mécanismes

Comme indiqué dans l'analyse de son architecture, Kitsune n'utilise aucun mécanisme pour scruter l'altérabilité du programme. Le programme de mise à jour des données traverse le tas immédiatement lors de l'étape de modification et transforme les données accédées instantanément. Le gestionnaire contrôle l'exécution du programme lors du redémarrage des fils d'exécution. Tous les fils d'exécution sont redémarrés lors de l'étape de reprise.

Le nouveau code est compilé en une bibliothèque dynamique par une chaîne de compilation spéciale, en même temps que le programme de mise à jour des données est compilé vers un format exécutable. Le nouveau code est chargé en utilisant les fonctionnalités de chargement dynamique du langage C.

Les types et les fonctions sont mises à jour en chargeant le nouveau code. Les nouvelles fonctions et les nouveaux types sont chargés à côté de leurs anciennes versions. C'est en redémarrant les fils d'exécution que les nouvelles fonctions sont exécutées. Les données sont mises à jour par le programme spécifique donné par le développeur de mise à jour.

Kitsune n'utilise aucun mécanisme pour vérifier la cohérence des types lors d'une mise à jour ou récupérer d'une éventuelle erreur.

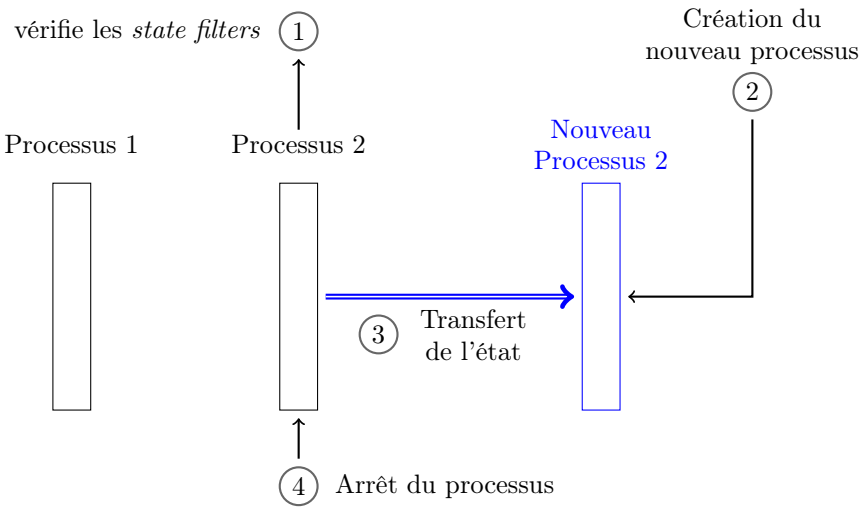


FIGURE 4.12 – Fonctionnement de ProteOS

4.4.2 ProteOS

ProteOS [36] est un système d'exploitation basé sur MINIX. Son architecture est orientée processus. Les processus de ProteOS communiquent par communication inter-processus (*IPC*) et peuvent être mis à jour dynamiquement. Chaque processus est constitué d'une boucle permanente répondant aux requêtes *IPC* qu'il reçoit.

La particularité de cette plate-forme est qu'elle permet aux développeurs des mises à jours de préciser des *state filters*. Il s'agit de critères d'altérabilité exprimés sous la forme de formules arithmétiques décrivant un état du programme (par exemple, *3 opérations d'écriture sont en attente*).

La figure 4.12 montre comment ProteOS applique les mises à jour aux processus. Lorsque les *state filters* sont vérifiés, une copie (à jour) du processus est crée et l'état de l'ancien processus lui est transféré.

Cycle de vie

② : (1re Préparation)	Envoi des *state filers* aux proc.
③ : (Attente altérabilité)	-
③ → ④ : (Détection altérabilité)	Début de boucle + *state filters*
④ : (Suspension)	Processus suspendus
⑤ : (Modification)	Créa. nouveau proc + transfert de l'état
⑥ : (2e Préparation)	Vérification cohérence de l'état
⑦ : (Reprise)	Arrêt vieux proc. Démarrage nouveau proc.
⑧, ⑨, ⑨ → : (Attente term. → Nouvelle vers.)	-

FIGURE 4.13 – Cycle de vie dans ProteOS

Lorsque la mise à jour d'un processus est demandée, les *state filters* donnés par le développeur de la mise à jour sont envoyés aux processus qui sont mis à jour, pendant la première étape de préparation. Ces *state filters* sont évalués à chaque début de la boucle principale de chaque processus. Lorsqu'ils sont vérifiés, l'altérabilité est détectée. Les processus à mettre à jour sont alors suspendus pendant l'étape de suspension. Des clones de ces processus sont alors créés pendant l'étape de modification et l'état des processus est recopié depuis leur ancienne version. Lors de la deuxième étape de préparation, l'état des deux versions de chaque processus est vérifié. Si l'état d'un processus contient des erreurs (par exemple, une corruption de mémoire), la mise à jour est annulée et l'ancien processus est conservé. Sinon, le nouveau processus est démarré pendant l'étape de reprise, alors que l'ancien processus est arrêté. La mise à jour est terminée une fois l'étape de reprise passée, aucune action n'est effectuée lors des étapes suivantes.

Architecture

Langage ciblé	**OS (Minix)**		
	Altérabilité		*Granularité*
Définit	**Plate-forme + Dév. MàJ**	Unité de MàJ.	**Processus**
Surveille	**Introspection dynamique**	Multi-version	**Non**
Évaluation	**Dynamique**		
	Gestionnaire		*Patch Dynamique*
Nature	**Processus spécial**	Script contrôle	**Plate-forme + Dév. MàJ.**
Portée	**Globale**	Nouveau code	**Dév. MàJ.**
Durée de vie	**Permanente**	Format fourni	**Nouveau code + *transformers* + *state filters***
Modifiable	**Non**		

FIGURE 4.14 – Architecture de ProteOS

Les critères d'altérabilité sont les *state filters* donnés par le développeur de la mise à jour ainsi que l'atteinte d'un début de boucle par le processus à mettre à jour. Les critères sont donc donnés par la plate-forme (atteinte d'un début de boucle) et par le développeur de la mise à jour (*state filters*. Ces critères sont évalués dynamiquement par un mécanisme d'introspection dynamique de ProteOS, indépendant du gestionnaire.

Le gestionnaire de ProteOS est un processus spécial qui peut être mis à jour dynamiquement, au même titre que les autres processus. Il a une portée globale (il peut mettre à jour tous les processus de ProteOS) et une durée de vie permanente (il est en permanence présent dans l'OS).

Comme précisé en début de sous-section, l'unité de mise à jour de ProteOS est un processus. Chaque modification, même partielle d'un processus entraînera la mise à jour complète de ce dernier. Il n'est pas possible dans ProteOS d'avoir plusieurs versions en même temps, lors de l'étape de reprise, les vieux processus sont arrêtés et les nouveaux sont démarrés.

Le script de contrôle des mises à jour est donné conjointement par la plate-forme qui précise comment mettre à jour les processus et par le développeur des mises à jour qui précise les *state filters* pour chacune d'entre elle.

Pour chaque mise à jour, le développeur doit donner le nouveau code, et les *state filters* en plus des *transformers* utilisés pour mettre à jour les données des processus mis à jour.

Mécanismes

	Altérabilité		*Flot d'exécution*
Scrutation	**Introspection dynamique**	Multi-version	**-**
	Accès et Transf. données	Sync. versions	**-**
Accès	**Immédiat**	MàJ. des fonctions	**Chargement de code**
Transformation	**Instantanée**	Redémarrage	**Démarrage nouveau proc.**
	Gestionnaire		*Modification des données*
Contrôle d'exéc.	**Non**	MàJ des types	**Chargement de code**
		Tranf. des données	***transformer* sur copie**
	Patch dynamique		*Gestion des erreurs*
Chargement	**Chargement de proc.**	Cohérence des types	**-**
Construction	**Compil. spécifique (LLVM)**	Récupération et erreurs	**Retour en arrière si erreur**

FIGURE 4.15 – Mécanismes de ProteOS

Les critères d'altérabilités sont évalués par introspection dynamique de l'état de l'OS et de ses processus. Cette introspection utilise l'instrumentation de code ajoutée au programme par LLVM [2], le compilateur utilisé par ProteOS.

Les données sont transférées du vieux processus au nouveau lors de l'étape de modification, elles sont transformées à ce moment. L'accès est donc immédiat et la transformation est instantanée. Les données sont mises à jour en y appliquant des *transformers* fournis par le développeur de la mise à jour.

2. LLVM est une infrastructure de compilation sur laquelle s'appuient différents compilateurs comme Clang, compilateur pour C,C++ et Objective C. http://llvm.org/

ProteOS fournit des mécanismes spécifiques permettant de charger un nouveau processus dans l'OS. Ce mécanisme est considéré comme le mécanisme employé par ProteOS pour charger le nouveau code. Le code source du processus est compilé par LLVM, en activant l'ajout des données d'introspection.

Les fonctions et les types sont mis à jour en chargeant le code du nouveau processus. Le nouveau processus est relié au reste de l'OS par indirection.

Comme c'est un nouveau processus qui est démarré à la place de l'ancien lors de l'étape de reprise, il est considéré que ProteOS redémarre ses processus pour les mettre à jour. Les IPC se font au travers de points d'entrée virtuels liés aux processus communiquant. Lors de la mise à jour d'un processus, les point d'entrée virtuels liés à l'ancienne version sont liés à la nouvelle version.

Enfin, ProteOS utilise des mécanismes d'introspection pour détecter des erreurs pendant l'application des mises à jour, afin de les annuler si des erreurs se produisent (corruption mémoire lors du transfert, impossibilité de vérifier les *state filters*, ...)

4.5 Bilan

L'analyse des plates-formes montre une variété de mécanismes et de choix architecturaux. Les tables de l'annexe A répertorient 72 mécanismes et 59 choix architecturaux différents. Elle permet d'identifier les différents mécanismes de l'état de l'art et fournit de premiers résultats quant à la façon dont ils peuvent être combinés. La combinaison la plus fréquente étant l'inspection des piles d'exécution et la redéfinition de fonction par indirection. Le premier mécanisme permet de détecter la quiescence des fonctions redéfinies en préparation du second.

Un autre constat est que certains mécanismes ou certaines architectures sont plus adaptés à certains types de plate-forme, souvent par facilité d'implémentation. Par exemple, les plates-formes pour programmes Java emploient généralement la fonctionnalité *hotswap* de la JVM standard pour redéfinir les fonctions par indirection. Pourtant, il est possible de remarquer des plates-formes dont l'architecture ou les mécanismes ne sont pas habituels compte tenu du programme qu'elles ciblent. Par exemple, Rubah [63] utilise des points de mise à jour alors qu'elle cible des programmes Java. D'une manière générale, les points de mise à jour sont utilisés par les plates-formes pour programmes C tandis que les plates-formes Java détectent l'altérabilité dynamiquement par introspection.

Ayant observé la diversité des combinaisons de mécanismes et d'éléments architecturaux des plates-formes de l'état de l'art, deux questions se posent : quelles sont les synergies entre ces aspects des plates-formes ? et quelles sont les principales familles de plates-formes ? Répondre à la première question donne des indications sur la manière dont une plate-forme configurable doit fournir les mécanismes. Répondre à la seconde question identifie des archétypes de stratégie de mise à jour qu'une telle plate-forme doit supporter.

Cette observation conforte la nouvelle approche de deux façons. Premièrement, si pour un type de programme donné, plusieurs stratégies de mises à jour sont envisageables, disposer d'une plate-forme proposant toutes ces stratégies permet à chaque mise à jour de choisir la stratégie la plus adaptée. Le choix de la plate-forme utilisée n'impose plus une seule stratégie qui peut s'avérer inadaptée ou incompatible avec certaines mises à jour. C'est d'autant plus préférable, que la plate-forme doit généralement être choisie au moment de lancer le programme, alors qu'il n'est pas possible de prévoir quelles seront les futures mises à jour.

Deuxièmement, à condition d'en adapter l'implémentation, il est possible d'utiliser certains mécanismes ou d'adopter certaines architectures quelque soit le type de programme ciblé par la plate-forme. Par exemple, Kitsune parcourt le tas du programme en partant des variables globales pour accéder immédiatement aux données tandis que Jvolve repose sur le ramasse-miettes de la JVM. Il est donc envisageable d'exprimer les mises à jour en utilisant un même langage quelque soit le type de programme ciblé. Mettre à jour une application complexe composée de plusieurs programmes de types différents devient alors plus simple car les mises à jours de chaque programme sont exprimées suivant la même logique.

Chapitre 5

Analyses statistiques

L'objectif de ce chapitre est d'identifier d'éventuelles synergies entre les propriétés des plates-formes. C'est-à-dire, identifier des mécanismes ou des choix de conception qui, lorsqu'ils sont fixés dans une plate-forme, ont tendance à fixer un autre choix ou mécanisme. Par exemple, l'accès immédiat aux données va souvent de pair avec un gestionnaire à portée globale. En effet, avoir une portée globale permet d'accéder plus simplement à l'ensemble des données du programme.

Dans ce chapitre, les choix de conception et les mécanismes sont appelés *propriétés des plates-formes*. Les combinaisons de ces propriétés sont étudiées dans ce chapitre en utilisant deux outils statistiques. La détection de motifs fréquents cherche dans les tables de l'annexe A quelles combinaisons de propriétés sont les plus fréquentes. Le *clustering* des plates-formes les regroupe en fonction de leurs propriétés communes et identifie des familles de plates-formes.

5.1 Une étude exploratoire des combinaisons de mécanismes

La nouvelle approche de la mises à jour dynamique défendue dans ce manuscrit accorde une grande importance à la combinaison et à la configuration de mécanismes. Concevoir une plate-forme adoptant cette approche nécessite de permettre aux développeurs de mises à jour de choisir quels mécanismes employer, de les combiner et de les configurer en fonction des besoins.

En recherchant les propriétés en synergie, il est possible d'anticiper les combinaisons de propriétés que la plate-forme doit permettre. Par exemple, la sous-section 5.2.2 montre une synergie entre stratégie d'accès aux données (mécanisme) et portée du gestionnaire (choix architectural). Si une plate-forme laisse le développeur de mise à jour choisir quelle stratégie d'accès employer mais impose la portée du gestionnaire, elle risque de limiter les possibilités offertes. Si la portée du gestionnaire est locale, comme dans K42 [8] où un gestionnaire différent est attaché à chaque composant, il sera plus difficile d'accéder immédiatement à toutes les données. C'est d'ailleurs pour palier à cela que les fabriques de K42 gardent une référence vers chaque composant existant. Si la plate-forme ne tient pas compte des synergies entre propriétés, elle risque de mal supporter certains des choix qu'elle permet. Dans le pire des cas, elle peut ne pas satisfaire les exigences de certains mécanismes.

Ce chapitre emploie une approche exploratoire pour identifier les synergies entre propriétés. Les tables de l'annexe A sont fusionnées en une base de données sur laquelle sont utilisés deux outils d'analyse statistiques. Dans un premier temps, les combinaisons fréquentes de propriétés sont recherchées et triées pour ne garder que les combinaisons qui indiquent de potentielles synergies. Dans un second temps, les plates-formes partageant les mêmes propriétés sont regroupées en familles dans l'objectif d'identifier des archétypes de combinaison. Les tables de l'annexe A ayant été remplies pour décrire précisément les propriétés des plates-formes, elles sont homogénéisées pour permettre une analyse automatique. La valeur de certaines cellules est changée afin d'abstraire certains détails. Deux cellules indiquant une même propriété ont alors la même valeur. Par exemple, deux plates-formes guidant l'exécution du programme vers son altérabilité peuvent avoir chacune la mention *Blocage des nouveaux appels* et *Blocage des fils entrants* dans la colonne *Attente de l'altérabilité*, la valeur de ces deux cellules est alors changée en *guidage de l'exécution*. La base de données homogénéisée est présentée en annexe A.4.

Le nouveau code est fourni par le Dév. de la mise à jour :	**35**	(83%)
Le Dév. MàJ fournit le nouveau code (et des transformers) :	**33**	(79%)
La plate-forme définit le script de mise à jour :	**32**	(76%)
Le gestionnaire a une portée globale :	**30**	(71%)
Le gestionnaire a une durée de vie permanente :	**27**	(64%)
Les données sont transformées instantanément :	**26**	(62%)
Une chaîne de compilation spéciale donne le patch dynamique :	**22**	(52%)
Les types sont mis à jour en chargeant le nouveau code :	**22**	(52%)
L'unité de mise à jour est intermédiaire (classe, composant, ...) :	**22**	(52%)
La plate-forme définit les critères d'altérabilité :	**22**	(52%)
C'est le gestionnaire qui surveille l'altérabilité :	**21**	(50%)
Tout le programme est suspendu pendant la mise à jour :	**21**	(50%)
Le gestionnaire est une partie du programme (code, fil, ...) :	**20**	(48%)
L'altérabilité est évaluée dynamiquement :	**19**	(45%)
Les données sont accédées immédiatement :	**19**	(45%)
Le langage ciblé est le C ou le C++ :	**18**	(43%)
Les fonctions sont mises à jour par indirection :	**17**	(40%)
Le langage ciblé s'exécute sur une machine virtuelle :	**15**	(36%)
Le critère d'altérabilité est la quiescence des éléments modifiés :	**14**	(33%)
Les fonctions sont mises à jour en chargeant le nouveau code :	**13**	(31%)
Les *transformers* sont appliqués directement sur les données :	**13**	(31%)
L'unité de mise à jour est petite (fonction, variable ...) :	**11**	(26%)
Des parties du programme sont mises à jour en jour parallèle :	**11**	(26%)
L'altérabilité est surveillée par introspection :	**11**	(26%)
Le gestionnaire contrôle l'exécution :	**10**	(24%)
Le critère d'altérabilité est l'atteinte d'un point de mise à jour :	**10**	(24%)

FIGURE 5.1 – Choix fréquents parmi les plates-formes étudiées

5.2 Détection des motifs fréquents

La base de donnée homogénéisée a été analysé à l'aide de l'algorithme APriori [1] qui cherche de manière exhaustive les combinaisons de valeurs dont le nombre d'occurrences est supérieur à un seuil donné.

Dans cette section, les combinaisons dont le nombre d'occurrences est supérieur ou égal à 10 (soit un taux d'occurrences de 24%) sont interprétées, utilisant les connaissances des mécanismes et de plates-formes accumulées lors de l'étude du domaine. Lorsqu'une combinaison fréquente s'explique par une synergie entre les propriétés qui la composent, celle-ci est présentée dans la sous-section 5.2.2. La sous-section 5.2.3 récapitule les synergies identifiées de cette manière.

5.2.1 Propriétés fréquentes

En préliminaire à l'analyse des combinaisons fréquentes, il est intéressant de repérer les propriétés qui sont fréquentes seules. Cela permet d'observer les tendances générales : quelles propriétés sont plus utilisées que d'autres. Une autre utilité est l'élimination de faux positifs lors de l'analyse des combinaisons. En effet, la fréquence de certaines combinaisons s'explique parfois juste par la fréquence des propriétés impliquées.

La figure 5.1 présente les propriétés les plus fréquentes. Pour 83% des plates-formes, c'est le développeur de la mise à jour qui donne le nouveau code, et dans 79% des cas, il ne fournit que le nouveau code (éventuellement accompagné de *transformers*). On remarque également que 76% des plates-formes fixent le script de mise à jour. Ces statistiques peuvent s'expliquer par l'approche usuelle de la mise à jour dynamique. Pour que le procédé de mise à jour dynamique reste le plus transparent possible, il est voulu que le développeur de la mise à jour ne fournisse que le code de la nouvelle version du programme, cela implique alors que le script de mise à jour est défini par la plate-forme. Cela peut également expliquer, dans certains cas, l'utilisation d'une chaîne de compilation spécifique pour générer un patch dynamique (52% des plates-formes) à partir du nouveau code. Le développeur de mise à jour fournissant peu d'informations, une analyse statique combinée à un procédé de compilation spécifique permet d'obtenir les informations nécessaires (par exemple, quelle fonction redéfinie quelle autre).

Dans 71% des cas, le gestionnaire a une portée globale et dans 64% des cas, il a une durée de vie permanente. En effet, lorsque l'unité de mise à jour n'est pas précisément cloisonnée (comme dans le cas d'un composant ou d'un processus), il est plus aisé d'avoir accès à l'intégralité des éléments du programme pour les modifier. Les mécanismes employés étant toujours les mêmes à chaque mise à jour, le gestionnaire n'a pas besoin d'être modifié et peut rester actif en permanence.

Il est également possible de remarquer une forte présence de certains choix tels que que la stratégie de transformation des données (instantanée, 62%) ou encore le mécanisme de mise à jour des types (chargement de code, 52%). La première statistique peut s'expliquer par une facilité d'implémentation. Transformer les données instantanément évite l'installation d'un *hook* pour retarder la transformation, économisant également le surcoût lié à son exécution. La deuxième statistique peut s'expliquer par le format exécutable du programme. En Java, les types sont présents dans le *bytecode* du programme et recharger le code permet de les mettre à jour. De plus, plusieurs JVM implémentent un *classloader* capable de recharger des classes pendant l'exécution du programme. En C, les types sont éliminés du binaire lors de la compilation. Mettre à jour un type revient à changer les parties du code qui utilisent ce type.

5.2.2 Combinaisons fréquentes de propriétés

Mise à jour des données

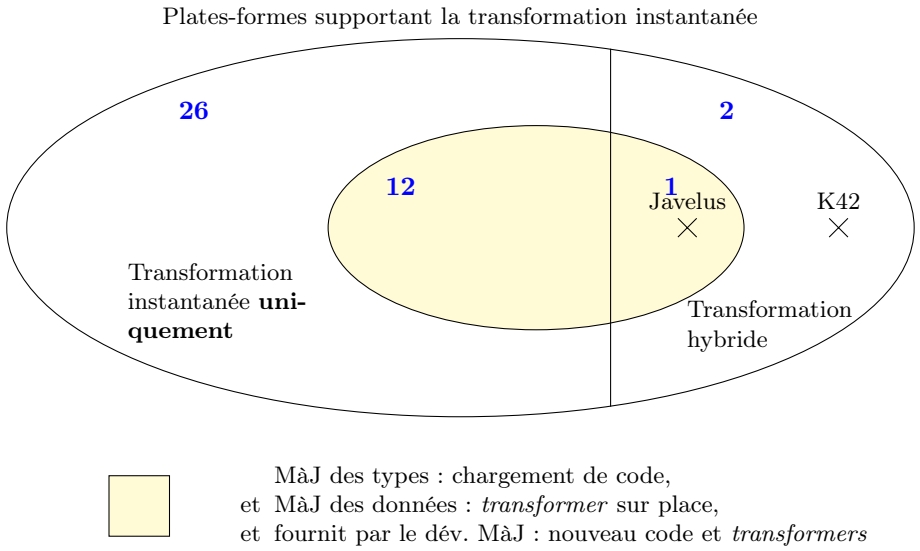
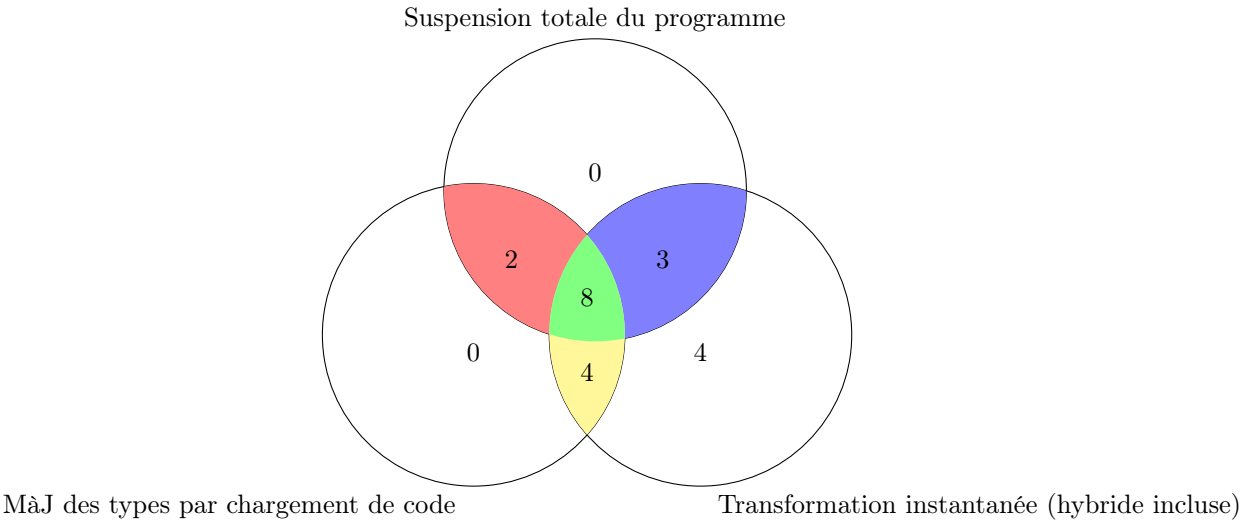


FIGURE 5.2 – Transformation instantanée des données

28 plates-formes supportent la transformation instantanée des données (dont 2 utilisant également la transformation retardée : Javelus [41] et K42 [8]). 13 de ces plates-formes (en jaune sur la figure 5.2) mettent à jour les types en chargeant leur code, appliquent des *transformers* directement sur les données et ne demandent que le nouveau code (éventuellement accompagné de *transformers*) au développeur de mise à jour. Cela peut s'expliquer par une synergie entre transformation instantanée et application de *transformer* sur place. Il est en effet plus facile d'appliquer un *transformer* sur place pour mettre à jour une donnée de manière instantanée. Une autre explication de cette statistique est la haute fréquence des propriétés : dans 79% des plates-formes étudiées, le développeur de la mise à jour ne fournit que le nouveau code, dans 62% des cas, les données sont transformées instantanément, ...

Synergies (figure 5.2)
- Méthode de transformation des données → Stratégie de transformation

21 Plates-formes accèdent immédiatement aux données (dont 2 qui supportent également l'accès progressif : Rubah [63] et LUCOS [22]). La figure 5.3 montre la répartition de ces plates-formes selon trois critères : les plates-formes qui suspendent totalement le programme, celles qui transforment les données instantanément et celles qui mettent à jour les types par chargement de code. Il est à remarquer que chacune de ces 21 plates-formes respecte au moins un de ces critères. La figure 5.3 permet les constats suivants :



Cette figure montre uniquement les plates-formes accédant aux données immédiatement

FIGURE 5.3 – Accès immédiat aux données (hybrides incluses)

— 13 plates-formes suspendent intégralement le programme. Cela souligne une forte synergie entre la stratégie d'accès et la suspension du programme. La définition de l'accès immédiat donnée dans le chapitre 3 indique que le programme ne doit exécuter aucune instruction entre la demande d'accès et l'accès effectif, suspendre le programme est un bon moyen de s'en assurer.
— 19 plates-formes utilisent la transformation instantanée (dont K42 qui emploie également la transformation retardée). En combinant transformation instantanée et accès immédiat, ces plates-formes utilisent la stratégie *pressée* de mise à jour des données. Cela indique une synergie entre stratégie d'accès et de transformation. D'autre part, il est remarquable que 62% des 42 plates-formes étudiées emploient cette stratégie de transformation.
— 14 plates-formes mettent à jour les types en chargeant leur code. Or, 22 plates-formes au total utilisent ce mécanisme. Ce sont donc deux tiers de ces 22 plates-formes qui accèdent immédiatement aux données. Cela peut s'expliquer par la logique suivante : Le type est mis à jour dès le chargement du nouveau code, l'accès immédiat aux données (généralement couplée à une transformation instantanée) permet de s'assurer que les données seront mises à jour au même moment que leurs types.
— 8 plates-formes combinent les trois propriétés. Cette statistique semble indiquer que les trois propriétés sont aisément combinables.

Par ailleurs, 13 des plates-formes accédant aux données immédiatement ont un ou plusieurs gestionnaires permanents à portée globale (ces plates-formes ne figurent pas sur la figure 5.3). La portée globale rend l'accès immédiat plus facile à l'intégralité des données tandis que la permanence du gestionnaire permet d'utiliser des techniques d'enregistrement des données pour pouvoir y accéder plus tard. Par exemple, beaucoup de plates-formes pour Java ont pour gestionnaire une partie de la JVM. Elles utilisent le ramasse-miettes pour accéder aux données de manière immédiate. Ces 13 plates-formes représentent 48% des 27 plates-formes dont les gestionnaires sont permanents et ont une portée globale. Inversement, lorsque la plate-forme dispose de ce type de gestionnaire, il est plus facile d'accéder immédiatement aux données (ramasse-miettes, mécanisme de collecte des données). Cependant, sur les 11 plates-formes accédant progressivement aux données (stratégies d'accès hybrides incluses), 9 ont un ou plusieurs gestionnaires permanent à portée globale. Ces choix architecturaux concernant les gestionnaires sont très fréquents (71% des plates-formes ont un ou plusieurs gestionnaires à portée globale, 64% des plates-formes ont un ou plusieurs gestionnaires permanents). La synergie identifiée reste toutefois pertinente.

Synergies (figure 5.3)
- Stratégie d'accès → Suspension du programme
- Stratégie d'accès → Stratégie de transformation
- Mise à jour des types → Stratégie d'accès
- Stratégie d'accès ↔ Portée et durée de vie du gestionnaire

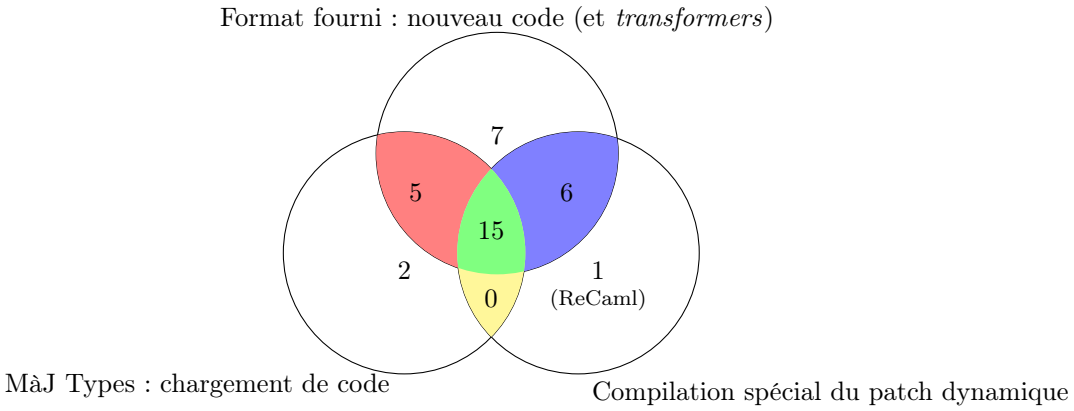


FIGURE 5.4 – Mise à jour des types par chargement de code

22 plates-formes mettent à jour leurs types en chargeant le nouveau code. Comme représenté sur la figure 5.4, 20 d'entre elles ne demandent que le nouveau code (éventuellement accompagné de *transformers*) au développeur de mise à jour. Et parmi ces 20 plates-formes, 15 emploient une chaîne de compilation spéciale pour générer la patch dynamique à partir de ce nouveau code. S'il est très fréquent (79% des plates-formes étudiées) que la plate-forme ne demande que le nouveau code, ces statistiques montrent une synergie entre ces propriétés. En effet, si la plate-forme met à jour les types en chargeant leur code, elle n'a pas besoin de demander plus d'informations au développeur de la mise à jour. Et si seul le nouveau code est demandé au développeur du programme, il est nécessaire d'employer une chaîne de compilation spécifique pour obtenir le patch dynamique à partir de ce dernier.

En effet, la figure 5.4 permet également de remarquer que la majorité (21 plates-formes sur 33) des plates-formes ne demandant que le nouveau code emploie une chaîne de compilation spéciale. Inversement, sur les 22 plates-formes employant une chaîne de compilation spécifique, 21 ne demandent que le nouveau code au développeur de mise à jour. Seule ReCaml [20] demande au développeur des fonctions dont l'exécution met à jour le programme. Ces statistiques indiquent une synergie à double sens : lorsque le patch dynamique est généré par une chaîne de compilation spéciale, peu d'informations sont nécessaires et lorsque peu d'informations sont données, une chaîne de compilation est nécessaire pour générer le patch dynamique. Cela est surtout vrai si les mécanismes employés demandent peu d'informations (comme l'indiquent les 15 plates-formes de la zone verte sur la figure 5.4).

Synergies (figure 5.4)
- Mise à jour des types → Format fourni par le développeur de mises à jours
- format fourni par le développeur de mises à jours ↔ Méthode de construction du patch dynamique

Altérabilité

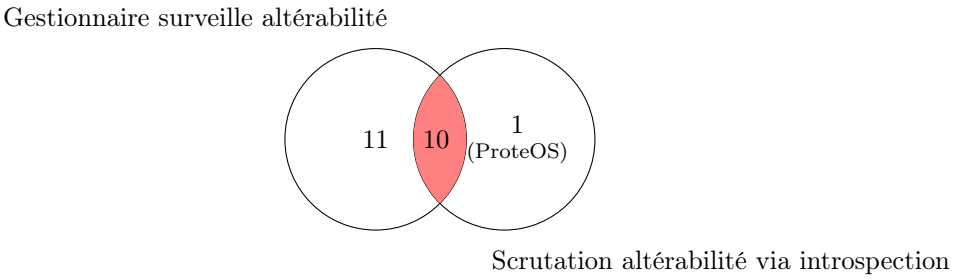


FIGURE 5.5 – Scrutation de l'altérabilité

11 plates-formes emploient de l'introspection pour scruter l'altérabilité. La quasi-totalité d'entre elles (10 plates-formes indiquées en rouge sur la figure 5.5) fait surveiller l'altérabilité par au moins un gestionnaire. Seule ProteOS [36] fait surveiller l'altérabilité par les processus modifiés eux-mêmes. Cela indique une synergie entre mécanisme de scrutation de l'altérabilité et entité chargée de cette surveillance. En effet, pour utiliser l'introspection, il est préférable de surveiller les critères de manière régulière ce qui est une tâche plus adaptée à un gestionnaire.

Synergies (figure 5.5)
- Mécanisme de scrutation de l'altérabilité → Qui surveille l'altérabilité ?

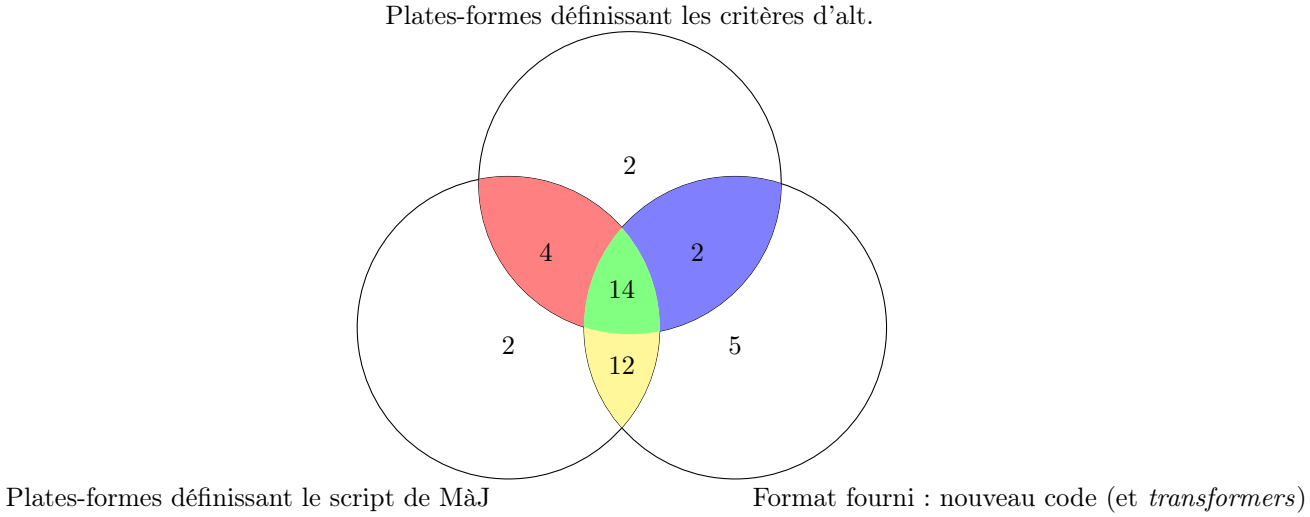
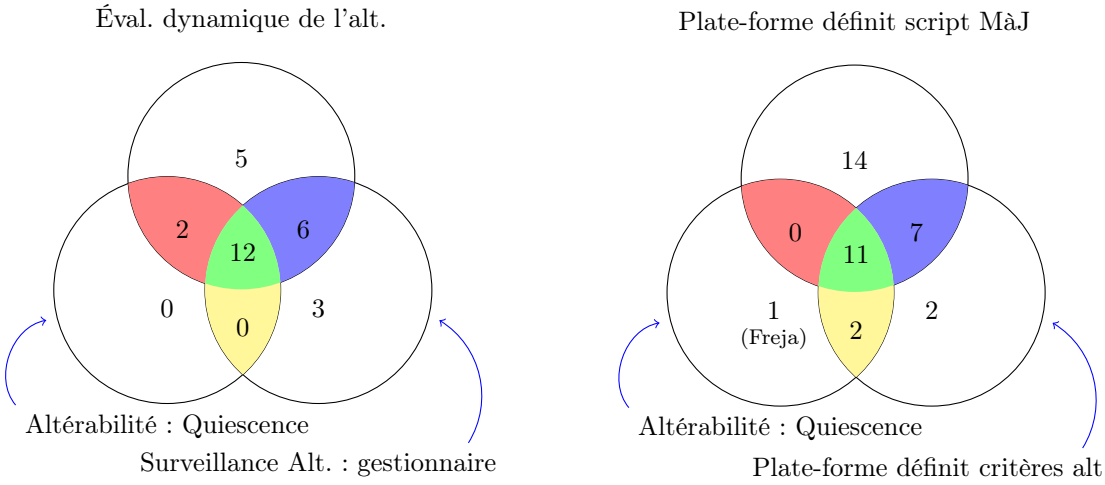


FIGURE 5.6 – Définition des scripts de mise à jour et patch dynamique

La figure 5.6 montre plusieurs synergies concernant les éléments du patch dynamique. On y remarque que sur les 22 plates-formes définissant elles-mêmes les critères d'altérabilité, 18 définissent également le reste du script de mise à jour. L'analyse des plates-formes du chapitre 4 a montré que dès qu'une plate-forme permet au développeur de définir une partie du script de mise à jour, elle lui permet de définir les critères d'altérabilité. D'une manière générale, quand une plate-forme fixe les critères d'altérabilité, elle fixe également le reste du script de mise à jour. On remarque également que 16 des plates-formes fixant les critères d'altérabilité ne demandent que le nouveau code au développeur de mise à jour (14 d'entre elles fixent l'intégralité sur script de mise à jour). Lorsque le script de mise à jour est fixé par la plate-forme, le développeur n'a plus qu'à fournir le nouveau code (et éventuellement des *transformers*). D'ailleurs, sur 32 plates-formes fixant le script de mise à jour, 26 ne demandent que le nouveau code.

Synergies (figure 5.6)
- Qui définit les critères → Qui définit le script de mise à jour
- Qui définit le script de mise à jour → Format fourni par le développeur de mise à jour



(a) Quiescence : surveillance

(b) Quiescence : définition

FIGURE 5.7 – Quiescence des éléments

Les critères d'altérabilité entrent en synergie avec plusieurs autres propriétés. La figure 5.7 montre les propriétés en combinaisons fréquentes dans les 14 plates-formes qui ont comme critère d'altérabilité la quiescence d'un ou plusieurs éléments.

La figure 5.7a montre le nombre de plates-formes utilisant ce type de critère d'altérabilité, mais

aussi le nombre de plates-formes évaluant dynamiquement l'altérabilité et/ou employant un gestionnaire pour la surveiller. Les 14 plates-formes qui utilisent la quiescence comme critère évaluent dynamiquement l'altérabilité. Et 12 de ces plates-formes utilisent au moins un gestionnaire pour la surveiller. En effet, il est plus facile d'évaluer dynamiquement la quiescence d'un élément, surtout quand il est possible d'utiliser des capacités d'introspection du système d'exécution. Il est aussi plus facile pour un gestionnaire de détecter la quiescence d'un élément car il a facilement accès à l'état du programme.

La figure 5.7b montre les plates-formes qui définissent elles-mêmes le script de mise à jour, celles qui définissent les critères d'altérabilité, et celles qui utilisent la quiescence comme critère. Pour 13 des 14 plates-formes utilisant la quiescence comme critère d'altérabilité, c'est la plate-forme elle-même qui définit ce critère. Seule Freja [5] laisse le développeur de mise à jour définir les critères d'altérabilité en précisant les composants qui doivent être quiescents lors de chaque mise à jour. A ce contre-exemple près, il est rare qu'une plate-forme ne fixe que partiellement les critères d'altérabilité (dans le cas de Freja, la plate-forme impose que les critères soient exprimés sous forme de quiescence de composants). Il y a une forte synergie entre ces deux propriétés. On retrouve également sur la figure 5.7b les 18 plates-formes qui définissent à la fois le script de mise à jour et les critères d'altérabilité. 11 de ces plates-formes (en vert sur la figure) utilisent la quiescence pour définir leurs critères d'altérabilité. Définir les critères d'altérabilité est donc indirectement en synergie avec la définition du script de mise à jour.

Synergies (figure 5.7)
- Critères d'altérabilité → Évaluation de l'altérabilité
- Critères d'altérabilité → Qui surveille l'altérabilité
- Critères d'altérabilité → Qui définit les critères et qui définit le script de mise à jour
- Qui définit les critères d'altérabilité → Qui définit le script de mise à jour (rappel)

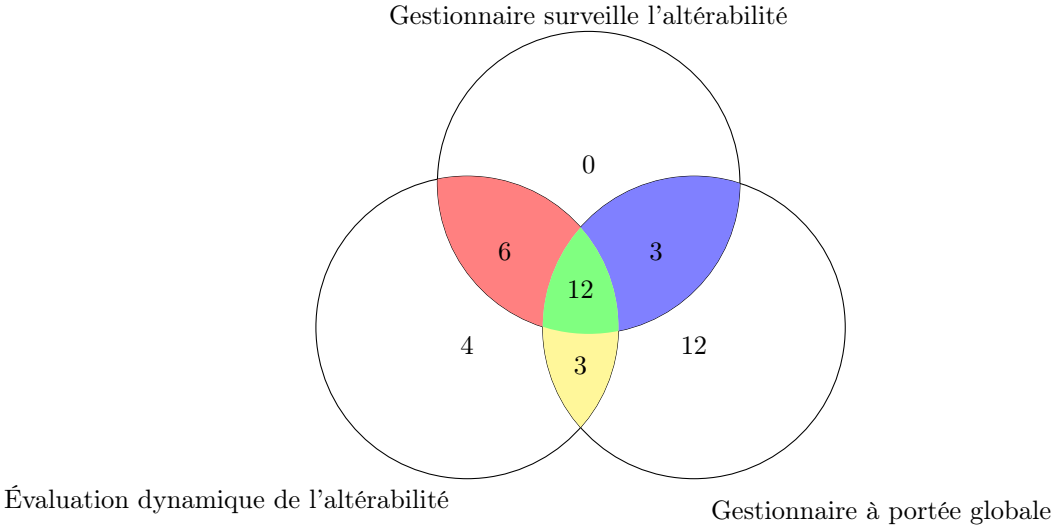


FIGURE 5.8 – Surveillance de l'altérabilité et gestionnaire

25 plates-formes évaluent l'altérabilité dynamiquement et 18 d'entre elles utilisent au moins un gestionnaire pour la surveiller (en jaune et rouge sur la figure 5.8). Pour 12 de ces plates-formes, les gestionnaires ont une portée globale. Il a déjà été établi que dans le cas où la quiescence est utilisée pour définir les critères d'altérabilité, ces derniers sont souvent surveillés par au moins un gestionnaire. Cette statistique confirme cela quelque soit les critères d'altérabilité définis. Le fait que les gestionnaire aient une portée globale peut s'expliquer par la grande fréquence de ce choix architectural (71% des plates-formes étudiées).

Synergies (figure 5.8)
- Évaluation de l'altérabilité → Qui surveille l'altérabilité

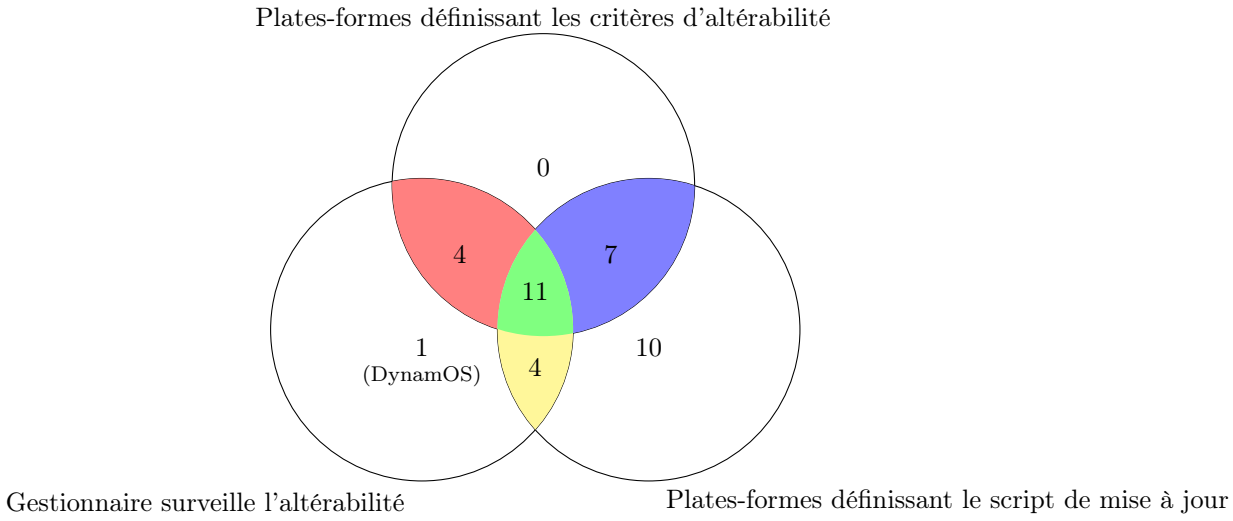


FIGURE 5.9 – Définition des scripts de mise à jour et critères d'altérabilité

La figure 5.9 montre des synergies entre la définition des critères d'altérabilité et le choix de l'entité qui les surveille. On y retrouve, en vert et bleu, les 18 plates-formes fixant à la fois les critères d'altérabilité et le script de mise à jour. On y remarque également que la quasi-totalité (19 plates-formes sur 20) des plates-formes utilisant un gestionnaire pour surveiller l'altérabilité définissent les critères, le script de mise à jour, ou les deux. Seule DynamOS [52] utilise un gestionnaire pour surveiller des critères fixés par le développeur de mise à jour. Ce chiffre indique une synergie entre les éléments qui surveillent l'altérabilité et ceux qui en définissent les critères (impliquant indirectement une synergie avec l'élément qui définit le script de mise à jour comme vu précédemment). Il est très rare que les gestionnaires soient modifiés au fil de la vie du programme. Lorsqu'un gestionnaire surveille l'altérabilité, les critères sont généralement fixés dans son code.

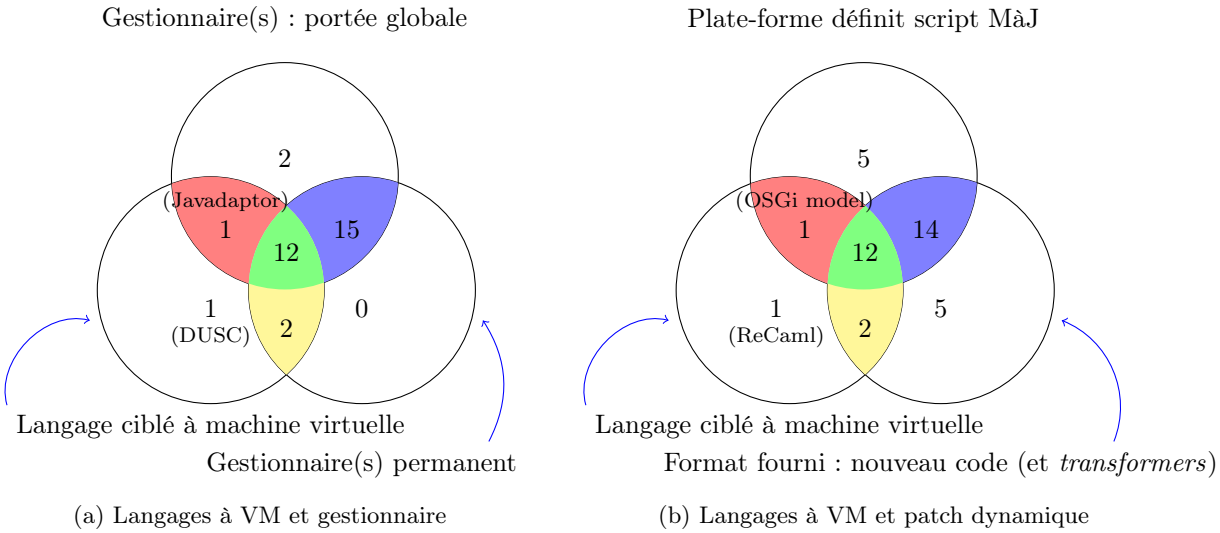
Synergies (figure 5.9)
- Qui surveille l'altérabilité → Qui définit le script de mise à jour et qui définit les critères d'altérabilité
- Qui définit les critères d'altérabilité → Qui définit le script de mise à jour (rappel)

Langage ciblé

Les prochaines statistiques montrent que le langage ciblé par une plate-forme impacte de nombreuses propriétés. L'algorithme APriori a détecté un grand nombre de combinaisons fréquentes impliquant le langage ciblé par la plate-forme. Cela confirme un des constats de l'analyse du chapitre 4 : le langage ciblé oriente généralement le choix des mécanismes qu'une plate-forme emploie.

16 plates-formes ciblent des langages s'exécutant sur des machines virtuelles ou des interpréteurs (Java, Smalltalk, ...). 12 d'entre elles définissent une unité de mise à jour de niveau intermédiaire (une classe ou un composant). Cela peut s'expliquer par la grande quantité de plates-formes Java dont l'unité de mise à jour est une classe. Le langage ciblé peut donc influencer le choix de l'unité de mise à jour. 10 des 16 plates-formes mettent à jour les classes en chargeant leur nouveau code. La majorité des plates-formes pour programmes Java utilise un *classloader* spécifique permettant de recharger des classes pour les mettre à jour. Il y a donc synergie entre langage ciblé et mécanisme de mise à jour des types.

La figure 5.10a montre que 15 plates-formes pour langages à machine virtuelles ont un ou plusieurs gestionnaires permanents, à portée globale ou les deux à la fois (12 plates-formes sont dans ce cas). DUSC [61] n'est pas dans ce cas car ces champs ne sont pas renseignés dans la base de données homogénéisée. Lorsque le langage utilise une machine virtuelle, il est fréquent que la plate-forme utilise une partie de cette dernière comme gestionnaire, permettant ainsi d'en utiliser les fonctionnalités pour détecter la quiescence d'éléments (points sûrs de la machine virtuelle, ramasse-miettes, ...). Le gestionnaire a alors la portée et la durée de vie de la machine virtuelle. Dans les cas où le gestionnaire a une portée ou une durée de vie limitée, les gestionnaires sont généralement des parties du programme. C'est par exemple le cas de Javadaptor.



(a) Langages à VM et gestionnaire

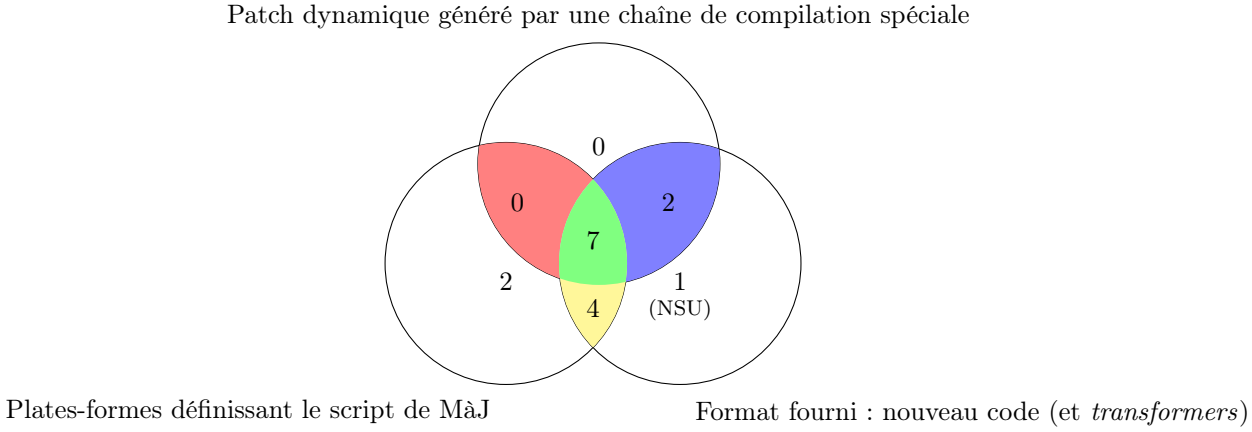
(b) Langages à VM et patch dynamique

FIGURE 5.10 – Langages à machine virtuelle

La figure 5.10b montre que la majorité des plates-formes ciblant un langage à machine virtuelle (12 plates-formes sur 16) définissent elles-mêmes le script de mise à jour et ne demandent que le nouveau code au développeur de mise à jour. Comme expliqué ci-dessus, ces plates-formes utilisent généralement des mécanismes basés sur des fonctionnalités de la machine virtuelle (*hotswap*, contextes, ...). Elles fixent alors le script de mise à jour en prédéfinissant les mécanismes utilisés et requièrent peu d'informations de la part du développeur de mise à jour.

Synergies (figure 5.10)

- Langage ciblé → Unité de mise à jour
- Langage ciblé → Mécanisme de mise à jour des types
- Langage ciblé → Portée et durée de vie du gestionnaire
- Langage ciblé → Qui fournit le script de mise à jour et format fourni par le développeur de mise à jour



Cette figure montre uniquement les plates-formes ciblant les langages C et C++

FIGURE 5.11 – Langages compilés (C/C++)

17 plates-formes ciblent des langages compilés de type C et C++. Comme dans le cas des langages à machine virtuelle, cela a un impact tant sur l'architecture des plates-formes que sur les mécanismes qu'elles emploient.

La figure 5.11 montre la répartition de 16 de ces 17 plates-formes selon trois critères : les plates-formes définissant elles-mêmes le script de mise à jour, celles qui ne demandent que le nouveau code au développeur de mise à jour et celles qui emploient une chaîne de compilation spécifique pour générer le patch dynamique. Afpac [17] est la seule plate-forme pour langage C ou C++ à ne pas avoir au moins

l'un de ces trois critères. Afpac est un modèle pour composants adaptables ne traitant pas ces trois sujets. 9 plates-formes (en bleu et vert sur la figure) ne demandent que le nouveau code au développeur de mise à jour et utilisent une chaîne de compilation spécifique. 11 plates-formes définissent elles mêmes le script de mise à jour et ne demandent que le nouveau code. 7 plates-formes ont les trois propriétés à la fois. Il a déjà été établi que le format fourni par le développeur de mise à jour impacte la manière dont le patch dynamique est compilé. Ici, les statistiques montrent une prédisposition des plates-formes pour un traitement statique des mises à jour, tirant profit des fonctionnalités des compilateurs disponibles qu'ils étendent. Par exemple, Ginseng [58] utilise une série d'outils en complément de GCC [33] pour analyser le nouveau code fourni par le développeur de mise à jour et le compiler en un patch dynamique. Ces outils sont programmés en tenant compte de la stratégie de mise à jour fixée par Ginseng.

D'autres synergies ont été constatées, comme résumé ci-dessous :

— 11 plates-formes sur 17 ont un gestionnaire qui est une partie du programme. Il s'agit généralement d'une bibliothèque dynamique chargée au démarrage du programme. La plate-forme tire profit de cette fonctionnalité du C pour intégrer un gestionnaire au programme.

— 10 plates-formes sur 17 mettent à jour les types en chargeant leur code. Dans beaucoup de plates-formes, le nouveau code est chargé sous forme d'une bibliothèque dynamique. Les nouvelles versions des types sont chargées comme de nouveaux types. C'est la technique la plus facile pour mettre à jour les types en C car cela évite de devoir changer le code en mémoire.

— 11 plates-formes sur 17 suspendent intégralement le programme. Cela peut s'expliquer par la difficulté à suspendre seulement certains fils d'exécution en C. Le chapitre 10 traitant de l'implémentation de Cmoult détaille ce problème et comment il peut être contourné.

Ces trois points ont montré des synergies entre langage ciblé et nature du gestionnaire, mécanisme de mise à jour des types et suspension du programme. Les deux premières synergies avaient déjà été mises en évidence lors de l'étude des plates-formes pour langages à machine virtuelle. Quelque soit le langage ciblé par une plate-forme, il impacte généralement les mêmes propriétés, mais de façon différente. Cela confirme l'observation faite dans le chapitre 4 : certains choix architecturaux, et certains mécanismes sont plus fréquents dans les plates-formes ciblant certains langages.

Synergies (figure 5.11)

- Langage ciblé → Nature du gestionnaire
- Langage ciblé → Suspension du programme
- Langage ciblé → Construction du patch dynamique
- Langage ciblé → Qui fournit le script de mise à jour et format fourni par le développeur de mise à jour
- Langage ciblé → Mécanisme de mise à jour des types

Mise à jour des fonctions

Sur 17 plates-formes utilisant de l'indirection pour mettre à jour les fonctions, 10 utilisent un gestionnaire interne au programme. Dans certains cas, ce gestionnaire est responsable de cette indirection : il peut agir comme connecteur entre deux composants ou maintenir une table d'indirection.

Synergies

- Mécanisme de mise à jour des fonctions → Nature du gestionnaire

Suspension

21 Plates-formes suspendent intégralement le programme lors de la mise à jour et 18 d'entre elles ont au moins un gestionnaire à portée globale. Un tel gestionnaire a accès à tous les fils d'exécution (ou tous les processus) et peut donc plus facilement suspendre l'intégralité du programme. Autrement il devient nécessaire de prévoir un algorithme permettant de suspendre tous les fils (ou processus) de manière asynchrone.

Synergies

- Suspension du programme → Portée du gestionnaire

Gestionnaire

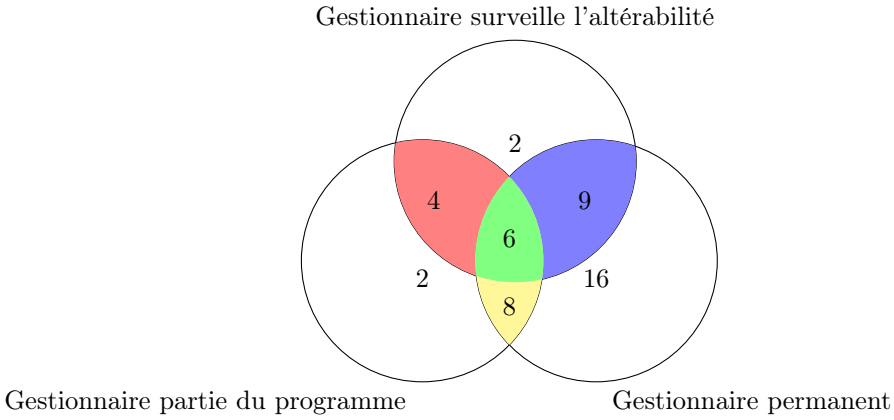
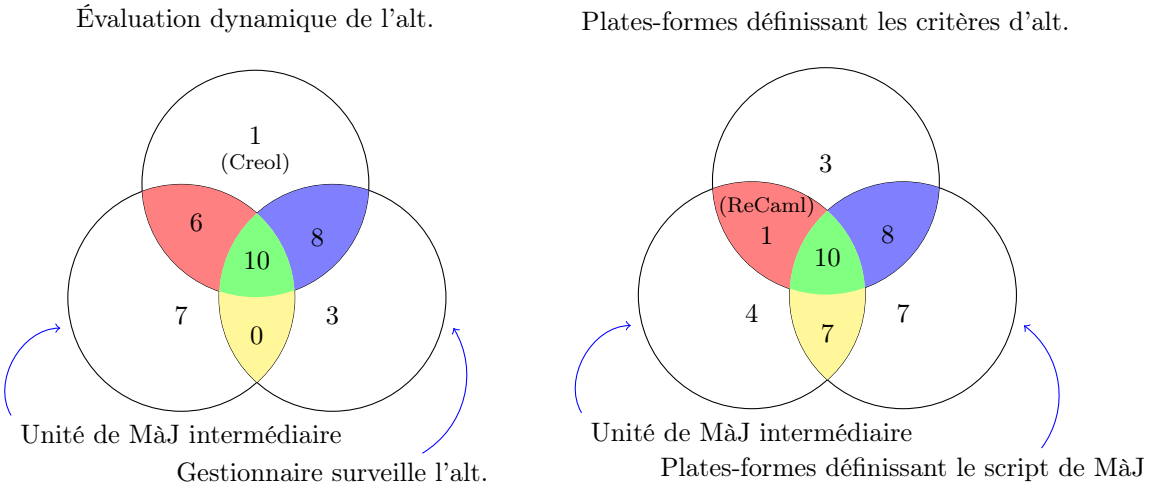


FIGURE 5.12 – Gestionnaires

La figure 5.12 montre des combinaisons de propriétés concernant les gestionnaires. On y remarque que sur 20 plates-formes dont le gestionnaire est une partie du programme, 14 ont un gestionnaire permanent (en vert et jaune sur la figure) et 10 utilisent ce gestionnaire pour surveiller l'altérabilité (en rouge et vert). Pour 6 plates-formes, le gestionnaire combine ces trois propriétés. Ces chiffres peuvent indiquer deux synergies indépendantes. Lorsque le gestionnaire est une partie du programme, il partage généralement sa durée de vie : il est lancé en même temps que le programme puis arrêté en même temps. C'est par exemple le cas de Ginseng qui charge le gestionnaire dans le programme à son démarrage et ne le décharge jamais. Lorsque le gestionnaire est interne au programme, il peut plus facilement accéder aux informations de ce dernier (piles d'appel, objets dans le tas, ...). Il peut alors facilement surveiller l'altérabilité. Inversement, pour qu'un gestionnaire puisse surveiller l'altérabilité, il est plus pratique qu'il soit interne au programme pour avoir accès à toutes ces informations.

Synergies (figure 5.12)
- Nature du gestionnaire ↔ Qui surveille l'altérabilité
- Nature du gestionnaire → Durée de vie du gestionnaire

Unité de mise à jour



(a) Unité de MàJ et surveillance d'altérabilité

(b) Unité et script de MàJ

FIGURE 5.13 – Unités de mise à jour

22 Plates-formes définissent une unité de mise à jour de niveau intermédiaire (classe, composant,...). Les figures 5.13a et 5.13b montrent l'impact de cette propriété sur la manière de surveiller l'altérabilité et de définir le script de mise à jour.

Sur la figure 5.13a, on peut remarquer que 16 plates-formes (en vert et rouge) évaluent dynamiquement les critères d'altérabilité. Et pour 10 d'entre elles, c'est un gestionnaire qui remplit ce rôle. Lorsque l'unité

de mise à jour est une classe ou un composant, elle est clairement délimitée et il est plus facile d'évaluer dynamiquement son altérabilité. Il est également possible d'attacher un gestionnaire aux unités pour détecter leur altérabilité comme c'est le cas dans K42.

Sur la figure 5.13b, on peut remarquer que 11 plates-formes (en rouge et vert) définissent elles-mêmes les critères d'altérabilité et que 17 plates-formes définissent le script de mise à jour. 10 plates-formes combinent ces deux propriétés. On remarque à nouveau la forte synergie entre définition des critères d'altérabilité et définition du script de mise à jour. Ces chiffres peuvent s'expliquer en partie par l'existence de plates-formes orientées composants pour lesquelles chaque mise à jour revient à reconfigurer le programme (remplacer des composants, changer les connections) lorsque les composants affectés sont quiescents. D'une manière générale, lorsque l'unité de mise à jour est clairement délimitée (une classe, un composant), les critères d'altérabilité sont basés sur la quiescence des éléments modifiés. Il y a donc synergie entre unité de mise à jour et définition des critères d'altérabilité et, par conséquent, définition du script de mise à jour.

Synergies (figure 5.13)
- Unité de mise à jour → Évaluation de l'altérabilité et Qui surveille l'altérabilité
- Unité de mise à jour → Qui définit les critères d'altérabilité et Qui fourni le script de mise à jour

5.2.3 Synergies entre propriétés

	Méc. MàJ fonctions	Méc. MàJ données	Méc. MàJ types	Transformation	Accès	Vie gestionnaire	Portée gestionnaire	Nature gestionnaire	Suspension	Format fourni	Compil. patch	Déf. critères alt.	Qui surveille alt.	Évaluation alt.	Critères alt.	Méc. détec. alt.	Déf. script MàJ	Langage ciblé	Unité de MàJ
Méc. MàJ fonc.								↗											
Méc. MàJ données				↗															
Méc. MàJ types						↗					↗								
Transformation																			
Accès				↗		↗	↗		↗										
Vie gest					↗														
Portée gest.					↗														
Nature du gest.						↗							↗						
Suspension							↗												
Format fourni											↗								
Compilation patch																			
Déf. critères d'alt.										↗							↗		
Qui surveille l'alt.								↗									↗		
Éval. de l'alt.							↗						↗						
Critères alt.												↗	↗	↗			↗		
Méc. détect. alt.													↗						
Déf. script MàJ																			
Langage ciblé			↗			↗	↗	↗	↗	↗	↗						↗		↗
Unité de MàJ												↗	↗	↗			↗		

FIGURE 5.14 – Synergies entre choix de conception

La figure 5.14 récapitule les synergies mises en évidences par les statistiques précédemment présentées. La table se lit de gauche à droite. Une case marqué d'une flèche bleue indique que le choix de la propriété en ligne peut influencer le choix de la propriété en colonne. Par exemple, la table montre que fixer le mécanisme de mise à jour des données peut impacter le choix de la stratégie de transformation des données. En effet, la sous-section 5.2.2 montre que lorsqu'une plate-forme applique des *transformers* directement sur les données, elle adopte généralement la stratégie de transformation instantanée.

Lors de la conception d'une plate-forme configurable, il est possible d'utiliser cette table des synergies pour identifier quelles propriétés doivent être configurables par le développeur de mise à jour. La plate-forme doit permettre au développeur de mise à jour de configurer toutes les propriétés d'un même groupe de synergies, pour les raison explicitées ci-dessus.

5.3 Familles de plates-formes

Distinguer des familles de plates-formes partageant des propriétés communes permet d'identifier des combinaisons typiques de mécanismes. Si plusieurs plates-formes utilisent une même combinaison de propriétés, cela peut indiquer que cette combinaison répond à un besoin type des programmes. Ces familles donnent alors des lignes directrices sur les mécanismes à implémenter et les combinaisons à supporter.

Pour identifier ces familles, la base de données homogénéisée est utilisée pour créer quatre matrices de distances chacune selon une mesure différente. Ces quatre mesures utilisent la formule suivante :

Définition 5. *Distance entre deux plates-formes*
La distance entre deux plates-formes P_1 et P_2 est définie par la formule suivante :
$dist(P_1, P_2) = \sum_{i=0}^{n} a_i * eq(prop(P_1, i), prop(P_2, i))$
où :
— $prop(P_k, i)$ *est la* i^e *propriété de la plate-forme* P_k
— $eq(x, y)$ *vaut 1 si* $x = y$*, 0 sinon*
— $\forall 0 \leq i \leq n,\ 0 \leq a_i \leq 1$.

Ce sont les coefficients a_i qui distinguent les mesures. Trois d'entre elles se concentrent uniquement sur une catégorie de propriétés (cycle de vie, architecture, mécanismes) et ont des coefficients nuls pour toute propriété hors de cette catégorie. La quatrième tient compte de toutes les catégories. Pour toutes les mesures, les coefficients sont ajustés pour tenir compte des synergies. Si plusieurs propriétés sont en synergie, leurs coefficients sont réduits pour éviter que la distance entre deux plates-formes soit principalement décidée par une propriété et ses synergies.

L'algorithme DBSCAN [67] est utilisé sur les quatre matrices de distances pour partitionner l'ensemble des plates-formes et identifier les différentes familles (ou *cluster* selon la terminologie de DBSCAN). DBSCAN prend en paramètres une distance ϵ et un nombre minimum de points M. Un cluster est formé si dans un rayon de longueur ϵ autour d'un point donné, au moins M points peuvent être trouvés.

Les figures 5.15 et 5.16 présentent les résultats de l'algorithme DBSCAN. La première montre une projection des plates-formes sur un plan. Les points colorés correspondent à des plates-formes d'un même cluster. Lorsque les points sont plus petits, il s'agit de plates-formes en bordure d'un cluster. La projection a tendance à perturber les distances, ce qui explique notamment le cluster rouge en deux parties sur la figure 5.15a. La figure 5.16 liste, pour chaque plate-forme, les familles dont elle fait partie.

Les prochaines sous-sections détaillent les coefficients a_i ainsi que les paramètres ϵ et M utilisés pour chaque matrice de distances. Elles analysent également les différentes familles identifiées.

5.3.1 Familles d'architecture

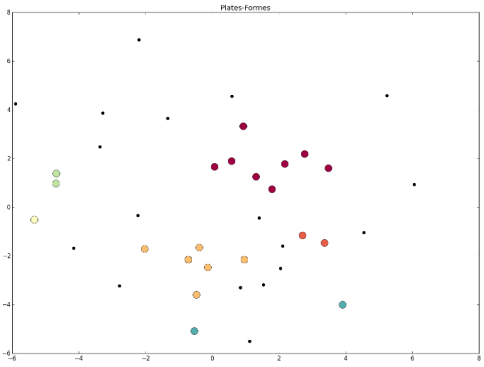
La matrice de distances concentrée sur les propriétés d'architecture utilise les coefficients a_i suivants :

Qui surveille l'altérabilité	0,5	Évaluation de l'altérabilité	0,5
Portée du(des) gestionnaire(s)	0,5	Durée de vie du(des) gestionnaire(s)	0,5
Cloisonnement des versions	0,5	Qui fournit le script de MàJ	0,25
Qui fournit le nouveau code	0,25	Que doit fournir le Dév. MàJ	0,5
Toute autre propriété	1		

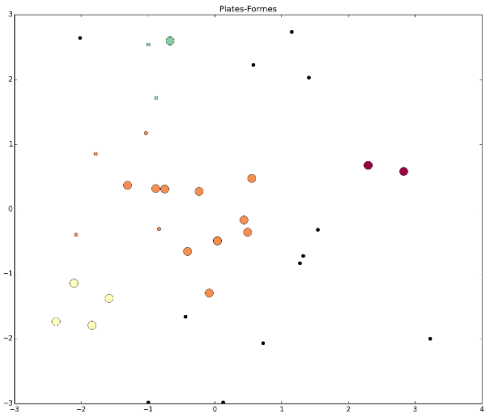
Ces coefficients tiennent compte des synergies ou des propriétés redondantes : la portée et la durée de vie du gestionnaire sont souvent liées à d'autres aspects. Ces deux propriétés sont donc pondérées pour qu'ensemble, elles équivalent à une propriété normale. La possibilité d'avoir plusieurs versions va de pair avec le cloisonnement des versions. Le second paramètre est donc pondéré par un coefficient 0,5. Trois aspects concernent le patch dynamique : qui fournit le script de mise à jour, qui fournit le nouveau code et ce que doit fournir le développeur de mise à jour. La somme des coefficients de ces trois aspects vaut donc 1. Les deux premiers ayant généralement toujours la même valeur, ils sont dépréciés au profit du troisième. L'évaluation de l'altérabilité est souvent liée à l'entité responsable de sa surveillance, ces deux aspects sont donc pondérés par un coefficient 0,5.

Les paramètres employés pour l'algorithme DBSCAN sont $\epsilon = 2$ et $M = 2$. Ils permettent de distinguer six familles de plates-formes.

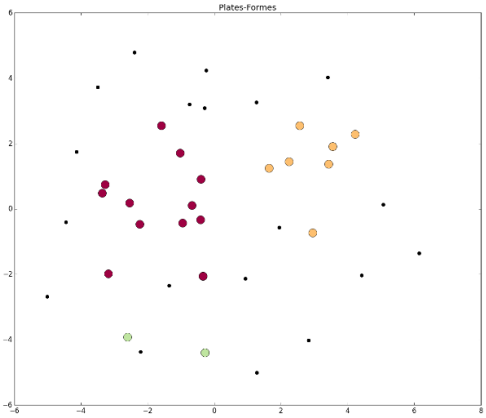
La première famille nommée *C/statique* regroupe 9 plates-formes pour programmes C, ayant un gestionnaire permanent et à portée globale. Pour toutes ces plates-formes sauf Kitsune, le gestionnaire est une partie du programme et pour toutes les plates-formes sauf LUCOS [22], l'altérabilité est évaluée statiquement (leur critère est l'atteinte de points de mise à jour).



(a) Clusters basés sur l'architecture des plates-formes



(b) Clusters basés sur le cycle de vie des plates-formes



(c) Clusters basés sur les mécanismes employés par les plates-formes

FIGURE 5.15 – Représentation des clusters de plates-formes

La deuxième famille nommée *Linux/dynamique* regroupe les deux plates-formes Ksplice [4] et DynamOS [52] qui permettent de mettre à jour Linux dynamiquement. Leur altérabilité est évaluée dynamiquement par un gestionnaire interne au programme.

La troisième famille nommée *VM/dynamique* regroupe 7 plates-formes utilisant un gestionnaire permanent à portée globale pour évaluer dynamiquement l'altérabilité. 7 plates-formes ciblent des langages à machine virtuelle (DynamicML [35] est présenté comme un modèle mais envisage une implémentation utilisant une machine virtuelle). 6 plates-formes utilisent une partie de cette machine virtuelle comme gestionnaire (*Incremental Dynamic Update for Java-Based Smart Cards* [59] utilise un gestionnaire externe).

La quatrième famille nommée *Altérable/contextes* regroupe les deux plates-formes Thesus [77] et ActiveContext [76] qui ont comme particularité d'appliquer les mises à jours à tout moment, considérant le programme comme toujours altérable. Elles appliquent les mises à jour dans un contexte ce qui résulte en la cohabitation de plusieurs versions. Il s'agit également de deux plates-formes pour le langage Smalltalk dont le gestionnaire permanent à portée globale est une partie de la machine virtuelle sur laquelle elles reposent.

La cinquième famille nommée *Altérable/Java* regroupe les deux plates-formes Javeleon [40] et Dynamic Correspondance Proxification [39] (DCP) qui ciblent des programmes Java et permettent à plusieurs versions de coexister, sous forme de versions différentes de classes pour Javeleon, de composants pour DCP. Elles appliquent les mises à jour en considérant le programme comme toujours altérable et emploient un gestionnaire permanent à portée globale.

Plate-forme	Architecture	Mécanismes	Cycle de vie	Général
Kitsune	C/statique	C.Sp./inst.	Redémarrage	Redémarrage
Ginseng	C/statique	C.Sp./inst.		C/statique
STUMP	C/statique	C.Sp./inst.		C/statique
Swap & Play	C/statique	C.Sp./inst.	Standard	Standard
UpStare	C/statique	C.Sp./inst.	Redémarrage	Redémarrage
Ekiden	C/statique	C.Sp./inst.	Redémarrage	Redémarrage
Proteus	C/statique			C/statique
How to have your cake and eat it too : ...	C/statique	Artéfact		
LUCOS	C/statique			
Ksplice	Linux/Dyn.		Standard	
DynamOS	Linux/Dyn.			
Jvolve	VM/Dyn.	C.Sp./inst.	Standard	Standard
Javelus	VM/Dyn.			Standard
TOS	VM/Dyn.	C.Sp./inst.	Standard	Standard
EmbedDSU	VM/Dyn.		Guidage	
DynamicML	VM/Dyn.	Artéfact	Standard	Standard
Incremental Dynamic Update for ...	VM/Dyn.		Standard	Standard
ReCaml	VM/Dyn.	Artéfact	Standard	
ActiveContext	Alt./Contexte	Altérable	Altérable	Altérable
Thesus	Alt./Contexte	Altérable	Altérable	Altérable
Dynamic Corresondance ...	Alt./Java		Standard	
Javeleon	Alt./Java	Altérable	Altérable	Altérable
Argus	Artéfact		Guidage	
DUSC	Artéfact		Standard	
K42			Guidage	
Polus				
Local Dynamic Update for Components ...			Altérable	
Hotspot			Standard	Standard
DDSU				
Freja			Guidage	
TimeAdapt				
NSU				
Dynamic software updates for ...			Standard	
ProteOS		C.Sp./inst.		
Rubah		C.Sp./inst.	Redémarrage	Redémarrage
Supporting Dynamic Service Updates ...		C.Sp./inst.	Standard	Standard
Javadaptor		C.Sp./inst.	Standard	Standard
A Formal Model for Supporting ...		C.Sp./inst.	Standard	
OPUS		Artéfact	Standard	
DUC		Artéfact	Standard	
Creol		Artéfact	Standard	
Afpac		Artéfact		

FIGURE 5.16 – Détail des clusters de plates-formes

La sixième famille regroupe les deux plates-formes Argus [13] et DUSC [61]. Cette famille est un artéfact. Dans les tables de l'annexe A, ces deux plates-formes ont beaucoup de propriétés non renseignées, ce qui les a rapprochées lors de l'utilisation de DBSCAN.

Les cinq familles identifiées montrent une vraie diversité des architectures de plates-formes. Deux principales familles peuvent être identifiées : la famille *C/statique* et la famille *VM/dynamique* qui sont les familles les plus nombreuses. La famille *Linux/dynamique* est spécialisée envers un programme spécifique (Linux) tandis que les familles *Altérable/Java* et *Altérable/contexte* présentent deux variations d'un genre de plate-forme à part : les plates-formes permettant à plusieurs versions de coexister.

Cette diversité conforte la défense de la nouvelle approche présentée dans ce manuscrit. Plusieurs types d'architectures se démarquent et pourtant, certaines de ces propriétés restent similaires quelques soient les familles. Par exemple, le développeur n'a généralement qu'à fournir le nouveau code à chaque mise à jour. Il n'existe pas d'architecture universelle ou même fortement dominante, ce qui était prévisible compte tenu de la diversité des programmes existants. Pour pouvoir s'adapter à un maximum de programmes différents, il est donc important que les plates-formes puissent changer d'architecture et soient configurables.

5.3.2 Familles de combinaison de mécanismes

La matrice de distances concentrée sur les mécanismes utilise les coefficients a_i suivants :

Surveillance de l'altérabilité	0,75	Contrôle de l'exécution	0,75
Mécanisme de chargement	0	Construction du patch dynamique	0,25
Mécanisme multi-version	0,75	Synchronisation des versions	0,75
Mécanisme MàJ des fonctions	1	Mécanisme MàJ des types	1
Mécanisme Transformation données	1	Toute autre propriété	0,5

Les différents mécanismes employés sont déterminants du type de la plate-forme qui les emploie. Seuls les mécanismes associés aux tâches principales ont un coefficient de 1 : mettre à jour les fonctions, les types et les données. Les autres mécanismes déterminants comme ceux impliqués dans la possibilité pour plusieurs versions de coexister ou encore le contrôle de l'exécution ont un coefficient égal 0,75. Le mécanisme de chargement du nouveau code étant peu déterminant, il est pondéré par 0. En effet, le mécanisme employé pour cette tâche est généralement fortement dépendant du langage du programme (*classloader* spécial pour Java, bibliothèque dynamique pour C) et donne peu d'information sur les procédés employés par la plate-forme pour modifier les éléments du programme. La méthode de construction du patch dynamique a un coefficient 0,25 compte tenu de sa synergie avec le mécanisme de mise à jour des types et son faible caractère déterminant.

Les paramètres employés pour l'algorithme DBSCAN sont $\epsilon = 2,5$ et $M = 3$. Ils permettent de distinguer trois familles de plates-formes.

La première famille nommée *Comp. Sp./instantanée* regroupe 13 plates-formes transformant les données instantanément. Elles génèrent leur patch dynamique grâce à une chaîne de compilation spéciale et mettent à jour les types en chargeant le nouveau code. 11 d'entre elles accèdent instantanément aux données (STUMP [56] et Ginseng [58] y accèdent progressivement), et 10 plates-formes mettent à jour les données en appliquant un *transformer* (sur place pour 8 plates-formes, sur une copie pour 2 plates-formes).

La seconde famille regroupe 7 plates-formes. Il s'agit d'un artefact de DBSCAN. La majorité de ces plates-formes sont des modèles dont certains aspects ne sont pas renseignés dans les tables de l'annexe A. Ces plates-formes sont uniquement regroupées car elles ne présentent pas de valeur pour plusieurs propriétés communes.

La troisième famille nommée *Altérable* regroupe 3 plates-formes mettant à jour les données de manière instantanée en y accédant progressivement. Elles permettent, toutes les trois, à plusieurs versions de coexister, Thesus [77] et ActiveContext [76] utilisent des contextes tandis que Javeleon [40] permet au programme d'exécuter plusieurs versions pendant une mise à jour. Ces 3 plates-formes considèrent le programme comme toujours altérable. Cette famille ressemble à la famille *Altérable/contextes* à laquelle a été ajoutée la plate-forme Javeleon.

Le faible nombre de familles par rapport à la sous section précédente peut s'expliquer par le grand nombre de combinaisons représentées par les plates-formes étudiées. Deux familles ont été identifiées et la famille *Comp. Sp./instantanée* montre une grande diversité de mécanismes employés. Comme la section précédente, ces résultats encouragent la combinaison de mécanismes lors de la conception des mises à jour. Il n'existe pas de combinaison *standard*, ce qui peut indiquer qu'il n'existe pas de solution universelle au choix des mécanismes à employer.

5.3.3 Familles de cycles de vie

La matrice de distances concentrée sur le cycle de vie des plates-formes utilise les coefficients a_i suivants :

Attente de l'altérabilité	1	Détection de l'altérabilité	1
Modification	0	Préparation à la reprise	0
Reprise	0	Attente terminaison	0,75
Détection de la terminaison	0,75	Toute autre propriété	0,5

Il y a assez peu de synergies entre les étapes du cycle de vie. Comme pour les mesures de distance consacrées aux mécanismes, les étapes les plus déterminantes ont un coefficient 1, les moins déterminantes ont un coefficient 0,5. Les étapes de préparation à la reprise et de modification ont un coefficient 0. Les plates-formes ont toutes un comportement différent lors de l'étape de modification. C'est la même chose pour les quelques plates-formes utilisant l'étape de préparation à la reprise. L'étape de reprise a un coefficient 0 car elle est entièrement déterminée par l'étape de suspension.

Les paramètres employés pour l'algorithme DBSCAN sont $\epsilon = 1$ et $M = 4$. Ils permettent de distinguer quatre familles de plates-formes.

La première famille nommée *Redémarrage* regroupe 4 plates-formes dont le critère d'altérabilité est l'atteinte d'un point de mise à jour. Elles redémarrent toutes le programme et guident son exécution pour le mettre à jour.

La deuxième famille nommée *Standard* regroupe 17 plates-formes aux propriétés très diverses. Elles n'ont aucune propriété en commun. 7 plates-formes ont comme critère d'altérabilité la quiescence des éléments modifiés et 9 suspendent totalement le programme lors de la mise à jour. Cette famille peut s'interpréter comme le regroupement de plates-formes combinant plusieurs propriétés courantes (quiescence, suspension du programme, aucune étape de préparation, ...)

La troisième famille nommée *Guidage* regroupe 4 plates-formes guidant l'exécution du programme vers la quiescence des éléments modifiés (qui constitue leurs critères d'altérabilité).

La quatrième famille nommée *Altérable* regroupe 4 plates-formes appliquant directement les mises à jour sans attendre l'altérabilité. Les modifications sont appliquées pendant la phase d'attente de terminaison, alors que le programme poursuit son exécution.

Le cycle de vie semble plus propice à l'expression de standards. Trois des quatre familles identifiées correspondent à des types de cycles de vie : redémarrer le programme, le guider vers l'altérabilité ou encore le mettre à jour sans attendre l'altérabilité. Il est d'ailleurs intéressant de remarquer que la famille *Altérable* correspond à la fusion des familles d'architectures *Altérable/Java* et *Altérable/Contexte* et contient les trois plates-formes de la famille de mécanismes *Altérable*.

La famille *standard* montre que le cycle de vie est aussi sujet à de nombreuses variations. Toutefois pour que 17 plates-formes au cycle de vie différent soient regroupées, la marge de variation sur le cycle de vie semble plus petite que sur l'architecture ou sur les mécanismes fournis.

5.3.4 Familles générales

La matrice de distances globale reprend une partie des coefficients des sections précédentes, mais tient compte également des synergies transversales entre propriétés. Ses coefficients a_i sont les suivants :

Préparation avant mise à jour	0,5	Attente de l'altérabilité	1
Détection de l'altérabilité	1	Modification	0
Préparation à la reprise	0	Reprise	0
Attente de la terminaison	0,5	Détection de la terminaison	0,5
Nettoyage	0,5	Exécution dans la nouvelle version	0
Surveillance altérabilité	0,75	Stratégie d'accès	0,5
Moment de transformation	0,5	Contrôle de l'exécution	0,75
Mécanisme de chargement	0	Mécanisme multi-version	0,5
Synchronisation des versions	0,75	Mécanisme MàJ fonctions	1
Redémarrage	0,75	Mécanisme MàJ types	1
Mécanisme transformation données	1	Langage ciblé	0.5
Qui définit les critères d'altérabilité	0,75	Gestionnaire modifiable	0,75
Unité de mise à jour	0,5	Cloisonnement des versions	0,5
Tout autre aspect	0,25		

D'une manière générale, les coefficients ont été réduits par rapport aux distances précédentes pour tenir compte du grand nombre d'aspects à différencier. Les aspects les plus déterminants comme les

mécanismes de mise à jour des données, des types et des fonctions gardent des coefficients entre 1 et 0,75. Certains coefficients comme le langage ciblé ou la nature du gestionnaire ont été plus réduits que la normale en raison de fortes synergies avec d'autres aspects.

Les paramètres employés pour l'algorithme DBSCAN sont $\epsilon = 4,5$ et $M = 3$. Ils permettent de distinguer quatre familles de plates-formes.

La première famille nommée *Redémarrage* regroupe les 4 plates-formes de la famille de cycle de vie *Redémarrage*.

La deuxième famille nommée *C/statique* regroupe 3 plates-formes dont le critère d'altérabilité est l'atteinte d'un point de mise à jour placé par le développeur du programme. Ces plates-formes mettent à jour les fonctions en utilisant de l'indirection, accèdent progressivement aux données et mettent à jour les types en chargeant le nouveau code. Les plates-formes de cette famille partagent d'autres propriétés communes comme la portée du gestionnaire (globale) ou encore le langage ciblé (C).

La troisième famille nommée *Standard* regroupe 9 plates-formes fournissant elles-mêmes le script de contrôle, et demandant au développeur de mise à jour uniquement le nouveau code (éventuellement accompagné de *transformers*. Elles utilisent un (ou plusieurs) gestionnaire(s) à portée globale. Elles définissent elles mêmes les critères d'altérabilité qu'elles font surveiller dynamiquement par le gestionnaire. Cette famille regroupe les plates-formes aux propriétés les plus répandues. L'existence de cette famille était prévisible étant donné la forte fréquence de certaines propriétés et le fonctionnement de l'algorithme DBSCAN.

La quatrième famille nommée *Altérable* regroupe les 3 plates-formes de la famille de mécanismes *Altérable*.

La recherche de familles sur l'ensemble des propriétés permet de confirmer l'existence de certains archétypes de plates-formes comme les plates-formes redémarrant le programme ou celles qui n'attendent pas l'altérabilité. Il est intéressant de remarquer que certaines plates-formes se retrouvent exclues de familles suivant la catégorie des propriétés évaluées. Par exemple, DCP [39] a été exclue de la famille de mécanismes *Altérable* et de la famille générale du même nom. Il est donc plus difficile de distinguer les différents types de plates-formes lorsqu'elles sont étudiées dans leur globalité que lorsque seulement certaines de leurs propriétés sont étudiées.

5.4 Diversité des combinaisons et construction de mises à jour

Les précédents chapitres de cette partie ont établi un modèle général qui permet d'étudier de la même manière les plates-formes et leurs propriétés. Dans le chapitre 4, 42 plates-formes de l'état de l'art on été analysés ce qui a permis de construire la base de données homogénéisée utilisée dans ce chapitre.

La recherche de combinaisons fréquentes a montré que certaines propriétés sont très répandues, du fait de l'approche usuelle de la mise à jour dynamique (le développeur de mise à jour ne fournit que le nouveau code, la plate-forme définit le script de mise à jour). Cela est appelé à changer avec la nouvelle approche défendue dans ce manuscrit. Malgré cela, une grande variété des combinaisons employées peut être remarquée. Cette grande variété a également été constatée lors de l'identification des familles de plates-formes. Si certains archétypes de plates-formes ont été identifiés (plates-formes redémarrant le programme, considérant le programme comme toujours altérable,...), un grand nombre de plates-formes est exclu de toute famille ou est classifié dans une famille aux propriétés très variées (comme par exemple les familles *standards*).

La variété des plates-formes montre qu'il n'existe pas de solution universelle au problème des mises à jour dynamiques. Les analyses de ce chapitre n'ont pas montré de combinaison universelle de propriétés. Au contraire, elles ont montré que quelque soit le type de programme ciblé, plusieurs combinaisons de propriétés sont possibles. Il y a donc un besoin de variabilité dans le domaine. Comme en témoignent les plates-formes spécifiques à un programme ou un type de programme (comme, par exemple, Argus ou K42), les besoins diffèrent d'un type de programme à un autre. En disposant d'une plate-forme configurable fournissant plusieurs mécanismes prêts à être combinés, il est possible de choisir la combinaison la plus adaptée au moment d'appliquer la mise à jour, au lieu de devoir choisir cette combinaison avant de lancer le programme. Il devient même possible de choisir la combinaison la plus adaptée à la mise à jour tandis que l'approche classique ne permet de s'adapter qu'au programme uniquement.

Ce chapitre a identifié des synergies entre propriétés, fournissant de premiers résultats sur les besoins des mécanismes et sur les directives à suivre lors de la conception de plates-formes configurables. Ainsi, lorsqu'une plate-forme permet de choisir la stratégie d'accès aux données, il est préférable qu'elle permette de choisir la portée des gestionnaires (ou qu'au moins elle fasse changer leur portée pour s'accorder à la stratégie d'accès choisie). Ces résultats constituent une base pour les prochaines parties qui visent à étudier les exigences des mécanismes et l'implémentation de plates-formes configurables.

Deuxième partie

Exigences des mécanismes

Exigences des mécanismes

La partie précédente identifie les mécanismes de l'état de l'art et donne de premiers résultats sur leurs exigences. Le chapitre 5 liste des combinaisons fréquentes de mécanismes et éléments architecturaux, analysant la raison de ces fréquences. Lorsqu'un mécanisme donné est combiné presque exclusivement avec un autre mécanisme (ou un élément architectural), cela peut indiquer une dépendance. Par exemple, la grande majorité des plates-formes (71%) dont les critères d'altérabilité sont la quiescence des éléments modifiés suspendent le programme au moins en partie. Ce résultat semble indiquer une dépendance entre ce critère d'altérabilité et la possibilité de suspendre le programme.
L'objectif de cette partie est d'étudier les exigences des mécanismes courants identifiés dans la partie précédente. Pour un mécanisme donné, il s'agit d'identifier les informations nécessaires à l'expression de mises à jour utilisant ce mécanisme, ainsi que les fonctionnalités nécessaires à son bon fonctionnement. Par exemple, changer la valeur d'une variable nécessite d'avoir accès à l'endroit où est enregistrée cette valeur à partir du nom de la variable. En C, cela implique l'instrumentation du code pour retrouver la zone mémoire contenant la valeur a modifier.
Pour atteindre cet objectif, le chapitre 6 page 59 définit une sémantique pour programmes modifiables dynamiquement, basée sur le λ-calcul, ainsi qu'une sémantique pour programmes de mise à jour. La combinaison de ces deux sémantiques est utilisée dans le chapitre 7 page 67 pour étudier les mécanismes courants. Les limitations de ces deux sémantiques, ainsi que les extensions nécessaires pour satisfaire les exigences des mécanismes étudiés sont également discutées dans ce chapitre.

Chapitre 6

Une sémantique opérationnelle pour la mise à jour dynamique

Afin d'étudier les mécanismes de mise à jour et d'identifier leurs exigences, tant en informations nécessaires à l'expression d'une mise à jour, qu'en fonctionnalités du système d'exécution, ce chapitre détaille une sémantique opérationnelle conçue spécialement pour exprimer et exécuter des mises à jour dynamiques.

Il s'agit d'une combinaison de deux langages : λ_{DSU}, une extension du λ-calcul et un langage permettant d'exprimer des mises à jour dynamiques. Ces deux langages embarquent les fonctionnalités clés nécessaires à la modification dynamique tels que la possibilité de redéfinir des variables ou des fonctions. La section 6.1 détaille la sémantique de λ_{DSU} et la section 6.2 détaille la sémantique du langage pour mises à jour dynamiques. Dans chacune de ces sections, les extensions ajoutées pour permettre la mise à jour dynamique sont indiquées et discutées. La sémantique détaillée dans ce chapitre est une version allégée d'une sémantique plus complète discutée dans le prochain chapitre et détaillée en annexe B.

6.1 λ_{DSU} : λ-calcul étendu

λ_{DSU} est une version modifiée du λ-calcul dans l'objectif de permettre la mise à jour dynamique des programmes. Le choix du λ-calcul, comme base pour la définition du langage pour les programmes dynamiquement modifiables, est motivé d'une part par son caractère turing-complet, et d'autre part par la simplicité de sa sémantique.

Trois extensions ont été ajoutées au λ-calcul pour répondre à trois besoins élémentaires de mécanismes courants : la suspension totale ou partielle d'un programme, la redéfinition d'une variable et l'enregistrement des fonctions actives.

Un programme λ_{DSU} est composé de plusieurs fils d'exécution évaluant chacun un terme de manière entrelacée. Chaque fil peut être suspendu individuellement, ce qui permet de suspendre le programme, en partie ou dans sa totalité.

Dans λ_{DSU}, définir une variable avec le terme $\texttt{def}\ n = v\ \texttt{in}\ t$ enregistre la valeur v dans une mémoire partagée entre tous les fils d'exécution. Il est possible par la suite de changer la valeur en mémoire pour redéfinir la variable n.

Chaque fil d'exécution d'un programme λ_{DSU} possède sa propre pile d'appel dans laquelle sont enregistrées les fonctions actives (en cours d'appel). Son fonctionnement est similaire à celui d'un système d'exécution pour langage à pile classique.

6.1.1 Grammaire

La figure 6.1 détaille la syntaxe de λ_{DSU}. Comme précisé précédemment, un programme est constitué de plusieurs fils d'exécution évaluant chacun un terme de manière concurrente. Un fil suspendu est noté $a \underset{\mathbb{S}}{\blacktriangleright} t$ où a est son identifiant unique, $\mathbb{S}$ est sa pile d'appel et t est son terme en cours de réduction. Un fil actif (non suspendu) est noté $a \underset{\mathbb{S}}{\triangleright} t$. Sur la figure, les termes colorés en bleu ne peuvent apparaître que lors de la réduction du programme, ils ne peuvent être écrits par le développeur. De même, le développeur ne peut écrire de programme dont les fils sont déjà suspendus.

Programme

$$\mathcal{P} ::= \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_k$$

Contexte (programme)

$$\mathrm{C} ::= \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel [\cdot] \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k$$

Fil d'exécution

$$\mathcal{T} ::= a \underset{\mathbb{S}}{\rhd} t \mid a \underset{\mathbb{S}}{\blacktriangleright} t$$

Terme

$$t ::= i \mid x \mid \lambda x.t \mid t\ t' \mid (t) \mid \texttt{spawn}\, t \mid a \mid n \mid n : \lambda x.t \mid \texttt{return}\, t \mid \texttt{def}\ n = t\ \texttt{in}\ t' \mid c^n$$

Contexte (terme)

$$\mathrm{CT} ::= [\cdot]t' \mid v[\cdot] \mid (n : \lambda x.t')[\cdot] \mid \texttt{return}\,[\cdot] \mid \texttt{def}\ n = [\cdot]\ \texttt{in}\ t'$$

Valeurs

$$v ::= i \mid \lambda x.t \mid a$$

Mémoire

$$\mathbb{M} ::= \varnothing \mid (c \mapsto v) + \mathbb{M}$$

Pile d'appel

$$\mathbb{S} ::= \varnothing \mid [n] :: \mathbb{S}$$

Méta-variables

x : variable, c : cellule, a : id fil d'exécution, n : nom, i : entier, k, l : indices

FIGURE 6.1 – Grammaire des programmes

Les termes ont une grammaire similaire à celle d'un λ-calcul habituel. Les productions suivantes ont été ajoutées :

- `spawn` t crée un nouveau fil dont le terme est t
- $n : \lambda x.t$ indique que la fonction désignée est nommée n
- `return` t indique que la réduction complète de t provoque le retour de la fonction nommée au sommet de la pile d'appel.

Les deux dernières productions sont liées à l'enregistrement des fonctions actives. La résolution de ces termes est détaillée dans la prochaine sous-section.

Dans λ_{DSU}, une distinction est faite entre variables et noms. Une variable est définie par substitution dans le cadre d'un appel de fonction tandis qu'un nom est défini dans la mémoire globale. Il n'est donc pas possible de redéfinir des variables, mais il est possible de redéfinir des noms. De même, seuls les appels de fonction nommées (définies sous un nom) sont enregistrés dans les piles d'appel. Dans l'exemple suivant, x est une variable et n est un nom : `def` $n = \lambda x.t_1$ `in` t_2. L'impact de cette distinction est discuté à la fin de ce chapitre.

La mémoire globale $\mathbb{M}$ est constituée de paires cellules/valeurs. Lorsqu'une valeur nommée est définie (`def` $n = t$ `in` t'), une nouvelle cellule c est crée et l'association $(c \mapsto v)$ est enregistrée en mémoire. Toute référence au nom n est alors remplacée par c^n dans le terme t'. La notation c^n permet de conserver l'association nom/cellule dans le terme à réduire. Il est à remarquer que les valeurs peuvent être des entiers, des fonctions ou également des identifiants de fil d'exécution. La grammaire des valeurs v est d'ailleurs une sous-grammaire de la grammaire des termes t.

6.1.2 Règles de réduction

Chaque fil d'exécution réduit son propre terme en concurrence avec les autres, suivant les règles usuelles du λ-calcul auxquelles sont ajoutées des règles supplémentaires décrites dans ce chapitre. Lorsque la réduction du terme d'un fil aboutit à une unique valeur $(a \underset{\mathbb{S}}{\rhd} v)$, le fil est considéré comme terminé et il est supprimé du programme comme indiqué dans la règle [PTERM]. Les fils sont crées par réduction du terme `spawn` t qui crée un nouveau fil dont la pile d'appel est vide et dont le terme courant est t. Le terme `spawn` est alors remplacé par l'identifiant du nouveau fil (règle [PSPAWN]).

Les termes de λ_{DSU} sont évalués de gauche à droite. Ce n'est que lorsque l'évaluation du terme le plus à gauche mène à une valeur (ou à une fonction nommée) que le terme suivant est évalué. Dans cette section, seules les règles additionnelles sont détaillées. Les règles de réduction de λ_{DSU} sont présentées

$\boxed{(c \mapsto v) \in \mathbb{M}}$

$$\frac{}{(c \mapsto v) \in (c \mapsto v) + \mathbb{M}}\ \textsc{mem1} \qquad \frac{(c \mapsto v) \in \mathbb{M}}{(c \mapsto v) \in (c' \mapsto v') + \mathbb{M}}\ \textsc{mem2}$$

$\boxed{\mathbb{M}, \mathbb{S}, t \longrightarrow_t \mathbb{M}', \mathbb{S}', t'}$

$$\frac{\mathbb{M}, \mathbb{S}, t \longrightarrow_t \mathbb{M}', \mathbb{S}', t'}{\mathbb{M}, \mathbb{S}, \mathrm{CT}[t] \longrightarrow_t \mathbb{M}', \mathbb{S}', \mathrm{CT}[t']}\ \textsc{Tcont}$$

$$\frac{c \textit{ is fresh}}{\mathbb{M}, \mathbb{S}, \mathtt{def}\ n = v\ \mathtt{in}\ t \longrightarrow_t (c \mapsto [n \to c^n]v) + \mathbb{M}, \mathbb{S}, [n \to c^n]t}\ \textsc{Tdef}$$

$$\frac{c \textit{ is fresh}}{\mathbb{M}, \mathbb{S}, \mathtt{def}\ n = n' : \lambda x.t\ \mathtt{in}\ t' \longrightarrow_t (c \mapsto [n \to c^n]\lambda x.t) + \mathbb{M}, \mathbb{S}, [n \to c^n]t'}\ \textsc{TdefNfun}$$

$$\frac{}{\mathbb{M}, \mathbb{S}, \lambda x.t\ v \longrightarrow_t \mathbb{M}, \mathbb{S}, [x \to v]t}\ \textsc{Tapp} \qquad \frac{}{\mathbb{M}, \mathbb{S}, n : \lambda x.t\ v \longrightarrow_t \mathbb{M}, [n] :: \mathbb{S}, \mathtt{return}\ (\lambda x.t\ v)}\ \textsc{Tcall}$$

$$\frac{}{\mathbb{M}, [n] :: \mathbb{S}, \mathtt{return}\ (v) \longrightarrow_t \mathbb{M}, \mathbb{S}, v}\ \textsc{Tret} \qquad \frac{(c \mapsto i) \in \mathbb{M}}{\mathbb{M}, \mathbb{S}, c^n \longrightarrow_t \mathbb{M}, \mathbb{S}, i}\ \textsc{Tnint}$$

$$\frac{(c \mapsto \lambda x.t) \in \mathbb{M}}{\mathbb{M}, \mathbb{S}, c^n \longrightarrow_t \mathbb{M}, \mathbb{S}, n : \lambda x.t}\ \textsc{Tnfun}$$

$\boxed{\mathbb{M}, \mathcal{P} \longrightarrow_p \mathbb{M}', \mathcal{P}'}$

$$\frac{\mathbb{M}, \mathbb{S}, t \longrightarrow_t \mathbb{M}', \mathbb{S}', t'}{\mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\triangleright} t] \longrightarrow_p \mathbb{M}', \mathrm{C}[a \underset{\mathbb{S}'}{\triangleright} t']}\ \textsc{Pcont}$$

$$\frac{a' \textit{ is fresh}}{\mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\triangleright} \mathrm{CT}[\mathtt{spawn}\ t]] \longrightarrow_p \mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\triangleright} \mathrm{CT}[a']] \parallel a' \underset{\varnothing}{\triangleright} t}\ \textsc{Pspawn}$$

$$\frac{}{\mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel a \underset{\mathbb{S}}{\triangleright} v \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k \longrightarrow_p \mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k}\ \textsc{Pterm}$$

FIGURE 6.2 – Règles de réduction des programmes

dans la figure 6.2.

La définition d'une valeur nommée ($\mathtt{def}\ n = v\ \mathtt{in}\ t$) crée une nouvelle cellule à laquelle elle est associée en mémoire (règle TDEF). Toutes les occurrences du nom sont alors remplacées par la cellule créée dans le terme t. Le nom n est apposé à la cellule dans t afin de conserver le lien entre nom et cellule. Comme selon la sémantique standard de la substitution, seules les occurrences de n liées à la définition sont remplacées. Les occurrences de n dans la portée d'une nouvelle définition ne sont donc pas impactées.

$\mathbb{M} : \varnothing\ ,\ \mathtt{def}\ n = 8\ \mathtt{in}\ (\,(\ \mathtt{def}\ n = \lambda x.3\ \mathtt{in}\ n\,)\ n\,) \longrightarrow_t \mathbb{M} : (c \mapsto 8) + \varnothing\ ,\ (\mathtt{def}\ n = \lambda x.3\ \mathtt{in}\ n\,)c^n$ [TDEF]

Lorsqu'une cellule correspond à une fonction dans la mémoire, le nom sous lequel elle a été enregistrée est apposé à la fonction lors de sa réduction. Cela indique que la fonction est nommée et qu'il faudra l'ajouter à la pile lors de son appel (voir plus loin).

$\mathbb{M} : (c_1 \mapsto \lambda x.3) + (c \mapsto 8) + \varnothing\ ,\ {c_1}^n\ c^n \longrightarrow_t \mathbb{M} : (c_1 \mapsto \lambda x.3) + (c \mapsto 8) + \varnothing\ ,\ n : \lambda x.3\ c^n$ [TNFUN]

Lorsqu'un appel à une fonction nommée est réduit, le mot clé `return` est ajouté et le nom de la fonction est empilé. Ici, la pile est présentée à côté de la mémoire globale.

$\mathbb{S} : \varnothing, \mathbb{M} : (c_1 \mapsto \lambda x.3) + (c \mapsto 8) + \varnothing\ ,\ n : \lambda x.3\,8 \longrightarrow_t \mathbb{S} : [n] :: \varnothing, \mathbb{M}\ ,\ \mathtt{return}\ \lambda x.3\,8$ [TCALL]

L'appel de fonction est ensuite réduit selon les règles usuelles. Une fois l'appel entièrement réduit, le nom de la fonction est dépilé et le mot clé `return` est supprimé. Le nom dépilé est le nom le plus à

gauche dans la pile, c'est à dire le nom appelé le plus récent.

$\mathbb{S} : [n] :: \varnothing, \mathbb{M}\,,\ \texttt{return}\,3 \longrightarrow_t \mathbb{S} : \varnothing, \mathbb{M}\,,\ 3$ [TRET]

Remarque : Mémoire et fonction nommée

Il peut arriver qu'une fonction nommée soit enregistrée sous un autre nom. Une nouvelle cellule est créée pour contenir la fonction et cette dernière est associée à son nouveau nom. La règle [TDEFNFUN] assure cela.

$\mathbb{M} : \varnothing\,,\ \texttt{def}\ n = \lambda x.5\ \texttt{in}\ ((\texttt{def}\ m = n\ \texttt{in}\ m)\ 2)$
$\longrightarrow_t \mathbb{M} : (c \mapsto \lambda x.5) + \varnothing\,,\ (\texttt{def}\ m = c^n\ \texttt{in}\ m)\ 2$ [TDEF]
$\longrightarrow_t \mathbb{M} : (c \mapsto \lambda x.5) + \varnothing\,,\ (\texttt{def}\ m = n : \lambda x.5\ \texttt{in}\ m)\ 2$ [TNFUN]
$\longrightarrow_t \mathbb{M} : (c_1 \mapsto \lambda x.5) + (c \mapsto \lambda x.5) + \varnothing\,,\ {c_1}^m\ 2$ [TDEFNFUN]
$\longrightarrow_t \ldots$

Dans λ_{DSU}, l'assignation d'un nom est effectuée par copie, comme dans l'exemple précédent, les valeurs sont copiées en mémoire. Si, dans l'exemple précédent, la valeur associée au nom n est changée, celle de m restera la même. C'est le fonctionnement par défaut du langage C où les valeurs sont passées par copie, mais pas du langage Python où la définition de m aurait enregistré une référence vers la cellule c à la place.

6.2 Programme de mise à jour dynamique

Les mises à jour sont décrites sous forme de programmes s'exécutant en parallèle du programme modifié. Ces programmes sont constitués d'une série d'instructions exécutées séquentiellement dès qu'une garde d'activation est vérifiée.

Cette garde d'activation permet à la fois d'exprimer les exigences (au sens de l'étape (1) du cycle de vie présenté chapitre 3) ainsi que les critères d'altérabilité de la mise à jour. Les instructions possibles sont la suspension ou la reprise de fils d'exécution ainsi que la redéfinition de nom. Il s'agit des instructions de mise à jour élémentaires permettant de décrire les mises à jour les plus simples.

6.2.1 Grammaire du langage de mise à jour

Programme de mise à jour
$\mathcal{R} ::= u \mid [\![\mathrm{I}_1 \,..\, \mathrm{I}_l]\!]_u$

Mise à jour
$u ::= \bot \mid \texttt{upgrade}\,\mathrm{I}_1 \,..\, \mathrm{I}_l\ \texttt{when}\ g_1\ \texttt{until}\ g_2$

Instruction
$\mathrm{I} ::= n \leftarrow v \mid \texttt{suspend}\ a \mid \texttt{resume}\ a$

Garde d'activation
$g ::= n\ \texttt{in stack} \mid g \wedge g' \mid g \vee g' \mid \neg g \mid (g)$

FIGURE 6.3 – Grammaire du langage de mise à jour

La figure 6.3 détaille la grammaire du langage de mise à jour. Un programme de mise à jour s'écrit comme une suite d'instructions associée à deux gardes : $\texttt{upgrade}\,\mathrm{I}_1 \,..\, \mathrm{I}_l\ \texttt{when}\ g_1\ \texttt{until}\ g_2$. Les instructions peuvent commander la redéfinition d'un nom ($n \leftarrow v$), la suspension ou la reprise d'un fil d'exécution ($\texttt{suspend}\ a$, $\texttt{resume}\ a$). La première garde indique à quel moment le programme de mise à jour doit commencer à exécuter les instructions $\mathrm{I}_1 \,..\, \mathrm{I}_l$. Quand cette garde est vérifiée, le programme de mise à jour devient *actif* et les instructions sont réduites. La deuxième garde indique quand le programme de mise à jour n'est plus valable et doit être abandonné. La garde `when` permet d'exprimer les critères d'altérabilités de la mise à jour, par exemple, en indiquant des fonctions qui doivent être quiescentes. Sur la figure 6.3, les productions en bleu ne peuvent apparaître que lors de la réduction d'un programme de mise à jour.

Un programme de mise à jour a deux formes : une forme *passive* notée $\texttt{upgrade}\,\mathrm{I}_1 \,..\, \mathrm{I}_l\ \texttt{when}\ g_1\ \texttt{until}\ g_2$ et une forme *active* notée $[\![\mathrm{I}_1 \,..\, \mathrm{I}_l]\!]_u$. Un programme de mise à jour devient *actif* dès que la garde `when` est vérifiée. Dans la formule active le u en indice indique la prochaine mise à jour à effectuer quand toutes les instructions $\mathrm{I}_1 \,..\, \mathrm{I}_l$ ont été exécutées.

6.2.2 Réduction d'une mise à jour

La figure 6.4 présente les règles de réduction d'un programme de mise à jour. De la même manière que la flèche $\longrightarrow_p$ indique la réduction d'un programme seul, la flèche $\longrightarrow_r$ indique la réduction d'un programme de mise à jour seul (c'est à dire sans que le programme modifié ne soit réduit). Lors d'une réduction *normale* (flèche $\longrightarrow$), le programme et le programme de mise à jour sont réduits de manière entrelacée. Les règles de réduction de **getCells**, fonction retournant les cellules associées à un nom donné dans le programme sont données en annexe B page 155. **getCells** $n\ \mathcal{P}$ parcourt la structure des termes de $\mathcal{P}$ à la recherche de l'ensemble des c^n.

Les exécutions du programme modifié et du programme de mise à jour sont entrelacées et chaque pas de réduction de l'un comme de l'autre est atomique. Tant que le programme de mise à jour est *passif*, ses gardes `when` et `until` sont évaluées. Si la garde `when` est évaluée à vrai, le programme de mise à jour passe en mode *actif*. Si la garde `until` est évaluée à vrai, la mise à jour est abandonnée. Si les deux gardes sont vraies, le programme de mise à jour passe en mode *actif* et une fois les instructions exécutées, la mise à jour est abandonnée.

$\mathbb{M} : (c \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} c^n\ 3\ ,\ \texttt{upgrade}\ n \leftarrow \lambda x.5\ \texttt{when}\ \mathit{true}\ \texttt{until}\ \mathit{false}$

$\longrightarrow \mathbb{M} : (c \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} c^n\ 3\ ,\ [\![n \leftarrow \lambda x.5]\!]_{\texttt{upgrade}\ n \leftarrow \lambda x.5\ \texttt{when}\ \mathit{true}\ \texttt{until}\ \mathit{false}}$

$\mathbb{M} : (c \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} c^n\ 3\ ,\ \texttt{upgrade}\ n \leftarrow \lambda x.5\ \texttt{when}\ \mathit{false}\ \texttt{until}\ \mathit{true}$

$\longrightarrow \mathbb{M} : (c \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} c^n\ 3\ ,\ \bot$

$\mathbb{M} : (c^n \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} c^n\ 3\ ,\ \texttt{upgrade}\ n \leftarrow \lambda x.5\ \texttt{when}\ \mathit{true}\ \texttt{until}\ \mathit{true}$

$\longrightarrow \mathbb{M} : (c^n \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} c^n\ 3\ ,\ [\![n \leftarrow \lambda x.5]\!]_\bot$

La réduction des instructions de suspension et de reprise des fils d'exécution ont pour effet de faire passer les fils ciblés en état actif ou suspendu de manière immédiate. La redéfinition d'un nom utilise la fonction **getCells** pour extraire des termes du programme toutes les cellules associées à un nom donné. Pour chacune de ces cellules, une nouvelle association cellule/valeur est ajoutée à la mémoire globale. Les anciennes associations cellules/valeurs ne sont pas supprimées, l'ordre de résolution dans la mémoire assure que les associations les plus récentes seront utilisées, rendant les anciennes associations inutiles. Par conséquent, la taille de la mémoire augmente à chaque redéfinition. Il est concevable de définir un mécanisme de ramasse-miettes pour éliminer les associations obsolètes de la mémoire, mais cela présente peu d'intérêt d'un point de vue théorique. Il est important de noter que *toutes* les cellules associées à un même nom sont affectées par une redéfinition. Si un même n a fait l'objet de deux définitions différentes par le passé, redéfinir n affectera les deux cellules qui lui correspondent (ce qui peut amener à des termes irréductibles comme dans l'exemple suivant).

$\mathbb{M} : (c_1 \mapsto \lambda x.3) + (c_0 \mapsto 5) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} {c_1}^n\ {c_0}^n\ ,\ [\![n \leftarrow 8]\!]_\bot$

$\longrightarrow \mathbb{M} : (c_0 \mapsto 8) + (c_1 \mapsto 8) + (c_1 \mapsto \lambda x.3) + (c_0 \mapsto 8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} {c_1}^n\ {c_0}^n\ ,\ \bot$

$\longrightarrow \ldots \longrightarrow \mathbb{M} : (c_0 \mapsto 8) + (c_1 \mapsto 8) + (c_1 \mapsto \lambda x.3) + (c_0 \mapsto 8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} 8\ 8\ ,\ \bot$

Cette situation est due à un manque d'information dans l'expression de la mise à jour. Sans plus d'information que le nom d'une valeur pour la désigner, il est impossible de différencier les différentes valeurs enregistrées sous ce même nom. L'exemple suivant montre une mise à jour où une fonction nommée est redéfinie.

$\mathbb{M} : \varnothing\ ,\ a \underset{\varnothing}{\rhd} \texttt{def}\ n = \lambda x.8\ \texttt{in}\ n\ 1\ ,\ \texttt{upgrade}\ \texttt{suspend}\ a\ n \leftarrow \lambda x.5\ \texttt{resume}\ a\ \texttt{when}\ \neg n\ \texttt{in}\ \texttt{stack}\ \texttt{until}\ \mathit{true}$

$\longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} {c_0}^n\ 1\ ,\ \texttt{upgrade}\ \texttt{suspend}\ a\ n \leftarrow \lambda x.5\ \texttt{resume}\ a\ \texttt{when}\ \neg n\ \texttt{in}\ \texttt{stack}\ \texttt{until}\ \mathit{true}$

$\longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} {c_0}^n\ 1\ ,\ [\![\texttt{suspend}\ a\ n \leftarrow \lambda x.5\ \texttt{resume}\ a]\!]_\bot$

$\longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\blacktriangleright} {c_0}^n\ 1\ ,\ [\![n \leftarrow \lambda x.5\ \texttt{resume}\ a]\!]_\bot$

$\longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.5) + (c_0 \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\blacktriangleright} {c_0}^n\ 1\ ,\ [\![\texttt{resume}\ a]\!]_\bot$

$\longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.5) + (c_0 \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} {c_0}^n\ 1\ ,\ \bot$

$\longrightarrow \ldots \longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.5) + (c_0 \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{[n]:\varnothing}{\rhd} \texttt{return}\ \lambda x.5\ 1\ ,\ \bot$

$\longrightarrow \ldots \longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.5) + (c_0 \mapsto \lambda x.8) + \varnothing\ ,\ a \underset{\varnothing}{\rhd} 5\ ,\ \bot$

$\boxed{n \in \mathbb{S}}$

$$\frac{}{n \in [n] :: \mathbb{S}}\ \textsc{stack1} \qquad \frac{n \in \mathbb{S}}{n \in [n'] :: \mathbb{S}}\ \textsc{stack2}$$

$\boxed{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2}$

$$\frac{\mathbb{M}_1, \mathcal{P}_1 \longrightarrow_p \mathbb{M}_2, \mathcal{P}_2}{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R} \longrightarrow \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}}\ \textsc{Rexec} \qquad \frac{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow_r \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2}{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2}\ \textsc{Rupdate}$$

$\boxed{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow_r \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2}$

$$\frac{}{\mathbb{M}, \mathcal{P}, [\![]\!]_u \longrightarrow_r \mathbb{M}, \mathcal{P}, u}\ \textsc{Rend} \qquad \frac{\mathcal{P} \vdash \neg g_1 \quad \mathcal{P} \vdash g_2}{\mathbb{M}, \mathcal{P}, \texttt{upgrade}\ \mathrm{I}_1 .. \mathrm{I}_l\ \texttt{when}\ g_1\ \texttt{until}\ g_2 \longrightarrow_r \mathbb{M}, \mathcal{P}, \bot}\ \textsc{Rdiscard}$$

$$\frac{\mathcal{P} \vdash g_1 \quad \mathcal{P} \vdash \neg g_2}{\mathbb{M}, \mathcal{P}, \texttt{upgrade}\ \mathrm{I}_1 .. \mathrm{I}_l\ \texttt{when}\ g_1\ \texttt{until}\ g_2 \longrightarrow_r \mathbb{M}, \mathcal{P}, [\![\mathrm{I}_1 .. \mathrm{I}_l]\!]_{\texttt{upgrade}\ \mathrm{I}_1 .. \mathrm{I}_l\ \texttt{when}\ g_1\ \texttt{until}\ g_2}}\ \textsc{Rreconf}$$

$$\frac{\mathcal{P} \vdash g_1 \quad \mathcal{P} \vdash g_2}{\mathbb{M}, \mathcal{P}, \texttt{upgrade}\ \mathrm{I}_1 .. \mathrm{I}_l\ \texttt{when}\ g_1\ \texttt{until}\ g_2 \longrightarrow_r \mathbb{M}', \mathcal{P}', [\![\mathrm{I}_1 .. \mathrm{I}_l]\!]_\bot}\ \textsc{Roneshot}$$

$$\frac{\textbf{getCells}\ n\ \mathcal{P} = c_1 .. c_l \quad \mathbb{M}' = (c_1 \mapsto v) + .. + (c_l \mapsto v) + \mathbb{M}}{\mathbb{M}, \mathcal{P}, [\![n \leftarrow v\ \mathrm{I}_1 .. \mathrm{I}_l]\!]_u \longrightarrow_r \mathbb{M}', \mathcal{P}, [\![\mathrm{I}_1 .. \mathrm{I}_l]\!]_u}\ \textsc{Rupd}$$

$$\frac{}{\mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\triangleright} t], [\![\texttt{suspend}\ a\ \mathrm{I}_1 .. \mathrm{I}_l]\!]_u \longrightarrow_r \mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\blacktriangleright} t], [\![\mathrm{I}_1 .. \mathrm{I}_l]\!]_u}\ \textsc{Rsups1}$$

$$\frac{}{\mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\blacktriangleright} t], [\![\texttt{suspend}\ a\ \mathrm{I}_1 .. \mathrm{I}_l]\!]_u \longrightarrow_r \mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\blacktriangleright} t], [\![\mathrm{I}_1 .. \mathrm{I}_l]\!]_u}\ \textsc{Rsups2}$$

$$\frac{}{\mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\triangleright} t], [\![\texttt{resume}\ a\ \mathrm{I}_1 .. \mathrm{I}_l]\!]_u \longrightarrow_r \mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\triangleright} t], [\![\mathrm{I}_1 .. \mathrm{I}_l]\!]_u}\ \textsc{Rres1}$$

$$\frac{}{\mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\blacktriangleright} t], [\![\texttt{resume}\ a\ \mathrm{I}_1 .. \mathrm{I}_l]\!]_u \longrightarrow_r \mathbb{M}, \mathrm{C}[a \underset{\mathbb{S}}{\triangleright} t], [\![\mathrm{I}_1 .. \mathrm{I}_l]\!]_u}\ \textsc{Rres2}$$

$\boxed{\mathcal{P} \vdash g}$

$$\frac{n \in \mathbb{S}}{\mathrm{C}[a \underset{\mathbb{S}}{\triangleright} t] \vdash n\ \texttt{in stack}}\ \textsc{Gisa} \qquad \frac{n \in \mathbb{S}}{\mathrm{C}[a \underset{\mathbb{S}}{\blacktriangleright} t] \vdash n\ \texttt{in stack}}\ \textsc{Ginp} \qquad \frac{n \notin \mathbb{S}}{a \underset{\mathbb{S}}{\triangleright} t \vdash \neg(n\ \texttt{in stack})}\ \textsc{Gnisa}$$

$$\frac{n \notin \mathbb{S}}{a \underset{\mathbb{S}}{\blacktriangleright} t \vdash \neg(n\ \texttt{in stack})}\ \textsc{Gninp} \qquad \frac{\mathcal{T}_1 \vdash \neg(n\ \texttt{in stack}) \quad .. \quad \mathcal{T}_l \vdash \neg(n\ \texttt{in stack})}{\mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \vdash \neg(n\ \texttt{in stack})}\ \textsc{Gnithl}$$

$$\frac{\mathcal{P} \vdash g_1 \quad \mathcal{P} \vdash g_2}{\mathcal{P} \vdash g_1 \wedge g_2}\ \textsc{Gand} \qquad \frac{\mathcal{P} \vdash g_1}{\mathcal{P} \vdash g_1 \vee g_2}\ \textsc{Gor1} \qquad \frac{\mathcal{P} \vdash g_2}{\mathcal{P} \vdash g_1 \vee g_2}\ \textsc{Gor2} \qquad \frac{\mathcal{P} \vdash \neg g_1}{\mathcal{P} \vdash \neg(g_1 \wedge g_2)}\ \textsc{Gnand1}$$

$$\frac{\mathcal{P} \vdash \neg g_2}{\mathcal{P} \vdash \neg(g_1 \wedge g_2)}\ \textsc{Gnand2} \qquad \frac{\mathcal{P} \vdash \neg g_1 \quad \mathcal{P} \vdash \neg g_2}{\mathcal{P} \vdash \neg(g_1 \vee g_2)}\ \textsc{Gnor}$$

FIGURE 6.4 – Règles de réduction des mises à jour

Il est à noter que rien n'empêche l'exécution du programme de se poursuivre entre le moment où le programme de mise à jour devient *actif* et le moment où la première instruction est exécutée (ici, `suspend` a). Ce problème peut se régler en posant comme règle que lorsque le programme de mise à jour devient *actif*, il exécute immédiatement la première instruction. Cette situation ainsi que les autres limitations de cette sémantique simplifiée sont discutées dans le prochain chapitre.

6.3 Autres sémantiques

D'autres sémantiques adressant des problématiques liées à la mise à jour dynamique ont été proposées.

Dans [79], Zhang et al. définissent trois sémantiques pour l'adaptation de programmes. Elles expriment les adaptations d'un programme en décrivant les contraintes sous lesquelles l'adaptation d'un programme doit se passer (quand elle peut avoir lieu, restrictions du comportement du programme pendant l'adaptation, ...) ainsi que la spécification du programme avant et après l'adaptation. Les spécifications et les contraintes sont des formules LTL (Linear Temporal Logic). Les sémantiques proposées par Zhang et al. n'offrent pas une granularité suffisante pour l'étude des mécanismes de mise à jour dynamique.

λ_{marsh} [10] est une sémantique basée sur celle du λ-calcul et lie les valeurs dynamiquement, afin de pouvoir les relier pendant l'exécution du programme. Il est possible de placer des marques dans un terme et d'empaqueter des variables sous cette marque. Les valeurs de ces variables sont enregistrées et peuvent être dépaquetées sous une autre marque, ce qui a pour effet de relier les variables dans le nouveau contexte, avec de nouvelles valeurs. L'exemple suivant empaquette la valeur 5 pour la variable x.

`mark` M `in let` $x = 5$ `in marshall` $M\ (x)$

Il est possible de dépaqueter dans un autre contexte grâce à `unmarshall` $M\ (x)$. λ_{marsh} est conçu pour exprimer des programmes distribués dont les valeurs des variables peuvent être reçues depuis un autre programme. Le mécanisme d'empaquetage des variables qu'il propose est une alternative intéressante à la mémoire globale utilisée dans λ_{DSU}.

PROTEUS [71] est composé de deux sémantiques permettant de décrire des programmes modifiables dynamiquement ainsi que des programmes de mise à jour. La sémantique de PROTEUS est proche de celle du langage C et inclue une expression `update` qui déclenche l'application des mises à jour. Les mises à jour sont un ensemble d'opérations pouvant être :

— Modification d'un type
— Ajout d'un type
— Modification d'une fonction ou d'une variable liée
— Ajout d'une fonction ou d'une variable liée

Un système de cœrcition de type permet aux mises à jour de n'être appliquées que lorsqu'elles conservent le bon typage du programme. PROTEUS utilise un tas pour enregistrer les variables et les fonctions définies dans le programme.

La sémantique de PROTEUS adopte une approche différente de λ_{DSU}. Les mises à jour sont exprimées sous forme de liste de modifications faites au programme et l'activation des mises à jour est gérée par une expression à l'intérieur du programme sous des contraintes de typage tandis que λ_{DSU} permet d'exprimer des critères d'altérabilité avec la condition `when`. λ_{DSU} ne permet pas d'ajouter de valeurs en mémoire, contrairement à PROTEUS. Bien que cette fonctionnalité soit utile pour appliquer des mises à jour, elle n'est pas nécessaire pour les mécanismes étudiés dans le chapitre 7 et n'a donc pas été ajoutée dans λ_{DSU}.

6.4 Premiers résultats

Les extensions apportées au λ-calcul permettent de redéfinir des valeurs nommées. La mémoire globale introduit une indirection naturelle à λ_{DSU}. Une fois un nom défini, une cellule conserve la valeur qui lui est associée. Il suffit alors de changer la valeur enregistrée dans la cellule pour redéfinir le nom. Afin d'identifier les cellules qui correspondent à un nom, le nom est ajouté à chaque occurrence de la cellule dans un terme (c^n). L'alternative est de remplacer les noms par leurs valeurs dans les termes dès leur définition. Une redéfinition consiste alors à remplacer ces valeurs par les nouvelles dans les termes. Cette technique nécessite également de savoir, pour chaque valeur nommée, sous quel nom elle a été définie. Dans l'exemple suivant, le nom n est défini et remplacé directement dans le terme. Il est ensuite redéfini avec une nouvelle valeur. L'exemple propose un moyen de retenir, dans les termes, le nom de chaque valeur nommée.

$\texttt{def}\ n\ = 5\ \texttt{in}\ \lambda x.2\,(n) \longrightarrow \lambda\, x.2\,(n : 5) \xrightarrow[n \leftarrow 7]{} \lambda\, x.2\,(n : 7)$

La capacité de suspendre ou reprendre l'exécution des fils d'exécution nécessite un moyen de les distinguer. Ici, chaque fil a un identifiant unique utilisé pour le désigner, ce qui est courant dans la plupart des systèmes d'exécution. La sémantique de λ_{DSU} prévoit également que les fils puissent être actifs ou inactifs, ce qui n'est pas standard dans la plupart des langages (par exemple, en Python, il n'existe pas de solution native permettant de suspendre les fils d'exécution).

Les mécanismes liés à la pile (empilement de nom, appels de fonctions nommées), permettent d'évaluer simplement si une fonction est quiescente ou non. Avant de redéfinir une fonction nommée, il suffit de parcourir les piles de chaque fil pour vérifier si cette fonction est quiescente ou non. Pour permettre cela, le mot clé `return` ainsi que l'étiquetage des fonctions dans les termes $(n : \lambda x.t)$ permettent de savoir quand une fonction nommée retourne (et donc quand dépiler un nom de la pile) et quel nom empiler lors de l'appel d'une fonction nommée.

Le langage pour les programmes de mise à jour suit un paradigme impératif qui convient aisément à la description de mises à jour : les modifications sont appliquées les unes après les autres. Les mises à jour ne sont activées que si une condition `when` est vérifiée. Cette condition peut être utilisée pour exprimer les critères d'altérabilité de la mise à jour. Les mises à jour ont également une condition `until` qui permet de limiter dans le temps la mise à jour. En effet, une modification peut ne plus avoir d'intérêt si l'exécution du programme a dépassé un certain point (par exemple, si la fonction redéfinie ne sera plus jamais appelée dans le programme, il est inutile de la redéfinir).

Les extensions listées ci-dessus permettent d'accomplir les tâches élémentaires d'une mise à jour dynamique (redéfinir un nom, suspendre un fil, détecter la quiescence d'une fonction, ...). Elles sont cependant limitées dans leurs capacités. Le prochain chapitre propose des implémentations des mécanismes de mise à jour courant et expose les limitations de λ_{DSU}. De nouvelles extensions sont proposées pour palier à ces limitations et permettre l'écriture de mises à jour plus complexes.

Chapitre 7

Étude de mises à jour dynamiques

Ce chapitre utilise la sémantique détaillée dans le chapitre précédent pour exprimer des mécanismes de mise à jour dynamique *classiques* et étudier leur fonctionnement. L'objectif final est d'identifier, pour chacun de ces mécanismes, quelles sont leurs exigences ; en termes de fonctionnalités du système d'exécution, ou d'informations accessibles. Par exemple, le chapitre précédent a montré que pour changer une valeur dans un programme, il est nécessaire de pouvoir l'identifier, par un nom ou par un autre moyen.

Les prochaines sections détaillent et discutent l'expression de l'altérabilité, des stratégies d'accès aux données, du redémarrage des fils d'exécution et de la redéfinition des fonctions. Certains de ces mécanismes nécessitent d'apporter des extensions à la sémantique du chapitre précédent. Ces extensions sont détaillées à chaque fois. La version complète de la sémantique incluant ces extensions est détaillée en annexe B.2 page 155.

Ces mécanismes ont été choisis car ils sont les plus fréquents parmi les plates-formes de la littérature et parce qu'ils constituent les briques de base de mises à jour plus complexes. Par exemple, Upstare [50] utilise un mécanisme semblable au redémarrage de fils d'exécution pour reconstruire les piles d'appel de ces fils.

7.1 Altérabilité

Les critères d'altérabilité indiquent sous quelles conditions une mise à jour peut être appliquée de manière sûre (sans provoquer d'erreurs). Dans la sémantique des programmes de mise à jour, ces critères peuvent s'exprimer avec la condition `when`.

7.1.1 Quiescence d'une fonction

Une fonction est quiescente à condition qu'elle ne soit pas en cours d'appel et qu'elle ne soit pas appelée dans un futur proche (pendant le temps d'appliquer la mise à jour). Les plates-formes implémentent cela en s'assurant qu'aucun appel de cette fonction n'est dans aucune pile d'exécution et en suspendant l'exécution du programme.

Les piles d'appel de λ_{DSU} et les instructions du langage de mise à jour permettent de reproduire ce fonctionnement. La quiescence d'une fonction f s'écrit alors ainsi :

`upgrade suspend` a ... `when` $\neg f$ `in stack until`...

Dans le cas d'un programme ayant plusieurs fils d'exécution susceptibles d'appeler f, le programme de mise à jour aurait une instruction `suspend` pour chacun de ces fils. Comme précisé dans le chapitre précédent, il n'y a aucune garantie que les fils à suspendre ne poursuivront pas leur exécution entre la détection de l'altérabilité (passage du programme de mise à jour en mode *actif* après vérification de la condition `when`) et l'exécution de la première instruction ou encore entre l'exécution des instructions.

Par exemple, cette résolution est possible :

$$\mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing\,,\ a_0 \underset{\varnothing}{\triangleright} {c_0}^f\ 1 \parallel\ a_1 \underset{\varnothing}{\triangleright} {c_0}^f\ 2\,,\ [\![\texttt{suspend}\ a_0\ \texttt{suspend}\ a_1]\!]_\perp$$

$$\longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing\,,\ a_0 \underset{\varnothing}{\blacktriangleright} {c_0}^f\ 1 \parallel\ a_1 \underset{\varnothing}{\triangleright} {c_0}^f\ 2\,,\ [\![\texttt{suspend}\ a_1]\!]_\perp \quad [\textsc{Rsusp1}]$$

$$\longrightarrow \ldots \longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing\,,\ a_0 \underset{\varnothing}{\blacktriangleright} {c_0}^f\ 1 \parallel\ a_1 \underset{[f]:\varnothing}{\triangleright} \texttt{return}\ \lambda x.8\ 2\,,\ [\![\texttt{suspend}\ a_1]\!]_\perp \quad [\textsc{Rsusp1}]$$

La fonction f n'est plus quiescente alors que le programme de mise à jour poursuit son exécution comme si f était bien quiescente. La sémantique complète présentée en annexe B.2 résout ce problème en modifiant l'instruction `suspend` de sorte à ce qu'elle puisse suspendre plusieurs fils à la fois

$$\frac{}{\mathbb{M}, \mathbb{S}, L : t \longrightarrow_t \mathbb{M}, \mathbb{S}, t}\ \textsc{Tlbl} \qquad \frac{}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} L : t], \mathcal{C} \vdash a@L}\ {}^{\textsc{at}}\_^{\textsc{L}}$$

FIGURE 7.1 – Réduction d'un point de mise à jour

(`suspend` $a_1..a_k$). Tous les fils sont alors suspendus de manière atomique. Le programme peut toujours poursuivre son exécution entre le passage de la mise à jour en mode *actif* et l'exécution de la première instruction. La sémantique complète impose également que la première instruction soit exécutée dès que le programme de mise à jour devient *actif*.

7.1.2 Point de mise à jour

Un autre critère d'altérabilité courant utilise des points de mise à jour indiqués dans le programme. Pour que le critère soit satisfait, l'exécution du programme doit avoir atteint un point de mise à jour. Le critère peut être l'atteinte d'un point en particulier ou de n'importe quel point.

Un tel critère d'altérabilité ne peut pas être exprimé par la sémantique simplifiée présentée au chapitre précédent. La sémantique complète de l'annexe B.2 apporte les extensions suivantes qui le permettent :

— Un nouveau terme $L : t$ qui indique un point de mise à jour nommé L.

— Une nouvelle condition $a@L$ stipulant que le fil a doit avoir atteint le point L

Le symbole L est une étiquette qui peut être attachée à un terme pour indiquer un point de mise à jour. la réduction du terme $L : t$ produit t et consomme l'étiquette L. Si le fil d'identifiant a est sur le point de réduire un terme de la forme $L : t$, alors la condition $a@L$ est vraie.

L'exemple suivant montre un programme ayant un point de mise à jour L et un programme de mise à jour ayant comme critère d'altérabilité l'atteinte de ce point.

$a \underset{\varnothing}{\triangleright}$ `def` $f = \lambda x.8$ `in` $L : f\,1$, `upgrade` ... `when` $a@L$ `until` *true*

Le programme de mise à jour ne deviendra *actif* que lorsque le terme courant du fil a aura le point L :

$\mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing$, $a \underset{\varnothing}{\triangleright} L : {c_0}^f\,1$, `upgrade` ... `when` $a@L$ `until` *true*

$\longrightarrow \mathbb{M} : (c_0 \mapsto \lambda x.8) + \varnothing$, $a \underset{\varnothing}{\triangleright} L : {c_0}^f\,1$, $[\![...]\!]_\perp$ [RONESHOT]

De même que pour la quiescence des fonctions, si le fil doit rester au point L toute la durée de la mise à jour, la première instruction de la mise à jour doit être `suspend` a.

Remarque : Altérabilité et condition `when`

Les gardes `when` n'expriment pas nécessairement des critères d'altérabilité. Elles peuvent, comme montré dans la prochaine section, indiquer quand des données doivent être mises à jour. Par définition, ces gardes expriment seulement quand le programme de mise à jour peut devenir *actif*.

7.2 Modification des données

Dans λ_{DSU}, les données modifiables sont uniquement les valeurs nommées : des entiers ou des fonctions. Ces données peuvent être modifiées uniquement par redéfinition. Il n'y a pas de *transformer* pour mettre à jour des données existantes dans λ_{DSU}. C'est donc l'instruction $n \leftarrow v$ qui permet de mettre à jour un entier.

Cette section traite des stratégies d'accès et de transformation des données : accès immédiat ou progressif et transformation instantanée ou retardée.

7.2.1 Accès aux données

Accès immédiat

L'accès immédiat est le plus facile à implémenter. Par défaut, l'instruction $n \leftarrow v$ accède immédiatement à la cellule correspondante. Il n'y a donc pas besoin d'écrire un programme de mise à jour spécifique pour cette stratégie.

$$\frac{\mathbb{M}_1, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[t]], \mathcal{R} \longrightarrow_r \mathbb{M}_2, \mathcal{P}', \mathcal{R}'}{\mathbb{M}_1, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[\texttt{breakpoint}\ t]], \mathcal{R} \longrightarrow \mathbb{M}_2, \mathcal{P}', \mathcal{R}'}\ \textsc{Rbreak}$$

$$\frac{\mathbb{M}_1, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[t]], \mathcal{R} \not\longrightarrow_r}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[\texttt{breakpoint}\ t]], \mathcal{R} \longrightarrow \mathbb{M}_1, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[t]], \mathcal{R}}\ \textsc{Rnbreak}$$

FIGURE 7.2 – Règles de réduction des `breakpoint`

Accès progressif

L'accès progressif nécessite de n'accéder à la donnée qu'au moment où elle est utilisée par le programme. La condition `when` permet d'exprimer cela. Il suffit d'écrire une mise à jour dont la condition exprime le moment où la valeur nommée est utilisée, c'est-à-dire au moment où la cellule qui lui est associée en mémoire est résolue.

La sémantique simplifiée ne permet pas d'exprimer une telle condition. L'extension permettant de placer des points de mise à jour permet cela à condition de placer un point spécifique à chaque utilisation de la variable comme dans l'exemple suivant.

$\mathbb{M} : \varnothing\,,\ a \underset{\varnothing}{\triangleright} \texttt{def}\ n = 3\ \texttt{in}\ \lambda x.8\, L : n\,,\ \texttt{upgrade}\ n \leftarrow 5\ \texttt{when}\ a@L\ \texttt{until}\ \mathit{true}$

$\longrightarrow \mathbb{M} : (c_0 \mapsto 3) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.8\, L : {c_0}^n\,,\ \texttt{upgrade}\ n \leftarrow 5\ \texttt{when}\ a@L\ \texttt{until}\ \mathit{true}$ [TDEF]

$\longrightarrow \mathbb{M} : (c_0 \mapsto 3) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.8\, L : {c_0}^n\,,\ [\![n \leftarrow 5]\!]_\perp$ [RONESHOT]

$\longrightarrow \mathbb{M} : ({c_0}^n \mapsto 5) + ({c_0}^n \mapsto 3) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.8\, L : {c_0}^n\,,\ [\![]\!]_\perp$ [RUPD]

Le nom n est mis à jour au moment où il est utilisé, c'est à dire au moment où la cellule ${c_0}^n$ est sur le point d'être résolue.

Le placement des différents points de mise à jour est fastidieux et maintenir une table des points associés à chaque noms n'est pas pratique. Il serait donc préférable de pouvoir exprimer l'utilisation d'un nom comme condition. Par exemple, la condition suivante pourrait être ajoutée : $a\,\texttt{use}\,n$. Elle deviendrait vraie dès qu'un fil d'exécution est sur le point de résoudre une cellule associée au nom n (${c_k}^n$). Cette condition ne fait pas partie de la sémantique présentée en annexe B.2 mais son comportement serait équivalent à l'utilisation des points de mise à jour présentée ci-dessus.

Les réductions du programme et du programme de mise à jour étant entrelacées, il est possible que le programme réduise la cellule sans laisser la main au programme de mise à jour :

$\mathbb{M} : (c_0 \mapsto 3) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.8\, {c_0}^n\,,\ \texttt{upgrade}\ n \leftarrow 5\ \texttt{when}\ a\ \texttt{use}\ n\ \texttt{until}\ \mathit{true}$

$\longrightarrow (c_0 \mapsto 3) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.8\, 3\,,\ \texttt{upgrade}\ n \leftarrow 5\ \texttt{when}\ a\ \texttt{use}\ n\ \texttt{until}\ \mathit{true}$ [TNINT]

Pour s'assurer que le programme laisse la main au programme de mise à jour, la sémantique complète ajoute un nouveau terme : `breakpoint` t. Lorsqu'un fil d'exécution réduit ce terme, il consomme le mot clé `breakpoint` et remplace le terme par t. Le programme de mise à jour est immédiatement réduit d'un pas, sans laisser la main au programme.

La figure 7.2 détaille les règles de réduction des `breakpoint`. La règle RBREAK montre le cas où l'état du programme permet de réduire le programme de mise à jour (au moins une des gardes `when` ou `until` est vérifiée). La réduction de $\mathcal{R}$ transforme $\mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[t]]$ en $\mathcal{P}'$ et $\mathbb{M}_1$ en $\mathbb{M}_2$. Le `breakpoint` est donc supprimé avant de réduire d'un pas le programme de mise à jour. La règle RNBREAK montre le cas où l'état du programme ne permet pas de réduire $\mathcal{R}$ (lorsque les deux gardes sont fausses). Le `breakpoint` est supprimé et l'exécution des deux programmes continue.

Les `breakpoint` peuvent être placé dans le programme par le programme de mise à jour si il utilise la condition $a\,\texttt{use}\,n$. Ce fonctionnement est similaire à celui d'un débogueur qui place un point d'arrêt sur un programme.

$\mathbb{M} : (c_0 \mapsto 3) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.8\, \texttt{breakpoint}\ {c_0}^n\,,\ \texttt{upgrade}\ n \leftarrow 5\ \texttt{when}\ a\ \texttt{use}\ n\ \texttt{until}\ \mathit{true}$

$\longrightarrow \mathbb{M} : (c_0 \mapsto 3) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.8\, {c_0}^n\,,\ [\![n \leftarrow 5]\!]_\perp$ [RBREAK2ONESHOT]

Remarque : La sémantique complète présentée en annexe B.2 ne présente pas comment le programme de mise à jour peut placer les `breakpoint`.

$$\frac{(c \mapsto \{t\}) \in \mathbb{M} \quad a' \textit{ is fresh}}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[c^n]] \longrightarrow_p (c \mapsto [\cdot]_{a'}) + \mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[c^n]] \parallel a' \underset{t,\varnothing}{\triangleright} t} \textsc{Plazy}$$

$$\frac{(c \mapsto [\cdot]_a) \in \mathbb{M}}{\mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel a \underset{t_0,\mathbb{S}}{\triangleright} v \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k \longrightarrow_p (c \mapsto v) + \mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k} \textsc{Plazyterm}$$

FIGURE 7.3 – Évaluation paresseuse des valeurs en mémoire

7.2.2 Transformation des données

Transformation instantanée

Comme pour l'accès immédiat, la transformation instantanée est le comportement par défaut de la redéfinition. Lorsque l'instruction $n \leftarrow v$ est résolue, la valeur associée à n est instantanément changée.

Transformation retardée

La transformation retardée consiste à transformer une donnée à un moment ultérieur, après que la donnée ait été accédée. La sémantique simplifiée ne permet pas une telle chose car toute modification d'une valeur nommée est instantanée.

La sémantique complète ajoute une possibilité d'évaluer paresseusement les valeurs associées à une cellule en mémoire. Une cellule peut être associée à un terme en utilisant la notation suivante : $(c_0 \mapsto \{t\})$. Les règles de la figure 7.3 montrent comment réduire le terme ${c_0}^n$ dans un fil d'exécution. Un nouveau fil est créé pour réduire le terme t (règle PLAZY). Lorsque ce nouveau fil a terminé la réduction et obtient une valeur v (un entier ou une fonction), une nouvelle association $(c_0 \mapsto v)$ est ajoutée en mémoire (règle PLAZYTERM). Tant que le nouveau fil n'a pas terminé la réduction de t, tous les fils qui doivent résoudre ${c_0}^n$ sont bloqués.

$\mathbb{M} : ({c_0}^n \mapsto \{t\}) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.5\ {c_0}^n$

$\longrightarrow \mathbb{M} : ({c_0}^n \mapsto [\cdot]_{a_1}) + ({c_0}^n \mapsto \{t\}) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.5\ {c_0}^n \parallel a_1 \underset{\varnothing}{\triangleright} t$ [PLAZY]

$\longrightarrow ... \longrightarrow \mathbb{M} : ({c_0}^n \mapsto [\cdot]_{a_1}) + ({c_0}^n \mapsto \{t\}) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.5\ {c_0}^n \parallel a_1 \underset{\varnothing}{\triangleright} v$

$\longrightarrow \mathbb{M} : ({c_0}^n \mapsto v) + ({c_0}^n \mapsto [\cdot]_{a_1}) + ({c_0}^n \mapsto \{t\}) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.5\ {c_0}^n$ [PLAZYTERM]

$\longrightarrow \mathbb{M} : ({c_0}^n \mapsto v) + ({c_0}^n \mapsto [\cdot]_{a_1}) + ({c_0}^n \mapsto \{t\}) + \varnothing\,,\ a \underset{\varnothing}{\triangleright} \lambda x.5\ v$ [TNINT/NFUN]

La mémoire paresseuse permet de transformer les données de manière retardée en redéfinissant une valeur avec un terme dont l'évaluation donne la nouvelle valeur. Le moment de la transformation est : la prochaine fois que le nom est utilisé par n'importe quel fil d'exécution.

Permettre de choisir le moment où la transformation est effectuée est plus difficile et demande davantage d'extensions. Ce cas particulier (et peu répandu, comme l'a montré la première partie de ce manuscrit) n'est pas traité ici et pourra faire l'objet de travaux futurs.

7.3 Modification du flot de contrôle

Les principales façons de modifier le flot de contrôle d'un programme sont la redéfinition des fonctions et le redémarrage de fils d'exécution. Ces deux mécanismes sont écrits en utilisant la sémantique présentée dans le chapitre précédent.

7.3.1 Redéfinition de fonction

Les fonctions nommées sont redéfinies de la même façon que les entiers car elles sont considérées comme des valeurs standards. La seule instruction $f \leftarrow \lambda x.t$ est suffisante pour redéfinir une fonction.

Usuellement, les fonctions sont redéfinies lorsqu'elles sont quiescentes, il suffit alors de choisir la condition `when` qui correspond et de suspendre l'exécution des fils concernés (qui sont susceptibles d'appeler f) :

`upgrade suspend` $a_0 ... a_k\ f \leftarrow \lambda x.t$ `resume` $a_0 ... a_k$ `when` $\neg f$ `in stack until`...

Dans le cas de la mise à jour d'une seule fonction, les instructions `suspend` et `resume` ne sont pas nécessaires car la première (et seule) instruction de la mise à jour est exécutée immédiatement dès que

$$\frac{s = a_1 \,..\, a_l \in \mathcal{C} \quad \textbf{map\_reboot}\ a_1 \,..\, a_l\ \mathcal{P} \mapsto \mathcal{P}'}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \texttt{reboot}\ s\ i_2..i_k]\!]_u \longrightarrow_r \mathbb{M}, \mathcal{P}', [\![\mathcal{C} \vdash i_2..i_k]\!]_u}\ \textsc{Rreboot}$$

$$\frac{s = a_1 \,..\, a_l \in \mathcal{C} \quad \textbf{map\_mainredef}\ a_1 \,..\, a_l\ t\ \mathcal{P} \mapsto \mathcal{P}'}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \texttt{mainredef}\ s\ t\ i_2..i_k]\!]_u \longrightarrow_r \mathbb{M}, \mathcal{P}', [\![\mathcal{C} \vdash i_2..i_k]\!]_u}\ \textsc{Rmainredef}$$

FIGURE 7.4 – Redémarrage d'un fil d'exécution et redéfinition de son terme initial

le programme de mise à jour devient actif. Dans le cas de la mise à jour de plusieurs fonctions, elles deviennent nécessaires.

7.3.2 Redémarrage des fils d'exécution

Redémarrer un fil d'exécution consiste à interrompre son exécution et à relancer ce fil avec un nouveau code à exécuter. C'est le mécanisme au cœur du fonctionnement de Kitsune [44]. Une telle chose n'est pas possible avec la sémantique simplifiée : le programme de mise a jour n'a pas d'instruction exprimant le redémarrage d'un fil ou le changement de son terme. De plus λ_{DSU} n'a pas de moyen de redémarrer des fils d'exécution.

La sémantique complète ajoute deux instructions $\texttt{reboot}\, a_0 ... a_k$ et $\texttt{mainredef}\, a_0 ... a_k\ t$ qui, respectivement, permettent de redémarrer un ou plusieurs fils d'exécution et de changer leurs termes initiaux. Dans la version complète de λ_{DSU} chaque fil d'exécution retient son terme initial t_{init} : $a \underset{\varnothing, t_{init}}{\triangleright} t$. Lorsque le fil est démarré (ou redémarré), son terme courant devient t_{init}.

$a \underset{\varnothing, t_1}{\triangleright} \texttt{spawn}\, t_{init} \longrightarrow_t a \underset{\varnothing, t_1}{\triangleright} a_1 \parallel a_1 \underset{\varnothing, t_{init}}{\triangleright} t_{init}$

Redémarrer un fil d'exécution après en avoir changé le terme initial s'exprime alors de la façon suivante :

$\texttt{upgrade mainredef}\, a\ t\ \texttt{reboot}\ a\ \texttt{when} \ldots \texttt{until} \ldots$

La figure 7.4 présente les règles pour le redémarrage de fils d'exécution (règle RREBOOT) et le changement du terme des fils (règle RMAINREDEF). **map_reboot** et **map_mainredef** permettent de redémarrer et redéfinir le terme de plusieurs fils d'exécution à la fois. Leurs règles de réduction sont détaillées dans l'annexe B.2. L'exemple suivant montre comment ces règles s'appliquent dans le cas d'un seul fil d'exécution.

$\mathbb{M} : \ldots, a \underset{\varnothing, t_{init}}{\triangleright} \lambda x.2\,3, \texttt{upgrade mainredef}\, a\, \texttt{def}\ n = 3\ \texttt{in}\ \lambda x.5\ n\ \texttt{reboot}\ a\ \texttt{when}\ \mathit{true}\ \texttt{until}\ \mathit{true}$

$\longrightarrow \mathbb{M} : \ldots, a \underset{\varnothing, t_{init}}{\triangleright} \lambda x.2\,3, [\![\texttt{mainredef}\, a\, \texttt{def}\ n = 3\ \texttt{in}\ \lambda x.5\ n\ \texttt{reboot}\ a]\!]_\perp$ [RONESHOT]

$\longrightarrow \mathbb{M} : \ldots, a \underset{\varnothing, \texttt{def}\ n=3\ \texttt{in}\ \lambda x.5\ n}{\triangleright} \lambda x.2\,3, [\![\ \texttt{reboot}\ a]\!]_\perp$ [RMAINREDEF]

$\longrightarrow \mathbb{M} : \ldots, a \underset{\varnothing, \texttt{def}\ n=3\ \texttt{in}\ \lambda x.5\ n}{\triangleright} \texttt{def}\ n = 3\ \texttt{in}\ \lambda x.5\ n\,, \perp$ [RREBOOT]

$\longrightarrow \ldots \longrightarrow \mathbb{M} : \ldots, a \underset{\varnothing, \texttt{def}\ n=3\ \texttt{in}\ \lambda x.5\ n}{\triangleright} 5\,, \perp$ [TAPP]

7.4 Bilan

7.4.1 Expression des mises à jour

Les mises à jour présentées dans cette partie sont exprimées de façon directe, ce qui implique que le développeur du programme de mise à jour a accès à toutes les informations du programme (identifiant des fils d'exécution, noms de valeurs, ...). Dans une situation réelle, cela est très difficile ou impossible. Il est rare qu'un développeur de mise à jour connaisse tous les noms des valeurs enregistrées en mémoire ou qu'il connaisse les identifiants des fils a suspendre. La sémantique complète propose un système de définitions et de capture de l'état du programme par introspection qui permet d'exprimer les mises à jour de façon plus réaliste. Les programmes de mise à jour ont une clause supplémentaire $\texttt{where}\ s_1 = ..., ..., s_k = ...$ qui permet de définir des ensembles de noms ou d'identifiants de fils qui peuvent être utilisés pour décrire des mises à jour. La définition de ces ensembles peut être directe (une liste de noms ou d'identifiants) ou utiliser un langage de requêtes qui permet de capturer des noms ou des identifiants du programme. Les ensembles définis dans la clause `where` d'une mise à jour sont évalués en même temps que les gardes `when` et `until`. Par conséquent, ces ensembles sont réévalués régulièrement, lors de la réduction du programme

Mécanisme	Exigences		
	Système d'exécution	Langage de MàJ	Informations
Quiescence fonction	suspension de fils, pile	condition f `in stack`, inspection de pile	nom des fonctions, pile, id fils
Points de MàJ	points de MàJ	condition @ L	-
Accès immédiat	-	redéfinition	nom
Accès progressif	`breakpoint` placé à chaque utilisation	condition `use` n	nom valeur
Transformation instantanée	redéfinition nom	redéfinition	nom valeur
Transformation retardée	mémoire paresseuse, redéfinition nom	redéfinition	nom valeur
Redéfinition fonction	redéfinition nom	redéfinition	nom des fonctions
Redémarrage fil	-	redémarrage fil, changement terme initial	id fils, terme initial

FIGURE 7.5 – Exigences des mécanismes classiques

de mise à jour lorsqu'il est *passif*. Et ils ne sont plus évalués lorsque le programme de mise à jour est *actif*. Ils gardent alors la valeur capturée au moment où le programme est devenu *actif*.

Il est rare que les valeurs à modifier soient directement adressées. Les valeurs sont généralement adressées par leur type, d'une part parce que les modifications affectent généralement toutes les valeurs d'un même type dans le cadre de la mise à jour de ce type, d'autre part parce que le type est plus facilement connu par le développeur de la mise à jour que les noms de toutes les valeurs de ce type. La sémantique complète reproduit ce comportement en permettant d'apposer une étiquette τ aux noms lors de leur définition (`def` $\tau : n = v$ `in` t). Il est possible alors d'utiliser la clause `where` pour capturer les noms de toutes les valeurs portant une même étiquette.

L'exemple suivant montre un programme de mise à jour qui suspend tous les fils d'exécution avant de redéfinir tous les noms portant l'étiquette τ, puis reprend l'exécution des fils.

`upgrade suspend` s_2 $s_1 \leftarrow 5$ `resume` s_2 `when` *true* `until` *true* `where` $s_1 = \{\tau\}, s_2 = \{$`all threads`$\}$

Dans cette partie, le programme de mise à jour emploie une sémantique simple qui permet de décrire des mécanismes à partir de briques de base. Cette approche est limitante quant aux programmes de mise à jour possible. Par exemple, il n'est pas possible de définir un *transformer* dans le programme de mise à jour. Il serait pourtant utile de pouvoir définir, dans le programme de mise à jour, des *transformer* calculant la nouvelle version d'une valeur à partir d'une valeur capturée dans le programme. La mise à jour des valeurs grâce à un *transformer* suivrait alors ce schéma : ① la valeur v définie sous le nom n est capturée ; ② un *transformer* t_r est appliqué à v et donne une nouvelle valeur $t_r(v)$; ③ la valeur de n est redéfinie avec la nouvelle valeur $t_r(v)$. Ce fonctionnement est le procédé standard d'application de *transformers* dans les plates-formes de la littérature.

Afin de permettre l'utilisation de *transformers*, et de permettre l'écriture de programme de mise à jour plus complexes, il serait intéressant de concevoir un langage dont la sémantique serait plus complète, comme par exemple un langage λ_{Upd} basé sur le lambda calcul.

7.4.2 Mécanismes et exigences

Plusieurs mécanismes classiques ont été présentés et leurs besoins en terme de fonctionnalités du système d'exécution, du programme de mise à jour et en terme d'informations accessibles ont été détaillés. Ils sont résumés dans la table de la figure 7.5. Les exigences de certains mécanismes étaient satisfaites d'emblée dans la version simplifiée de λ_{DSU} et du langage de mise à jour. C'est le cas de la quiescence, de l'accès immédiat et de la transformation instantanée. Les exigences de ces mécanismes sont satisfaites dans beaucoup de langages de programmation, au moins en partie. Par exemple, un programme C instrumenté permet d'accéder aux valeurs enregistrées en mémoire à partir de leur nom. Il peut donc être modifié en écrivant les nouvelles valeurs dans la mémoire, ce qui correspond à un accès immédiat et une transformation instantanée. La quiescence d'une fonction peut être détectée en inspectant la pile des fils d'exécution et le programme entier peut être suspendu (la suspension des fils n'étant pas possible naturellement).

Les autres mécanismes étudiés ont demandé des modifications soit de λ_{DSU} soit du langage de mise à jour. Pour beaucoup de langages de programmation, les exigences de ces mécanismes ne sont

pas satisfaites par défaut et la plate-forme de mise à jour dynamique doit apporter les fonctionnalités manquantes. Par exemple, pour permettre le redémarrage des fils d'exécution, Kitsune définit un nouveau type de fils d'exécution qui peuvent être redémarrés. Le programme doit utiliser ce type de fil d'exécution plutôt que celui de la bibliothèque `pthread`.

D'autres mécanismes de la littérature nécessitent d'autres augmentations, parfois très complexes, de la sémantique présentée dans cette partie. Par exemple, la possibilité de reconstruire la pile comme dans UpStare [50] ou ReCaml [20] nécessite un moyen d'accéder à chaque élément de la pile et de les manipuler. Dans ReCaml, la pile d'exécution est incluse dans le terme et les mises à jour sont exprimées dans le programme par son développeur. Les sémantiques de ReCaml et de Upstare sont très différentes de la sémantique de λ_{DSU} et montrent qu'il peut être complexe voir impossible de satisfaire les exigences de certains mécanismes pour un langage donné.

La complexité de certaines exigences peut les rendre incompatibles avec un langage de programmation ou une catégorie de programmes. Par exemple, la reconstruction de pile n'est pas possible en Python sans modifier l'interpréteur car elle n'est accessible qu'en lecture seule. La plate-forme de mise à jour peut pallier à ce problème en ajoutant une couche logicielle entre le système d'exécution du langage et le programme comme Kitsune qui définit un nouveau type de fil d'exécution.

Troisième partie

Implémentation de plates-formes

Implémentation de plates-formes

Les parties précédentes ont permis d'étudier l'architecture des plates-formes de mise à jour dynamique et leur mécanismes. La première partie propose un modèle générique de plate-forme et établit une synthèse des combinaisons fréquentes de propriétés des plates-formes, identifiant des synergies entre mécanismes et choix architecturaux. La deuxième partie détaille les exigences des différents mécanismes et propose un langage pour exprimer des mises à jour dynamiques.
Les résultats des deux précédentes parties sont utilisés pour implémenter deux plates-formes configurables décrites dans cette partie : Pymoult et Cmoult. Elles ont toutes les deux une architecture similaire basée sur le modèle présenté dans le chapitre 3 et fournissent des mécanismes de mise à jour configurables et combinables, permettant ainsi aux développeurs de mise à jour de définir leurs propres scripts de mise à jour.
Le chapitre 8 page 79 présente l'architecture Starmoult, partagée par Pymoult et Cmoult. Le chapitre 9 page 85 présente la plate-forme Pymoult qui permet la modification dynamique de programme Python. Le chapitre 10 page 93 présente le développement de la plate-forme Cmoult pour programmes C. Dans le chapitre 11, Pymoult est utilisée pour mettre à jour Pyftpdlib (serveur ftp) et Django (*framework* web). Les choix de mécanismes employés sont également discutés dans ce chapitre.

Chapitre 8

Architecture Starmoult

Starmoult est une architecture pour plates-formes configurables permettant aux développeurs de mises à jour de sélectionner et combiner les mécanismes qu'elle fournit. Une telle chose n'est actuellement pas possible avec les plates-formes de l'état de l'art car d'une part, elles ne fournissent qu'un ensemble limité de mécanismes, et d'autre part, elles laissent peu de contrôle aux développeurs de mise à jour.

Starmoult est une architecture universelle ayant comme objectif d'être adaptable au plus grand nombre de langages de programmation possible. Comme le montrent les chapitres 9 et 10, Starmoult est compatible, au moins, avec les langages Python et C. L'universalité de Starmoult permet d'exprimer de manière similaire les mises à jour de programmes très différents, comme le montre ce chapitre.

Starmoult est basée sur le modèle générique détaillé dans le chapitre 3. Elle adopte le cycle de vie générique qui y est présenté et se centre autour de la notion de gestionnaire.

Dans un premier temps, ce chapitre présente comment les utilisateurs (développeurs de programmes ou de mises à jour) interagissent avec les plates-formes conformes à l'architecture Starmoult (désignées par *plates-formes Starmoult* dans la suite de cette partie). Dans un second temps, l'architecture Starmoult est présentée : les éléments qui la composent ainsi que son cycle de vie sont détaillés.

8.1 Utilisation d'une plate-forme Starmoult

Deux types d'utilisateurs d'une plate-forme Starmoult se distinguent : les développeurs du programme et les développeurs des mises à jour. Chacun de ces rôles interagit de manière différente et à un stade différent de l'exécution du programme.

La figure 8.1 présente les différentes étapes de la vie du programme où intervient la plate-forme Starmoult. D'abord, le développeur prépare le programme avant qu'il soit lancé, soit immédiatement avant en modifiant son code source, soit lors de son développement. Il doit placer des points de mise à jour (si besoin), installer éventuellement un ou plusieurs gestionnaires [1] ou installer des éléments spécifiques à la plate-forme (*code loader* dans le cas de Cmoult, *listener* dans le cas de Pymoult).

Une fois le programme préparé, ce dernier est lancé et charge une partie des fonctionnalités de la plate-forme (une bibliothèque dans le cas de Cmoult, des modules Python dans Pymoult). Les éléments

1. cela n'est pas toujours nécessaire, comme indiqué plus loin dans ce chapitre

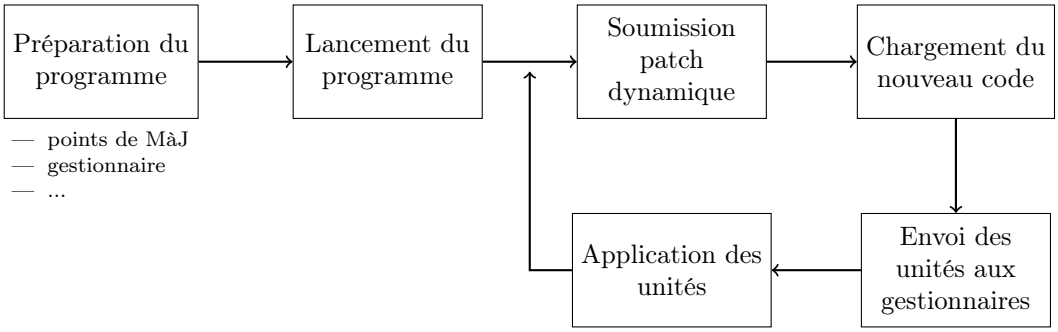


FIGURE 8.1 – Cycle d'utilisation d'une plate-forme Starmoult

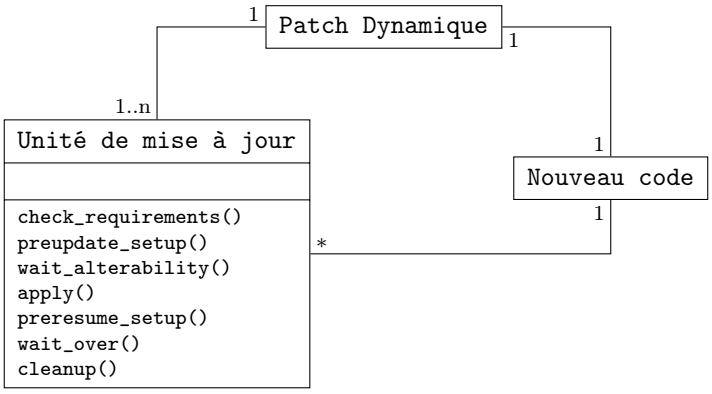


FIGURE 8.2 – Carte d'un patch dynamique

de la plate-forme installés par le développeur du programme restent inactifs jusqu'à ce qu'une mise à jour soit soumise à la plate-forme.

Lorsqu'une mise à jour doit être appliquée, son patch dynamique est envoyé à la plate-forme. Le nouveau code est chargé et le script de mise à jour est exécuté. Dans Starmoult, le script de mise à jour est constitué de une ou plusieurs *unités de mise à jour* qui indiquent les opérations à effectuer pour modifier le programme. Ces unités sont envoyées aux gestionnaires qui les appliquent (ils exécutent les opérations que les unités indiquent). Quand toutes les unités ont été appliquées, la mise à jour est terminée et la plate-forme attend la prochaine mise à jour.

La figure 8.2 présente la carte d'un patch dynamique Starmoult. Il contient le nouveau code (nouvelles fonctions, nouveaux types, ...) ainsi que plusieurs unités de mise à jour qui constituent le script de mise à jour. Chaque unité de mise à jour définit une fonction différente pour chaque étape du cycle de vie de Starmoult détaillé dans la sous-section 8.2.2. Chacune de ces fonctions est appelée par un gestionnaire lors de l'étape qui lui correspond. Ce sont ces fonctions qui utilisent les mécanismes de mise à jour fournis par la plate-forme. Toutes les plates-formes Starmoult fournissent des modèles d'unité de mise à jour prêtes à être configurées et utilisées dans une mise à jour, ainsi que la possibilité pour le développeur de définir ses propres unités de mise à jour.

A chaque mise à jour, le rôle du développeur est de définir le patch dynamique en fournissant le nouveau code ainsi que les différentes unités de mise à jour.

La forme du patch dynamique varie suivant les plates-formes. Dans Pymoult, il s'agit d'un module Python tandis que dans Cmoult, il s'agit d'un ensemble de bibliothèques dynamiques et de fichiers texte.

8.2 L' Architecture Starmoult

Starmoult se base sur le modèle générique détaillé dans le chapitre 3. Un ou plusieurs gestionnaires sont chargés d'appliquer les mises à jour en suivant le cycle de vie générique.

Le but de l'architecture Starmoult est de permettre aux développeurs du programme et des mises à jour d'adapter la plate-forme aux besoins du programme, par exemple, en choisissant les types de gestionnaires les plus adaptés. Starmoult permet également aux développeurs des mises à jour de choisir quels mécanismes utiliser et de les configurer selon leurs besoins. Starmoult est donc conçue pour être la plus souple possible.

8.2.1 Gestionnaires et unités de mise à jour

L'architecture Starmoult possède deux éléments centraux : les gestionnaires et les unités de mise à jour. Les gestionnaires peuvent se trouver dans le programme ou à l'extérieur, les gestionnaires internes pouvant être ajoutés au programme avant ou après son lancement. Chaque gestionnaire peut être associé à un ou plusieurs éléments (ou à un ou plusieurs types d'éléments) du programme. Dans ce cas, il ne peut modifier que ces éléments. Sinon, les gestionnaires peuvent modifier n'importe quelle partie du programme. La figure 8.3 montre un exemple de programme contenant un gestionnaire interne associé à ses fonctions et un gestionnaire externe qui n'est associé à aucun élément. Trois unités de mise à jour modifient chacune un élément différent du programme et sont réparties entre les deux gestionnaires.

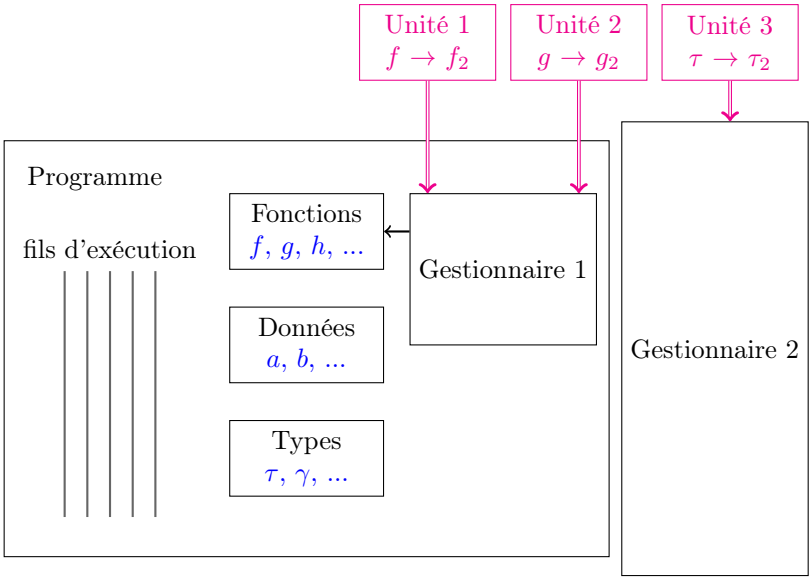


FIGURE 8.3 – Architecture Starmoult

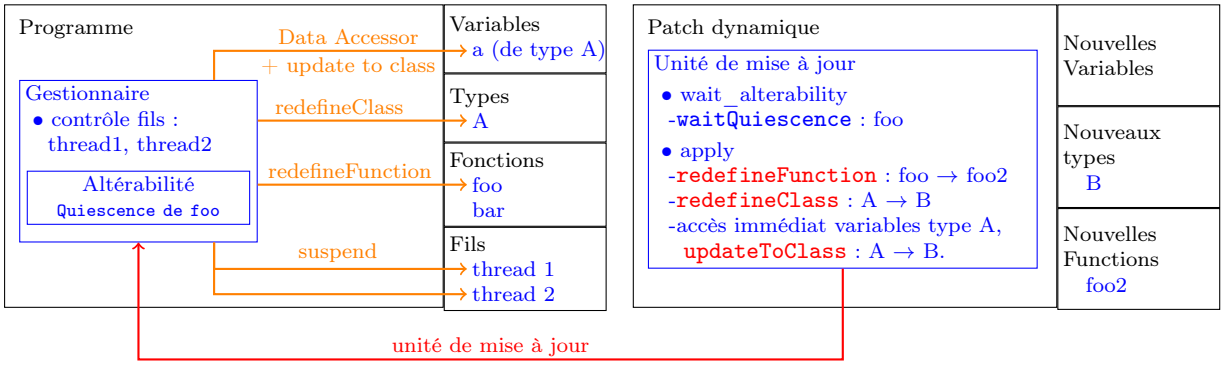


FIGURE 8.4 – Carte d'une mise à jour dans les plates-formes Starmoult

Les gestionnaires appliquent les modifications décrites par les unités de mise à jour. Elles indiquent quels mécanismes doivent être utilisés et sur quelles cibles. Par exemple, dans la figure 8.4, l'unité de mise à jour indique au gestionnaire que la fonction `foo` doit être redéfinie en `foo2`. Pour chaque unité de mise à jour définie, le patch dynamique indique quel gestionnaire doit appliquer cette dernière. Chaque gestionnaire ne peut appliquer qu'une unité de mise à jour à la fois mais plusieurs gestionnaires peuvent appliquer une unité de mise à jour différente en même temps.

Les unités de mise à jour définissent une fonction par étape du cycle de vie de Starmoult. Le gestionnaire qui applique une unité de mise à jour appelle ces fonctions dans l'ordre jusqu'à ce que l'unité de mise à jour soit complètement appliquée (c'est-à-dire lorsque la fonction correspondant à l'étape de nettoyage retourne). Le gestionnaire peut ensuite appliquer l'unité de mise à jour suivante, dans l'ordre indiqué dans le patch dynamique par le développeur de la mise à jour.

Lorsque tous les gestionnaires ont complètement appliqué toutes les unités de mise à jour qui leur ont été attribuées, la mise à jour est terminée et les gestionnaires redeviennent passifs jusqu'à la prochaine mise à jour. Il est également possible de désinstaller les gestionnaires à la fin d'une mise à jour et de les réinstaller au moment d'appliquer la prochaine mise à jour.

Les plates-formes Starmoult fournissent différents types de gestionnaires et différents types d'unité de mise à jour. Par exemple, Pymoult propose un type de gestionnaire s'exécutant dans son propre fil et un type de gestionnaire s'exécutant chaque fois que le programme lui donne la main. Il est possible également pour les développeurs (du programme ou des mises à jour) de définir leurs propres types de gestionnaire ou d'unité de mise à jour.

D'une manière générale, les types d'unité de mise à jour fournies par les plates-formes Starmoult implémentent des combinaisons de mécanismes courants comme par exemple la *redéfinition sûre* qui redéfinit une fonction quand elle est quiescente.

Étape	Fonction	Valeur retour	Description
(1)	`check_requirements`	`"yes"` satisfaites `"no"` insatisfaites `"never"` insatisfaisables	Vérifie que les exigences de l'unité sont satisfaites
(2)	`preupdate_setup`	aucune	Prépare la mise à jour
(3)	`wait_alterability`	`True` détectée `False` non détectée dans les délais	Scrute l'altérabilité
-	`clean_failed_alterability`	aucune	Nettoie avant report de l'unité
(5)	`apply`	aucune	Applique les modifications
(6)	`preresume_setup`	aucune	Prépare à la reprise
(8)	`wait_over`	aucune	Retourne quand l'unité est entièrement appliquée
(9)	`cleanup`	aucune	Nettoie avant fin de l'unité

FIGURE 8.5 – Fonctions des unités de mise à jour

8.2.2 Cycle de vie

Le cycle de vie de Starmoult est basé sur le cycle de vie générique détaillé dans le chapitre 3. Il est suivi par chaque gestionnaire lorsqu'il applique une unité de mise à jour. Chaque étape de ce cycle correspond à l'appel d'une fonction définie par l'unité de mise à jour en cours d'application. Les valeurs retournées par chacune de ces fonctions sont définies par Starmoult tandis que les paramètres en entrée peuvent varier suivant les plates-formes. La figure 8.5 récapitule, pour chaque étape, la fonction appelée, ses valeurs de retour possible et sa sémantique.

La figure 8.6 montre le processus selon lequel les différentes fonctions des unités de mise à jour sont appelées. Lorsqu'un gestionnaire commence à appliquer une unité de mise à jour, il appelle sa fonction `check_requirements` pour vérifier les exigences de l'unité. Si la fonction retourne `"yes"`, les exigences sont satisfaites et le gestionnaire passe a l'étape suivante. Si `"no"` est retourné, les exigences ne sont pas satisfaites et le gestionnaire reporte l'unité (il applique d'éventuelles autres unités de mise à jour avant de retenter d'appliquer l'unité reportée). Si `"never"` est retourné, le gestionnaire abandonne l'unité et passe à la suivante.

Une fois les exigences satisfaites, la fonction `preupdate_setup` est appelée pour installer les éléments nécessaires à la scrutation de l'altérabilité (si besoin). Lorsque cette fonction retourne, le gestionnaire appelle `wait_alterability` qui scrute l'altérabilité. Lorsque l'altérabilité est détectée, cette fonction retourne `True`. Si l'altérabilité n'est pas détectée dans un laps de temps défini par l'unité de mise à jour, `False` est retournée et le gestionnaire appelle `clean_failed_alterability` pour nettoyer d'éventuelles installations entreprises lors de l'étape (1) et l'unité est reportée.

Quand `wait_alterability` retourne `True`, le gestionnaire suspend les fils d'exécution du programme et appelle la fonction `apply` qui applique les modifications demandées par l'unité. Puis, la fonction `preresume_setup` est appelée pour installer, si besoin, des éléments avant de reprendre l'exécution des fils.

Lorsque tous les fils sont repris, la fonction `wait_over` est appelée. Cette fonction retourne lorsque toutes les modifications demandées par l'unité ont été complètement terminées (par exemple, lorsque toutes les données ont été mises à jour dans le cas d'un accès progressif). Le gestionnaire peut ensuite appeler la fonction `cleanup` et enregistrer l'unité comme terminée.

8.3 Une architecture universelle

Utiliser plusieurs plates-formes se conformant à une même architecture permet de les combiner, par exemple en utilisant une plate-forme par programme d'une même application. Leur principe de fonctionnement étant similaire et les mises à jour étant exprimées de la même façon, il devient possible pour ces plates-formes de partager des informations communes. Il est envisageable de soumettre un patch dynamique global s'adressant à tous les programmes dont les unités de mise à jour seraient attribuées aux gestionnaires des bons programmes. Il pourrait même être possible pour ces plates-formes de communi-

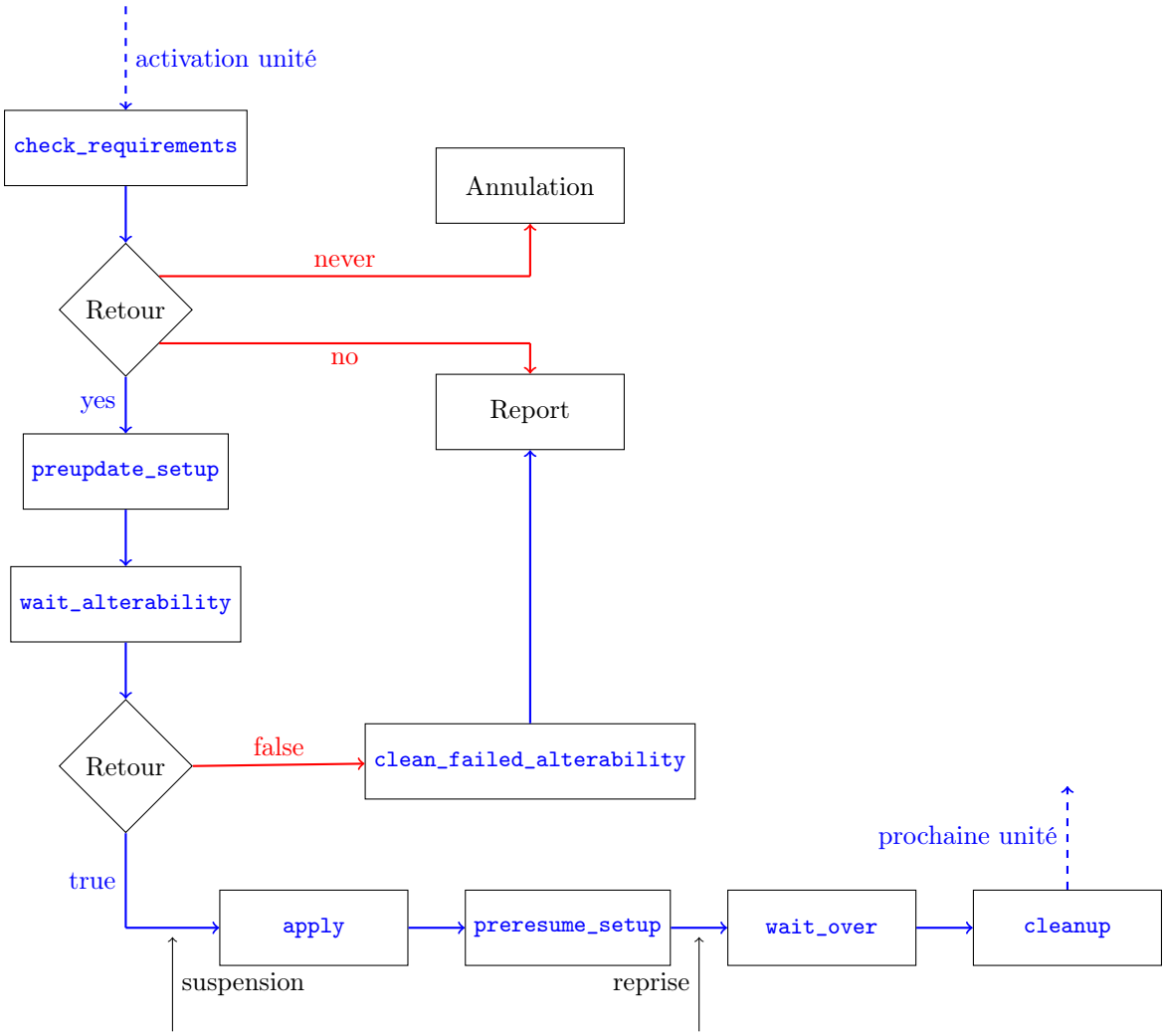


FIGURE 8.6 – Cycle de vie des mise à jour des plates-formes Starmoult

quer lors de l'application des mises à jour. Par exemple, une unité de mise à jour dans un programme P pourrait avoir comme critères d'altérabilité la quiescence d'une fonction dans un autre programme P'. Utilisant un protocole dédié, un gestionnaire du programme P' indiquerait au gestionnaire appliquant l'unité de mise à jour quand la fonction est quiescente.

Chapitre 9

Pymoult

Pymoult est une plate-forme configurable de mise à jour dynamique pour programmes Python adoptant l'architecture Starmoult. Elle adopte donc l'approche proposée dans ce manuscrit : à chaque mise à jour, le développeur choisit et configure les mécanismes qui doivent être employés pour modifier le programme.

Le développement de Pymoult est motivé par deux objectifs. Premièrement, il s'agit d'étudier l'implémentation des mécanismes de la littérature afin, par une approche pratique, d'identifier leurs besoins (informations accessibles, fonctionnalités, ...). Deuxièmement, il s'agit de proposer une interface de programmation générique qui pourrait être partagée par plusieurs plates-formes quelque soit le type de programme qu'elles ciblent. Pymoult fournit une interface de programmation à deux niveaux. L'interface de haut niveau permet d'ajouter des gestionnaires au programme et d'utiliser des unités de mise à jour prêtes à être configurées. L'interface de bas niveau fournit tous les mécanismes de mise à jour de la plate-forme et peut être utilisée pour définir de nouveaux types d'unité de mise à jour.

Pymoult nécessite d'utiliser une version étendue d'un des interpréteurs Python standard : Cpython (interpréteur écrit en C) ou PyPy (interpréteur écrit en Python). Les extensions apportées à ces deux interpréteurs sont détaillées dans la section 9.4 de ce chapitre.

Ce chapitre décrit les particularités de l'architecture de Pymoult et détaille l'implémentation de certains mécanismes. La section 9.1 détaille les spécificités de l'architecture de Pymoult. La section 9.2 présente les fonctionnalités de Python utilisées pour implémenter les mécanismes dans la section 9.3. La section 9.4 présente les différentes modifications des interpréteurs PyPy et CPython, effectuées pour répondre aux besoins de certains mécanismes. Enfin, la section 9.5 conclut le chapitre.

9.1 Retour sur l'architecture

Pymoult étant une plate-forme Starmoult, son architecture respecte les spécifications détaillées dans le chapitre 8 : des gestionnaires appliquent des unités de mise à jour selon le cycle de vie générique détaillé dans le précédent chapitre. Dans Pymoult, chaque type de gestionnaire et d'unité de mise à jour correspond à une classe. Ces classes peuvent être étendues pour définir de nouveaux types de gestionnaires ou d'unités.

Pymoult utilise également un nouvel élément devant impérativement être installé dans le programme avant son démarrage : un `listener` dont le rôle et de permettre le chargement des patchs dynamiques.

```
manager = Manager(...)
def main():
    while True:
        manager.apply_next_update()
        command = recv_command()
        ...
        manager.apply_next_update()
```

FIGURE 9.1 – Exemple d'utilisation d'un `Manager`

9.1.1 Gestionnaires

Pymoult propose deux types de gestionnaire : les gestionnaires de la classe `ThreadedManager` qui s'exécutent dans leurs propres fils et les gestionnaires de la classe `Manager` qui s'exécutent chaque fois que le programme leur donne la main. Les gestionnaires du premier type peuvent être placés dans le programme avant son démarrage ou après, pendant une mise à jour et fonctionnent exactement tels que décrits dans le chapitre 8. Les gestionnaires du second type ne peuvent être placés facilement qu'avant le démarrage du programme. Comme le montre la figure 9.1, une fois un gestionnaire `Manager` crée, le développeur du programme indique à quels moments le gestionnaire peut appliquer des mises à jour en appelant sa méthode `apply_next_update`. Une fois le programme démarré, il est toujours possible d'utiliser des gestionnaires de type `Manager`, il faut cependant mettre à jour le programme pour ajouter les appels à `apply_next_update`.

Remarque : gestionnaire `Manager` et point de mise à jour

Le placement des appels à `apply_next_update` dans le code ressemble fortement au placement de points de mise à jour. Il s'agit de permettre l'application de mises à jour lorsque l'exécution du programme a atteint un certain point. Pourtant, ces deux choses sont sémantiquement très différentes : un gestionnaire de type `Manager` applique une mise à jour dès que n'importe quel fil d'exécution lui donne la main tandis qu'un point de mise à jour ne peut déclencher de mise à jour que lorsque **tous** les fils ont atteint un point de mise à jour (en général et dans Pymoult).

9.1.2 Unités de mise à jour

```
class EagerUpdate(Update):
    def __init__(self,type,newtype,transformer):
        self.type = type
        self.newtype = newtype
        self.transformer = transformer
    ...
    def apply(self):
        for obj in DataAccessor("immediate",self.type):
            updateToClass(obj,self.newtype)
            self.transformer(obj)
    ...
```

FIGURE 9.2 – Exemple de classes d'unité

Dans Pymoult, les unités de mises à jour sont les instances de certaines classes fournies dans l'interface de haut niveau. Toutes ces classes héritent de la classe abstraite `Update`. Définir un nouveau type d'unité de mise à jour revient à définir une nouvelle classe héritant de `Update`, et définir une unité de mise à jour revient à instancier une de ces classes (dites *classes d'unité de mise à jour* ou *classe d'unité*).

L'interface de haut niveau de Pymoult propose plusieurs classes d'unité utilisant des combinaisons courantes de mécanismes (par exemple, accès immédiat et transformation instantanée). Ces classes sont prêtes à être instanciées avec comme paramètre les configurations des mécanismes qu'elles emploient.

Les méthodes de chaque classe d'unité appellent les fonctions de l'interface de bas niveau pour employer les mécanismes fournis par Pymoult. Il est possible d'étendre la classe `Update` ou n'importe quelle classe d'unité pour définir une nouvelle unité utilisant une combinaison de mécanismes qui n'est pas proposée dans l'interface de haut niveau.

La figure 9.2 montre un exemple de classe d'unité `EagerUpdate` qui implémente la mise à jour *pressée* d'objets d'un type donné. La classe définit la méthode `apply` qui utilise deux éléments de l'interface de bas niveau de Pymoult : `DataAccessor` qui implémente l'accès aux objets et `updateToClass` qui met à jour un objet donné.

9.1.3 Patch dynamique

```
#New code
def foo_v2():...
class Spam_v2():...
...
#Update units
unit1 = SafeRedefineUpdate(foo,foo_v2)
unit2 = EagerUpdate(Spam,Spam_v2,...)

#Adding manager and distribu-
ting units
manager = ThreadedManager(...)
manager.add_update(unit1)
manager.add_update(unit2)
```

FIGURE 9.3 – Exemple (simplifié) de patch dynamique

Dans Pymoult, un patch dynamique est un module Python qui est chargé par le `listener` intégré au programme avant son démarrage. Ce module contient le nouveau code ainsi que les unités de de mise à jour. La figure 9.3 montre un exemple de patch dynamique redéfinissant une fonction et une classe.

Le patch dynamique contient du code dont l'exécution génère le nouveau code, les unités de mise à jour et les associe aux gestionnaires. En effet, lorsqu'un patch dynamique est chargé par le `listener`, il est immédiatement exécuté ce qui permet de lancer un gestionnaire dans le programme comme le montre la figure 9.3. En Python, l'exécution de `def foo(...):...` définit la fonction `foo`. L'exécution de la partie *nouveau code* du patch dynamique définit donc les nouvelles fonctions, variables et classes dans le programme.

9.1.4 Listener

Pour pouvoir fonctionner, Pymoult nécessite qu'un élément appelé `listener` soit ajouté au programme avant son démarrage. Il s'agit d'une instance de la classe `Listener` fournie dans l'interface de haut niveau. Elle permet de charger les patchs dynamiques et de les exécuter alors que le programme est en cours d'exécution. Il s'agit d'un fil d'exécution indépendant du reste du programme qui attend des commandes du développeur de mise à jour. Lorsque celui-ci indique au `listener` le chemin d'un patch dynamique, ce dernier est chargé et exécuté, ce qui a les effets listés précédemment.

L'ajout d'un `listener` est la seule modification obligatoire du programme avant son lancement. Certains mécanismes peuvent nécessiter davantage de modifications (placement de points de mise à jour, utilisation d'un type de fil d'exécution spécifique, ...). Ces besoins sont présentés dans la section 9.3 qui traite de l'implémentation des mécanismes.

9.2 Quelques fonctionnalités de Python

Python est un langage orienté objet dynamiquement typé et interprété. Les variables contiennent toutes des références vers des objets alloués dans le tas. L'assignation de variable ou le passage d'arguments de fonction est fait par copie de cette référence. Certains types de base comme les entiers ou les chaînes de caractères sont *non mutables*. C'est-à-dire que chaque modification de l'objet crée un nouvel objet qui remplace le premier, comme montré dans la figure 9.4. C'est le même fonctionnement pour la définition de classe et de fonction. Il est donc possible de redéfinir une fonction avec l'expression `foo = newFoo`. De plus, Python est unitypé : chaque variable est considérée comme d'un type unique (un objet) jusqu'à ce qu'elle soit utilisée. L'interpréteur vérifie alors que l'opération demandée soit bien compatible avec le type de l'objet manipulé. Cette méthode de typage est surnommé *duck typing* en raison de l'exemple suivant : une variable `animal` peut pointer vers un canard ou un dragon. Lorsqu'il est demandé à cet animal de voler (`animal.fly()`), l'interpréteur vérifie seulement que la classe de `animal` a bien une méthode `fly`. Tant qu'on ne demande à cet animal que de voler, impossible de savoir s'il s'agit d'un canard ou d'un dragon. Il est donc possible de changer le type des variables tant que leur nouveau type est compatible avec le reste du programme. En conséquence, il est possible de mettre à jour un

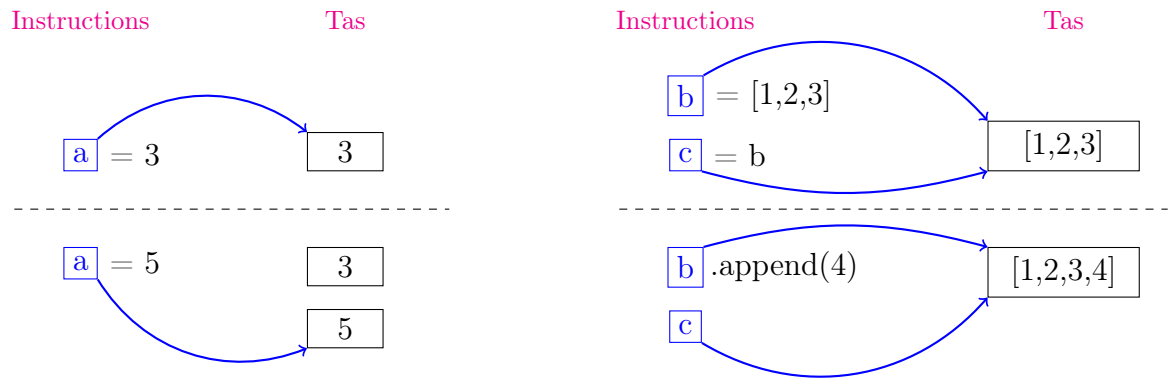


FIGURE 9.4 – Gestion des références en Python

objet dont la classe a changé (en changeant son type pour la nouvelle version de la classe) tant que la nouvelle version de la classe est compatible avec le reste du programme [1].

Comme dans beaucoup de langages objets, les fils d'exécution Python sont des instances d'une classe `Thread`. Cette classe définie une méthode `run` qui doit être redéfinie et qui est exécutée lorsque le fil est démarré. A chaque fil d'exécution peut être associée une trace. Il s'agit d'une fonction exécutée par le fil entre chaque instruction. Il est alors possible, lors d'une mise à jour, de faire exécuter du code à un fil d'exécution, pour le suspendre par exemple en attendant un verrou.

En Python, un programme est divisé en modules contenant des variables, des fonctions, des classes, etc. Il est possible de charger un module depuis un autre avec l'instruction `import`. Les définitions du module chargé deviennent alors accessibles. Lorsqu'un module est chargé, il est exécuté, ce qui a pour effet de définir les variables, fonctions et autres définitions que le module contient, comme indiqué précédemment. Dans Pymoult, un patch dynamique est un module chargé après le démarrage du programme.

Les fonctionnalités de base de Python permettent d'écrire simplement certains mécanismes de mise à jour, comme par exemple la redéfinition par indirection. Certains autres mécanismes nécessitent plus de développement, voir la modification de l'interpréteur.

9.3 Implémentation de mécanismes

Cette section détaille l'implémentation de cinq des mécanismes dont les besoins ont été identifiés dans la partie II. Il s'agit ici d'étudier le coût en développement de la satisfaction de ces besoins. L'implémentation des trois autres mécanismes décrits dans la partie II n'est pas détaillée dans ce manuscrit car elle n'apporte pas de nouveaux éléments de discours par rapport aux autres mécanismes présentés dans cette section.

9.3.1 Détection de quiescence

Pour détecter la quiescence d'une fonction, Pymoult utilise les capacités d'introspection de Python. Grâce au module `inspect`, il est possible d'accéder à la pile d'un fil depuis ce dernier. Il est alors possible de parcourir la pile à la recherche du code de la fonction. Si le code est trouvé, la fonction n'est pas quiescente.

L'implémentation de ce mécanisme n'a demandé qu'une modification légère de l'interpréteur pour permettre à un fil d'accéder à la pile d'exécution de n'importe quel autre fil. Cela permet à un gestionnaire de détecter la quiescence d'une fonction dans tous les fils du programme si besoin.

La partie II a montré que la détection de la quiescence nécessite une capacité d'introspection ainsi qu'une notion de pile. Il s'agit de deux capacités de base des interpréteurs de référence. La suspension de fils, autre besoin du mécanisme, a nécessité l'extension des interpréteurs, cela n'étant pas possible dans un interpréteur standard. Cette extension est détaillée dans la section 9.4.

1. soit parce qu'elle ne modifie pas *trop* le comportement de la classe, soit parce le programme est mis à jour pour assurer la compatibilité

9.3.2 Redéfinition de fonctions

Comme indiqué dans la section 9.2, l'indirection naturelle de Python permet aisément de redéfinir les fonctions et les classes. Il suffit de changer la valeur associée au nom de la fonction (ou de la classe), en précisant éventuellement le module où ce nom est enregistrée : `module.foo= newFoo`.

La partie II a montré une dépendance de mécanisme envers la possibilité de redéfinir un nom. Ceci étant possible de base en Python, la redéfinition de fonction a un coût de développement nul en Python.

9.3.3 Redémarrage de fil d'exécution

D'après la partie II, redémarrer les fils d'exécution nécessite une capacité du système d'exécution à effectivement redémarrer un fil et à en changer la fonction initiale (`run` dans le cas de Python). En Python, il est possible de redéfinir la méthode `run` d'un fil d'exécution mais il n'est pas possible de redémarrer un fil.

Pour palier à cela, Pymoult propose une classe `DSUThread` qui étend la classe `Thread` et permet aux fils d'exécution d'être redémarrés. La classe `DSUThread` fixe la méthode `run` et laisse le développeur définir une méthode `main` à la place. Lorsqu'une instance de `DSUThread` est lancée, la méthode `run` entre dans une boucle infinie appelant la méthode `main` dans une clause *try ... except* interceptant une exception `RebootException`. Ainsi, lorsque le fil lève cette exception, le fil appelle une nouvelle fois sa méthode `main` et redémarre. L'exception `RebootException` est levée par une trace associée au fil qui doit être redémarré. Mais le fonctionnement des traces Python a des limitations qui rendent leur utilisation impossible dans ce contexte. Pour y remédier, les interpréteurs de référence ont été étendus.

En Python, l'implémentation du redémarrage des fils d'exécution est coûteuse. Elle nécessite l'implémentation d'une nouvelle classe de fils d'exécution ainsi que l'extension des interpréteurs de référence. Son utilisation est également intrusive car le développeur du programme doit utiliser la classe `DSUThread` au lieu de `Thread`. Si le code du programme n'a pas été adapté avant son lancement, il est impossible de redémarrer les fils d'exécution lors des futures mises à jour.

9.3.4 Accès aux données

L'interface de bas niveau de Pymoult propose une classe `DataAccessor` qui permet d'accéder à tous les objets d'un type donné. Il suffit d'instancier cette classe pour obtenir un itérateur sur l'ensemble des objets du type demandé. Lors de l'instanciation, il est possible de préciser quelle stratégie d'accès doit être employée. La figure 9.2 page 86 montre un exemple d'utilisation de la classe `DataAccessor`.

Accès immédiat

La partie II n'a identifié aucun besoin particulier de l'accès immédiat. En Python, l'accès immédiat utilise le ramasse-miettes de l'interpréteur. Lorsqu'un accès est demandé, le tas est parcouru en utilisant la liste des objets recensés par le ramasse-miettes.

Le coût de développement de l'accès immédiat est donc presque nul grâce au ramasse-miettes de l'interpréteur Python.

Remarque : Accès immédiat avec Pypy

En raison d'optimisations des performances par Pypy, le ramasse-miettes n'est pas utilisable pour accéder aux données. Il est alors nécessaire d'ajouter au programme un élément qui conserve des références faibles vers tous les objets créés. Cet élément a nécessité la modification de PyPy et doit être ajouté au programme avant son démarrage. L'implémentation de l'accès immédiat est donc plus complexe dans PyPy que dans CPython.

Accès progressif

La partie II utilise une condition spécifique `use` n qui place un point d'arrêt à chaque utilisation du nom à accéder. Pymoult utilise le protocole à méta-objets de Python pour remplir le rôle du point d'arrêt.

Lorsqu'une méthode ou un attribut d'un objet est accédé (pour appeler la méthode, lire la valeur de l'attribut ou pour le modifier), l'interpréteur appelle la méthode `__getattribute__` (ou `__setattr__` pour modifier un attribut) de la classe de l'objet. Lorsqu'un accès progressif aux objets d'un type donné

est demandé, Pymoult surcharge les méthodes `__getattribute__` et `__setattr__` de la classe associée à ce type. Il est alors possible d'accéder aux données lorsqu'elles sont utilisées.

Bien qu'elle ne nécessite aucune modification de l'interpréteur, l'implémentation de l'accès progressif est plus complexe que celle d'autres mécanismes simples comme la redéfinition de fonctions. Il s'agit de surcharger les méthodes `__getattribute__` et `__setattr__` pour leur faire ajouter une référence vers l'objet utilisé dans l'itérateur créé par instanciation de `DataAccessor`. Le coût de cette implémentation est toutefois très inférieur à celui du redémarrage des fils.

9.4 Interpréteurs

Certaines fonctionnalités des interpréteurs CPython et Pypy sont limitées, ce qui rend complexe ou impossible l'implémentation de certains mécanismes. Par exemple, il n'est pas possible d'associer une trace à un fil d'exécution depuis un autre fil, ce qui empêche un gestionnaire de type `ThreadedManager` de redémarrer un fil du programme. CPython-Dsu et PyPy-Dsu, les versions étendues des deux interpréteurs sus-cités, pallient aux limitations de ces fonctionnalités et ajoutent également de nouvelles fonctionnalités. La table suivante résume quelles extensions ont été apportées aux interpréteurs pour permettre l'implémentation de mécanismes dans Pymoult.

Extension	**Mécanismes concernés**
Suspension (et reprise) des fils d'exécutions	Détection de Quiescence
Accès aux piles d'exécution depuis n'importe quel fil	
Collecte des objets à leur création	Accès immédiat (PyPy)
Ajout de traces depuis n'importe quel fil	Redémarrage de fils, Suspension (PyPy)
Déclenchement immédiat des traces	
Reconstruction de piles d'exécution	Reconstruction de pile (CPython)

En Python, une trace ne peut pas être associée à un fil d'exécution depuis un autre fil. De plus la trace ne se déclenche que la prochaine fois que le fil appelle une fonction. Cela pose un problème car un gestionnaire de type `ThreadedManager` ne peut pas ajouter de trace à un fil d'exécution et la trace peut s'activer trop tard. Par exemple, la trace déclenchant le redémarrage du fil peut s'activer une fois que le programme n'est plus altérable. Ces limitations sont corrigées dans CPython-Dsu et PyPy-Dsu.

Afin de permettre l'accès immédiat aux données, PyPy-Dsu permet d'intercepter la création de tous les objets. Pymoult utilise alors cette fonctionnalité pour créer une référence faible vers chaque objet créé et l'enregistrer dans une liste, reproduisant ainsi, en partie, le comportement d'un ramasse-miettes.

Dans CPython-Dsu, deux méthodes `suspend` et `resume` sont ajoutées à la classe `Thread` pour permettre la suspension et la reprise des fils d'exécution. Dans PyPy-Dsu, les fils sont suspendus en leur associant des traces qui attendent un verrou. Lorsque les fils doivent reprendre leur exécution, le verrou est désactivé et les traces sont supprimées.

Remarque : Suspension de fils d'exécution

Suspendre des fils d'exécution sans prendre de précautions est dangereux. Si un fil est suspendu au milieu d'une tâche sémantique (par exemple un calcul complexe non atomique), l'exécution des autres fils peut modifier l'état du fil suspendu et provoquer des erreurs. Il faut s'assurer que l'état d'un fil suspendu est correct avant qu'il ne reprenne son exécution, soit en corrigeant son état s'il contient des erreurs, soit en s'assurant que l'exécution des autres fils ne modifiera pas son état (avec des verrous par exemple).
CPython-Dsu et PyPy-Dsu permettent de suspendre et reprendre l'exécution de fils et n'apportent aucune garantie sur l'état des fils suspendus. C'est au développeur de s'assurer que la suspension et la reprise d'un fil ne provoque pas d'erreur. Pymoult (et les plates-formes de mise à jour dynamique d'une manière générale) traitent ce problème en utilisant l'altérabilité. Lorsque le programme est altérable, il peut être suspendu et modifié sans risque d'erreur.

9.5 Une plate-forme de prototypage

Pymoult a commencé comme support à l'exploration du domaine : il s'agissait d'implémenter les mécanismes de l'état de l'art afin d'en comprendre leur fonctionnement. Au fil de ces implémentations, les réflexions et les analyses empiriques présentées dans la partie I ont permis d'établir l'architecture Starmoult à laquelle Pymoult s'est ensuite conformée. De même, les premiers résultats de Pymoult ont conduit à une étude plus théorique des besoins des mécanismes comme présenté dans la partie II et les résultats de cette étude ont impacté le développement de Pymoult.

Au moment de la rédaction de ce manuscrit Pymoult est devenue une plate-forme configurable fournissant une interface de programmation qui permet tant aux développeurs de concevoir les mises à jour qu'aux chercheurs du domaine d'expérimenter de nouvelles combinaisons ou d'implémenter de nouveaux mécanismes. Ainsi, le *framework* orienté composants Pycots permet de développer des programmes reconfigurables (dont les composants peuvent être inter-changés, reconnectés, ...) et utilise les mécanismes de Pymoult. Pycots est l'implémentation du modèle Coqcots qui permet de prouver la sûreté des reconfigurations.

Chapitre 10

Cmoult

Cmoult est une plate-forme de mise à jour pour programmes C qui adopte l'architecture Starmoult. Comme Pymoult, elle permet aux développeurs de mises à jour de sélectionner les mécanismes à employer, de les combiner et de les configurer.

Au moment où ce manuscrit est rédigé, Cmoult est encore à l'état de développement. Plusieurs expériences ont été conduites dans l'objectif d'étudier l'implémentation de mécanismes en C. Ce chapitre présente les résultats de ces expériences ainsi que l'architecture de Cmoult.

10.1 Instrumentation du code

La partie II a établi l'importance des informations pour mettre à jour des éléments du programme. Le développeur de mises à jour doit avoir accès au nom des éléments à modifier (variables,fonctions,...), ainsi qu'aux informations concernant l'état du programme (fonctions actives, ...). Ces informations sont généralement extraites du code source du programme, ce qui est possible quand les informations ne sont pas perdues lors de sa compilation vers un format exécutable. C'est le cas en Python ou en Java où le code source est compilé en *bytecode* très proche du code source. Il est donc possible de retrouver une fonction par son nom dans le *bytecode* de programme.

Pour d'autres langages compilés comme le C, une partie des informations est perdue. Le binaire généré par la compilation est trop différent du code source pour que toutes les informations soient conservées facilement. Il devient plus difficile de localiser une fonction dans le binaire juste en connaissant son nom. De plus, la compilation du programme le transforme pour l'optimiser, ce qui conduit à des pertes d'informations. Par exemple, pour économiser des appels de fonctions et éviter les instructions d'appel et de retour, le code de certaines fonctions peut être internalisé comme montré dans la figure 10.1. La partie gauche de la figure représente le code source avant compilation, et la partie droite représente le code source après internalisation de la fonction `mul3`. Le code de la fonction `mul3` est extrait de la fonction et placée à la place de son appel dans la fonction `main`. Une fois `mul3` internalisée, elle n'existe plus dans le programme, ce qui rend impossible sa localisation en se basant uniquement sur son nom. C'est pour cette raison que certaines plates-formes comme OPUS [2] interdisent l'utilisation de certaines optimisations lors de la compilation du programme.

Pour conserver les informations du programme tout en permettant au programme d'être optimisé lors de sa compilation, Cmoult utilise les informations de débogage au format DWARF qui peuvent être ajoutées au binaire par le compilateur (l'option `-g` de `gcc` ajoute ces informations au binaire). Le format

```c
int mul3(int x){
  return x*3;
}
int main(){
  int i = 2;
  int k = mul3(i);
  printf("result : \%d",k);
}
```

```c
int main(){
  int i = 2;
  int k = i*3;
  printf("result : %d",k);
}
```

FIGURE 10.1 – Exemple d'internalisation d'appel

```
pid_t pidT1, pidT2;
pthread_t t1, t2;
static void * t1run(void * arg){
  pidT1= syscall(SYS_gettid);
  //thread T1 core
  ....
}

static void * t2run(void * arg){
  pidT2= syscall(SYS_gettid);
  //thread T2 core
  ....
}

void main(){
  pthread_create(&t1,NULL,&t1run,NULL);
  pthread_create(&t2,NULL,&t2run,NULL);
  printf("t1: \%d, t2: \%d\n",pidT1,pidT2);
}
```

```
void main(int argc, char** argv){
  pid_t tid = atoi(argv[1]);
  ptrace(PTRACE_ATTACH, thread, NULL, NULL);
  waitpid(thread, NULL, 0);
  //Update, do stuff
  ...
  ptrace(PTRACE_DETACH, (pid_t) thread, NULL, NULL);
}
```

FIGURE 10.2 – Suspension d'un fil d'exécution

DWARF est compatible avec n'importe quel binaire au format ELF [1] qui enregistre les informations du programme dans des registres DIE (*Debugging Information Entity*). Un registre DIE est associé à chaque entité du programme (variable, fonction, type, ...) et indique son adresse dans la mémoire ainsi que d'autres informations. Les informations DWARF permettent également d'analyser plus facilement la pile d'exécution. Le format DWARF n'est pas détaillé dans ce manuscrit. Pour plus d'informations, le lecteur est invité à lire le tutoriel du format [31].

Cmoult repose sur l'infrastructure de débogage du langage C car les débogueurs partagent des problématiques communes avec la mise à jour dynamique. La conservation des informations dans le binaire, le contrôle des fils d'exécution ou encore la modification des variables sont des tâches communes à ces deux domaines. Utiliser les informations DWARF pour conserver les informations ou encore l'outil `ptrace` pour écrire en mémoire et contrôler les fils d'exécution permet à Cmoult de reposer sur des outils complets remplissant déjà une partie des tâches que la plate-forme doit accomplir.

10.2 Implémentation de mécanismes

L'implémentation de trois mécanismes a fait l'objet d'expériences qui ont permis d'en évaluer le coût. D'autres implémentations que celles présentées dans cette section existent. Ces alternatives sont présentées et discutées.

10.2.1 Suspension de fils d'exécution

La suspension des fils d'exécution est une tâche complexe pour les programmes C. L'implémentation standard des fils d'exécution en C est celle fournie par la bibliothèque `pthread` qui implémente le standard POSIX. Hors, les fils d'exécution POSIX ne peuvent être suspendus naturellement. Heureusement, leur implémentation sous Linux utilise des processus légers, c'est-à-dire des processus qui n'ont pas leur mémoire propre (ils la partagent avec le processus parent qui les a démarré). Il est donc possible de les suspendre comme des processus *normaux* avec la bibliothèque `ptrace`. Cette bibliothèque, utilisée principalement par les débogueurs, fournit des fonctions permettant de contrôler des programmes en cours d'exécution. Ces fonctions permettent, entre autre, de suspendre et reprendre l'exécution du programme, de lire et d'écrire dans sa mémoire.

Lorsqu'un processus P_1 s'attache à un autre processus P_2 en utilisant `ptrace`, le processus P_2 est suspendu. Il est alors possible pour P_1 de lire ou écrire dans la mémoire du processus P_2. Lorsque P_1 se détache de P_2, ce dernier reprend son exécution. Cmoult utilise `ptarce` pour s'attacher aux fils d'exécution (qui sont des processus légers) et les suspendre. La figure 10.2 montre un exemple de suspension de fil

1. d'où le nom DWARF

d'exécution par un processus externe. La partie de gauche montre un programme contenant deux fils `t1` et `t2` et la partie de droite montre le code d'un programme qui suspend puis reprend un fil d'exécution dont l'identifiant est donné en paramètre.

Comme le montre l'exemple de la figure 10.2, il est nécessaire de capturer l'identifiant du fil d'exécution à son démarrage. Par conséquent, Cmoult définit un nouveau type de fil : `DSUthread` qui enrobe un fil `pthread` et capture son identifiant à son lancement.

Dans l'état de l'art, les plates-formes pour programme C ne suspendent pas les fils du programme un à un mais suspendent l'intégralité du programme. Deux raisons expliquent cela. Premièrement, il est complexe de ne suspendre que certains fils. Deuxièmement, les patchs dynamiques étant générés automatiquement, il est difficile d'identifier quels fils doivent être suspendus et quels autres peuvent s'exécuter pendant une mise à jour. Suspendre tout le programme est à la fois plus simple et plus prudent.

D'autres moyens de suspendre les fils d'exécution sont envisageables. Par exemple, utiliser `ptrace` pour insérer une instruction attendant un verrou dans le code d'un fil est une alternative. L'utilisation directe de `ptrace` étant plus simple et étant parfaitement adaptée, c'est cette solution qui est utilisée dans Cmoult.

10.2.2 Redéfinition de fonctions

0x4200	function foo :	0x4200	function foo :
0x4232	instr1	0x4232	**jump 0x5725**
0x4264	instr2	0x4264	instr2
0x4296		0x4296	
...	...	...	...
0x5724	function foo_v2 :	0x5724	function foo_v2 :
0x5756	instr1'	0x5756	instr1'
0x5788	instr2'	0x5788	instr2'

FIGURE 10.3 – Insertion d'un trampoline dans une fonction

Cmoult emploie la méthode la plus répandue pour redéfinir les fonctions : l'utilisation d'un trampoline. Une fois le code d'une fonction localisé dans le binaire, la première instruction de la fonction est remplacée par une instruction `jump` vers la première instruction de la nouvelle version. La figure 10.3 montre un exemple d'insertion de trampoline.

Les informations DWARF permettent de retrouver le code d'une fonction dans le binaire à partir de son nom. Si certains appels ont été internalisés lors de la compilation, les informations DWARF permettent de localiser le code internalisé et d'insérer un trampoline à cet endroit. Il faut alors également insérer un trampoline pour qu'à son retour le programme continue à la fin de la fonction internalisée.

D'autres méthodes pour redéfinir des fonctions existent. Par exemple, Ginseng [58] utilise des pointeurs de fonctions générés par une passe de compilation. L'insertion de trampoline a l'avantage de ne pas demander de passes supplémentaires lors de la compilation.

Aucune des méthodes de redéfinition de fonction indiquée ci-dessus ne permet de changer les paramètres d'une fonction lors de sa mise à jour. Au moment où ce manuscrit est rédigé, nous n'avons aucune piste sérieuse permettant cela.

10.2.3 Détection de la quiescence

Cmoult utilise deux méthodes pour détecter la quiescence des fonctions. La première méthode est la scrutation des piles d'exécution. Les registres FDE (*Frame Description Entry*) de DWARF permettent de parcourir la pile pour identifier les fonctions actives.

La deuxième méthode utilise également les registres FDE pour trouver l'appel à la fonction le plus ancien. Un point d'arrêt est ensuite placé au retour de la *frame* qui correspond à cet appel. Les fils sont ensuite automatiquement suspendus par les points d'arrêts placés de cette manière lorsque la fonction devient quiescente.

Ces deux techniques, implémentées dans Cmoult, sont les principales méthodes employées par les plates-formes de l'état de l'art. Une autre méthode est d'intercepter les appels et retours de fonctions

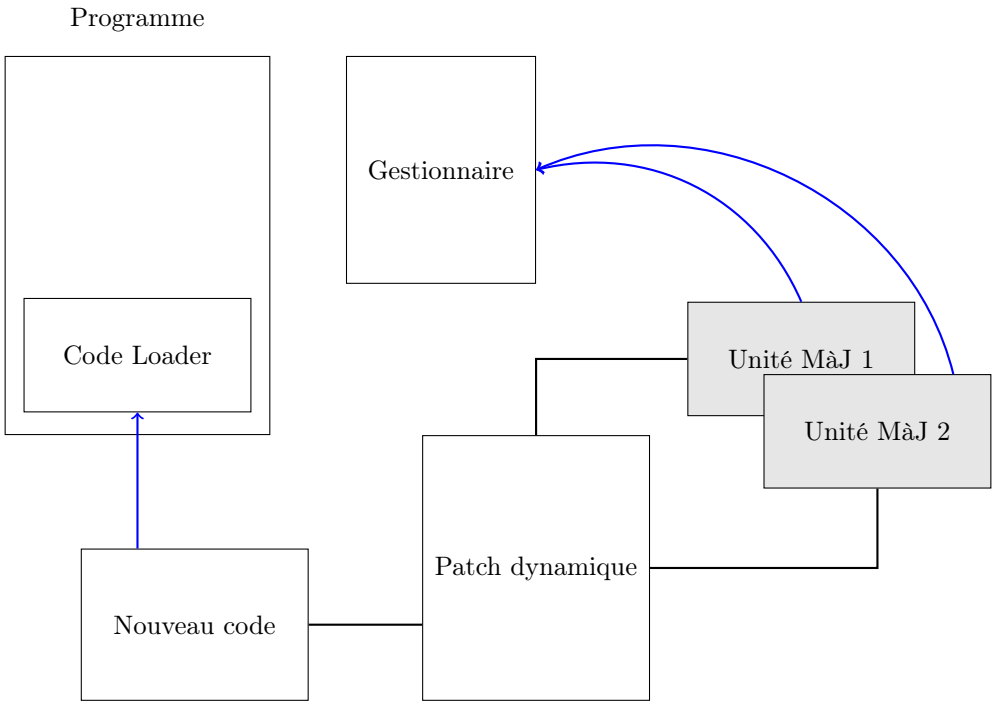


FIGURE 10.4 – Cartographie de la plate-forme Cmoult

(en insérant du code à ces endroits lors de la compilation) afin de compter le nombre d'appels en cours pour une fonction donnée. Détecter la quiescence revient alors à vérifier que ce compteur vaut zéro.

10.3 Impact sur l'architecture

Les outils employés ainsi que les choix d'implémentation des mécanismes ont leurs propres besoins qui se traduisent par des orientations de l'architecture de Starmoult. Par exemple, l'outil `ptrace` ne peut fonctionner que d'un processus P_1 vers un autre processus P_2 différent du premier. Hors, selon l'architecture Starmoult, ce sont les gestionnaires qui suspendent les fils et appliquent les modifications. Il est donc plus aisé que les gestionnaires de Cmoult soient des processus externes s'attachant au programme en cours d'exécution. Cmoult propose également un type de gestionnaire interne au programme, lié à un processus externe qui effectue toutes les opérations utilisant `ptrace` (suspension, écriture en mémoire,...).

L'externalisation par défaut des gestionnaires dissocie la problématique du chargement du nouveau code de celle du chargement du patch dynamique. En effet, le script de mise à jour s'adresse au gestionnaire tandis que le nouveau code doit être chargé dans le programme. Cmoult introduit alors un nouvel élément dans le programme avant son démarrage : le `codeloader`. Cet élément, sur instruction d'un gestionnaire, peut charger une bibliothèque dynamique avec `dlopen`. Le nouveau code est donc une bibliothèque dynamique dissociée du reste du patch dynamique. `dlopen` est une fonction permettant de charger une bibliothèque dynamique dans la mémoire d'un programme en cours d'exécution. Elle doit être appelée par le programme en question. Lors du chargement, elle édite les liens entre le binaire en cours d'exécution et la bibliothèque afin que le programme puisse utiliser les noms définis dans cette dernière. Il s'agit de la méthode utilisée par la majorité des plates-formes de l'état de l'art.

Les choix d'implémentation des mécanismes définissent des exigences sur la structure et la façon d'utiliser une plate-forme. Dans le cas de Cmoult, l'implémentation du mécanisme de suspension des fils oblige à utiliser `DSUThread` au lieu de `ptrace`, ce qui nécessite soit une passe de compilation supplémentaire, soit que le développeur du programme adapte le code de ce dernier. D'autre part l'implémentation de `ptrace` oriente l'architecture de Cmoult vers l'utilisation de gestionnaires externes. Il existe différentes façons d'implémenter un mécanisme donné, chaque fois avec des exigences différentes pouvant impacter la plate-forme qui fournit ces mécanismes ou même le programme mis à jour. Les choix d'implémentation des mécanismes de Cmoult ne l'empêche pas de se conformer à l'architecture Starmoult.

Chapitre 11

Quelques cas d'utilisation de Pymoult

Ce chapitre donne des exemples de mises à jour dynamiques employant Pymoult et décrit certaines des expériences conduites avec cette plate-forme.

Dans un premier temps, un programme de mise en cache de données, spécialement conçu pour servir de cas d'exemple, est mis à jour deux fois selon des stratégies différentes. L'objectif de cette première section est de donner des exemples d'utilisation de Pymoult et de montrer comment l'architecture Starmoult permet de choisir et concevoir une stratégie mieux adaptée à une mise à jour donnée. C'est également l'occasion d'identifier des lignes directrices pour le choix des stratégies.

Dans un second temps, trois utilisations de Pymoult sur des applications *réelles* sont présentées. Deux applications, Pyftpdlib [65] et Django [27], ont été mises à jour dynamiquement. Les sous-sections 11.2.1 et 11.2.2 décrivent comment les mises à jour ont été conçues à partir des codes sources de chacune de leurs versions. La sous-section 11.2.3 présente Coqcots [21], modèle pour programmes reconfigurables orientés composants et Pycots, implémentation Python de ce modèle qui repose sur Pymoult.

L'objectif de ce chapitre est de montrer comment une plate-forme Starmoult peut être utilisée pour concevoir des stratégies adaptées à chaque mise à jour, tout en discutant des stratégies les plus adaptées pour une situation donnée. Seulement une partie des expériences conduites avec Pymoult est présentée dans ce chapitre. Les autres peuvent êtres trouvées sur le dépôt de Pymoult [11]. À titre indicatif, parmi ces expériences figurent une démonstration interactive qui guide l'utilisateur dans le choix des mécanismes à employer pour la mise à jour d'un programme prédéfini, un serveur mis à jour selon les stratégies de K42 [8] et de Kitsune [44], ainsi que plusieurs cas d'exemples destinés à tester le fonctionnement des mécanismes de Pymoult.

11.1 Exemples de combinaisons de mécanismes

Cette section propose un programme de mise en cache de données conçu spécialement comme support d'exemple. Le programme permet de récupérer des données (sur internet par exemple) et de les enregistrer dans un cache. Les données mises en cache sont associées à la session sous laquelle la donnée a été demandée (il est considéré qu'une même requête peut obtenir une donnée différente suivant la session, ce qui est le cas de certains sites internet).

Deux mises à jour successives sont appliquées au programme. La première compresse les données mises en cache pour réduire la consommation mémoire du programme. La seconde invalide les données mises en cache au delà d'un certain temps.

Les prochaines sous-sections présentent des versions simplifiées du code du programme et des mises à jour. Les versions complètes de ces codes sont disponibles en annexe C.

11.1.1 Le programme

Le programme de mise en cache est composé principalement de deux classes `CachedData` et `Session` qui représentent respectivement une donnée mise en cache et une session à laquelle sont associées une ou plusieurs données, et de deux fonctions `cache` et `main`. La première fonction enregistre une donnée dans une session avec comme clé la requête qui a permis de l'obtenir. La seconde fonction lit les requêtes reçues, affiche les données obtenues, soit depuis le cache, soit via la fonction `get`. La figure 11.1 présente le code simplifié du programme.

```
class CachedData(object):
    def __init__(self,data):...
    def get(self):...

class Session(object):
    def __init__(self):
        self.cache = {}
        self.active = False
    def add_cache(self,key,data):...
    def has_cached(self,request):...
    def load_from_cache(self,request):...
    def enter(self):...
    def leave(self):...

requests = Queue()
def get(request):...
def display(data):...
```

```
def cache(data,session,request):
    cached = CachedData(data)
    session.add_cache(request,cached)

def main():
    current = Session()
    while True:
        (session,request) = requests.get()
        if session != current:
            current.leave()
            current = session
            current.enter()
        if current.has_cached(request):
            display(current.load_from_cache(request))
        else:
            data = get(request)
            cache(data,current,request)
            display(data)
```

FIGURE 11.1 – Code simplifié du programme de mise en cache

Les requêtes sont envoyées au programme via une file d'attente. Elles sont formées d'une chaîne de caractères qui correspond à la requête de donnée (par exemple, une adresse web) et d'une session dans laquelle la donnée obtenue sera enregistrée (ou d'où elle sera chargée). Le prochain exemple montre comment utiliser le programme pour obtenir une donnée.

```
from program import requests, Session
mysession = Session()
requests.put(mysession,"adress")
```

Pour exécuter le programme normalement, il suffit d'appeler la fonction `main`. Pour permettre l'utilisation de Pymoult, il est nécessaire d'ajouter également un `Listener` qui permet de charger des patchs dynamiques tout au long de l'exécution du programme. Afin d'être compatible avec tous les mécanismes de Pymoult, il est aussi nécessaire que tous les fils d'exécution du programme utilisent la classe `DSU_Thread` au lieu de la classe `Thread` usuelle. Appeler `main` directement revient à employer la classe `Thread`. Il faut donc créer un fil de type `DSU_Thread` en plus des autres modifications à apporter au code du programme. L'exemple suivant montre le code à ajouter au code de la figure 11.1 pour permettre l'utilisation complète de Pymoult.

```
from pymoult.highlevel.listener import Listener
from pymoult.threads import DSU_Thread
listener = Listener()
listener.start()
mainThread = DSU_Thread(target=main,name="mainThread")
mainThread.start()
```

Les fonctions `get` et `display` représentent des fonctionnalités qu'aurait un réel programme de mise en cache : récupérer les données demandées (sur internet dans le cas d'un serveur mandataire, par exemple) et les afficher ou les transmettre à l'agent qui les a demandées. Ce programme étant un exemple, ces deux fonctions sont très simplifiées car leur comportement n'a pas d'impact sur les mises à jour présentées dans cette section.

11.1.2 Compression des données en cache

Si le programme est exécuté sur une longue période de temps, la quantité de données en cache peut augmenter au point que la consommation mémoire du programme devienne problématique. Pour remédier à cela, la première mise à jour propose de compresser les données mises en cache. Comme le montre la figure 11.2, cela implique de modifier la méthode `get` de la classe `CachedData` et la fonction `cache`. Les données en cache doivent également être compressées. Pour appliquer cette mise à jour, les trois tâches suivantes doivent êtres accomplies.

1. Redéfinir la classe `CachedData` ;
2. Transformer les instances de `CachedData` ;
3. Redéfinir la fonction `cache`.

```
#New code
class CachedData_v2(object):
    def __init__(self,data):
        self.data = data
    def get(self):
        raw = zlib.decompress(self.data)
        return raw.decode()
def cache_v2(data,session,request):
    raw = data.encode()
    comp = zlib.compress(raw)
    cached = CachedData_v2(comp)
    session.add_cache(request,cached)

#Update script
#Transformer for class CachedData
def datatransf(obj):
    raw = obj.data.encode()
    comp = zlib.compress(raw)
    obj.data = comp

#Create update units
update_data =
  LazyConversionUpdate(CData,CData_v2,datatransf,...)
update_cache =
  SafeRedefineUpdate(cache,cache_v2,"cacheUpdate")

#Create a manager
manager = ThreadedManager(name="manager")
manager.start()
#Send update units
manager.add_update(update_data)
manager.add_update(update_cache)
```

FIGURE 11.2 – Code simplifié de la première mise à jour

Pour accomplir ces tâches, des types d'unité de mise à jour fournis dans l'interface de haut niveau de Pymoult sont employés. Deux de ces types permettent de redéfinir une classe et de transformer ses instances : `LazyConversionUpdate` et `EagerConversionUpdate` qui adoptent la stratégie de mise à jour paresseuse (accès progressif, transformation instantanée) ou pressée (accès immédiat, transformation instantanée). Un autre type d'unité fourni par Pymoult permet de redéfinir une fonction lorsqu'elle est quiescente : `SafeRedefineUpdate`.

Les données en cache sont potentiellement très nombreuses, c'est d'ailleurs pour cette raison que cette mise à jour est appliquée. Employer une unité de `EagerConversionUpdate` causerait la suspension du programme pendant la transformation de toutes les données en cache, ce qui rendrait le programme indisponible pendant un long moment. La mise à jour présentée en figure 11.2 emploie donc une unité de type `LazyConversionUpdate` qui met à jour les données au moment où elles sont utilisées et permet au programme de continuer son exécution lors de la mise à jour. La limitation de ce choix est que certaines données peuvent rester non compressées très longtemps en mémoire avant d'être enfin utilisées. Il est possible d'employer des stratégies d'accès plus complexes pour ne pas avoir un tel problème, moyennant d'autres limitations. Par exemple, il est possible d'accéder aux données immédiatement pour leur attacher un gestionnaire spécial dont le rôle est d'attendre que la donnée ne soit plus utilisée pendant une durée fixée. Lorsque c'est le cas, le gestionnaire met à jour (et compresse) la donnée. Les solutions possibles ne sont pas détaillées ici. Cependant, des pistes sont proposées dans la sous-section 11.1.4 qui discute de la conception de stratégies de mise à jour.

La figure 11.2 présente le patch dynamique de la mise à jour. Après la définition du nouveau code et du script de mise à jour, un gestionnaire est créé et démarré dans le programme et les unités de mise à jour lui sont envoyées.

11.1.3 Durée de vie des données en cache

Certaines données peuvent changer au cours du temps, impliquant qu'une même requête puisse obtenir une donnée différente à deux moments différents. La seconde mise à jour propose donc d'invalider les données en cache à partir d'un certain temps d'ancienneté. Dans la mise à jour présentée ici, ce temps est fixé arbitrairement pour toutes les sessions et toutes les données. Or, certaines données mises en cache peuvent devenir obsolètes plus rapidement que d'autres. Par exemple, dans le cas de pages internet, les pages dont le contenu est généré dynamiquement deviennent obsolètes plus rapidement que les pages statiques. La mise à jour présentée ici n'adresse pas cet aspect du problème.

Comme le montre la figure 11.3, il est nécessaire de mettre à jour la classe `Session`, et en particulier ses méthodes `add_cache` et `has_cached`. Au lieu de contenir uniquement des données, le dictionnaire `cache` contient désormais une paire (donnée,date de mise en cache). Toute donnée en cache dont l'ancienneté dépasse un seuil fixé est considérée comme absente du cache et sera rechargée à la prochaine demande.

Au moment d'appliquer la mise à jour, les données en cache ne sont pas datées et il est impossible de savoir si elles sont obsolètes ou non. Le cache de chaque session mise à jour est donc invalidé (les données qu'il contient sont supprimées). Mais si cela est fait alors qu'une session est encore active, des données juste mises en cache peuvent être supprimées et rechargées à nouveau. Il est donc préférable de

```
#New code
TIMEOUT = 5
class Session_v2(object):
    def __init__(self):...
    def add_cache(self,key,data):
        timestamp = int(time())
        self.cache[key] = (timestamp,data)
    def has_cached(self,request):
        if request in self.cache.keys():
            d = int(time())-self.cache[request][0]
            return d < TIMEOUT
        return False
    def load_from_cache(self,request):...
    def enter(self):...
    def leave(self):...

#Update script
#Transformer for class Session
def sessiontransf(obj):
    obj.cache = {}
    obj.active = False

#micro-manager for delayed update of sessions
def micromanage_session(s):
    def leave_and_update():
        Session.leave()
        updateToClass(s,Session_v2,sessiontransf)
        delattr(s,"leave")
    s.leave = leave_and_update

#Create update unit classes
class SessionUpdate(Update):
    def __init__(self,name):
        self.pending_sessions = []
        super(SessionUpdate,self).__init__(name)
    def wait_alterability(self):
        return True
    def apply(self):
        sessions = DataAccessor(Session,"immediate")
        for s in sessions:
            if s.active:
                micromanage_session(s)
                self.pending_sessions.append(s)
            else:
                updateToClass(s,Session_v2,sessiontransf)
        redefineClass(Session,Session_v2)
    def wait_over(self):
        finished = False
        while not finished:
            finished = True
            for s in self.pending_sessions:
                if type(s) != Session_v2:
                    finished = False
        self.pending_sessions = None

#Create update units
update_session = SessionUpdate("sessionUpdate")
#Get the previously started manager
manager = fetch_manager("manager")
#Send update units
manager.add_update(update_session)
```

FIGURE 11.3 – Code simplifié de la seconde mise à jour

n'invalider le cache d'une session donnée que lorsqu'elle n'est pas utilisée. Pymoult ne fournissant pas de type d'unité de mise à jour correspondant à cette stratégie, il est nécessaire de définir un nouveau type.

La figure 11.3 définit le type d'unité `SessionUpdate` qui suit la stratégie suivante :

1. Accéder immédiatement à toutes les instances de `Session`.
2. Pour chaque instance :

   si la session est active un micro-gestionnaire lui est attaché ;

   sinon la session est transformée instantanément.
3. Redéfinir la classe `Session`.
4. Attendre que toutes les sessions soient transformées.

Les étapes 1 à 3 de cette stratégie sont effectuées lors de l'étape d'application des modifications, elles sont donc implémentées dans la méthode `apply` de `SessionUpdate`. L'étape 4 correspond à l'attente de la terminaison de l'unité de mise à jour et est implémentée dans la méthode `wait_over`.

Cette mise à jour utilise trois éléments de l'interface de bas niveau de Pymoult : `DataAccessor` pour accéder aux sessions, `updateToClass` pour les transformer et `redefineClass` pour redéfinir la classe `Session`. En instanciant la classe `DataAccessor`, un itérateur sur l'ensemble des instances de `Session` est créé. Ici, les instances sont accédées immédiatement : à la création de l'itérateur, l'ensemble des sessions est collecté et enregistré dans l'itérateur.

Le micro-gestionnaire est une surcharge de la méthode `leave` des sessions. Il porte ce nom car il remplit une des fonctions d'un gestionnaire (appliquer des modifications). La fonction `leave_and_update` est enregistrée comme un attribut de la session ciblée, sous le nom *leave*. Lorsque cette session est quittée, c'est cette fonction qui est appelée au lieu de la méthode `leave` de la classe `Session`. La session est alors mise à jour après avoir appelé la méthode `leave` de la classe `Session`. Puis, l'attribut `leave` est supprimé de la session. Les prochaines fois que la session sera quittée, ce sera la méthode `leave` de `Session_v2` qui sera appelée.

Le micro-gestionnaire exploite le mécanisme de résolution des méthodes et des attributs en Python. Les deux sont résolus selon le même procédé. D'abord le nom de l'attribut (ou de la méthode) est cherché parmi les attributs de l'objet. Si il correspond à une entrée, cette dernière est retournée. Sinon, le nom est cherché dans la classe de l'objet, puis dans ses classes parentes. Lorsque `session.leave` est appelée, c'est `leave_and_update` qui est retournée en premier en tant qu'attribut de la session. Une fois cette attribut supprimé, c'est à nouveau la méthode `leave` de la classe `Session` qui est retournée. Il est possible de ne pas employer cette propriété de Python en utilisant une première fois `updateToClass` pour changer les sessions actives et leur faire adopter une classe transitoire `SessionT` dont la méthode `leave` reproduit le comportement du micro gestionnaire.

Cet exemple de mise à jour montre comment l'interface de bas niveau de Pymoult peut être utilisé pour combiner des mécanismes et concevoir des scripts de mise à jour originaux. Ici, les sessions sont accédées immédiatement puis transformées de manière instantanée si elles sont actives ou de manière retardée sinon. En définissant un nouveau type d'unité de mise à jour, il est possible de décrire les tâches effectuées à chaque étape du cycle de vie. L'unité définie ici n'exprime aucun critère d'altérabilité et applique directement les modifications avant d'attendre que les sessions transformées de manière retardée soient toutes à jour. Cet exemple montre comment il est possible de concevoir des stratégies complexes en les décomposant en modifications successives du programmes. Ici, les sessions actives sont modifiées une première fois pour mettre en place la stratégie d'accès. Elles sont modifiées une seconde fois pour les mettre à jour.

11.1.4 Concevoir la stratégie de mise à jour

Les deux mises à jour appliquées au programme de mise en cache ont été appliquées selon des stratégies différentes. La première stratégie, simple, présente une limitation : les données rarement utilisées restent longtemps décompressées tandis que les données utilisées fréquemment le sont. Par souci de performance, le contraire serait préférable. Les données couramment utilisées ne sont pas compressées pour éviter les décompressions permanentes tandis que que les données rarement utilisées le sont pour réduire la consommation de mémoire du programme.

Une telle stratégie de mise à jour est plus complexe que celle présentée en premier exemple. Elle nécessite d'identifier le bon moment pour accéder et/ou transformer les données, ce qui peut être complexe (comment détecter qu'une donnée est rarement utilisée ?) et coûteux (les outils détectant le bon moment et transformant les données ajoutent un surcoût au programme). Chaque stratégie a ses propres avantages et inconvénients. Il convient au développeur de la mise à jour de choisir laquelle correspond le mieux à ses besoins (simplicité de réalisation, surcoût introduit, performances obtenues ...).

La stratégie de la seconde mise à jour a fait le choix de plus de complexité pour de meilleurs performances du programme. En choisissant de ne pas vider le cache des sections tant qu'elles sont encore ouvertes, le programme évite de recharger des données qui étaient encore en cache, mais la mise à jour est plus longue à appliquer (il faut attendre que toutes les sessions aient été fermées une fois) et plus complexe a concevoir (un micro-gestionnaire a été conçu).

Ces deux exemples montrent l'intérêt de choisir les stratégies d'accès et de transformation des données lors de la conception des mises à jour. Les autres expériences conduites avec Pymoult ont montré que d'une manière générale, il est intéressant de pouvoir choisir quels mécanismes employer pour chaque tâche de mise à jour (accès aux données, détection de l'altérabilité, mise à jour des fonctions ...). Les expériences conduites sur Django présentées dans la sous-section 11.2.2 de ce chapitre ont également amené à sélectionner et concevoir les stratégies de mise à jour employées.

11.2 Applications concrètes

Cette section présente trois cas d'utilisation de Pymoult sur des applications concrètes. Les deux prochaines sous-sections décrivent comment deux applications serveurs ont été mises à jour dynamiquement et la troisième sous-section décrit Pycots, un *framework* Python pour applications orientées composants.

11.2.1 Pyftpdlib

Pyftpdlib [65] est une bibliothèque Python fournissant une interface de haut niveau pour développer des serveurs FTP. Un programme employant cette bibliothèque pour créer un serveur FTP basique (accès anonyme, un seul dossier servi) a été développé. Le programme a été modifié lors de quatre mises à jour successives de pyftpdlib (de la version 1.1.0 à la version 1.4.0).

Ces mises à jours concernent principalement les méthodes des classes et n'affectent pas le format des données. Les modifications de ces méthodes sont assez légères et changent peu la sémantique du programme. Les patchs dynamiques emploient donc surtout la classe d'unité `SafeRedefineMethodUpdate` qui redéfinit une méthode d'une classe lorsqu'elle est quiescente. La nature des modifications peut s'expliquer par une maturité de la bibliothèque dont l'architecture ne change plus.

En moyenne, les patchs dynamiques emploient trois unités de mise à jour différentes, chaque unité correspondant à la mise à jour d'une méthode, d'une fonction ou d'une classe (lorsque toute la classe est redéfinie). Il s'agit de petites mises à jour impactant des parties localisées du programme. Sur le programme de serveur FTP basique, les mises à jour sont appliquées en 2,86 secondes en moyenne (temps mesuré entre l'envoi de la première unité au gestionnaire et la fin de la dernière unité) et aucune interruption de service ni aucun ralentissement n'ont été constatés. Ces temps d'application des mises à jour sont négligeables devant le temps séparant la publication de deux versions consécutives. Entre les publications des versions 1.1.0 et 1.2.0, 15 jours se sont écoulés. Plus d'un an s'est écoulé entre la version 1.2.0 et la version 1.3.0. Les deux versions suivantes (1.3.1 et 1.4.0) ont respectivement été publiées 35 et 52 jours après leur précédente version.

Il est possible qu'une application plus complexe employant la bibliothèque pyftpdlib nécessite que le script de mise à jour soit plus complexe et que des combinaisons de mécanismes différents soient utilisées au fil des mises à jour. En effet, une partie des modifications appliquées affecte des éléments de la bibliothèque qui ne sont pas employés par le programme utilisé ici.

11.2.2 Django

Django [27] est un framework permettant de développer des applications web en Python. Une application de test a été crée et un serveur Django dans la version 1.6.8 a été utilisé pour servir cette application. Il a été mis à jour deux fois successivement pour atteindre la version 1.6.10.

Les mises à jour consistent majoritairement en la modification de fonctions et de méthodes de classe et ne changent pas le format des données de manière significative. Comme dans le cas de pyftpdlib, il s'agit d'une application mature dont l'architecture change peu.

Les deux mises à jour appliquées emploient des stratégies différentes. La première mise à jour redéfinit un petit nombre de méthodes et de fonctions indépendantes. Elles ont été redéfinies chacune indépendamment lorsqu'elles étaient quiescentes. Compte tenu du petit nombre de modifications, employer la stratégie la plus simple à réaliser (plusieurs unités de type `SafeRedefineUpdate`/ `SafeRedefineMethodUpdate`) est une possibilité. La deuxième mise à jour redéfinit un grand nombre de méthodes et de fonctions de petite taille. La complexité de Django ne permettant pas simplement de juger de l'interdépendance de ces fonctions, il a été préféré de mettre à jour toutes ces fonctions et méthodes au même moment, lorsqu'elles sont toutes quiescentes. Heureusement, la constitution de Django avait permis, au préalable, de placer un point de mise à jour entre le traitement de deux requêtes, alors que le programme est globalement quiescent. Les fonctions et méthodes sont alors redéfinies lorsque ce point de mise à jour est atteint.

11.2.3 Coqcots & Pycots

Dans le domaine de la mise à jour dynamique, les programmes orientés composants sont mis à jour en les reconfigurant, c'est à dire en remplaçant des composants et/ou en changeant les connections entre eux. Pour que l'opération ne cause pas de problème, il faut qu'aucun autre composant n'utilise ceux qui sont affectés par la reconfirguration. S'assurer de la quiescence des composants affectés est un moyen de satisfaire cette condition. Une méthode possible est d'arrêter les composants affectés ainsi que les composants qui dépendent d'eux. Le défaut de cette méthode est qu'en parcourant les dépendances des composants arrêtés, les composants fournissant les services du programme peuvent être arrêtés, ce qui affecte sévèrement la disponibilité du programme.

Coqcots [18] est un modèle à composant pour applications reconfigurables qui propose, au lieu d'arrêter les composants dépendant d'un composant C reconfiguré, de les mettre à jour dynamiquement pour qu'ils n'emploient pas C le temps de la reconfiguration. Pycot est une implémentation Python du modèle Coqcots et emploie Pymoult effectuer les tâches de mise à jour dynamique. Coqcots permet, à l'aide de l'assistant de preuve Coq, de développer des reconfigurations et de prouver leur validité avant de les traduire en Python et d'utiliser Pycots pour les appliquer.

Dans Pycots, l'implémentation du modèle utilise des éléments de base de Python pour construire des éléments plus sophistiqués. Les composants sont instances de classes spécifiques réifiant la notion de composant. Pour mettre à jour des composants, Pycots utilise les mécanismes de Pymoult mettant à jour

les éléments de base (classe, objet, méthode). Par conséquent, Pycots emploie principalement l'interface de bas niveau de Pymoult. Les opérations de reconfiguration correspondent à des unités de mise à jour définies par Pycots employant des mécanismes pour détecter la quiescence des méthodes des composants ou encore transformer les composants.

Comme la deuxième mise à jour de l'exemple présenté en section 11.1, Pycots applique des modifications dans une première phase pour préparer la véritable mise à jour du programme. Une fois la mise à jour appliquée, les modifications appliquées dans la première phase sont nettoyées. Dans l'exemple de la section 11.1, il s'agissait de la fonction `leave_and_update`. Ici, il s'agit de l'implémentation des composants dépendants des composants affectés par la reconfiguration du programme.

11.3 Discussion

Concevoir la stratégie selon laquelle une mise à jour est appliquée permet, en donnant le contrôle au développeur de mise à jour, de s'assurer que les besoins de chaque mise à jour soient satisfaits au mieux. Il est possible d'adopter la stratégie assurant de meilleurs performances, la plus simple à réaliser ou celle qui est le moins propice aux erreurs. L'exemple de la section 11.1 montre des choix de stratégie entre simplicité et performance, tandis que le cas de Django 11.2.2 a montré le cas d'une stratégie prudente où un point de mise à jour est attendu pour mettre à jour un groupe de fonctions.

L'architecture Starmoult, implémentée dans Pymoult permet de concevoir ces stratégies en laissant le développeur définir le script de mise à jour. En définissant de nouveaux types d'unité, en sélectionnant et en combinant les mécanismes a utiliser, il peut choisir et réaliser la stratégie la plus adaptée pour chaque mise à jour. La plate-forme est alors compatible et mieux adaptée à chaque mise à jour (dans la limite des mécanismes fournis). Lorsqu'une mise à jour n'a aucun besoin particulier, il est possible de choisir la stratégie la plus simple à définir en employant des types d'unité prêtes à être configurées.

Au moment où ce manuscrit est rédigé, Pymoult, et l'architecture Starmoult attendent du développeur de mise à jour qu'il choisisse seul la stratégie à employer et l'implémente en utilisant l'interface de programmation de la plate-forme. Or, il est difficile de tenir compte à la fois des besoins du programme, de la mise à jour et des mécanismes employés. Pour aider le développeur de mise à jour dans ses choix, des lignes directrices peuvent être proposées. Elles indiqueraient quelle combinaison de mécanismes est la plus adaptée pour un ensemble de besoins donnés. Les expériences conduites avec Pymoult ont permis d'identifier quelques-unes de ces lignes directrices. Par exemple, il est préférable d'accéder progressivement aux données et/ou de les transformer de manière retardée lorsqu'elles sont en grand nombre.

En définissant des métriques permettant de quantifier les stratégies de mise à jour, il serait possible de comparer ces dernières et d'identifier les situations dans lesquelles elles sont les plus adaptées. Par exemple, en mesurant le temps nécessaire pour qu'une mise à jour soit entièrement appliquée, il est possible d'identifier la stratégie la plus efficace. D'autres métriques pourraient mesurer la sûreté d'une stratégie (la probabilité que la mise à jour cause des problèmes), l'impact sur les performances du programme (ralentissement, dégradation du service lors de la mise à jour) ou encore le temps d'attente moyen avant de pouvoir appliquer la mise à jour (par exemple, le temps nécessaire au programme pour devenir altérable).

Chapitre 12

Conclusion et perspectives

Le domaine de la mise à jour dynamique permet de modifier des programmes pendant leur exécution, sans nécessairement interrompre les services qu'ils fournissent. En combinant un programme avec une plate-forme, il est possible d'utiliser des mécanismes spécifiques pour redéfinir des fonctions ou des types du programme, accéder à ses données et les transformer. L'approche habituelle suivie par ces plates-formes cherche à rendre l'opération la plus transparente possible, demandant peu d'interventions de la part du développeur de mise à jour. Par conséquent, les plates-formes emploient une même série de mécanismes pour appliquer toutes les mises à jour (en général, un mécanisme par tâche). Ces mécanismes ayant leurs propres besoins, leurs propres défauts et leurs propres avantages, la plate-forme qui les combine se spécialise pour certains types de programmes ou de mises à jour. Pour un programme donné, il est donc nécessaire de choisir la plate-forme la plus adaptée, en tenant compte des exigences du programme et en prévoyant à l'avance les mises à jour. Si les mises à jour anticipées sont plus faciles à appliquer, celles dont les exigences sont incompatibles avec les mécanismes fixés par la plate-forme peuvent s'avérer impossible à appliquer.

Le présent manuscrit a proposé une nouvelle approche permettant d'éviter cette situation en permettant aux plates-formes de s'adapter aux programmes et aux mises à jour. Les développeurs choisissent et combinent les mécanismes les plus adaptés à chaque mise à jour, utilisant une interface de programmation fournie par la plate-forme. Pour atteindre ce résultat, la première partie a analysé les plates-formes de l'état de l'art et leurs mécanismes et a identifié des façons de les combiner pour concevoir des mises à jour. La deuxième partie a étudié les mécanismes individuellement pour identifier leurs exigences et a proposé un langage d'expression des mises à jour. La troisième partie a défini l'architecture Starmoult pour plates-formes configurables et a présenté ses deux implémentations : Pymoult et Cmoult.

12.1 Une nouvelle approche pour la mise à jour dynamique

La mise à jour dynamique permet de considérer les programmes comme en évolution permanente. Les mises à jour sont appliquées aux programmes à mesure qu'elles sont disponibles, sans que l'exécution des programmes ne soit interrompue. Ainsi, un programme en exécution n'est plus *figé* et peut, si la plate-forme le permet, aller jusqu'à changer de sémantique.

La figure 12.1 montre les étapes du processus de mise à jour dynamique des programmes. Le nouveau code est extrait, le script de mise à jour est implémenté puis le patch dynamique est compilé et chargé. Lorsqu'une plate-forme de l'état de l'art est employée, implémenter le script se résume à écrire des *transformers* ou d'autres codes complémentaires (traverseur de tas dans Kitsune [44], critères d'altérabilité dans ProteOS [36]). Lorsqu'une plate-forme configurable est employée, l'implémentation du script est à la charge du développeur de la mise à jour. Et parce que cette tâche est complexe, elle est précédée d'une étape de conception où les mécanismes à employer sont sélectionnés, combinés et configurés.

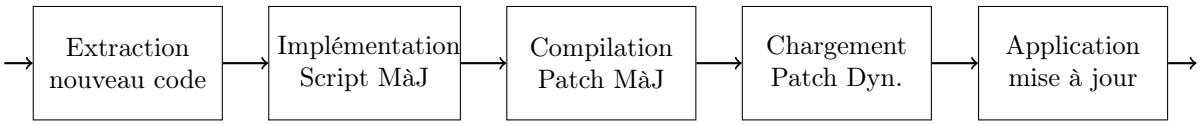


FIGURE 12.1 – Mise à jour d'un programme

Cette étape demande de connaître les besoins du programme et des mises à jour tout en connaissant les capacités des mécanismes. Elle a ses propres problématiques et manipule ses propres concepts (altérabilité, stratégie de transformation des données, ...). La conception des mises à jour est donc une étape spécifique du processus de développement du logiciel, similaire à la conception d'un programme *normal*. Les prochaines sous-sections récapitulent les enjeux de cette étape.

12.1.1 Choisir les mécanismes et les combiner

L'étude détaillée de la partie I a listé les différents mécanismes de mise à jour de l'état de l'art. Il a été constaté que pour une tâche de mise à jour donnée, il existe plusieurs mécanismes différents, chacun ayant ses propres exigences et capacités. Pour une mise à jour donnée, il s'agit alors de choisir pour chaque tâche de mise à jour les mécanismes les plus adaptés à la situation. Les expériences de la partie III ont montré que certains mécanismes sont plus adaptés à certaines situations.

Un premier critère de choix se base sur les exigences des mécanismes. La partie II a montré que chaque mécanisme a des besoins différents, soit en informations sur le programme (introspection, instrumentation, ...), soit en fonctionnalités du système d'exécution (suspension de fils d'exécution, collecte de données, ...). Il est préférable de choisir les mécanismes dont les besoins sont déjà satisfaits par le programme ou la plate-forme.

Un second critère de choix se base sur la manière dont les mécanismes peuvent se combiner. La partie I a identifié des synergies entre mécanismes, choix architecturaux et étapes du cycle de vie des plates-formes. Il convient de choisir les mécanismes qui peuvent au mieux se combiner ou qui sont, par exemple, en synergie avec des choix architecturaux de la plate-forme.

La partie III a montré comment combiner des mécanismes pour définir des stratégies de mise à jour complexes. La deuxième mise à jour appliquée au programme de mise en cache emploie trois mécanismes différents pour répondre à l'exigence principale de la mise à jour : mettre à jour les sessions lorsqu'elles sont fermées. La partie II a également utilisé des mécanismes élémentaires pour définir des mécanismes plus sophistiqués. Par exemple, le mécanisme de redémarrage des fils d'exécution de Kitsune [44] a été implémenté en combinant les mécanismes `mainredef` et `reboot`.

L'idéal est de choisir les mécanismes et de les combiner en se basant sur leurs capacités. Par exemple, dans le chapitre 11, lors de la première mise à jour du programme de mise en cache, la stratégie d'accès progressive a été préférée. En effet, les données en cache étant nombreuses et utilisées de manière non-homogène, il est préférable d'y accéder et de les mettre à jour uniquement lorsqu'elles sont utilisées. Le surcoût lié à la mise à jour des données est réparti dans le temps au lieu d'être concentré au moment où l'unité de mise à jour est appliquée. La sous-section 12.2.2 donne des pistes pour établir des lignes directrices permettant de choisir les mécanismes les plus adaptés dans une situation donnée.

12.1.2 Exprimer les mises à jour

Le langage utilisé dans la partie II pour écrire des programmes de mise à jour a montré l'importance de l'accès aux informations du programme pour l'expression des mises à jour. Lors des expériences autour de Cmoult, ces besoins ont été confirmés et ont mené à l'utilisation des informations de débogage du format ELF. La partie II a également précisé l'intérêt d'avoir un langage de programmation complet pour exprimer des mises à jour complètes.

L'architecture Starmoult présentée dans la partie III définit une interface de programmation permettant d'écrire des mises à jour. Cette interface basée sur le modèle générique établi dans la partie I permet d'exprimer n'importe quelle mise à jour même si certaines demandent plus d'efforts que d'autres (comme l'a montré la deuxième mise à jour du programme de mise en cache).

L'analyse de la partie I a montré que les plates-formes manipulent les mêmes concepts (redéfinir les fonctions, accéder aux données, ...) et partagent des mécanismes communs. C'est ce constat qui a permis de définir l'architecture Starmoult qui propose une méthode générique d'expression des mises à jour au travers d'une interface de programmation se voulant universelle.

12.1.3 Fournir les mécanismes

La diversité des combinaisons de propriétés identifiées dans la partie I a confirmé l'intérêt d'une plate-forme configurable pouvant s'adapter aux besoins des programmes et des mises à jour. Il devient alors possible de reproduire le comportement des différentes familles de plates-formes qui ont été identifiées.

Les plates-formes Starmoult fournissent les mécanismes qu'elles implémentent au travers d'une interface de programmation offrant une abstraction de haut niveau qui rend l'interface indépendante de

l'implémentation des mécanismes. Cependant, les mécanismes fournis sont tout de même impactés par le langage de la plate-forme. Par exemple, dans Pymoult, la redéfinition d'une fonction permet d'en changer la signature tandis que dans Cmoult, cela n'est pas possible.

L'interface de programmation proposée par Starmoult permet deux niveaux de développement des mises à jour. En utilisant les unités prêtes à être configurées de l'interface de haut niveau, il est possible de développer rapidement des mises à jour simples. En définissant ses propres unités utilisant les mécanismes de l'interface de bas niveau, le développeur de mise à jour peut concevoir des stratégies complexes mieux adaptées à des situations spécifiques. Cela permet de faciliter le travail du développeur de mise à jour sans pénaliser la capacité de la plate-forme à s'adapter.

L'approche des plates-formes configurables incite à implémenter un grand nombre de mécanismes et de stratégies pour être mieux adaptables à toutes les situations. Elle incite également à voir les mécanismes comme des briques de base utilisées pour développer d'autres mécanismes ou des stratégies originales. Les plates-formes configurables doivent donc fournir des mécanismes élémentaires, même si seuls ils ne permettent pas d'accomplir une tâche de mise à jour. Par exemple, Pymoult fournit séparément les deux mécanismes équivalents aux instructions `mainredef` et `reboot` du langage de mise à jour de la partie II.

12.2 Pistes pour de futures recherches

Les enjeux de la nouvelle approche ouvrent de nouveaux axes de recherche. Il s'agit d'étudier comment étendre les capacités d'adaptation des plates-formes configurables, comment aider le développeur à concevoir les mises à jour ou encore définir un langage universel spécifique à ces dernières. Des expériences ont été conduites pour identifier les enjeux de ces axes. Si elles ont permis d'identifier des pistes de recherche, elles ont besoin d'êtres approfondies avant d'obtenir des résultats concluants. Ces expériences sont présentées dans les prochaines sous-sections.

Fournir les mécanismes prêts à l'emploi en laissant les développeurs les configurer permet d'autres utilisations que la mise à jour des programmes. En effet, comme indiqué dans le chapitre 2, d'autres domaines emploient des mécanismes ou des notions semblables aux mécanismes et aux notions de la mise à jour dynamique. Il devient alors possible d'employer des plates-formes configurables dans d'autres domaines, offrant de nouvelles possibilités. La sous-section 12.2.4 propose des exemples de telles utilisations.

12.2.1 De nouveaux mécanismes et de nouvelles stratégies

En concevant le script d'une mise à jour donnée, le développeur peut être amené à employer une stratégie spécifique, peu efficace dans la majorité des cas, mais mieux adaptée dans ce cas précis. Dans une plate-forme suivant l'approche usuelle du domaine, cette stratégie n'est très certainement pas implémentée, à moins que la plate-forme ne soit spécialisée pour des mises à jour similaires à celle-ci. Elle est inadaptée pour la majorité des cas et l'utiliser pour chaque mise à jour est indésirable.

Les plates-formes configurables peuvent alors proposer des mécanismes et des stratégies peu fréquents car moins adaptés à la majorité des cas. Par exemple, les mécanismes de contextes utilisés par ActiveContext [76] et Thesus [77] font s'exécuter les fils dans différentes versions du code du programme, les données partagées étant synchronisées par un mécanisme de la plate-forme. Ces mécanismes ajoutent un surcoût lors de l'exécution du programme, les rendant inadaptés pour des mises à jour *simples*. Pour une mise à jour ne pouvant pas attendre un état d'altérabilité (mise à jour de sécurité critique, ...), ces mécanismes ont l'avantage de considérer le programme comme toujours altérable. Une plate-forme configurable pourrait proposer ces mécanismes en prévoyant ce genre de situation.

Une implémentation de ces mécanismes de contextes a été commencée dans Pymoult. Elle nécessite une modification profonde et complexe du fonctionnement de l'interpréteur. L'implémentation des contextes telle que prévue dans PyPy-DSU introduit une couche d'indirection supplémentaire entre valeur et variables. Chaque variable a plusieurs valeurs selon le contexte courant. Disposer de mécanismes de contextes dans CPython-DSU ou PyPy-DSU permettrait l'implémentation de nouveaux mécanismes ainsi que d'apporter plus de flexibilité à la plate-forme. Il deviendrait possible de créer un nouveau contexte avant d'appliquer une mise à jour temporaire et de quitter ce contexte pour désinstaller cette mise à jour.

Des expérimentations sur des programmes temps réel, ont montré l'importance de la mémoire du programme lors de sa mise à jour. En effet, le calcul de l'état du programme à un cycle donné peut dépendre de son état lors des cycles précédents. Il est possible que la nouvelle version du programme dépende d'états antérieurs qui ne sont pas enregistrés. Une solution est de décomposer la mise à jour en

plusieurs étapes, comme vu dans l'exemple du chapitre 11 où les sessions étaient mises à jour une première fois pour mettre en place la stratégie d'accès. Une première mise à jour permet de conserver les états antérieurs demandés par la nouvelle version. Une fois ces états collectés, il devient possible d'appliquer la mise à jour. Bien que ce genre de stratégie soit peu commun dans les plates-formes *classiques*, l'idée de décomposer une mises à jour en plusieurs mises à jour plus petites est déjà exposée dans A-LTL [79] où l'adaptation d'un programme peut être décomposée en plusieurs autres adaptations.

12.2.2 Aider la conception des scripts de mise à jour

Développer une mise à jour à partir du code de la nouvelle version du programme, et plus particulièrement concevoir le script de mise à jour, peut être complexe dans certaines situations. Le développeur doit connaître les besoins du programme, de la mise à jour mais aussi les capacités, les avantages et les exigences des mécanismes.

Pour faciliter le choix et la configuration des mécanismes, des lignes directrices pourraient spécifier les avantages et les inconvénients des mécanismes dans des situations *typiques*. Les expériences conduites avec Pymoult ont donné de premiers résultats vers ces lignes directrices. Par exemple, comme vu lors de la première mise à jour de l'exemple du chapitre 11, lorsqu'une grande quantité de données doit être mise à jour, il est préférable d'y accéder progressivement ou de les transformer de manière retardée.

Pour dresser un inventaire des avantages et des inconvénients des différents mécanismes, il est nécessaire de comparer différentes combinaisons et configurations de mécanismes et de quantifier leurs capacités à appliquer des mises à jour *typiques*. Cela implique d'identifier quelles sont les mises à jour *typiques* et de définir des jeux de tests représentatifs de ces dernières. Et pour chacun de ces types de mise à jour, il s'agirait de donner une combinaison recommandée de mécanismes.

Pour quantifier objectivement les capacités des mécanismes, il faut également définir des métriques spécifiques (impact sur les performances du programme, coût d'implémentation d'une stratégie, temps nécessaire au programme pour être altérable, ...). Les lignes directrices pourraient alors être exprimées sous forme de diagrammes araignées (ou de Kiviat [1]), indiquant les mesures des mécanismes pour chaque mise à jour *typique*.

Implémenter les métriques dans une plate-forme configurable permettrait aux développeurs de mises à jour de tester et comparer différentes stratégies sur une copie du programme mis à jour pour choisir la mieux adaptée aux besoins d'une mise à jour donnée. Il serait également possible de prototyper de nouvelles stratégies et de nouveaux mécanismes et d'en mesurer les performances.

Par ailleurs, il est important de vérifier que la complexité de la conception des mises à jour passe à l'échelle dans le cas de gros programmes complexes. Pour cela, il est nécessaire de compléter les expériences présentées et mentionnées dans ce manuscrit par le développement de mises à jour pour de tels programmes.

12.2.3 Un langage spécifique pour les mises à jour dynamiques

Les plates-formes Starmoult prévoient que le script de mise à jour soit écrit dans le langage de programmation de la plate-forme (C, Python) en employant les éléments (fonctions, classes, ...) fournis par son interface de programmation. La poursuite des expériences autour de Cmoult devraient confirmer qu'une même interface de programmation peut être utilisée pour mettre à jour des langages différents. L'interface définie par Starmoult donne alors une base pour un langage spécifique au développement des mises à jour. L'hypothèse de la possibilité d'un langage de mise à jour universel est également supportée par l'analyse de la partie I qui a montré qu'un grand nombre de mécanismes est indépendant du langage de la plate-forme, à l'implémentation près. Si le langage de mise à jour est d'assez haut niveau, il pourra être adapté à tous les langages de programmation.

Une autre possibilité est d'écrire le script de mise à jour sous forme d'annotations ajoutées au nouveau code (`redéfinir fonction foo` sur une fonction, `appliquer instantanément transformer ftrans` au dessus d'une classe dont les instances doivent être transformées, ...). L'implémentation d'un mécanisme d'annotations dans PyPy-Dsu a été commencée pour étudier les enjeux liées à cette option. Cependant, des études plus approfondies sont nécessaires pour déterminer sa faisabilité et sa pertinence.

Une fois le script de mise à jour écrit, soit dans un langage spécifique, soit avec des annotations, un compilateur peut ensuite générer le patch dynamique employant l'interface de programmation de la plate-forme. Il devient alors possible d'exprimer les mises à jour d'une application complexe utilisant plusieurs plates-formes et langages différents d'une même manière. Le compilateur génère ensuite les

1. http://fr.wikipedia.org/wiki/Diagramme_de_Kiviat

patchs dynamiques dans les formats correspondants à chaque plate-forme. Cela permettrait de mettre à jour des programmes multi-langages (par exemple, un programme Python dont certaines parties sont écrites en C).

12.2.4 De nouvelles utilisations de la mise à jour dynamique

Jusqu'à présent, la mise à jour dynamique était centrée sur l'application de modifications visant à corriger des bugs ou étendant certaines fonctionnalités du programme. Très peu de plates-formes envisagent une modification de la sémantique du programme.

Les plates-formes configurables, assurant que chaque mise à jour puisse être appliquée, ouvrent de nouvelles possibilités d'utilisation de la mise à jour dynamique. Il devient possible d'implémenter un compilateur juste-à-temps employant des mécanismes de mise à jour dynamique pour remplacer à la volée des parties du code interprété par du code natif, plus performant. Il est d'ailleurs à remarquer que les compilateurs juste-à-temps emploient des mécanismes semblables à certains mécanismes de mise à jour dynamique, notamment le remplacement de fonctions sur la pile.

Les programmes autonomes pourraient se mettre à jour pour s'adapter aux changements de leur environnement. Par exemple, lors d'une attaque informatique, un programme pourrait interroger une base de donnée pour obtenir un patch modifiant son comportement et le protégeant de cette attaque. Lors d'une attaque de type déni de service, un serveur pourrait être modifié pour servir uniquement des clients de confiance identifiés comme *sûrs*.

12.3 Une étape pour les concevoir toutes

Comme pour un programme, le développement d'une mise à jour doit tenir compte d'exigences imposées. La mise à jour doit être appliquée le plus vite possible, ou encore doit impacter le moins possible les performances du programme modifié. Ce manuscrit a montré la diversité des plates-formes et des stratégies qu'elles suivent, adaptées au mieux à des types de programmes spécifiques (système d'exploitation, programmes orientés composants, serveurs, systèmes distribués, ...). En concevant des plates-formes configurables et en ajoutant une étape de conception au processus de développement des mises à jour, il est possible, au lieu de choisir une plate-forme adaptée au programme et de se limiter aux mécanismes qu'elle emploie, d'adapter une plate-forme au programme et d'adapter chaque mise à jour à ses besoins.

Cela est possible grâce à la grande diversité des mécanismes existants et à leur portabilité. Même si implémenter un mécanisme peut s'avérer plus compliqué dans un langage donné, très peu de mécanismes sont exclusifs à un langage ou à un type de plate-forme. À condition d'utiliser un niveau assez haut d'abstraction, il est possible d'exprimer une mise à jour de la même manière quelque soit la plate-forme qui l'applique. En effet, chaque plate-forme manipule les mêmes concepts (altérabilité, stratégie d'accès, ...) et répond aux mêmes problématiques. Cela permet d'établir un modèle générique pour la description des plates-formes et des mécanismes. Il en résulte un modèle de plate-forme configurable permettant de concevoir et développer tout type de mise à jour. Si les plates-formes ont la capacité de communiquer entre elles, il devient alors possible de distribuer une mise à jour sur plusieurs plates-formes. Pour mettre à jour une application composée de deux programmes combinés avec des plates-formes différentes, il serait possible de considérer l'application comme un tout et de ne développer qu'une seule mise à jour. Chaque plate-forme effectuerait alors les tâches qui correspondent au programme auquel elle est combinée.

Cette nouvelle approche à la mise à jour dynamique ouvre la voie à de nouvelles utilisations. Les programmes pouvant être modifiés en pleine exécution, il est possible d'adapter un programme à une situation particulière, par exemple, pour protéger un serveur sous attaque informatique ou encore pour répondre à une panne d'un composant matériel.

L'étape de conception des mises à jour est la clé de la flexibilité des plates-formes configurables. Elle correspond cependant à une tâche complexe. En fournissant des stratégies prêtes à êtres configurées, les plates-formes assurent une certaine simplicité pour les situations classiques. Dans les autres cas, il faut choisir et combiner les mécanismes appropriés en suivant certaines lignes directrices, puis implémenter la mise à jour. Ce processus ressemblant à la façon dont un compilateur analyse le code d'un programme pour le transformer et l'optimiser, il serait intéressant d'étudier la possibilité d'un compilateur analysant une mise à jour et combinant les mécanismes selon ces lignes directrices. Appliquer des mises à jour deviendrait aussi transparent que selon l'approche usuelle, sans restreindre le champ des possibilités.

Annexes

Annexe A

Analyse des plates-formes

A.1 Cycle de vie

Plate-forme	Préparation (1)	Attente Altérabilité	Détection Altérabilité	Suspension	Modification	Préparation (2)
Kitsune	-	-	Point MaJ	Suspension totale	MàJ tas	-
ProteOS	Envoi de *state filters* au processus	-	Début de boucle + *state filters*	Suspension partielle	création nouv. proc. , transfert de l'état	Vérification de cohérence de l'état transféré
Ksplice	-	-	Quiescence fonctions	Suspension totale	Redéfintion fonctions, MàJ donneés	-
OPUS	Attachement gestionnaire	-	Quiescence fonctions	Suspension totale	Redéfinition fonctions	-
K42	Attachement gestionnaire, MàJ fabriques	Blocages nouv. requêtes	Quiescence composants	-	Remplacement du composant	-
Ginseng	-	-	Point MaJ	Suspension totale	Redef. Fonctions, MàJ donneés	-
Jvolve	-	-	Quiescence classes	Suspension totale	Redef classes et méthodes, MàJ tas	-
Rubah	-	-	Point MaJ	Suspension totale	MàJ tas (ou début MàJ tas)	-
ActiveContext	-	-	-	-	Changement du contexte principal	-
Javelus	-	-	Quiescence classes	Suspension totale	Mise à jour des classes	-
DCP	-	-	-	-	Chargement composant, instal. proxys	-
Polus	callback dév MàJ, init. nouvelles variables	-	Quiescence fonctions	Suspension totale	MàJ fonctions, instal. *traps* (MàJ données)	-

Plate-forme	Reprise	Attente terminaison	Détection terminaison	Nettoyage	Nouvelle version
Kitsune	Redémarrage fils	Guidage exécution	Point de MàJ atteint	-	-
ProteOS	Arrêt ancien proc. , démarrage nouv. proc.	-	-	-	-
Ksplice	Reprise	-	-	-	-
OPUS	Reprise des fils	-	-	Détachement gestionnaire	-
K42	Déblocage des fils	-	-	Détachement gestionnaire	-
Ginseng	Reprise des fils	-	-	-	Transformartion progressive des données
Jvolve	Reprise des fils	-	-	-	-
Rubah	Redémarrage fils	Guidage exécution	Point de MàJ atteint	-	(Transformartion progressive des données)
ActiveContext	-	MaJ données progressive	Fin d'ugage des fonctions	Nettoyage code obsolète	-
Javelus	Reprise des fils	MaJ données progressive	Fin d'usage des données	Nettoyage code obsolète	-
DCP	-	-	-	-	Plusieurs version coexistent (MàJ pour les suppr.)
Polus	Reprise des fils	MaJ données progressive	Fin d'ugage des fonctions	Callback + suppr. *traps* synchro	-

Plate-forme	Préparation (1)	Attente Altérabilité	Détection Altérabilité	Suspension	Modification	Préparation (2)
Thesus	-	-	-	-	Changement du contexte principal	-
STUMP	-	-	Point MaJ	Suspension totale	Redéfinition fonctions, MàJ donneés	-
Local Dynamic ...	-	-	-	-	Remplacement de composant	-
DUC	-	-	-	Suspension partielle	Remplacement du composant	Désérialisation des connections
Swap & Play	Copie de l'état depuis ancienne version	-	-	Suspension partielle	Transfert d'l'exécution et MàJ de l'état	-
Supporting ...	-	-	Quiescence service	Blocage des requêtes	Remplacement du service	-
UpStare	-	-	Point MaJ	Suspension totale	MàJ fonctions, var. globales, capture des piles	-
TOS	-	-	Point MàJ + quiescence	Suspension totale	Redef classes et méthodes, MàJ tas	-
Javadaptor	-	-	-	Suspension totale	Redéf. méthodes, instal. conteneurs et proxy	-
Javeleon	-	-	-	-	Copie données et MàJ, début synchronisation versions	-
Hotspot	-	-	-	-	Redéfinition méthodes	-
EmbedDSU	-	Blocage nouv. appels méthodes	Quiescence classes	-	Redéfinition classes et méthodes, MàJ tas	-
DynamOS	Attachement de gestionnaires	Comptage entrées et sorties des fonctions	Quiescence	Suspension totale	Redéfinition fonctions, MàJ données	Purge des caches
DynamicML	-	-	Passage du ramasse miettes	Suspension totale	MàJ des données	-
Ekiden	-	-	Point MaJ	Suspension totale	Sérialisation des données	-

Plate-forme	Reprise	Attente terminaison	Détection terminaison	Nettoyage	Nouvelle version
Thesus	-	MaJ données progressive	Fin d'ugage des fonctions	Nettoyage code obsolète	-
STUMP	Reprise des fils	-	-	-	Transformartion progressive des données
Local Dynamic ...	-	-	Fin d'utilisation composants	Nettoyage code obsolète	-
DUC	-	-	-	-	-
Swap & Play	Reprise des CPU	-	-	Nettoyage code obsolète	-
Supporting ...	Déblocage des requêtes	-	-	-	-
UpStare	Redémarrage programme	Guidage exécution	Point de MàJ atteint	-	-
TOS	Reprise des fils	-	-	-	-
Javadaptor	Reprise des fils	-	-	-	-
Javeleon	-	MaJ données progressive	Fin d'usage des données	Nettoyage code obsolète	-
Hotspot	-	-	-	-	-
EmbedDSU	Déblocage des appels	-	-	Nettoyage code obsolète	-
DynamOS	Déblocage interruptions	-	-	-	Mise à jour manuelle des variables
DynamicML	Reprise (fin passage ramasse miettes)	-	-	-	-
Ekiden	Démar. nouv. version, désérialisation des données	Guidage exécution	Point de MàJ atteint	-	-

Plate-forme	Préparation (1)	Attente Altérabilité	Détection Altérabilité	Suspension	Modification	Préparation (2)
DDSU	Insertion de *mediators*	-	-	Blocage + sérialisation requêtes	Installation nouveau service	-
Creol	-	-	-	-	MàJ des classes et méthodes	-
Argus	-	Annulation de requêtes	Quiescence composants	-	Erreur provoquée pour recharger les composants	-
Afpac	-	Recherche d'un futur point de MàJ	Point MaJ	-	MàJ	-
Freja	Envoi de la MàJ à tous les clients	Blocage classes vers la quiescence	Quiescence	-	MàJ classe	-
Proteus	-	-	Point MaJ valide	Suspension totale	Redéfition / ajout des types, fonctions et valeurs	-
TimeAdapt	Calcul ordonancement de la MàJ	Prise de verrous écriture du code	Tous les verrous sont pris	Suspension partielle	Exécution des tâches de reconfiguration	-
How to have ...	N/R	N/R	Point MaJ	N/R	Reconstruction de la pile	N/R
NSU	Création des nouveaux fils	-	Point MaJ + critères physiques	Suspension totale	Capture de l'état des fils	-
DSU for real-time systems	-	-	Entre deux cycles de l'application	-	Transfert d'état	-
Incremental Dynamic ...	-	-	Quiescence méthodes	Suspension totale	MàJ objets, remplacement des références	-
A Formal Model ...	-	Blocage requêtes entrantes	Plus de rquêtes en cours	-	Transfert d'état et modification des liens	-
LUCOS	fonc. actives → rempl. addr. retour par stub	-	Aucun fil exéc. 5 premiers octets fonc. modifée	Suspension totale	MàJ fonctions et données quiesc., prép. MàJ autres données	-
DUSC	-	-	Quiescence classes	Suspension totale	MàJ données	-
ReCaml	-	-	Une fonction retourne	Capture continuation	Exécution de la fonction de MàJ	-

Plate-forme	Reprise	Attente terminaison	Détection terminaison	Nettoyage	Nouvelle version
DDSU	Déblocage + déserialisation des requêtes	Guidage exécution	Fin d'usage des données	Suppr. *mediators* + service obsolète	-
Creol	-	MaJ données progressive	-	-	-
Argus	-	-	-	-	-
Afpac	-	-	-	-	-
Freja	Déblocage des classes	-	-	-	-
Proteus	Reprise des fils	-	-	-	Anciennes vers. fonc. terminent leurs appels
TimeAdapt	Verrous rendus	-	-	-	-
How to have ...	N/R	-	Fin d'ugage des fonctions	Passage à une version moins instrumentée	-
NSU	Transfert d'état vers nouveaux fils, démarrage nouveaux fils	-	-	Nettoyage code obsolète	-
DSU for real-time systems	-	-	-	-	-
Incremental Dynamic ...	Reprise des fils	-	-	-	-
A Formal Model ...	Déblocage des requêtes entrantes	-	-	-	-
LUCOS	Retour de hypercall	-	Fin d'ugage des fonctions	Nettoyage code obsolète	-
DUSC	Reprise des fils	-	Fin d'usage des données	-	-
ReCaml	(évaluation de la continuation capturée)	-	-	-	-

A.2 Architecture

		Altérabilité			Patch dynamique		
Plate-forme	Langage ciblé	Définit	Surveille	Évaluation	Script de contrôle	Nouveau code	Format fourni
Kitsune	C	Dév. Programme	-	Statique	Plate-forme + Dév. MàJ	Dév. MàJ	Code, *transformers*
ProteOS	OS (Minix)	Plate-forme + Dév MàJ	Processus	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Code, *transformers*, *state filters*
Ksplice	OS (Linux)	Plate-forme	Gestionnaire	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Code + code supplémentaire
OPUS	C	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code
K42	OS	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Nouveaux composants
Ginseng	C	Dév. Programme	Gestionnaire	Statique	Plate-forme	Dév. MàJ	Code
Jvolve	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code, *transformers*, critères d'alt.
Rubah	VM	Dév. Programme	-	Statique	Plate-forme + Dév. MàJ	Dév. MàJ	Code, classe de MàJ
ActiveContext	VM	-	-	-	Plate-forme	Dév. MàJ	Code, fonctions de transfert
Javelus	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code, fonctions de transfert
DCP	VM	-	-	-	Plate-forme	Dév. MàJ	Code, code init. nouv. champs
Polus	C	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code, mapping fonc./var., *callbacks*
Thesus	VM	-	-	-	Plate-forme	Dév. MàJ	Code, fonctions de transfert
STUMP	C	Dév. Programme	Gestionnaire	Statique	Plate-forme	Dév. MàJ	Code

	Granularité			Gestionnaire			
Plate-forme	Unité de MàJ	Multi. Vers	Cloisonnement des vers.	Nature	Portée	Durée de vie	Modifiable
Kitsune	Tout le code	Non	-	Entité externe	Globale	Permanente	Non
ProteOS	processus	Non	-	Entité externe	Globale	Permanente	Oui
Ksplice	Fonction	Non	-	Partie du programme	Globale	Permanente	Non
OPUS	Fonction	Non	-	Entité externe	Globale	Une MàJ	Non
K42	Composant	Oui	Composant	Partie du programme	Un composant	Une MàJ	Non
Ginseng	Fonction, Type	Non	Variable	Partie du programme	Globale	Permanente	Non
Jvolve	Classe, Méthode	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
Rubah	Tout le code	Non	Objet	Entité externe	Globale	Permanente	Non
ActiveContext	Contexte	Oui	Fil d'exécution	Plate-forme d'exécution	Globale	Permanente	Non
Javelus	Classe, Méthode	Non	Objet	Plate-forme d'exécution	Globale	Permanente	Non
DCP	Composant	Oui	Composant	Partie du programme	Globale	Permanente	Non
Polus	Fonction, variables globales	Oui	Variable	Entité externe	Globale	Une MàJ	Non
Thesus	Contexte	Oui	Fil d'exécution	Plate-forme d'exécution	Globale	Permanente	Non
STUMP	Fonction, Type	Non	Variable	Partie du programme	Globale	Permanente	Non

		Altérabilité			Patch dynamique		
Plate-forme	Langage ciblé	Définit	Surveille	Évaluation	Script de contrôle	Nouveau code	Format fourni
Local Dynamic ...	C	Gestionnaire	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Nouveaux composants
DUC	C	-	-	-	Plate-forme	-	-
Swap & Play	C	-	-	-	Plate-forme	Dév. MàJ	Code, *transformers*
Supporting ...	Autre	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code
UpStare	C	Plate-forme	-	Statique	Plate-forme	Dév. MàJ	Code
TOS	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code, critères d'alt.
Javadaptor	VM	-	-	-	Plate-forme	Dév. MàJ	Code, *transformers*
Javeleon	VM	-	-	-	Plate-forme	Dév. MàJ	Code, *transformers*
Hotspot	VM	-	-	-	Plate-forme	Dév. MàJ	Code
EmbedDSU	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code
DynamOS	OS (Linux)	Plate-forme	Gestionnaire	Dynamique	Dév. MàJ	Dév. MàJ	Code, script MàJ, code suppl.
DynamicML	Modèles	Plate-forme	Gestionnaire	Dynamique	Plate-forme	-	*transformers*
Ekiden	C	Dév. Programme	-	Statique	Plate-forme	Dév. MàJ	Code, sérialiseurs
DDSU	Autre	-	-	-	Plate-forme	-	-

	Granularité			Gestionnaire			
Plate-forme	Unité de MàJ	Multi. Vers	Cloisonnement des vers.	Nature	Portée	Durée de vie	Modifiable
Local Dynamic ...	Composant	Oui	Composant	Partie du programme	composant	Vie du composant	Non
DUC	Composant	Non	-	-	-	-	-
Swap & Play	Tout le code	Non	-	Partie du programme	Globale	Permanente	Non
Supporting ...	Service	Non	-	Partie de la plate-forme	Globale	Permanente	Non
UpStare	Tout le code	Non	-	Partie du programme	Globale	Permanente	Non
TOS	Classe, Méthode	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
Javadaptor	Classe	Non	-	Partie du programme	Globale	Une MàJ	Non
Javeleon	Classe	Oui	Objet	Partie du programme	Globale	Permanente	Non
Hotspot	Méthode, classe	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
EmbedDSU	Méthode, classe	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
DynamOS	Fonction, Type	Non	-	Partie du programme	Globale / Fonction	Permanente / Une MàJ	Non
DynamicML	Signature, Structure	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
Ekiden	Tout le code	Non	-	Partie du programme	Globale	Permanente	Non
DDSU	Un service	Oui	Service	Partie du programme	Un service	Une MàJ	Non

		Altérabilité			Patch dynamique		
Plate-forme	Langage ciblé	Définit	Surveille	Évaluation	Script de contrôle	Nouveau code	Format fourni
Creol	Modèles	Plate-forme	-	Dynamique	Plate-forme	-	-
Argus	C	Plate-forme	N/R	Dynamique	Plate-forme	Dév. MàJ	N/R
Afpac	C	Plate-forme + Dév . Programme	Gestionnaire	Dynamique	-	-	-
Freja	VM	Dév. MàJ	Gestionnaire	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Code, critères d'alt.
Proteus	C	Plate-forme	-	Statique	Plate-forme	Dév. MàJ	Code, *transformers*
TimeAdapt	Autre	Plate-forme	Ordonnanceur	Dynamique	Plate-forme	Dév. Programme	-
How to have ...	C	Dév. Programme	Gestionnaire	Statique	Plate-forme	Dév. MàJ	Code
NSU	C	Dév. Programme	-	Statique + Dynamique	N/R	Dév. MàJ	Code, critères d'alt.
DSU for real-time systems	C	Plate-forme	Ordonnanceur	Dynamique	Plate-forme	Dév. MàJ	Code
Incremental Dynamic ...	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code
A Formal Model ...	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	N/R
LUCOS	C	Plate-forme	Gestionnaire	Dynamique	Dév. MàJ	Dév. MàJ	Code, *transformers*, *callbacks*
DUSC	VM	Plate-forme	*Wrapper*	Dynamique	Plate-forme	Dév. MàJ	Bytecode nouv. vers.
ReCaml	C	Plate-forme	Gestionnaire	Dynamique	Dév. Programme	Dév. Programme	Fonction de MàJ

	Granularité			Gestionnaire			
Plate-forme	Unité de MàJ	Multi. Vers	Cloisonnement des vers.	Nature	Portée	Durée de vie	Modifiable
Creol	Attribut, Méthode	Non	Objet	-	-	-	-
Argus	Composant	Non	-	N/R	N/R	N/R	N/R
Afpac	Composant	Non	-	Partie du programme	Composant	Vie du composant	Non
Freja	Une classe	Non	-	Entité externe	Un serveur / Les *update clients*	Permanente	Non
Proteus	Fonction, Type, Variable	Non	-	Partie du programme	Globale	Permanente	Non
TimeAdapt	Composant	Non	-	Partie du programme	Globale	Permanente	Non
How to have ...	Tout le code	Non	-	Partie du programme	Globale	Permanente	N/R
NSU	Fil d'exécution	Oui	Fil d'exécution	N/R	N/R	N/R	N/R
DSU for real-time systems	Composant (processus)	Non	-	Partie du programme	Globale	Permanente	Oui
Incremental Dynamic ...	Classe	Non	-	Entité externe	Globale	Permanente	N/R
A Formal Model ...	Service	Non	-	Partie du programme	Service (*delegate*) + Globale (*update manager*)	Permanente	Non
LUCOS	Fonction	Non	Variable	Partie du programme	Globale	Permanente	Non
DUSC	Classe	Non	-	N/R	N/R	N/R	N/R
ReCaml	Pile d'exécution	Non	-	Plate-forme d'exécution	Globale	Permanente	Non

A.3 Mécanismes

	Altérabilité	Accès et Transformation des données		Patch dynamique		Gestion des Erreurs	
Plate-forme	Scrutation	Accès	Transformation	Chargement	Construction	Cohérence des Types	Récupération et Erreurs
Kitsune	-	Immédiat	Instantanée	Bibliothèque dynamique	Compilation spéciale	-	-
ProteOS	Introspection dynamique	Immédiat	Instantanée	Mécanisme de chargement de processus	Compilateur spécifique	-	Erreur application MàJ provoque retour en arrière
Ksplice	Parcours piles	-	-	Chargement de module noyeau	Compilateur générique (opt. imposées)	-	-
OPUS	Parcours piles	-	-	Bibliothèque dynamique	Compilation spéciale	Vérification effets de bord des fonctions	-
K42	Comptage d'appels	Immédiat	Hybride	Chargeur de module spécial	Compilation vers un format spécial	-	-
Ginseng	-	Progressif	Instantanée	Bibliothèque dynamique	Compilation spéciale	Fonction de vérification des variables	-
Jvolve	Introspection, points sûrs JVM	Immédiat	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Rubah	-	Hybride	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
ActiveContext	-	Progressif	Instantanée	chargement dynamique	Écriture de code mettant à jour	-	-
Javelus	Introspection, points sûrs JVM	Progressif	Hybride	Chargement dynamique des classes	Compilation spéciale	-	-
DCP	-	Progressif	Instantanée	Chargement dynamique des classes	Préprocesseur spécial	-	-
Polus	Parcours piles	Immédiat	Non	Bibliothèque dynamique	Compilation spéciale	-	*callback* Dév. MàJ récuprère d'un état contaminé
Thesus	-	Progressif	Instantanée	chargement dynamique	Écriture de code mettant à jour	-	-
STUMP	-	Progressif	Instantanée	Bibliothèque dynamique	Compilation spéciale	Fonction de vérification des variables	-

	Flot d'exécution				Modification des données		Gestionnaire
Plate-forme	Multi. Vers.	Synchro. des Vers.	MàJ des fonctions	Redémarrage	MàJ des Types	Transf. données	Contrôle exec.
Kitsune	-	-	Chargement de code	Oui	Chargement de code	Script Dév. MàJ	Oui (guidage/-redémarrage)
ProteOS	-	-	Chargement de code	Oui	Chargement de code	Transformer sur place	Non
Ksplice	-	-	Indirrection	Non	Script Dév. MàJ	Script Dév. MàJ	Non
OPUS	-	-	Indirrection	Non	-	-	Non
K42	Mise à jour parallèle	-	-	Non	Remplacement des factory	Indirrection	Oui (blacage de fils)
Ginseng	Mise à jour parallèle	-	Indirrection	Non	Chargement de code	Transformer sur place	Non
Jvolve	-	-	Indirrection	Non	Chargement de code	Transformer sur place	Non
Rubah	Mise à jour parallèle	-	Chargement de code	Oui	Chargement de code	Transformer sur place	Oui (guidage/-redémarrage)
ActiveContext	Contextes	fonctions de transfert	Chargement de code	Non	Chargement de code	Transformer sur place	Non
Javelus	Mise à jour parallèle	-	Réécriture forme compilée	Non	Chargement de code	Transformer sur place	Non
DCP	Architecture multi version	Proxy vers vers. récente, conversion à chaque accès	Indirrection	Non	Indirrection	Indirrection	Non
Polus	Mise à jour parallèle	Trap synchronisant les écritures	Indirrection	Non	Chargement de code	copie	Non
Thesus	Contextes	fonctions de transfert	Chargement de code	Non	Chargement de code	Transformer sur place	Non
STUMP	Mise à jour parallèle	-	Indirrection	Non	Chargement de code	Transformer sur place	Non

	Altérabilité	Accès et Transformation des données		Patch dynamique		Gestion des Erreurs	
Plate-forme	Scrutation	Accès	Transformation	Chargement	Construction	Cohérence des Types	Récupération et Erreurs
Local Dynamic ...	Maintien d'un graphe de flot de contrôle	-	-	-	-	-	-
DUC	-	-	-	-	-	-	-
Swap & Play	-	Immédiat	Instantanée	Mécanisme de chargement de code	Compilation spéciale	-	-
Supporting ...	Comptage d'appels	Immédiat	Instantanée	voir OSGi (implémentation)	voir OSGi (implémentation)	Vérif. types du service+ interfaçage d'adaptateur	Points de sauvegarde pour revenir en arrière
UpStare	-	Immédiat	Instantanée	Bibliothèque dynamique	Compilation spéciale	-	-
TOS	Introspection, points sûrs JVM	Immédiat	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Javadaptor	-	Immédiat	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Javeleon	-	Progressif	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Hotspot	-	-	-	Chargement dynamique des classes	Compilateur générique	-	-
EmbedDSU	Comptage d'appels entrants	Immédiat	Instantanée	Chargement dynamique du bytecode	Compilation spéciale	-	Erreur application MàJ provoque retour en arrière
DynamOS	Parcours piles, comptages entrées/sorties	Script MàJ	Script MàJ	Chargement de module noyeau	Compilateur générique	-	Possibilité d'annuler une mise à jour
DynamicML	-	Immédiat	Instantanée	-	-	-	-
Ekiden	-	Immédiat	Instantanée	Lancement du nouveau binaire	Compilation spéciale	-	-
DDSU	-	Immédiat	Instantanée	-	-	-	-

	Flot d'exécution				Modification des données		Gestionnaire
Plate-forme	Multi. Vers.	Synchro. des Vers.	MàJ des fonctions	Redémarrage	MàJ des Types	Transf. données	Contrôle exec.
Local Dynamic ...	Architecture multi version	-	-	Non	-	Transformer + copie	Non
DUC	-	-	Indirrection	Non	-	Transformer + copie	Non
Swap & Play	-	-	Chargement de code	Non	Chargement de code	Transformer sur place	Non
Supporting ...	-	-	Indirrection	Non	Chargement de code	Transformer + copie	Non
UpStare	-	-	Indirrection	Oui	Chargement de code	Transformer sur place	Oui (guidage/-redémarrage)
TOS	-	-	Indirrection	Non	Chargement de code	Transformer + copie	Non
Javadaptor	-	-	Indirrection	Non	Chargement de code	Transformer sur place	Non
Javeleon	Mise à jour parallèle	interception des écritures et lectures de champs	Chargement de code	Non	Chargement de code	Transformer sur place	Oui (choisit versions à appeler)
Hotspot	Non	-	Indirrection	Non	Chargement de code	-	Non
EmbedDSU	-	-	Réécriture forme compilée	Non	Modification du bytecode	copie	Oui (blacage de fils)
DynamOS	-	-	Indirrection	Non	Chargement de shadow type	copie	Oui (code adaptation Dév. MàJ)
DynamicML	-	-	-	Non	-	Transformer + copie	Non
Ekiden	-	-	Chargement de code	Oui	Chargement de code	Sérialisation	Oui (guidage/-redémarrage)
DDSU	Architecture multi version	Mediator synchronise l'état	-	Non	-	-	Non

	Altérabilité	Accès et Transformation des données		Patch dynamique		Gestion des Erreurs	
Plate-forme	Scrutation	Accès	Transformation	Chargement	Construction	Cohérence des Types	Récupération et Erreurs
Creol	-	Progressif	Instantanée	-	-	-	-
Argus	N/R	N/R	N/R	N/R	N/R	-	-
Afpac	Calcul du futur de l'exécution	-	-	-	-	-	-
Freja	Introspection à base de *hooks*	Progressif	Instantanée	Chargement dynamique des classes	N/R	-	-
Proteus	-	Immédiat	Retardée	N/R	N/R	Analyse statique	-
TimeAdapt	Ordonnancement	-	-	-	-	-	-
How to have ...	-	-	-	-	-	-	-
NSU	Calcul des critères Dev. MàJ	-	-	Bibliothèque dynamique	N/R	-	-
DSU for real-time systems	Ordonnancement	Immédiat	Instantanée	chargement dynamique	Compilateur générique	-	-
Incremental Dynamic ...	Parcours piles	Immédiat	Instantanée	Chargement DIFF	Calcul DIFF après compil. standard	-	-
A Formal Model ...	Comptage des requêtes entrantes	Immédiat	Instantanée	N/R	N/R	Vérif. statique (moins de dép., plus de service)	-
LUCOS	Parcours piles	Hybride	Instantanée	chargement module noyeau	Compilation générique + prelink	-	Mécanisme de rollback
DUSC	Comptage des méthodes actives	Immédiat	Instantanée	Chargement dynamique des classes	Calcul DIFF après compil. standard	Pas de modification de l'interface publique	-
ReCaml	-	-	-	Inclus dans l'application	Compilation spéciale	Typage fort du compilateur	-

	Flot d'exécution				Modification des données		Gestionnaire
Plate-forme	Multi. Vers.	Synchro. des Vers.	MàJ des fonctions	Redémarrage	MàJ des Types	Transf. données	Contrôle exec.
Creol	Mise à jour parallèle	-	-	Non	-	-	Non
Argus	-	-	Chargement de code	Non	Chargement de code	N/R	Oui (annulation requêtes)
Afpac	-	-	-	Non	-	-	Non
Freja	Mise à jour parallèle	-	Chargement de code	Non	Indirrection	Transformer + copie	Oui (blacage de requêtes)
Proteus	Mise à jour parallèle	-	Indirrection	Non	Chargement de code	Indirrection	Non
TimeAdapt	-	-	Ajout / suppression de composants	-	-	Pas de transfert d'état (reconf.uniquement)	Non
How to have ...	-	-	Chargement de code	Oui	-	-	Non
NSU	Architecture multi version	Interpolation des résultats	Chargement de code	Oui	-	-	Non
DSU for real-time systems	-	-	Chargement de code	Non	Script Dév. MàJ	Script Dév. MàJ	Non
Incremental Dynamic ...	-	-	Réécriture forme compilée	Non	Réécriture forme compilée	copie	Non
A Formal Model ...	-	-	Indirrection	Non	Chargement de code	Sérialisation	Non
LUCOS	Mise à jour parallèle	fonctions de transfert	Indirrection	Non	Chargement de code	Transformer sur place	Exécution pas à pas lors de la MàJ des données
DUSC	-	-	Mise à jour de la classe	Non	Indirrection	Transformer + copie	Non
ReCaml	-	-	-	-	-	-	Non

A.4 Base de données homogénéisée

A.4.1 Cycle de vie

Plate-forme	Préparation (1)	Attente Altérabilité	Détection Altérabilité	Suspension	Modification	Préparation (2)
Kitsune	-	-	Point MaJ	Suspension totale	MàJ tas	-
ProteOS	envoi des state filters au processus	-	Début de boucle + state filters	Suspension partielle	création nouv. proc. , transfert de l'état	Vérification de cohérence de l'état transféré
Ksplice	-	-	Quiescence	Suspension totale	Redef. Fonctions, MàJ donneés	-
OPUS	Attachement Gestionnaire	-	Quiescence	Suspension totale	Redef. Fonctions	-
K42	Attachement Gestionnaire	Guidage Quiescence	Quiescence	-	Remplacement du composant	-
Ginseng	-	-	Point MaJ	Suspension totale	Redef. Fonctions, MàJ donneés	-
Jvolve	-	-	Quiescence	Suspension totale	Redef classes et méthodes, MàJ tas	-
Rubah	-	-	Point MaJ	Suspension totale	MàJ tas (ou début)	-
ActiveContext	-	-	-	-	Changement du contexte principal	-
Javelus	-	-	Quiescence	Suspension totale	Mise à jour des classes	-
DCP	-	-	-	-	Chargement des composant et instalation des proxys	-
Polus	callback dév MàJ, init. nouvelles variables	-	Quiescence	Suspension totale	MàJ fonctions, instal. *traps* (MàJ données)	-
Thesus	-	-	-	-	Changement du contexte principal	-
STUMP	-	-	Point MaJ	Suspension totale	Redef. Fonctions, MàJ donneés	-

Plate-forme	Reprise	Attente terminaison	Détection terminaison	Nettoyage	Nouvelle version
Kitsune	Redémarrage fils	Guidage exécution	Point de MàJ atteint	-	-
ProteOS	arrêt du vieux proc. , Démarrage nouveau proc.	-	-	-	-
Ksplice	reprise	-	-	-	-
OPUS	Reprise des fils	-	-	Détachement Gestionnaire	-
K42	Déblocage des fils	-	-	Détachement Gestionnaire	-
Ginseng	Reprise des fils	-	-	-	Transformartion progressive des données
Jvolve	Reprise des fils	-	-	-	-
Rubah	Redémarrage fils	Guidage exécution	Point de MàJ atteint	-	Transformartion progressive des données
ActiveContext	-	MaJ données progressive	Fin d'ugage des fonctions	Nettoyage code obsolète	-
Javelus	Reprise des fils	MaJ données progressive	Fin d'usage des données	Nettoyage code obsolète	-
DCP	-	-	-	-	Plusieurs version coexistent (MàJ pour les suppr.)
Polus	Reprise des fils	MaJ données progressive	Fin d'ugage des fonctions	Callback postupdate + suppression des traps de synchro	-
Thesus	-	MaJ données progressive	Fin d'ugage des fonctions	Nettoyage code obsolète	-
STUMP	Reprise des fils	-	-	-	Transformartion progressive des données

Plate-forme	Préparation (1)	Attente Altérabilité	Détection Altérabilité	Suspension	Modification	Préparation (2)
Local Dynamic ...	-	-	-	-	Remplacement de composant	-
DUC	-	-	-	Suspension partielle	Remplacement du composant	désérialisation des connections
Swap & Play	Copie de l'état depuis l'ancienne version	-	-	Suspension partielle	Transfert de l'exécution et mise à jour de l'état	-
Supporting ...	-	-	Quiescence	Suspension partielle	Remplacement du service	-
UpStare	-	-	Point MaJ	Suspension totale	MàJ fonctions, var. globales, capture des piles	-
TOS	-	-	Quiescence	Suspension totale	Redef classes et méthodes, MàJ tas	-
Javadaptor	-	-	-	Suspension totale	Redéf. méthodes, instal. conteneurs et proxy	-
Javeleon	-	-	-	-	Copie des données et mie à jour, début de la synchronisation	-
Hotspot	-	-	-	-	Mise à jour des méthodes	-
EmbedDSU	-	Guidage Quiescence	Quiescence	-	Redef classes et méthodes, MàJ tas	-
DynamOS	Attachement Gestionnaire	Comptage des entrées et sorties des fonctions	Quiescence	Suspension totale	MàJ des fonctions, MàJ des données	Purge des caches
DynamicML	-	-	Passage du ramasse miettes	Suspension totale	MàJ des données	-
Ekiden	-	-	Point MaJ	Suspension totale	Sérialisation des données,	-
DDSU	Attachement Gestionnaire	-	-	Suspension partielle	Installation du nouveau service	-

Plate-forme	Reprise	Attente terminaison	Détection terminaison	Nettoyage	Nouvelle version
Local Dynamic ...	-	-	Fin d'ugage des fonctions	Nettoyage code obsolète	-
DUC	-	-	-	-	-
Swap & Play	reprise des CPU	-	-	Nettoyage code obsolète	-
Supporting ...	déblocage des requêtes	-	-	-	-
UpStare	Redémarrage de l'exécution	Guidage exécution	Point de MàJ atteint	-	-
TOS	Reprise des fils	-	-	-	-
Javadaptor	Reprise des fils	-	-	-	-
Javeleon	-	MaJ données progressive	Fin d'usage des données	Nettoyage code obsolète	-
Hotspot	-	-	-	-	-
EmbedDSU	Déblocage des appels	-	-	Nettoyage code obsolète	-
DynamOS	Déblocage des interruptions	-	-	-	Mise à jour manuelle des variables
DynamicML	Reprise (fin du passage du ramasse miettes)	-	-	-	-
Ekiden	Démarrage de la nouvelle version, déserialisation des données	Guidage exécution	Point de MàJ atteint	-	-
DDSU	Déblocage et déserialisation des requêtes	Guidage exécution	Fin d'usage des données	Détachement Gestionnaire	-

Plate-forme	Préparation (1)	Attente Altérabilité	Détection Altérabilité	Suspension	Modification	Préparation (2)
Creol	-	-	-	-	Mise à jour des classes et méthodes	-
Argus	-	Guidage Quiescence	Quiescence	-	Erreur provoquée pour recharger les composants	-
Afpac	-	Recherche d'un futur point de MàJ	Point MaJ	-	MàJ	-
Freja	Envoi de la MàJ à tous les clients	Guidage Quiescence	Quiescence	-	Mise à jour de la classe	-
Proteus	-	-	Point MaJ	Suspension totale	Remplacement et ajout des types, fonctions et valeurs	-
TimeAdapt	calcul ordonancement de la MàJ	prise des verrous écriture du code	tous les verrous sont pris	Suspension partielle	exécution des tâches de reconfiguration	-
How to have ...	N/R	N/R	Point MaJ	N/R	Reconstruction de la pile	N/R
NSU	Création des nouveaux fils	-	Point MaJ	Suspension totale	capture de l'état des fils	-
DSU for real-time systems	-	-	Entre deux cycles de l'application	-	Transfert d'état	-
Incremental Dynamic ...	-	-	Quiescence	Suspension totale	MàJ des objets et remplacement des références	-
A Formal Model ...	-	Guidage Quiescence	-	-	Transfert d'état et modification des liens	-
LUCOS	fonc. actives → rempl. addr. retour par stub	-	Aucun fil exéc. 5 premiers octets fonc. modifée	Suspension totale	MàJ fonctions et données quiesc., prép. MàJ autres données	-
DUSC	-	-	Quiescence	Suspension totale	MàJ données	-
ReCaml	-	-	Une fonction retourne	Suspension totale	exécution de la fonction de MàJ	-

Plate-forme	Reprise	Attente terminaison	Détection terminaison	Nettoyage	Nouvelle version
Creol	-	MaJ données progressive	-	-	-
Argus	-	-	-	-	-
Afpac	-	-	-	-	-
Freja	Déblocage des instances et des classes	-	-	-	-
Proteus	Reprise des fils	-	-	-	Anciennes vers. fonc. terminent leurs appels
TimeAdapt	verrous rendus	-	-	-	-
How to have ...	N/R	-	Fin d'ugage des fonctions	passage à une version moins instrumentée	-
NSU	Transfert d'état vers nouveaux fils, démarrage nouveaux fils	-	-	Nettoyage code obsolète	-
DSU for real-time systems	-	-	-	-	-
Incremental Dynamic ...	Reprise des fils	-	-	-	-
A Formal Model ...	Déblocage des requêtes entrantes	-	-	-	-
LUCOS	retour de hypercall	-	Fin d'ugage des fonctions	Nettoyage code obsolète	-
DUSC	Reprise des fils	-	Fin d'usage des données	-	-
ReCaml	(évaluation de la continuation capturée)	-	-	-	-

A.4.2 Architecture

		Altérabilité			Patch dynamique		
Plate-forme	Langage ciblé	Définit	Surveille	Évaluation	Script de contrôle	Nouveau code	Format fourni
Kitsune	C	Dév. Programme	-	Statique	Plate-forme + Dév. MàJ	Dév. MàJ	Code (+ transformers)
ProteOS	OS (Minix)	Plate-forme + Dév MàJ	Élément du programme	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Code (+ transformers)
Ksplice	OS (Linux)	Plate-forme	Gestionnaire	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Code + autres instructions
OPUS	C	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
K42	OS	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
Ginseng	C	Dév. Programme	Gestionnaire	Statique	Plate-forme	Dév. MàJ	Code (+ transformers)
Jvolve	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
Rubah	VM	Dév. Programme	-	Statique	Plate-forme + Dév. MàJ	Dév. MàJ	Code (+ transformers)
ActiveContext	VM	-	-	-	Plate-forme	Dév. MàJ	Code (+ transformers)
Javelus	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
DCP	VM	-	-	-	Plate-forme	Dév. MàJ	Code (+ transformers)
Polus	C	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
Thesus	VM	-	-	-	Plate-forme	Dév. MàJ	Code (+ transformers)
STUMP	C	Dév. Programme	Gestionnaire	Statique	Plate-forme	Dév. MàJ	Code (+ transformers)

	Granularité			Gestionnaire			
Plate-forme	Unité de MàJ	Multi. Vers	Cloisonnement des vers.	Nature	Portée	Durée de vie	Modifiable
Kitsune	Tout le code	Non	-	Entité externe	Globale	Permanente	Non
ProteOS	Intermédiaire	Non	-	Entité externe	Globale	Permanente	Oui
Ksplice	Faible grain	Non	-	Partie du programme	Globale	Permanente	Non
OPUS	Faible grain	Non	-	Entité externe	Globale	Une MàJ	Non
K42	Intermédiaire	Oui	Composant	Partie du programme	Un composant	Une MàJ	Non
Ginseng	Faible grain	Non	Variable	Partie du programme	Globale	Permanente	Non
Jvolve	Intermédiaire	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
Rubah	Tout le code	Non		Entité externe	Globale	Permanente	Non
ActiveContext	Intermédiaire	Oui	Fil d'exécution	Plate-forme d'exécution	Globale	Permanente	Non
Javelus	Intermédiaire	Non		Plate-forme d'exécution	Globale	Permanente	Non
DCP	Intermédiaire	Oui	Composant	Partie du programme	Globale	Permanente	Non
Polus	Faible grain	Oui	Variable	Entité externe	Globale	Une MàJ	Non
Thesus	Intermédiaire	Oui	Fil d'exécution	Plate-forme d'exécution	Globale	Permanente	Non
STUMP	Faible grain	Non	Variable	Partie du programme	Globale	Permanente	Non

		Altérabilité			Patch dynamique		
Plate-forme	Langage ciblé	Définit	Surveille	Évaluation	Script de contrôle	Nouveau code	Format fourni
Local Dynamic ...	C	Gestionnaire	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
DUC	C	-	-	-	Plate-forme	-	-
Swap & Play	C	-	-	-	Plate-forme	Dév. MàJ	Code (+ transformers)
Supporting ...	Autre	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
UpStare	C	Plate-forme	-	Statique	Plate-forme	Dév. MàJ	Code (+ transformers)
TOS	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
Javadaptor	VM	-	-	-	Plate-forme	Dév. MàJ	Code (+ transformers)
Javeleon	VM	-	-	-	Plate-forme	Dév. MàJ	Code (+ transformers)
Hotspot	VM	-	-	-	Plate-forme	Dév. MàJ	Code (+ transformers)
EmbedDSU	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
DynamOS	OS (Linux)	Plate-forme	Gestionnaire	Dynamique	Dév. MàJ	Dév. MàJ	Code (+ transformers)
DynamicML	Modèles	Plate-forme	Gestionnaire	Dynamique	Plate-forme	-	Code (+ transformers)
Ekiden	C	Dév. Programme	-	Statique	Plate-forme	Dév. MàJ	Code (+ transformers)
DDSU	Autre	-	-	-	Plate-forme	-	-

	Granularité			Gestionnaire			
Plate-forme	Unité de MàJ	Multi. Vers	Cloisonnement des vers.	Nature	Portée	Durée de vie	Modifiable
Local Dynamic ...	Intermédiaire	Oui	Composant	Partie du programme	composant	Vie du composant	Non
DUC	Intermédiaire	Non	-	-	-	-	-
Swap & Play	Tout le code	Non	-	Partie du programme	Globale	Permanente	Non
Supporting ...	Service	Non	-	Partie de la plate-forme	Globale	Permanente	Non
UpStare	Tout le code	Non	-	Partie du programme	Globale	Permanente	Non
TOS	Intermédiaire	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
Javadaptor	Intermédiaire	Non	-	Partie du programme	Globale	Une MàJ	Non
Javeleon	Intermédiaire	Oui		Partie du programme	Globale	Permanente	Non
Hotspot	Faible grain	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
EmbedDSU	Faible grain	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
DynamOS	Faible grain	Non	-	Partie du programme	Globale / Une fonction	Permanente / Une MàJ	Non
DynamicML	Intermédiaire	Non	-	Plate-forme d'exécution	Globale	Permanente	Non
Ekiden	Tout le code	Non	-	Partie du programme	Globale	Permanente	Non
DDSU	Un service	Oui	Service	Partie du programme	Un service	Une MàJ	Non

		Altérabilité			Patch dynamique		
Plate-forme	Langage ciblé	Définit	Surveille	Évaluation	Script de contrôle	Nouveau code	Format fourni
Creol	Modèles	Plate-forme	-	Dynamique	Plate-forme	-	-
Argus	C	Plate-forme	N/R	Dynamique	Plate-forme	Dév. MàJ	N/R
Afpac	C	Plate-forme + Dév . Programme	Gestionnaire	Dynamique	-	-	-
Freja	VM	Dév. MàJ	Gestionnaire	Dynamique	Plate-forme + Dév. MàJ	Dév. MàJ	Code (+ transformers)
Proteus	C	Plate-forme	-	Statique	Plate-forme	Dév. MàJ	Code (+ transformers)
TimeAdapt	Autre	Plate-forme	Ordonnanceur	Dynamique	Plate-forme	Dév. Programme	-
How to have ...	C	Dév. Programme	Gestionnaire	Statique	Plate-forme	Dév. MàJ	Code (+ transformers)
NSU	C	Dév. Programme	-	Statique + Dynamique	N/R	Dév. MàJ	Code (+ transformers)
DSU for real-time systems	C	Plate-forme	Ordonnanceur	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
Incremental Dynamic ...	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
A Formal Model ...	VM	Plate-forme	Gestionnaire	Dynamique	Plate-forme	Dév. MàJ	N/R
LUCOS	C	Plate-forme	Gestionnaire	Dynamique	Dév. MàJ	Dév. MàJ	Code (+ transformers)
DUSC	VM	Plate-forme	Élément du programme	Dynamique	Plate-forme	Dév. MàJ	Code (+ transformers)
ReCaml	C	Plate-forme	Gestionnaire	Dynamique	Dév. Programme	Dév. Programme	Code + autres instructions

	Granularité			Gestionnaire			
Plate-forme	Unité de MàJ	Multi. Vers	Cloisonnement des vers.	Nature	Portée	Durée de vie	Modifiable
Creol	Faible grain	Non		-	-	-	-
Argus	Intermédiaire	Non	-	N/R	N/R	N/R	N/R
Afpac	Intermédiaire	Non	-	Partie du programme	Composant	Vie du composant	Non
Freja	Intermédiaire	Non	-	Entité externe	Un serveur / Les *update client*	Permanente	Non
Proteus	Faible grain	Non	-	Partie du programme	Globale	Permanente	Non
TimeAdapt	Intermédiaire	Non	-	Partie du programme	Globale	Permanente	Non
How to have ...	Tout le code	Non	-	Partie du programme	Globale	Permanente	N/R
NSU	Intermédiaire	Oui	Fil d'exécution	N/R	N/R	N/R	N/R
DSU for real-time systems	Intermédiaire	Non	-	Partie du programme	Globale	Permanente	Oui
Incremental Dynamic ...	Intermédiaire	Non	-	Entité externe	Globale	Permanente	N/R
A Formal Model ...	Service	Non	-	Partie du programme	Service (delegate) + Globale (Update Manager)	Permanente	Non
LUCOS	Faible grain	Non	variable	Partie du programme	Globale	Permanente	Non
DUSC	Intermédiaire	Non	-	N/R	N/R	N/R	N/R
ReCaml	Intermédiaire	Non	-	Plate-forme d'exécution	Globale	Permanente	Non

A.4.3 Méanismes

	Altérabilité	Accès et Transformation des données		Patch dynamique		Gestion des Erreurs	
Plate-forme	Scrutation	Accès	Transformation	Chargement	Construction	Cohérence des Types	Récupération et Erreurs
Kitsune	-	Immédiat	Instantanée	Bibliothèque dynamique	Compilation spéciale	-	-
ProteOS	Introspection	Immédiat	Instantanée	Mécanisme de chargement de processus	Compilation spéciale	-	Erreur application MàJ provoque retour en arrière
Ksplice	Introspection	-	-	Chargement de module noyeau	Compilateur générique	-	-
OPUS	Introspection	-	-	Bibliothèque dynamique	Compilation spéciale	Vérification des effets de bord des fonctions	-
K42	Comptage d'appels	Immédiat	Hybride	Chargeur de module spécial	Compilation spéciale	-	-
Ginseng	-	Progressif	Instantanée	Bibliothèque dynamique	Compilation spéciale	Fonction de vérification des variables	-
Jvolve	Introspection	Immédiat	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Rubah	-	Hybride	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
ActiveContext	-	Progressif	Instantanée	chargement dynamique	Écriture de code mettant à jour	-	-
Javelus	Introspection	Progressif	Hybride	Chargement dynamique des classes	Compilation spéciale	-	-
DCP	-	Progressif	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Polus	Introspection	Immédiat	Non	Bibliothèque dynamique	Compilation spéciale	-	*callback* Dév. MàJ récuprère d'un état contaminé
Thesus	-	Progressif	Instantanée	chargement dynamique	Écriture de code mettant à jour	-	-
STUMP	-	Progressif	Instantanée	Bibliothèque dynamique	Compilation spéciale	Fonction de vérification des variables	-

	Flot d'exécution				Modification des données		Gestionnaire
Plate-forme	Multi. Vers.	Synchro. des Vers.	MàJ des fonctions	Redémarrage	MàJ des Types	Transf. données	Contrôle exec.
Kitsune	-	-	Chargement de code	Oui	Chargement de code	Script Dév. MàJ	Oui
ProteOS	-	-	Chargement de code	Oui	Chargement de code	Transformer sur place	Non
Ksplice	-	-	Indirrection	Non	Script Dév. MàJ	Script Dév. MàJ	Non
OPUS	-	-	Indirrection	Non	-	-	Non
K42	Mise à jour parallèle	-	-	Non	Remplacement des factory	Indirrection	Oui
Ginseng	Mise à jour parallèle	-	Indirrection	Non	Chargement de code	Transformer sur place	Non
Jvolve	-	-	Indirrection	Non	Chargement de code	Transformer sur place	Non
Rubah	Mise à jour parallèle	-	Chargement de code	Oui	Chargement de code	Transformer sur place	Oui
ActiveContext	Contextes	fonctions de transfert	Chargement de code	Non	Chargement de code	Transformer sur place	Non
Javelus	Mise à jour parallèle	-	Réécriture forme compilée	Non	Chargement de code	Transformer sur place	Non
DCP	Architecture multi version	Proxy vers vers. récente, conversion à chaque accès	Indirrection	Non	Indirrection	Indirrection	Non
Polus	Mise à jour parallèle	Trap synchronisant les écritures	Indirrection	Non	Chargement de code	copie	Non
Thesus	Contextes	fonctions de transfert	Chargement de code	Non	Chargement de code	Transformer sur place	Non
STUMP	Mise à jour parallèle	-	Indirrection	Non	Chargement de code	Transformer sur place	Non

	Altérabilité	Accès et Transformation des données		Patch dynamique		Gestion des Erreurs	
Plate-forme	Scrutation	Accès	Transformation	Chargement	Construction	Cohérence des Types	Récupération et Erreurs
Local Dynamic ...	Maintien d'un graphe de flot de contrôle	-	-	-	-	-	-
DUC	-	-	-	-	-	-	-
Swap & Play	-	Immédiat	Instantanée	Mécanisme de chargement de code	Compilation spéciale	-	-
Supporting ...	Comptage d'appels	Immédiat	Instantanée	voir OSGi (implémentation)	voir OSGi (implémentation)	Vérif. types du service+ interfaçage d'adaptateur	Points de sauvegarde pour revenir en arrière
UpStare	-	Immédiat	Instantanée	Bibliothèque dynamique	Compilation spéciale	-	-
TOS	Introspection	Immédiat	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Javadaptor	-	Immédiat	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Javeleon	-	Progressif	Instantanée	Chargement dynamique des classes	Compilation spéciale	-	-
Hotspot	-	-	-	Chargement dynamique des classes	Compilateur générique	-	-
EmbedDSU	Comptage d'appels	Immédiat	Instantanée	Chargement dynamique du bytecode	Compilation spéciale	-	Erreur application MàJ provoque retour en arrière
DynamOS	Introspection	Script MàJ	Script MàJ	Chargement de module noyeau	Compilateur générique	-	Possibilité d'annuler une mise à jour
DynamicML	-	Immédiat	Instantanée	-	-	-	-
Ekiden	-	Immédiat	Instantanée	Lancement du nouveau binaire	Compilation spéciale	-	-
DDSU	-	Immédiat	Instantanée	-	-	-	-

	Flot d'exécution				Modification des données		Gestionnaire
Plate-forme	Multi. Vers.	Synchro. des Vers.	MàJ des fonctions	Redémarrage	MàJ des Types	Transf. données	Contrôle exec.
Local Dynamic ...	Architecture multi version	-	-	Non	-	Transformer + copie	Non
DUC	-	-	Indirrection	Non	-	Transformer + copie	Non
Swap & Play	-	-	Chargement de code	Non	Chargement de code	Transformer sur place	Non
Supporting ...	-	-	Indirrection	Non	Chargement de code	Transformer + copie	Non
UpStare	-	-	Indirrection	Oui	Chargement de code	Transformer sur place	Oui
TOS	-	-	Indirrection	Non	Chargement de code	Transformer + copie	Non
Javadaptor	-	-	Indirrection	Non	Chargement de code	Transformer sur place	Non
Javeleon	Mise à jour parallèle	interception des écritures et lectures de champs	Chargement de code	Non	Chargement de code	Transformer sur place	Oui
Hotspot	Non	-	Indirrection	Non	Chargement de code	-	Non
EmbedDSU	-	-	Réécriture forme compilée	Non	Modification du bytecode	copie	Oui
DynamOS	-	-	Indirrection	Non	Chargement de shadow type	copie	Oui
DynamicML	-	-	-	Non	-	Transformer + copie	Non
Ekiden	-	-	Chargement de code	Oui	Chargement de code	Sérialisation	Oui
DDSU	Architecture multi version	Mediator synchronise l'état	-	Non	-	-	Non

	Altérabilité	Accès et Transformation des données		Patch dynamique		Gestion des Erreurs	
Plate-forme	Scrutation	Accès	Transformation	Chargement	Construction	Cohérence des Types	Récupération et Erreurs
Creol	-	Progressif	Instantanée	-	-	-	-
Argus	N/R	N/R	N/R	N/R	N/R	-	-
Afpac	Calcul du futur de l'exécution	-	-	-	-	-	-
Freja	Introspection	Progressif	Instantanée	Chargement dynamique des classes	N/R	-	-
Proteus	-	Immédiat	Retardée	N/R	N/R	Analyse statique	-
TimeAdapt	ordonnancement	-	-	-	-	-	-
How to have ...	-	-	-	-	-	-	-
NSU	Calcul des critères Dev. MàJ	-	-	Bibliothèque dynamique	N/R	-	-
DSU for real-time systems	ordonnancement	Immédiat	Instantanée	chargement dynamique	Compilateur générique	-	-
Incremental Dynamic ...	Introspection	Immédiat	Instantanée	Chargement DIFF	Compilation spéciale	-	-
A Formal Model ...	Comptage d'appels	Immédiat	Instantanée	N/R	N/R	Vérif. statique (moins de dép., plus de service)	-
LUCOS	Introspection	Hybride	Instantanée	chargement module noyeau	Compilation spéciale	-	Mécanisme de rollback
DUSC	Comptage d'appels	Immédiat	Instantanée	Chargement dynamique des classes	Compilation spéciale	Pas de modification de l'interface publique	-
ReCaml	-	-	-	Inclus dans l'application	Compilation spéciale	Typage fort du compilateur	-

	Flot d'exécution				Modification des données		Gestionnaire
Plate-forme	Multi. Vers.	Synchro. des Vers.	MàJ des fonctions	Redémarrage	MàJ des Types	Transf. données	Contrôle exec.
Creol	Mise à jour parallèle	-	-	Non	-	-	Non
Argus	-	-	Chargement de code	Non	Chargement de code	N/R	Oui
Afpac	-	-	-	Non	-	-	Non
Freja	Mise à jour parallèle	-	Chargement de code	Non	Indirrection	Transformer + copie	Oui
Proteus	Mise à jour parallèle	-	Indirrection	Non	Chargement de code	Indirrection	Non
TimeAdapt	-	-	Ajout / suppression de composants	-	-	Pas de transfert d'état (reconf.uniquement)	Non
How to have ...	-	-	Chargement de code	Oui	-	-	Non
NSU	Architecture multi version	Interpolation des résultats	Chargement de code	Oui	-	-	Non
DSU for real-time systems	-	-	Chargement de code	Non	Script Dév. MàJ	Script Dév. MàJ	Non
Incremental Dynamic ...	-	-	Réécriture forme compilée	Non	Réécriture forme compilée	copie	Non
A Formal Model ...	-	-	Indirrection	Non	Chargement de code	Sérialisation	Non
LUCOS	Mise à jour parallèle	fonctions de transfert	Indirrection	Non	Chargement de code	Transformer sur place	Exécution pas à pas lors de la MàJ des données
DUSC	-	-	Mise à jour de la classe	Non	Indirrection	Transformer + copie	Non
ReCaml	-	-	-	-	-	-	Non

Annexe B

Modèles et étude de mécanismes

B.1 Recherche des cellules

$\boxed{\mathbf{getCells}\ n\ \mathcal{P} = c_1\ ..\ c_l}$

$$\frac{}{\mathbf{getCells}\ n\ =}\ \textsc{GCemp} \qquad \frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l \quad \mathbf{getCells}\ n\ \mathcal{P} = c'_1\ ..\ c'_k}{\mathbf{getCells}\ n\ a \underset{\mathbb{S}}{\rhd} t \parallel \mathcal{P} = c_1\ ..\ c_l\ c'_1\ ..\ c'_k}\ \textsc{GCth1}$$

$$\frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l \quad \mathbf{getCells}\ n\ \mathcal{P} = c'_1\ ..\ c'_k}{\mathbf{getCells}\ n\ a \underset{\mathbb{S}}{\blacktriangleright} t \parallel \mathcal{P} = c_1\ ..\ c_l\ c'_1\ ..\ c'_k}\ \textsc{GCth2}$$

$\boxed{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l}$

$$\frac{}{\mathbf{getCells}\ n\ c^n = c}\ \textsc{GCcell1} \qquad \frac{n_1 \neq n_2}{\mathbf{getCells}\ n_1\ c^{n_2} =}\ \textsc{GCcell2} \qquad \frac{}{\mathbf{getCells}\ n\ i =}\ \textsc{GCint}$$

$$\frac{}{\mathbf{getCells}\ n\ x =}\ \textsc{GCvar} \qquad \frac{}{\mathbf{getCells}\ n\ a =}\ \textsc{GCtid} \qquad \frac{}{\mathbf{getCells}\ n_1\ n_2 =}\ \textsc{GCnam}$$

$$\frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l}{\mathbf{getCells}\ n\ \lambda x.t = c_1\ ..\ c_l}\ \textsc{GClam} \qquad \frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l}{\mathbf{getCells}\ n_1\ n_2 : \lambda x.t = c_1\ ..\ c_l}\ \textsc{GCnlam}$$

$$\frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l \quad \mathbf{getCells}\ n\ t' = c'_1\ ..\ c'_k}{\mathbf{getCells}\ n\ t\ t' = c_1\ ..\ c_l\ c'_1\ ..\ c'_k}\ \textsc{GCapp} \qquad \frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l}{\mathbf{getCells}\ n\ \texttt{spawn}\ t = c_1\ ..\ c_l}\ \textsc{GCspa}$$

$$\frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l}{\mathbf{getCells}\ n\ \texttt{return}\ t = c_1\ ..\ c_l}\ \textsc{GCret} \qquad \frac{\mathbf{getCells}\ n\ t = c_1\ ..\ c_l \quad \mathbf{getCells}\ n\ t' = c'_1\ ..\ c'_k}{\mathbf{getCells}\ n\ \texttt{def}\ n = t\ \texttt{in}\ t' = c_1\ ..\ c_l\ c'_1\ ..\ c'_k}\ \textsc{GCdef}$$

B.2 Sémantique complète

B.2.1 Variables

x
c
a
n
i
L
s
τ
k, l, j

B.2.2 Grammaire

$\mathcal{P}$ ::=
- $\mathcal{T}_1 \parallel .. \parallel \mathcal{T}_k$

C ::=
- $\mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel [\cdot] \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k$

$\mathcal{T}$ ::=
- $a \underset{t_0,\mathbb{S}}{\triangleright} t$
- $a \underset{t_0,\mathbb{S}}{\blacktriangleright} t$

t ::=

i	
x	
$\lambda x.t$	bind x in t
$t\ t'$	
(t)	S
$\texttt{spawn}\ t$	
$\texttt{breakpoint}\ t$	
a	
n	
$n : \lambda x.t$	bind x in t
$\texttt{return}\ t$	
$\texttt{def}\ n = t\ \texttt{in}\ t'$	bind n in t bind n in t'
$\texttt{def}\ \tau : n = t\ \texttt{in}\ t'$	bind n in t bind n in t'
c^n	
c^n_τ	
$L : t$	

CT ::=
- $[\cdot]t'$
- $v[\cdot]$
- $(n : \lambda x.t)[\cdot]$
- $\texttt{return}\ [\cdot]$
- $\texttt{def}\ n = [\cdot]\ \texttt{in}\ t'$
- $\texttt{def}\ n = v\ \texttt{in}\ [\cdot]$

v ::=
- i
- $\lambda x.t$
- a

mv ::=
- i
- $\lambda x.t$
- a
- $\{t\}$

	|	$[\cdot]_a$	
$\mathbb{M}$	::=		
	|	$\varnothing$	
	|	$(c \mapsto mv) + \mathbb{M}$	
	|	$(c_1 \mapsto mv_1) + \,..\, + (c_l \mapsto mv_l) + \mathbb{M}$	
$\mathbb{S}$	::=		
	|	$\varnothing$	
	|	$[n] :: \mathbb{S}$	
u	::=		
	|	`upgrade` I `when` g_1 `until` g_2 `where` d	bind binders(d) in I bind binders(d) in g_1 bind binders(d) in g_2
	|	$u_1, \,..\, , u_k$	
$\mathcal{R}$	::=		
	|	u	
	|	$[\![\mathcal{C} \vdash \mathrm{I}]\!]_u$	
d	::=		
	|	$s = \{req\}$	binders $= s$
	|	$d_1 \,..\, d_l$	binders $=$ binders($d_1 \,..\, d_l$)
req	::=		
	|	`all threads`	
	|	τ	
	|	set	
	|	$req \bigcup req'$	
	|	$req \bigcap req'$	
	|	$req - req'$	
$\mathcal{C}$	::=		
	|	$s_1 = set_1 \,..\, s_l = set_l$	
set	::=		
	|	$a_1 \,..\, a_l$	
	|	$n_1 \,..\, n_l$	
g	::=		
	|	s `in stack`	
	|	$s@L$	
	|	$g \wedge g'$	
	|	$g \parallel g'$	
	|	$\neg g$	
	|	(g)	S
I	::=		
	|	$\mathrm{I}_1 \,..\, \mathrm{I}_k$	
	|	$s \leftarrow v$	

| $\texttt{mainredef}\ s\ t$
| $\texttt{suspend}\ s$
| $\texttt{resume}\ s$
| $\texttt{reboot}\ s$

B.2.3 Règles de réduction

$\boxed{(c \mapsto mv) \in \mathbb{M}}$

$$\frac{}{(c \mapsto mv) \in (c \mapsto mv) + \mathbb{M}}\ \text{S1} \qquad \frac{(c \mapsto mv) \in \mathbb{M}}{(c \mapsto mv) \in (c' \mapsto mv') + \mathbb{M}}\ \text{S2}$$

$\boxed{n \in \mathbb{S}}$

$$\frac{}{n \in [n] :: \mathbb{S}}\ \text{STACK1} \qquad \frac{n \in \mathbb{S}}{n \in [n'] :: \mathbb{S}_2}\ \text{STACK2}$$

$\boxed{\mathbb{M}, \mathbb{S}, t \longrightarrow_t \mathbb{M}', \mathbb{S}', t'}$

$$\frac{\mathbb{M}, \mathbb{S}, t \longrightarrow_t \mathbb{M}', \mathbb{S}', t'}{\mathbb{M}, \mathbb{S}, \mathrm{CT}[t] \longrightarrow_t \mathbb{M}', \mathbb{S}', \mathrm{CT}[t']}\ \textsc{Tcont}$$

$$\frac{c \textit{ is fresh}}{\mathbb{M}, \mathbb{S}, \texttt{def}\ n = v\ \texttt{in}\ t \longrightarrow_t (c \mapsto [n \to c^n]v) + \mathbb{M}, \mathbb{S}, [n \to c^n]t}\ \textsc{Tdef}$$

$$\frac{c \textit{ is fresh}}{\mathbb{M}, \mathbb{S}, \texttt{def}\ n = n' : \lambda x.t\ \texttt{in}\ t' \longrightarrow_t (c \mapsto [n \to c^n]\lambda x.t) + \mathbb{M}, \mathbb{S}, [n \to c^n]t'}\ \textsc{TdefNfun}$$

$$\frac{}{\mathbb{M}, \mathbb{S}, \lambda x.t\ v \longrightarrow_t \mathbb{M}, \mathbb{S}, [x \to v]t}\ \textsc{Tapp} \qquad \frac{}{\mathbb{M}, \mathbb{S}, n : \lambda x.t\ v \longrightarrow_t \mathbb{M}, [n] :: \mathbb{S}, \texttt{return}\ (\lambda x.t\ v)}\ \textsc{Tcall}$$

$$\frac{}{\mathbb{M}, [n] :: \mathbb{S}, \texttt{return}\ (v) \longrightarrow_t \mathbb{M}, \mathbb{S}, v}\ \textsc{Tret} \qquad \frac{(c \mapsto i) \in \mathbb{M}}{\mathbb{M}, \mathbb{S}, c^n \longrightarrow_t \mathbb{M}, \mathbb{S}, i}\ \textsc{Tnint}$$

$$\frac{(c \mapsto \lambda x.t) \in \mathbb{M}}{\mathbb{M}, \mathbb{S}, c^n \longrightarrow_t \mathbb{M}, \mathbb{S}, n : \lambda x.t}\ \textsc{Tnfun} \qquad \frac{(c \mapsto i) \in \mathbb{M}}{\mathbb{M}, \mathbb{S}, c^n_\tau \longrightarrow_t \mathbb{M}, \mathbb{S}, i}\ \textsc{Tntagint}$$

$$\frac{(c \mapsto \lambda x.t) \in \mathbb{M}}{\mathbb{M}, \mathbb{S}, c^n_\tau \longrightarrow_t \mathbb{M}, \mathbb{S}, n : \lambda x.t}\ \textsc{Tntagfun}$$

$$\frac{c \textit{ is fresh}}{\mathbb{M}, \mathbb{S}, \texttt{def}\ \tau : n = v\ \texttt{in}\ t \longrightarrow_t (c \mapsto [n \to c^n_\tau]v) + \mathbb{M}, \mathbb{S}, [n \to c^n_\tau]t}\ \textsc{Tdeftag}$$

$$\frac{}{\mathbb{M}, \mathbb{S}, L : t \longrightarrow_t \mathbb{M}, \mathbb{S}, t}\ \textsc{Tlabel}$$

$\boxed{\mathbb{M}, \mathcal{P} \longrightarrow_p \mathbb{M}', \mathcal{P}'}$

$$\frac{\mathbb{M}, \mathbb{S}, t \longrightarrow_t \mathbb{M}', \mathbb{S}', t'}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} t] \longrightarrow_p \mathbb{M}', \mathrm{C}[a \underset{t_0,\mathbb{S}'}{\triangleright} t']} \textsc{Pcont}$$

$$\frac{a' \textit{ is fresh}}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[\texttt{spawn}\ t]] \longrightarrow_p \mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[a']] \parallel a' \underset{t,\varnothing}{\triangleright} t} \textsc{Pspawn}$$

$$\frac{}{\mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel a \underset{t_0,\mathbb{S}}{\triangleright} v \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k \longrightarrow_p \mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k} \textsc{Pterm}$$

$$\frac{(c \mapsto \{t\}) \in \mathbb{M} \quad a' \textit{ is fresh}}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[c^n]] \longrightarrow_p (c \mapsto [\cdot]_{a'}) + \mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[c^n]] \parallel a' \underset{t,\varnothing}{\triangleright} t} \textsc{Plazy}$$

$$\frac{(c \mapsto \{t\}) \in \mathbb{M} \quad a' \textit{ is fresh}}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[c^n_\tau]] \longrightarrow_p (c \mapsto [\cdot]_{a'}) + \mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[c^n_\tau]] \parallel a' \underset{t,\varnothing}{\triangleright} t} \textsc{Plazytag}$$

$$\frac{(c \mapsto [\cdot]_a) \in \mathbb{M}}{\mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel a \underset{t_0,\mathbb{S}}{\triangleright} v \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k \longrightarrow_p (c \mapsto v) + \mathbb{M}, \mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \parallel \mathcal{T}'_1 \parallel .. \parallel \mathcal{T}'_k} \textsc{Plazyterm}$$

$\boxed{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow_r \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2}$

$$\frac{}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash]\!]_u \longrightarrow_r \mathbb{M}, \mathcal{P}, u} \textsc{Rend}$$

$$\frac{M, prog, d \mapsto \mathcal{C} \quad C, \mathcal{P} \vdash \neg g_1 \quad C, \mathcal{P} \vdash g_2}{\mathbb{M}, \mathcal{P}, \texttt{upgrade}\ \mathrm{I}\ \texttt{when}\ g_1\ \texttt{until}\ g_2\ \texttt{where}\ d, u_2, .., u_k \longrightarrow_r \mathbb{M}, \mathcal{P}, u_2, .., u_k} \textsc{Rdiscard}$$

$$\frac{M, prog, d \mapsto \mathcal{C} \quad C, \mathcal{P} \vdash g_1 \quad C, \mathcal{P} \vdash \neg g_2 \quad \mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \mathrm{I}_1]\!] \longrightarrow_r \mathbb{M}', \mathcal{P}'}{\begin{array}{l}\mathbb{M}, \mathcal{P}, \texttt{upgrade}\ \mathrm{I}_1\ \mathrm{I}_2\ ..\ \mathrm{I}_k\ \texttt{when}\ g_1\ \texttt{until}\ g_2\ \texttt{where}\ d, u_2, .., u_l \\ \quad \longrightarrow_r \mathbb{M}', \mathcal{P}', [\![\mathcal{C} \vdash \mathrm{I}_2\ ..\ \mathrm{I}_k]\!]_{\texttt{upgrade}\ \mathrm{I}_1\ \mathrm{I}_2\ ..\ \mathrm{I}_k\ \texttt{when}\ g_1\ \texttt{until}\ g_2\ \texttt{where}\ d, u_2, .., u_l}\end{array}} \textsc{Rreconf}$$

$$\frac{M, prog, d \mapsto \mathcal{C} \quad C, \mathcal{P} \vdash g_1 \quad C, \mathcal{P} \vdash g_2 \quad \mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \mathrm{I}_1]\!] \longrightarrow_r \mathbb{M}', \mathcal{P}'}{\mathbb{M}, \mathcal{P}, \texttt{upgrade}\ \mathrm{I}_1\ \mathrm{I}_2\ ..\ \mathrm{I}_k\ \texttt{when}\ g_1\ \texttt{until}\ g_2\ \texttt{where}\ d, u_2, .., u_l \longrightarrow_r \mathbb{M}', \mathcal{P}', [\![\mathcal{C} \vdash \mathrm{I}_2\ ..\ \mathrm{I}_k]\!]_{u_2, .., u_l}} \textsc{Roneshot}$$

$$\frac{\mathcal{C} \vdash s = n_1\ ..\ n_k \quad \texttt{map\_getCells}\ n_1\ ..\ n_k\ \mathcal{P} = c_1\ ..\ c_l \quad \mathbb{M}' = (c_1 \mapsto v) + .. + (c_l \mapsto v) + \mathbb{M}}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash s \leftarrow v\ \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_j} \longrightarrow_r \mathbb{M}', \mathcal{P}, [\![\mathcal{C} \vdash \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_j}} \textsc{Rupd}$$

$$\frac{\mathcal{C} \vdash s = a_1\ ..\ a_k \quad \texttt{map\_suspend}\ a_1\ ..\ a_k\ \mathcal{P} \mapsto \mathcal{P}'}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \texttt{suspend}\ s\ \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l} \longrightarrow_r \mathbb{M}, \mathcal{P}', [\![\mathcal{C} \vdash \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l}} \textsc{Rsups}$$

$$\frac{\mathcal{C} \vdash s = a_1\ ..\ a_k \quad \texttt{map\_resume}\ a_1\ ..\ a_k\ \mathcal{P} \mapsto \mathcal{P}'}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \texttt{resume}\ s\ \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l} \longrightarrow_r \mathbb{M}, \mathcal{P}', [\![\mathcal{C} \vdash \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l}} \textsc{Rres}$$

$$\frac{\mathcal{C} \vdash s = a_1\ ..\ a_k \quad \texttt{map\_reboot}\ a_1\ ..\ a_k\ \mathcal{P} \mapsto \mathcal{P}'}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \texttt{reboot}\ s\ \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l} \longrightarrow_r \mathbb{M}, \mathcal{P}', [\![\mathcal{C} \vdash \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l}} \textsc{Rreboot}$$

$$\frac{\mathcal{C} \vdash s = a_1\ ..\ a_k \quad \texttt{map\_mainRedef}\ a_1\ ..\ a_k\ t\ \mathcal{P} \mapsto \mathcal{P}'}{\mathbb{M}, \mathcal{P}, [\![\mathcal{C} \vdash \texttt{mainredef}\ s\ t\ \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l} \longrightarrow_r \mathbb{M}, \mathcal{P}', [\![\mathcal{C} \vdash \mathrm{I}_1\ ..\ \mathrm{I}_l]\!]_{u_2, .., u_l}} \textsc{Rmainredef}$$

$\boxed{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2}$

$$\frac{\mathbb{M}_1, \mathcal{P}_1 \longrightarrow_p \mathbb{M}_2, \mathcal{P}_2}{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R} \longrightarrow \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}} \textsc{Rexec} \qquad \frac{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow_r \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2}{\mathbb{M}_1, \mathcal{P}_1, \mathcal{R}_1 \longrightarrow \mathbb{M}_2, \mathcal{P}_2, \mathcal{R}_2} \textsc{Rupdate}$$

$$\frac{\mathbb{M}_1, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[t]], \mathcal{R} \longrightarrow_r \mathbb{M}_2, \mathcal{P}', \mathcal{R}'}{\mathbb{M}_1, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[\texttt{breakpoint}\ t]], \mathcal{R} \longrightarrow \mathbb{M}_2, \mathcal{P}', \mathcal{R}'} \textsc{Rbreak}$$

$$\frac{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[t]], \mathcal{R} \not\longrightarrow_r}{\mathbb{M}, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[\texttt{breakpoint}\ t]], \mathcal{R} \longrightarrow \mathbb{M}_1, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[t]], \mathcal{R}} \textsc{Rnbreak}$$

$\boxed{\textbf{getCells}\ n\ t = c_1 .. c_l}$

$$\frac{}{\textbf{getCells}\ n\ c^n = c} \textsc{GCcell1} \qquad \frac{}{\textbf{getCells}\ n\ c^n_\tau = c} \textsc{GCcell2} \qquad \frac{n_1 \neq n_2}{\textbf{getCells}\ n_1\ c^{n_2} =} \textsc{GCcell3}$$

$$\frac{}{\textbf{getCells}\ n\ i =} \textsc{GCint} \qquad \frac{}{\textbf{getCells}\ n\ x =} \textsc{GCvar} \qquad \frac{}{\textbf{getCells}\ n\ a =} \textsc{GCtid}$$

$$\frac{}{\textbf{getCells}\ n_1\ n_2 =} \textsc{GCnam} \qquad \frac{\textbf{getCells}\ n\ t = c_1 .. c_l}{\textbf{getCells}\ n\ \lambda x.t = c_1 .. c_l} \textsc{GClam}$$

$$\frac{\textbf{getCells}\ n\ t = c_1 .. c_l}{\textbf{getCells}\ n\ \texttt{breakpoint}\ t = c_1 .. c_l} \textsc{GCbreak} \qquad \frac{\textbf{getCells}\ n\ t = c_1 .. c_l}{\textbf{getCells}\ n\ L : t = c_1 .. c_l} \textsc{GClabel}$$

$$\frac{\textbf{getCells}\ n\ t = c_1 .. c_l}{\textbf{getCells}\ n_1\ n_2 : \lambda x.t = c_1 .. c_l} \textsc{GCnlam} \qquad \frac{\textbf{getCells}\ n\ t = c_1 .. c_l \quad \textbf{getCells}\ n\ t' = c'_1 .. c'_k}{\textbf{getCells}\ n\ t\ t' = c_1 .. c_l\ c'_1 .. c'_k} \textsc{GCapp}$$

$$\frac{\textbf{getCells}\ n\ t = c_1 .. c_l}{\textbf{getCells}\ n\ \texttt{spawn}\ t = c_1 .. c_l} \textsc{GCspa} \qquad \frac{\textbf{getCells}\ n\ t = c_1 .. c_l}{\textbf{getCells}\ n\ \texttt{return}\ t = c_1 .. c_l} \textsc{GCret}$$

$$\frac{\textbf{getCells}\ n\ t = c_1 .. c_l \quad \textbf{getCells}\ n\ t' = c'_1 .. c'_k}{\textbf{getCells}\ n\ \texttt{def}\ n' = t\ \texttt{in}\ t' = c_1 .. c_l\ c'_1 .. c'_k} \textsc{GCdef}$$

$$\frac{\textbf{getCells}\ n\ t = c_1 .. c_l \quad \textbf{getCells}\ n\ t' = c'_1 .. c'_k}{\textbf{getCells}\ n\ \texttt{def}\ \tau : n' = t\ \texttt{in}\ t' = c_1 .. c_l\ c'_1 .. c'_k} \textsc{GCdeftag}$$

$\boxed{\texttt{map\_getCells}\ n_1 .. n_k\ t = c_1 .. c_l}$

$$\frac{}{\texttt{map\_getCells}\ t =} \textsc{MGCnone}$$

$$\frac{\textbf{getCells}\ n_1\ t = c_1 .. c_l \quad \texttt{map\_getCells}\ n_2 .. n_k\ t = c'_1 .. c'_j}{\texttt{map\_getCells}\ n_1\ n_2 .. n_k\ t = c_1 .. c_l\ c'_1 .. c'_j} \textsc{MGCmany}$$

$\boxed{\texttt{map\_getCells}\ n_1 .. n_k\ \mathcal{P} = c_1 .. c_l}$

$$\frac{}{\texttt{map\_getCells}\ n_1 .. n_k\ =} \textsc{MGCemp}$$

$$\frac{\texttt{map\_getCells}\ n_1 .. n_k\ t = c_1 .. c_l \quad \texttt{map\_getCells}\ n_1 .. n_k\ \mathcal{P} = c'_1 .. c'_k}{\texttt{map\_getCells}\ n_1 .. n_k\ a \underset{t_0,\mathbb{S}}{\triangleright} t \parallel \mathcal{P} = c_1 .. c_l\ c'_1 .. c'_k} \textsc{MGCth1}$$

$$\frac{\texttt{map\_getCells}\ n_1 .. n_k\ t = c_1 .. c_l \quad \texttt{map\_getCells}\ n_1 .. n_k\ \mathcal{P} = c'_1 .. c'_k}{\texttt{map\_getCells}\ n_1 .. n_k\ a \underset{t_0,\mathbb{S}}{\blacktriangleright} t \parallel \mathcal{P} = c_1 .. c_l\ c'_1 .. c'_k} \textsc{MGCth2}$$

$\boxed{\mathcal{P} \vdash n\,\texttt{not\_in\_stack}}$

$$\frac{n \notin \mathbb{S}}{a \underset{t_0,\mathbb{S}}{\triangleright} t \vdash n\,\texttt{not\_in\_stack}}\ \textsc{Nstack} \qquad \frac{n \notin \mathbb{S}}{a \underset{t_0,\mathbb{S}}{\blacktriangleright} t \vdash n\,\texttt{not\_in\_stack}}\ \textsc{Nstackp}$$

$$\frac{\mathcal{T}_1 \vdash n\,\texttt{not\_in\_stack} \,..\, \mathcal{T}_l \vdash n\,\texttt{not\_in\_stack}}{\mathcal{T}_1 \parallel .. \parallel \mathcal{T}_l \vdash n\,\texttt{not\_in\_stack}}\ \textsc{Nnstackl}$$

$\boxed{\mathcal{P} \vdash a@L}$

$$\frac{}{\mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} \mathrm{CT}[L:t]] \vdash a@L}\ \textsc{Labl} \qquad \frac{}{\mathrm{C}[a \underset{t_0,\mathbb{S}}{\blacktriangleright} \mathrm{CT}[L:t]] \vdash a@L}\ \textsc{Lablp}$$

$\boxed{C, \mathcal{P} \vdash g}$

$$\frac{\mathcal{C} \vdash s = n_1 \,..\, n_l \quad \mathcal{P} \vdash n_1\,\texttt{not\_in\_stack} \,..\, \mathcal{P} \vdash n_l\,\texttt{not\_in\_stack}}{C, \mathcal{P} \vdash \neg(s\ \texttt{in stack})}\ \textsc{Ginstack}$$

$$\frac{\mathcal{C} \vdash s = n_1 \,..\, n_l \quad n_1 \in \mathbb{S}}{C, \mathrm{C}[a \underset{t_0,\mathbb{S}}{\triangleright} t] \vdash (s\ \texttt{in stack})}\ \textsc{Gninstack} \qquad \frac{\mathcal{C} \vdash s = a_1 \,..\, a_l \quad \mathcal{P} \vdash a_1@L \,..\, \mathcal{P} \vdash a_l@L}{C, \mathcal{P} \vdash s@L}\ \textsc{Gatlabel}$$

$$\frac{\mathcal{C} \vdash s = a_1 \,..\, a_l \quad t \neq t@L}{C, \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t] \vdash \neg(s@L)}\ \textsc{Gnatlabel} \qquad \frac{C, \mathcal{P} \vdash g_1 \quad C, \mathcal{P} \vdash g_2}{C, \mathcal{P} \vdash g_1 \wedge g_2}\ \textsc{Gand} \qquad \frac{C, \mathcal{P} \vdash g_1}{C, \mathcal{P} \vdash g_1 \parallel g_2}\ \textsc{Gor1}$$

$$\frac{C, \mathcal{P} \vdash g_2}{C, \mathcal{P} \vdash g_1 \parallel g_2}\ \textsc{Gor2} \qquad \frac{C, \mathcal{P} \vdash \neg g_1}{C, \mathcal{P} \vdash \neg(g_1 \wedge g_2)}\ \textsc{Gnand1} \qquad \frac{C, \mathcal{P} \vdash \neg g_2}{C, \mathcal{P} \vdash \neg(g_1 \wedge g_2)}\ \textsc{Gnand2}$$

$$\frac{C, \mathcal{P} \vdash \neg g_1 \quad C, \mathcal{P} \vdash \neg g_2}{C, \mathcal{P} \vdash \neg(g_1 \parallel g_2)}\ \textsc{Gnor}$$

$\boxed{\texttt{map\_suspend}\ a_1 \,..\, a_k\ \mathcal{P} \mapsto \mathcal{P}'}$

$$\frac{}{\texttt{map\_suspend}\ \mathcal{P} \mapsto \mathcal{P}}\ \textsc{Suspend0} \qquad \frac{\texttt{map\_suspend}\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t] \mapsto \mathcal{P}}{\texttt{map\_suspend}\ a_1\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t] \mapsto \mathcal{P}}\ \textsc{suspend2}$$

$$\frac{\texttt{map\_suspend}\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t] \mapsto \mathcal{P}}{\texttt{map\_suspend}\ a_1\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t] \mapsto \mathcal{P}}\ \textsc{suspend2s}$$

$\boxed{\texttt{map\_resume}\ a_1 \,..\, a_k\ \mathcal{P} \mapsto \mathcal{P}'}$

$$\frac{}{\texttt{map\_resume}\ \mathcal{P} \mapsto \mathcal{P}}\ \textsc{Resume0} \qquad \frac{\texttt{map\_resume}\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t] \mapsto \mathcal{P}}{\texttt{map\_resume}\ a_1\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t] \mapsto \mathcal{P}}\ \textsc{resume2}$$

$$\frac{\texttt{map\_resume}\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t] \mapsto \mathcal{P}}{\texttt{map\_resume}\ a_1\ a_2 \,..\, a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t] \mapsto \mathcal{P}}\ \textsc{resume2r}$$

$\boxed{\texttt{map\_reboot}\ a_1 .. a_k\ \mathcal{P} \mapsto \mathcal{P}'}$

$$\frac{}{\texttt{map\_reboot}\ \mathcal{P} \mapsto \mathcal{P}}\ \textsc{Reboot0} \qquad \frac{\texttt{map\_reboot}\ a_2 .. a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t_0] \mapsto \mathcal{P}}{\texttt{map\_reboot}\ a_1\ a_2 .. a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t] \mapsto \mathcal{P}}\ \textsc{reboot2}$$

$$\frac{\texttt{map\_reboot}\ a_2 .. a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t_0] \mapsto \mathcal{P}}{\texttt{map\_reboot}\ a_1\ a_2 .. a_l\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t_0] \mapsto \mathcal{P}}\ \textsc{reboot2s}$$

$\boxed{\texttt{map\_mainRedef}\ a_1 .. a_k\ t\ \mathcal{P} \mapsto \mathcal{P}'}$

$$\frac{}{\texttt{map\_mainRedef}\ t\ \mathcal{P} \mapsto \mathcal{P}}\ \textsc{Mredef0} \qquad \frac{\texttt{map\_mainRedef}\ a_2 .. a_l\ t\ \mathrm{C}[a_1 \underset{t,\mathbb{S}}{\triangleright} t'] \mapsto \mathcal{P}}{\texttt{map\_mainRedef}\ a_1\ a_2 .. a_l\ t\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\triangleright} t'] \mapsto \mathcal{P}}\ \textsc{Mredef2}$$

$$\frac{\texttt{map\_mainRedef}\ a_2 .. a_l\ t\ \mathrm{C}[a_1 \underset{t,\mathbb{S}}{\blacktriangleright} t'] \mapsto \mathcal{P}}{\texttt{map\_mainRedef}\ a_1\ a_2 .. a_l\ t\ \mathrm{C}[a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t'] \mapsto \mathcal{P}}\ \textsc{Mredef2s}$$

$\boxed{M, prog, d \mapsto \mathcal{C}}$

$$\frac{}{M, prog, s = \{set\} \mapsto s = set}\ \textsc{CapSet} \qquad \frac{\texttt{getAllThreads}\ \mathcal{P} = a_1 .. a_l}{M, prog, s = \{\texttt{all threads}\} \mapsto s = a_1 .. a_l}\ \textsc{CapAllT}$$

$$\frac{\texttt{getTag}\ \tau\ \mathcal{P} = n_1 .. n_l}{M, prog, s = \{\tau\} \mapsto s = n_1 .. n_l}\ \textsc{CapTau}$$

$$\frac{M, prog, s_1 = \{req_1\} \mapsto s_1 = n_1 .. n_l \quad M, prog, s_2 = \{req_2\} \mapsto s_2 = n'_1 .. n'_k}{M, prog, s = \{req_1 \bigcup req_2\} \mapsto s = n_1 .. n_l\ n'_1 .. n'_k}\ \textsc{CapUnionN}$$

$$\frac{M, prog, s_1 = \{req_1\} \mapsto s_1 = a_1 .. a_l \quad M, prog, s_2 = \{req_2\} \mapsto s_2 = a'_1 .. a'_k}{M, prog, s = \{req_1 \bigcup req_2\} \mapsto s = a_1 .. a_l\ a'_1 .. a'_k}\ \textsc{CapUnionA}$$

$$\frac{M, prog, s_1 = \{req_1\} \mapsto s_1 = n_1 .. n_l\ n'_1 .. n'_j \quad M, prog, s_2 = \{req_2\} \mapsto s_2 = n'_1 .. n'_j\ n''_1 .. n''_k}{M, prog, s = \{req_1 \bigcap req_2\} \mapsto s = n'_1 .. n'_j}\ \textsc{CapInterN}$$

$$\frac{M, prog, s_1 = \{req_1\} \mapsto s_1 = a_1 .. a_l\ a'_1 .. a'_j \quad M, prog, s_2 = \{req_2\} \mapsto s_2 = a'_1 .. a'_j\ a''_1 .. a''_k}{M, prog, s = \{req_1 \bigcap req_2\} \mapsto s = a'_1 .. a'_j}\ \textsc{CapInterA}$$

$$\frac{M, prog, s_1 = \{req_1\} \mapsto s_1 = n_1 .. n_l\ n'_1 .. n'_j \quad M, prog, s_2 = \{req_2\} \mapsto s_2 = n'_1 .. n'_j}{M, prog, s = \{req_1 - req_2\} \mapsto s = n'_1 .. n'_j}\ \textsc{CapMinN}$$

$$\frac{M, prog, s_1 = \{req_1\} \mapsto s_1 = a_1 .. a_l\ a'_1 .. a'_j \quad M, prog, s_2 = \{req_2\} \mapsto s_2 = a'_1 .. a'_j}{M, prog, s = \{req_1 - req_2\} \mapsto s = a'_1 .. a'_j}\ \textsc{CapMinA}$$

$$\frac{M, prog, s_2 = \{req_2\} .. s_l = \{req_l\} \mapsto s_2 = set_2 .. s_l = set_l \quad M, prog, s_1 = \{req_1\} \mapsto s_1 = set_1}{M, prog, s_1 = \{req_1\}\ s_2 = \{req_2\} .. s_l = \{req_l\} \mapsto s_1 = set_1\ s_2 = set_2 .. s_l = set_l}\ \textsc{CapTwo}$$

$\boxed{\texttt{getAllThreads}\ \mathcal{P} = a_1\ ..\ a_l}$

$$\frac{}{\texttt{getAllThreads}\ a \underset{t_0,\mathbb{S}}{\triangleright} t = a}\ \textsc{GetTone} \qquad \frac{}{\texttt{getAllThreads}\ a \underset{t_0,\mathbb{S}}{\blacktriangleright} t = a}\ \textsc{GetTones}$$

$$\frac{\texttt{getAllThreads}\ \mathcal{T}_2 \parallel\ ..\ \parallel \mathcal{T}_l = a_2\ ..\ a_l}{\texttt{getAllThreads}\ \mathcal{T}_2 \parallel\ ..\ \parallel \mathcal{T}_l \parallel a_1 \underset{t_0,\mathbb{S}}{\triangleright} t = a_1\ a_2\ ..\ a_l}\ \textsc{GetTmany}$$

$$\frac{\texttt{getAllThreads}\ \mathcal{T}_2 \parallel\ ..\ \parallel \mathcal{T}_l = a_2\ ..\ a_l}{\texttt{getAllThreads}\ \mathcal{T}_2 \parallel\ ..\ \parallel \mathcal{T}_l \parallel a_1 \underset{t_0,\mathbb{S}}{\blacktriangleright} t = a_1\ a_2\ ..\ a_l}\ \textsc{GetTmanys}$$

$\boxed{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l}$

$$\frac{}{\texttt{getTag}\ \tau\ n\ c^n =}\ \textsc{GTag1} \qquad \frac{}{\texttt{getTag}\ \tau\ n\ c^n_\tau = n}\ \textsc{GTag2} \qquad \frac{\tau_1 \neq \tau_2}{\texttt{getTag}\ \tau_1\ c^n_{\tau_2} =}\ \textsc{GTag3}$$

$$\frac{}{\texttt{getTag}\ \tau\ i =}\ \textsc{GTagint} \qquad \frac{}{\texttt{getTag}\ \tau\ x =}\ \textsc{GTagvar} \qquad \frac{}{\texttt{getTag}\ \tau\ a =}\ \textsc{GTagtid}$$

$$\frac{}{\texttt{getTag}\ \tau\ n =}\ \textsc{GTagnam} \qquad \frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l}{\texttt{getTag}\ \tau\ \lambda x.t = n_1\ ..\ n_l}\ \textsc{GTaglam}$$

$$\frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l}{\texttt{getTag}\ \tau\ \texttt{breakpoint}\ t = n_1\ ..\ n_l}\ \textsc{GTagbrk} \qquad \frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l}{\texttt{getTag}\ \tau\ L : t = n_1\ ..\ n_l}\ \textsc{GTaglabel}$$

$$\frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l}{\texttt{getTag}\ \tau\ n : \lambda x.t = n_1\ ..\ n_l}\ \textsc{GTagnlam} \qquad \frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l \quad \texttt{getTag}\ \tau\ t' = n'_1\ ..\ n'_k}{\texttt{getTag}\ \tau\ t\ t' = n_1\ ..\ n_l\ n'_1\ ..\ n'_k}\ \textsc{GTagapp}$$

$$\frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l}{\texttt{getTag}\ \tau\ \texttt{spawn}\ t = n_1\ ..\ n_l}\ \textsc{GTagspa} \qquad \frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l}{\texttt{getTag}\ \tau\ \texttt{return}\ t = n_1\ ..\ n_l}\ \textsc{GTagret}$$

$$\frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l \quad \texttt{getTag}\ \tau\ t' = n'_1\ ..\ n'_k}{\texttt{getTag}\ \tau\ \texttt{def}\ n = t\ \texttt{in}\ t' = n_1\ ..\ n_l\ n'_1\ ..\ n'_k}\ \textsc{GTagdef}$$

$$\frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l \quad \texttt{getTag}\ \tau\ t' = n'_1\ ..\ n'_k}{\texttt{getTag}\ \tau\ \texttt{def}\ \tau' : n = t\ \texttt{in}\ t' = n_1\ ..\ n_l\ n'_1\ ..\ n'_k}\ \textsc{GTagdeftag}$$

$\boxed{\texttt{getTag}\ \tau\ \mathcal{P} = n_1\ ..\ n_l}$

$$\frac{}{\texttt{getTag}\ \tau\ =}\ \textsc{GTagemp} \qquad \frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l \quad \texttt{getTag}\ \tau\ \mathcal{P} = n'_1\ ..\ n'_k}{\texttt{getTag}\ \tau\ a \underset{t_0,\mathbb{S}}{\triangleright} t \parallel \mathcal{P} = n_1\ ..\ n_l\ n'_1\ ..\ n'_k}\ \textsc{Gtafth1}$$

$$\frac{\texttt{getTag}\ \tau\ t = n_1\ ..\ n_l \quad \texttt{getTag}\ \tau\ \mathcal{P} = n'_1\ ..\ n'_k}{\texttt{getTag}\ \tau\ a \underset{t_0,\mathbb{S}}{\blacktriangleright} t \parallel \mathcal{P} = n_1\ ..\ n_l\ n'_1\ ..\ n'_k}\ \textsc{Gtagth2}$$

Annexe C

Implémentation de plates-formes

C.1 Code complet du programme de mise en cache

```
from pymoult.highlevel.listener import Listener
from pymoult.threads import DSU_Thread
from queue import Queue

#Cached Data
class CachedData(object):
    def __init__(self,data):
        self.data= data
    def get(self):
        return self.data

#Caches data in memory
def cache(data,session,request):
    cached = CachedData(data)
    session.add_cache(request,cached)

#Session
class Session(object):
    def __init__(self):
        self.cache = {}
        self.active = False
    def add_cache(self,key,data):
        self.cache[key] = data
    def has_cached(self,request):
        return request in self.cache.keys()
    def load_from_cache(self,request):
        return self.cache[request].get()
    def enter(self):
        self.active = True
    def leave(self):
        self.active = False

#Dummy get
def get(request):
    return "Data : "+str(request)
#Dummy display
def display(data):
    print(data)

#Event for comunicating with the main loop
requests = Queue()

#Main loop
def main():
```

```
    current_session = Session() #dummy session
    while True:
        (session,request) = requests.get()
        if session != current_session:
            current_session.leave()
            current_session = session
            current_session.enter()
        if current_session.has_cached(request):
            display(current_session.load_from_cache(request))
        else:
            data = get(request)
            cache(data,current_session,request)
            display(data)


listener = Listener()
listener.start()
mainThread = DSU_Thread(name="mainThread",target=main)
mainThread.start()
```

C.2 Code complet des mises à jour

C.2.1 Compression des données

```
#parsed
from pymoult.highlevel.managers import ThreadedManager
from pymoult.highlevel.updates import SafeRedefineUpdate, LazyConversionUpdate
import sys
import zlib

#New code
#———

class CachedData_v2(object):
    def __init__(self,data):
        self.data = data
    def get(self):
        raw = zlib.decompress(self.data)
        return raw.decode()

def cache_v2(data,session,request):
    raw = data.encode()
    comp = zlib.compress(raw)
    cached = CachedData_v2(comp)
    session.add_cache(request,cached)

#Update script
#————————-

mainmod = sys.modules['__main__']

#Transformer for class CachedData
def datatransf(obj):
    raw = obj.data.encode()
    comp = zlib.compress(raw)
    obj.data = comp

#Create update units
update_data = LazyConversionUpdate(mainmod.CachedData,CachedData_v2,datatransf,"dataUpdate")
update_cache = SafeRedefineUpdate(mainmod,mainmod.cache,cache_v2,"cacheUpdate")

#Create a manager
manager = ThreadedManager(name="manager")
```

```
manager.start()
#Send update units
manager.add_update(update_data)
manager.add_update(update_cache)
```

C.2.2 Durée de vie des données en cache

```
#parsed
from pymoult.highlevel.managers import fetch_manager
from pymoult.highlevel.updates import Update
from pymoult.lowlevel.data_update import updateToClass,redefineClass
from pymoult.lowlevel.data_access import DataAccessor
import sys
import time

mainmod = sys.modules['__main__']

#New code
#———
TIMEOUT = 5

class Session_v2(object):
    def __init__(self):
        self.cache = {}
        self.active = False
    def add_cache(self,key,data):
        timestamp = int(time.time())
        self.cache[key] = (timestamp,data)
    def has_cached(self,request):
        if request in self.cache.keys():
            d = int(time.time()) - self.cache[request][0]
            return d < TIMEOUT
        return False
    def load_from_cache(self,request):
        return self.cache[request][1].get()
    def enter(self):
        self.active = True
    def leave(self):
        self.active = False


#Update script
#————————-

#Transformer for class Session
def sessiontransf(obj):
    #Invalidate the whole cache as we cannot know if it is still valid
    obj.cache = {}
    obj.active = False

#micro-manager for delayed update of sessions
def micromanage_session(session):
    def leave_and_update():
        session.active = False
        #Do the update
        updateToClass(session,mainmod.Session,Session_v2,transformer=sessiontransf)
        #Rechage the leave method
        delattr(session,"leave")
    session.leave = leave_and_update

#Create update unit classes
class SessionUpdate(Update):
    def __init__(self,name):
        self.pending_sessions = []
```

```
        super(SessionUpdate,self).__init__(name=name)
    def wait_alterability(self):
        #Always alterable
        return True
    def apply(self):
        sessions = DataAccessor(mainmod.Session,strategy="immediate")
        for s in sessions:
            if s.active:
                micromanage_session(s)
                self.pending_sessions.append(s)
            else:
                updateToClass(session,mainmod.Session,Session_v2,transformer=sessiontransf)
        redefineClass(mainmod,mainmod.Session,Session_v2)
    def wait_over(self):
        finished = False
        while not finished:
            Session.leave()
            for s in self.pending_sessions:
                if type(s) != Session_v2:
                    finished = False
        self.pending_sessions = None

#Create update units
update_session = SessionUpdate("sessionUpdate")
#Get the previously started manager
manager = fetch_manager("manager")
#Send update units
manager.add_update(update_session)
```

Bibliographie

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining association rules between sets of items in large databases. *SIGMOD Rec.*, 22(2) :207–216, June 1993.

[2] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. Opus : online patches and updates for security. In *USENIX Security Symposium*, pages 287–302, Baltimore, Maryland, USA, August 2005.

[3] J. Appavoo, K. Hui, C. A. N. Soules, R. W. Wisniewski, D. M. Da Silva, O. Krieger, M. A. Auslander, D. J. Edelsohn, B. Gamsa, G. R. Ganger, and Others. Enabling autonomic behavior in systems software with hot swapping. *IBM Systems Journal*, 42(1) :60–76, 2003.

[4] Jeff Arnold and M. Frans Kaashoek. Ksplice : automatic rebootless kernel updates. In *European Conference on Computer Systems*, pages 187–198, April 2009.

[5] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana. Handling consistent dynamic updates on distributed systems. In *Computers and Communications (ISCC), 2010 IEEE Symposium on*, pages 471–476, June 2010.

[6] F. Banno, D. Marletta, G. Pappalardo, and E. Tramontana. Tackling consistency issues for runtime updating distributed systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, pages 1–8, April 2010.

[7] Andrew Baumann, Jonathan Appavoo, Robert W. Wisniewski, Dilma Da Silva, Orran Krieger, and Gernot Heiser. Reboots are for hardware : Challenges and solutions to updating an operating system on the fly. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 337–350, Santa Clara, CA, USA, jun 2007.

[8] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. Providing dynamic update in an operating system. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[9] Rida A. Bazzi, Bryan Topp, and Iulian Neamtiu. How to have your cake and eat it too : Dynamic software updating with just-in-time overhead. In *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, HotSWUp '12, pages 1–5, Piscataway, NJ, USA, 2012. IEEE Press.

[10] Gavin Bierman, Michael Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time ? In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 99–110, New York, NY, USA, 2003. ACM.

[11] Pymoult, site du projet : `http://bitbucket.org/smartinezgd/pymoult`.

[12] T. Bloom and M. Day. Reconfiguration and module replacement in argus : theory and practice. *Software Engineering Journal*, 8(2) :102–108, Mar 1993.

[13] Toby Bloom and Mark Day. Reconfiguration in Argus. In *Intl. Workshop on Configurable Dist. Systems*, pages 176–187, London, England, March 1992. Also in [swej93mar], pages 102–108.

[14] Franz Ferdinand Brasser, Mihai Bucicoiu, and Ahmad-Reza Sadeghi. Swap and play : Live updating hypervisors and its application to xen. In *Proceedings of the 6th Edition of the ACM Workshop on Cloud Computing Security*, CCSW '14, pages 33–44, New York, NY, USA, 2014. ACM.

[15] Einar Broch, Olaf Owe, and Isabelle Simplot-Ryl. A Dynamic Class Construct for Asynchronous Concurrent Objects. In M. Steffen and G. Zavattaro, editors, *Proc. 7th International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 15–30, Athens, Greece, 2005. Springer-Verlag.

[16] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a taxonomy of software change : Research articles. *J. Softw. Maint. Evol.*, 17(5) :309–332, September 2005.

[17] Jérémy Buisson, Françoise André, and Jean-Louis Pazat. Afpac : enforcing consistency during the adaptation of a parallel component. *Scalable Computing : Practice and Experience*, 7(3) :61, September 2006.

[18] Jérémy Buisson, Everton Calvacante, Fabien Dagnat, Sébastien Martinez, and Elena Leroux. Coqcots & Pycots : non-stopping components for safe dynamic reconfiguration. In ACM, editor, *Proc of the 17th Symposium on Component-Based Software Engineering*, CBSE, pages 85 – 90, New York, USA, 2014.

[19] Jérémy Buisson and Fabien Dagnat. Introspecting continuations in order to update active code. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, pages 4 :1–4 :5, New York, NY, USA, 2008. ACM.

[20] Jérémy Buisson and Fabien Dagnat. Recaml : execution state as the cornerstone of reconfigurations. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 27–38, 2010.

[21] Jérémy Buisson, Fabien Dagnat, Elena Leroux, and Sébastien Martinez. Safe reconfiguration of Coqcots and Pycots components. *Journal of Systems and Software*, 2015.

[22] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2Nd International Conference on Virtual Execution Environments*, VEE '06, pages 35–44, New York, NY, USA, 2006. ACM.

[23] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. Polus : A powerful live updating system. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 271–281, Washington, DC, USA, 2007. IEEE Computer Society.

[24] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic software updating using a relaxed consistency model. *IEEE Transactions on Software Engineering*, 37(5) :679–694, 2011.

[25] Junqing Chen and Linpeng Huang. Supporting dynamic service updates in pervasive applications. In *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*, pages 273–278, July 2011.

[26] Junqing Chen, Linpeng Huang, Siqi Du, and Wenjia Zhou. A formal model for supporting frameworks of dynamic service update based on osgi. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 234–241, Nov 2010.

[27] Django, site du project : http://www.djangoproject.com/.

[28] M. Dmitriev. Towards flexible and safe technology for runtime evolution of java language applications. In *In Proceedings of the Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001.

[29] Mikhail Dmitriev. Safe class and data evolution in large and long-lived java[tm] applications. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2001.

[30] Siqi Du, Linpeng Huang, and Junqing Chen. A formal model for dynamic service updates in pervasive computing. In *Proceedings of the 2011 Fifth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, IMIS '11, pages 114–119, Washington, DC, USA, 2011. IEEE Computer Society.

[31] Michael J. Eager. Introduction to the dwarf debugging format, 2007.

[32] Serena Fritsch and Siobhán Clarke. Timeadapt : Timely execution of dynamic software reconfigurations. In *Proceedings of the 5th Middleware Doctoral Symposium*, MDS '08, pages 13–18, New York, NY, USA, 2008. ACM.

[33] Gnu c compiler, site du projet : http://gcc.gnu.org/.

[34] Mohammad Ghafari, Pooyan Jamshidi, Saeed Shahbazi, and Hassan Haghighi. An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '12, pages 177–182, New York, NY, USA, 2012. ACM.

[35] Stephen Gilmore, Dilsun Kirli, and Chris Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, The University of Edinburgh, 1997.

[36] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 279–292, New York, NY, USA, 2013. ACM.

[37] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. Safe and automatic live update for operating systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 279–292, New York, NY, USA, 2013. ACM.

[38] Cristiano Giuffrida and Andrew Tanenbaum. A taxonomy of live updates. In *ASCI Conference*, Veldhoven, The Netherlands, November 2010.

[39] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic update of java applications&mdash ;balancing change flexibility vs programming transparency. *J. Softw. Maint. Evol.*, 21(2) :81–112, March 2009.

[40] Allan Raundahl Gregersen, Douglas Simon, and Bo Nørregaard Jørgensen. Towards a dynamic-update-enabled jvm. In *Proceedings of the Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, pages 2 :1–2 :7, New York, NY, USA, 2009. ACM.

[41] Tianxiao Gu, Chun Cao, Chang Xu, Xiaoxing Ma, Linghao Zhang, and Jian Lu. Javelus : A low disruptive approach to dynamic software updates. In Karl R. P. H. Leung and Pornsiri Muenchaisri, editors, *APSEC*, pages 527–536. IEEE, 2012.

[42] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *Software Engineering, IEEE Transactions on*, 22(2) :120–131, Feb 1996.

[43] C. Hayden, E. Smith, Michael W. Hicks, and Jeffrey S. Foster. State transfer for clear and efficient runtime upgrades. *HotSWUp'11*, 2011.

[44] Christopher M. Hayden, Edward K. Smith, Michail Denchev, Michael Hicks, and Jeffrey S. Foster. Kitsune : efficient, general-purpose dynamic software updating for C. In Gary T. Leavens and Matthew B. Dwyer, editors, *OOPSLA*, pages 249–264. ACM, 2012.

[45] E. Horiuchi. Nonstop update of running robot controllers. In *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 4, pages 3662–3667 vol.4, Sept 2004.

[46] E. Horiuchi, O. Matsumoto, and N. Koyachi. Safety issues in nonstop update of running programs for mobile robots. In *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, pages 120–126, Aug 2005.

[47] Jeff Kramer and Jeff Magee. The evolving philosophers problem : Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11) :1293–1306, November 1990.

[48] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42 : Building a complete operating system. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '06, pages 133–145, New York, NY, USA, 2006. ACM.

[49] Stephen Magill, Michael Hicks, Suriya Subramanian, and Kathryn S. McKinley. Automating object transformations for dynamic software updating. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 265–280, New York, NY, USA, 2012. ACM.

[50] Kristis Makris and Rida A. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 31–31, Berkeley, CA, USA, 2009. USENIX Association.

[51] Kristis Makris and Kyung Dong Ryu. On-the-fly kernel updates for high-performance computing clusters. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 361–361, Washington, DC, USA, 2006. IEEE Computer Society.

[52] Kristis Makris and Kyung Dong Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. *SIGOPS Oper. Syst. Rev.*, 41(3) :327–340, March 2007.

[53] Sébastien Martinez, Fabien Dagnat, and Jérémy Buisson. Prototyping DSU techniques using Python. In USENIX, editor, *HotSWUp 2013 : 5th Workshop on Hot Topics in Software Upgrades*, 2013.

[54] Sébastien Martinez, Fabien Dagnat, and Jérémy Buisson. Pymoult : On-line updates for python programs. In *Proceedings of the 2015 International Conference on Software Engineering Advances*, Barcelona, Spain, nov 2015.

[55] E. Miedes and F. D. Muñoz-Escoí. A survey about dynamic software updating. Technical Report ITI-SIDI-2012/003, Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, May 2012.

[56] Iulian Neamtiu and Michael Hicks. Safe and timely updates to multi-threaded programs. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 13–24, New York, NY, USA, 2009. ACM.

[57] Iulian Neamtiu, Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 37–49, New York, NY, USA, 2008. ACM.

[58] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *Proc of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, 2006.

[59] A.C. Noubissi, J. Iguchi-Cartigny, and J. Lanet. Incremental dynamic update for java-based smart cards. In *Systems (ICONS), 2010 Fifth International Conference on*, pages 110–113, April 2010.

[60] Agnes C. Noubissi, Julien Iguchi-Cartigny, and Jean-Louis Lanet. Hot updates for java based smart cards. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering Workshops*, ICDEW '11, pages 168–173, Washington, DC, USA, 2011. IEEE Computer Society.

[61] A. Orso, A. Rao, and M.J. Harrold. A technique for dynamic updating of java software. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 649–658, 2002.

[62] Valerio Panzica La Manna. Local dynamic update for component-based distributed systems. In *Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering*, CBSE '12, pages 167–176, New York, NY, USA, 2012. ACM.

[63] Luís Pina and Michael Hicks. Rubah : Efficient, general-purpose dynamic software updating for java. In *Presented as part of the 5th Workshop on Hot Topics in Software Upgrades*, Berkeley, CA, 2013. USENIX.

[64] Mario Pukall, Christian Kästner, Walter Cazzola, Sebastian Götz, Alexander Grebhahn, Reimar Schröter, and Gunter Saake. Javadaptor - flexible runtime updates of java applications. *Softw., Pract. Exper.*, 43(2) :153–185, 2013.

[65] Pyftpdlib, site du projet : `http://github.com/giampaolo/pyftpdlib`.

[66] Emmanuel Riou, Erven Rohou, Philippe Clauss, Nabil Hallou, and Alain Ketterlin. PADRONE : a Platform for Online Profiling, Analysis, and Optimization. In *DCE 2014 - International workshop on Dynamic Compilation Everywhere*, Vienne, Austria, January 2014.

[67] Jörg Sander, Martin Ester, Hans-Peter Kriegel, and Xiaowei Xu. Density-based clustering in spatial databases : The algorithm gdbscan and its applications. *Data Min. Knowl. Discov.*, 2(2) :169–194, June 1998.

[68] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The lam/mpi checkpoint/restart framework : System-initiated checkpointing. In *in Proceedings, LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[69] Habib Seifzadeh, Hassan Abolhassani, and Mohsen Sadighi Moshkenani. A survey of dynamic software updating. *Journal of Software : Evolution and Process*, 2012.

[70] Junrong Shen, Xi Sun, Gang Huang, Wenpin Jiao, Yanchun Sun, and Hong Mei. Towards a unified formal model for supporting mechanisms of dynamic component update. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 80–89, New York, NY, USA, 2005. ACM.

[71] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis mutandis : Safe and predictable dynamic software updating, 2007.

[72] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates : A vm-centric approach. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 1–12, New York, NY, USA, 2009. ACM.

[73] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility : A low disruptive alternative to quiescence for ensuring safe dynamic updates. *Software Engineering, IEEE Transactions on*, 33(12) :856–868, Dec 2007.

[74] Michael Wahler, Stefan Richter, and Manuel Oriol. Dynamic software updates for real-time systems. In *Proceedings of the 2Nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 2 :1–2 :6, New York, NY, USA, 2009. ACM.

[75] E. Wernli. Theseus : Whole updates of java server applications. In *Hot Topics in Software Upgrades (HotSWUp), 2012 Fourth Workshop on*, pages 41–45, June 2012.

[76] Erwann Wernli, David Gurtner, and Oscar Nierstrasz. Using first-class contexts to realize dynamic software updates. In *Proceedings of the International Workshop on Smalltalk Technologies*, IWST '11, pages 2 :1–2 :11, New York, NY, USA, 2011. ACM.

[77] Erwann Wernli, Mircea Lungu, and Oscar Nierstrasz. Incremental dynamic updates with first-class contexts. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 304–319, Berlin, Heidelberg, 2012. Springer-Verlag.

[78] Guowu Xie, Jianbo Chen, and I. Neamtiu. Towards a better understanding of software evolution : An empirical study on open source software. In *Software Maintenance, 2009. IEEE International Conference on*, pages 51–60, Sept 2009.

[79] Ji Zhang and B. H. C. Cheng. Adaptation semantics. Technical report, Department of Computer Science, Michigan State University, East Lansing, Michigan, February 2005.

[80] Min Zhang, K. Ogata, and K. Futatsugi. An algebraic approach to formal analysis of dynamic software updating mechanisms. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 664–673, Dec 2012.

Résumé

La mise à jour dynamique des logiciels permet de modifier ces derniers sans interrompre les services qu'ils fournissent. C'est un enjeu important à une époque où les logiciels sont omniprésents et où leur indisponibilité peut être coûteuse (service commercial) ou même dangereuse (système de sécurité). De nombreux mécanismes aux propriétés et besoins variés permettent d'atteindre cet objectif. Ces mécanismes sont employés par des plates-formes dédiées à des types de logiciel et/ou de mises à jour spécifiques. En se spécialisant, ces plates-formes facilitent l'écriture de mises à jour dynamiques mais peuvent être mal adaptées à l'application de certaines modifications imprévues.

Il convient alors de sélectionner et combiner les mécanismes les mieux adaptés à chaque mise à jour afin d'assurer une meilleure compatibilité des plates-formes avec les différents logiciels et mises à jour. C'est autour de cet objectif que s'organisent les contributions de ce manuscrit:

- Étudier les plates-formes et identifier des modèles génériques de plate-forme et de mise à jour
- Étudier les besoins et les propriétés des mécanismes de mise à jour ainsi que leurs capacités à être combinés.
- Développer des plates-formes configurables permettant de sélectionner les mécanismes les mieux adaptés pour chaque mise à jour.

Les résultats obtenus ouvrent des pistes vers une nouvelle génération de plates-formes ainsi que vers de nouvelles utilisations de la mise à jour dynamique. Le troisième axe a mené au développement de Pymoult, plate-forme configurable pour programmes Python. Cette plate-forme fournit de nombreux mécanismes au travers d'une API de haut niveau adaptée à la conception de mises à jour dynamiques.

Mots-clés : Mise à jour dynamique de logiciel, Génie logiciel, Langages de programmation, Pymoult, Reconfiguration logicielle

Abstract

Dynamic software updating allows applications to be modified without interrupting the services it provides. Because today's systems rely heavily on software and its availability, such a possibility is an important issue. Many mechanisms with diverse needs and properties enable dynamic updates. They are used by platforms targeting specific types of applications and/or updates. While the specialization of these platforms make the development of dynamic updates easier, it can cause the platform to be ill suited in the case of unforeseen updates.

A solution is to select and combine best-suited mechanisms for each update in order to guarantee a best compatibility of platforms with the different kinds of applications and updates. The three contributions detailed in this thesis follow this objective:

- Studying platforms and identify generic models for platforms and updates
- Studying the needs and properties of mechanisms as well as their capacity to be combined
- Develop configurable platforms allowing the selection of best-suited mechanisms for each update.

Theses contributions open leads towards a new generation of platforms and towards new uses of dynamic updates. The third contribution lead to the development of Pymoult, a configurable platform for Python programs. Pymoult provides several mechanisms through a high-level API suited to the conception of dynamic updates.

Keywords : Dynamic software updating, Software engineering, Programing languages, Pymoult, software reconfiguration

TELECOM Bretagne

n° d'ordre : 2016telb0395
Télécom Bretagne
Technopôle Brest-Iroise - CS 83818 - 29238 Brest Cedex 3
Tél : + 33(0) 29 00 11 11 - Fax : + 33(0) 29 00 10 00