UNIVERSITÉ DU QUÉBEC À MONTRÉAL

CELLULONIT: UNE IMPLÉMENTATION DU MODÈLE ACTEUR EN NIT

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

ROMAIN CHANOIR

DECEMBRE 2017

REMERCIEMENTS

Je tiens à remercier toutes les personnes qui m'ont soutenu au cours des dernières années. Jean, mon directeur, pour m'avoir aidé dans ce travail inconditionnellement, en plus d'accepter humblement des kilos de bergamotes et des cafés faits avec amour. Mon co-directeur, Guy, et son chalet à Saint-Jean-de-Matha, sans lesquels je ne serais pas en train d'écrire ces remerciements, mais putôt encore en train de réfléchir au contenu de mon premier chapitre. Tous les membres du projet Nit : Alexis pour son aide et surtout ces moments passés à développer des jeux ou à faire de la VR ; Alexandre, toujours prêt à faire oublier les moments de doute à coups de Jameson au Benelux, et toujours de belles histoires à raconter ; Lucas, qui essayait probablement de casser son clavier (ou ses doigts) en codant, et pour ses multiples tentatives de me faire utiliser GDB pour déboger mon code ; Philippe, que je pèterai un jour à smash, peut-être... Tous mes amis à Montréal, dont mes colocs Mathieu, Lucas et Alex, qui m'ont supporté et ont su respecter mon culte de la raclette ; Florent, Marion, Nico, Anthony, Peggy, JP, Ariane, Arabelle, etc. Mes amis en France, pour avoir pris des nouvelles régulièrement et avoir été présents lorsque je rentrais en France en vacances : Valentin, Arthur, Mathieu, Clément, et tous ceux que j'oublie. Un merci particulier à Arthur qui a conservé mes identifiants Steam pendant la phase de rédaction de ce mémoire, même si me donner son compte NetFlix en plus était moins judicieux. Enfin, merci à ma famille, surtout mes parents qui m'ont encouragé et soutenu durant toute cette aventure et tout au long de ma vie.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

Ces dernières années, la programmation parallèle s'est développée et se décline maintenant de plusieurs manières. Dans ce mémoire, nous nous intéressons au modèle acteur, composé d'entités autonomes et asynchrones communiquant par l'intermédiaire de messages.

D'abord énoncé comme un modèle théorique au début des années 70, le modèle acteur a ensuite été popularisé par son implémentation dans le langage Erlang. Depuis, plusieurs autres implémentations ont vu le jour sous différentes formes — langages, bibliothèques ou extensions de langage — et notamment Akka et Celluloid.

Akka, sous la forme de bibliothèques Java et Scala, propose une implémentation robuste et complète du modèle acteur. Toutefois, son interface de programmation est verbeuse et complexe, et profite peu du typage statique offert par Java et Scala. Celluloid, de son coté, propose une implémentation sous la forme d'un *gem* (bibliothèque) Ruby. Son interface de programmation simple et proche de la programmation à objets classique permet d'écrire des programmes avec acteurs quasi identiques à leur équivalent Ruby, toutefois sans aucun typage statique, Ruby étant un langage dynamique.

Nous proposons une implémentation du modèle acteur en Nit, appelée **CelluloNit**, qui bénéficie à la fois de la simplicité de Celluloid et des avantages procurés par le typage statique.

L'exécution d'une série de programmes de tests de performance place CelluloNit entre Akka et Celluloid en termes de performance. CelluloNit représente donc une première implémentation intéressante du modèle acteur en Nit.

MOTS-CLÉS : modèle acteur, programmation parallèle, compilation, language à objets, Celluloid.

INTRODUCTION

L'avènement des machines à processeurs multi-coeurs depuis le milieu des années 2000 a popularisé une approche ancienne, mais peu utilisée jusqu'alors, la programmation parallèle. Cette approche permet notamment d'exploiter tous les coeurs de tels processeurs pour exécuter des tâches exigeantes en calcul.

Il existe deux approches principales à la programmation parallèle : en mémoire partagée, généralement avec des *threads*, ou en mémoire distribuée, généralement avec des processus. La première vise principalement l'exploitation parallèle des ressources d'une seule machine et utilise des bibliothèques telles que Pthreads (Butenhof, 1997) ou OpenMP (Chandra et al., 2001; Quinn, 2003), tandis que la deuxième se concentre sur l'exploitation d'un réseau de machines, notamment avec une bibliothèque telle que MPI (Pacheco, 1997; Quinn, 2003). Certains programmes ou langages utilisent quant à eux une approche hybride. Une telle approche combine le découpage en « grosses » tâches sur un réseau de machines — par exemple avec MPI — tout en exploitant le parallélisme disponible sur chacune d'elles — par exemple avec OpenMP.

Le but initial de ce projet de maîtrise était d'implémenter une extension parallèle pour le langage Nit, qui ne possèdait alors qu'un support minimal pour le parallélisme. Le langage Nit est un langage de programmation orienté objet statiquement typé, développé principalement à l'UQAM par l'équipe du professeur Jean Privat (Privat, 2006). Afin de déterminer quel type d'interface pour la programmation parallèle nous voulions implémenter, nous avons étudié et analysé différentes approches existantes. Notre objectif était de fournir au programmeur Nit une interface de programmation parallèle la plus simple possible.

Nous avons commençé par regarder du côté du modèle PGAS (De Wael et al., 2015) —

Partitionned Global Address Space. Ce modèle définit un espace mémoire partagé, mais partitionné, dans un contexte où cet espace est en réalité distribué et où un *thread* doit distinguer entre accès local (peu coûteux et immédiat) et accès global (coûteux, avec un temps de latence imprévisible). Les langages PGAS introduisent des constructions qui permettent de lancer des tâches asynchrones à être exécutées à certaines *places* — une *place* est une portion cohérente de l'espace mémoire, qui regroupe à la fois des données et des *threads* opérant sur ces données. Différents mécanismes permettent de partager ou protéger des données entre différentes *places*, notamment avec la possibilité de synchroniser plusieurs tâches. Ce modèle est intéressant, mais nécessite des langages spécialisés ou des extensions de langage conséquentes — par exemple, Habanero-Java (Cavé et al., 2011), Chapel (Chamberlain, Callahan et Zima, 2007), X10 (Charles et al., 2005). Il nous a semblé trop ambitieux de tenter une implémentation d'une telle approche en Nit, tant pour la complexité de la mise en oeuvre que pour la complexité d'utilisation par le programmeur.

D'autres cadres parallèles existent, notamment SCOOP (Torshizi et al., 2009; Eiffel, 2017) — *Simple Concurrent Object Oriented Programming* — pour le langage Eiffel. SCOOP divise la mémoire en régions indépendantes, et chaque objet appartient exactement à une région. La communication entre régions se fait alors par l'envoi de messages asynchrones. SCOOP étend le système de type d'Eiffel en ajoutant le mot-clé `separate`. Lorsqu'un objet est de type `separate`, cela signifie qu'il appartient potentiellement à une autre région. Un ensemble de règles déterminent alors la validité d'un appel de méthode sur un objet `separate`, ainsi que le caractère synchrone ou asynchrone des appels effectués. La compilation s'occupe de vérifier la bonne utilisation du type `separate`, tandis que le système d'exécution (*runtime system*) s'occupe de la gestion des synchronisations nécessaires au bon déroulement de l'exécution. On peut même utiliser des objets `separate` dans les post-conditions et les pré-conditions, en respectant aussi un ensemble de règles. Le modèle de SCOOP est intéressant lui aussi, mais les règles appliquées à l'utilisation des objets `separate` sont strictes et complexes. En pratique, de nombreux appels deviennent synchrones et le parallélisme est limité. De plus, cela force

le programmeur à prendre en compte la localisation de ses objets, modifiant radicalement sa manière de programmer. Pour ces raisons, nous avons aussi décidé de ne pas nous inspirer de SCOOP.

En plus de SCOOP et du modèle PGAS, on trouve le modèle acteur (Agha, 1986; Tilkov, 2016). Un acteur est une entité concurrente autonome, qui encapsule son état et communique exclusivement par envoi de messages asynchrones. L'implémentation la plus populaire du modèle acteur est le langage fonctionnel Erlang (Armstrong et al., 1993). Dans le cadre de notre étude du modèle acteur, nous avons concentré notre attention sur deux implémentations en particulier : Akka en Java (Akka, 2017) et Celluloid en Ruby (Celluloid, 2017).

Akka est un *framework* volumineux, offrant une API — Application Programming Interface — en Java et une en Scala, les deux s'inspirant fortement d'Erlang pour fournir une implémentation la plus proche possible du modèle d'acteur théorique. Néanmoins, la création et gestion des acteurs est dynamique — notamment via la réflexion —, et ce bien que Java et Scala soient des langages statiquements typés. De cette dualité découle une interface de programmation verbeuse et complexe.

Celluloid de son côté propose une implémentation du modèle acteur moins complète et moins robuste qu'Akka, mais plus simple à utiliser et beaucoup plus proche de la programmation orientée objet classique. Ainsi, passer d'un programme séquentiel écrit avec des classes Ruby normales à un programme parallèle/concurrent utilisant des acteurs Celluloid se fait rapidement et facilement. L'interface de programmation fournie par Celluloid est intéressante et a fortement inspiré notre propre implémentation du modèle acteur, d'où son nom : CelluloNit.

Un inconvénient de Celluloid, basé sur le langage dynamique Ruby, est que les erreurs de type dans le programme sont uniquement signalées au cours de l'exécution d'un programme, et pas avant. De son côté, le typage statique de Nit permet de repérer d'eventuelles fautes de type lors de la compilation, sans avoir à exécuter le programme. Le défi principal rencontré pour notre implémentation a donc été de faire cohabiter

l'interface et le modèle simple de Celluloid avec le typage statique de Nit.

Akka et Celluloid se présentent tous deux sous la forme de bibliothèques respectivement pour les langages Java et Ruby. Elles reposent sur des mécanismes inhérents à ces langages, notamment la réflexion, les clôtures, et le fait que le code est interprété — par machine virtuelle Java directement pour Akka, et par l'interpréteur JRuby écrit en Java pour Celluloid. De son côté, CelluloNit est implémenté comme une extension du langage Nit. En effet, c'est une combinaison de bibliothèques et de deux phases ajoutées au compilateur Nit. Ces modifications apportées directement au langage nous ont permis d'avoir plus de liberté et de compenser le fait que Nit ne possède pas les mêmes mécanismes que Java ou Ruby.

Dans le premier chapitre, nous commençons par présenter le modèle acteur théorique, puis les implémentations de ce modèle par Akka et par Celluloid, afin d'avoir des points de référence pour l'API de notre propre implémentation. Le deuxième chapitre présente les fonctionnalités proposées au programmeur par notre implémentation, CelluloNit — donc l'API (*Application Programming Interface*) de CelluloNit. Le troisième chapitre, quant à lui, présente l'implémentation de CelluloNit. Il explique notamment les différents problèmes auxquels nous avons été confrontés ainsi que les solutions d'implémentation qui nous ont permis de les règler afin d'obtenir une version à typage statique de Celluloid, en Nit. Dans le quatrième chapitre, nous présentons et discutons des résultats d'expérimentation qui comparent les performances d'Akka, Celluloid et CelluloNit. Finalement, nous concluons avec des pistes de travail futur.

CHAPITRE I

ACTEURS : MODÈLE THÉORIQUE ET LANGAGES

Le terme acteur a été employé pour la première fois dans les années 70 par Hewitt (Hewitt, Bishop et Steiger, 1973; Hewitt, 1977). La définition utilisée de nos jours est celle énoncée par Agha une dizaine d'années plus tard (Agha, 1986). Nous présentons tout d'abord cette définition, puis nous présentons quelques mises en oeuvre de ce modèle, dont deux de façon plus détaillée : Akka (Java) et Celluloid (Ruby).

1.1 Définition générale du modèle d'acteurs

1.1.1 Acteurs et système d'acteurs

Le modèle acteur est une théorie mathématique qui traite les "acteurs" comme *primitives universelles de calcul* pour la **concurrence** (Hewitt, Bishop et Steiger, 1973).

Un système d'acteurs est composé d'une collection d'acteurs. Les acteurs sont des entités concurrentes et autonomes qui communiquent de façon asynchrone par l'envoi de messages. Un acteur possède son propre fil d'exécution — pas obligatoirement un thread système — et traite les messages qu'il reçoit dans leur ordre d'arrivée. L'état interne d'un acteur est dissimulé et encapsulé, donc le seul moyen de modifier son état est en lui envoyant un message approprié.

Le parallélisme dans le modèle acteur apparait lorsqu'un programme regroupe plusieurs acteurs et qu'à chaque acteur est associé une unité d'exécution. En effet, si chaque acteur traite ses messages de façon séquentielle sur son propre fil d'exécution, plusieurs acteurs

qui travaillent au même moment conduisent naturellement à une exécution parallèle.

De par la nature du modèle, la programmation parallèle par acteurs supprime presqu'entièrement le besoin d'écrire du code spécifique pour la synchronisation du système. De plus, comme chaque acteur est une entité autonome qui agit en fonction des messages qu'il reçoit, cela permet de décomposer les programmes en composants autonomes. Cette modularité permet d'implémenter des stratégies pour des applications « auto-régénerantes » et robustes, dans lesquelles un composant qui *plante* peut tout simplement être redémarré.

1.1.2 Caractéristiques d'un acteur

Un acteur possède les caractéristiques suivantes — voir Figure 1.1 :

- Un **nom** globalement unique qui permet de l'identifier ;
- Une **boîte** aux lettres qui lui est propre, pour la réception des messages qui lui sont destinés ;
- Un **comportement**, qui détermine les traitements à effectuer en fonction des messages reçus ;
- Un **état interne**, encapsulé, qui peut être uniquement modifié par le biais des traitements effectués en réaction aux messages reçus.

Sur réception d'un message, un acteur peut effectuer une ou plusieurs des actions suivantes :

- Mettre à jour son état interne ;
- Envoyer des messages à des acteurs — y compris à lui-même ;
- Créer des nouveaux acteurs.

1.1.3 Propriétés du modèle théorique

Le modèle acteur théorique impose de respecter plusieurs propriétés, décrites plus en détails ci-bas :

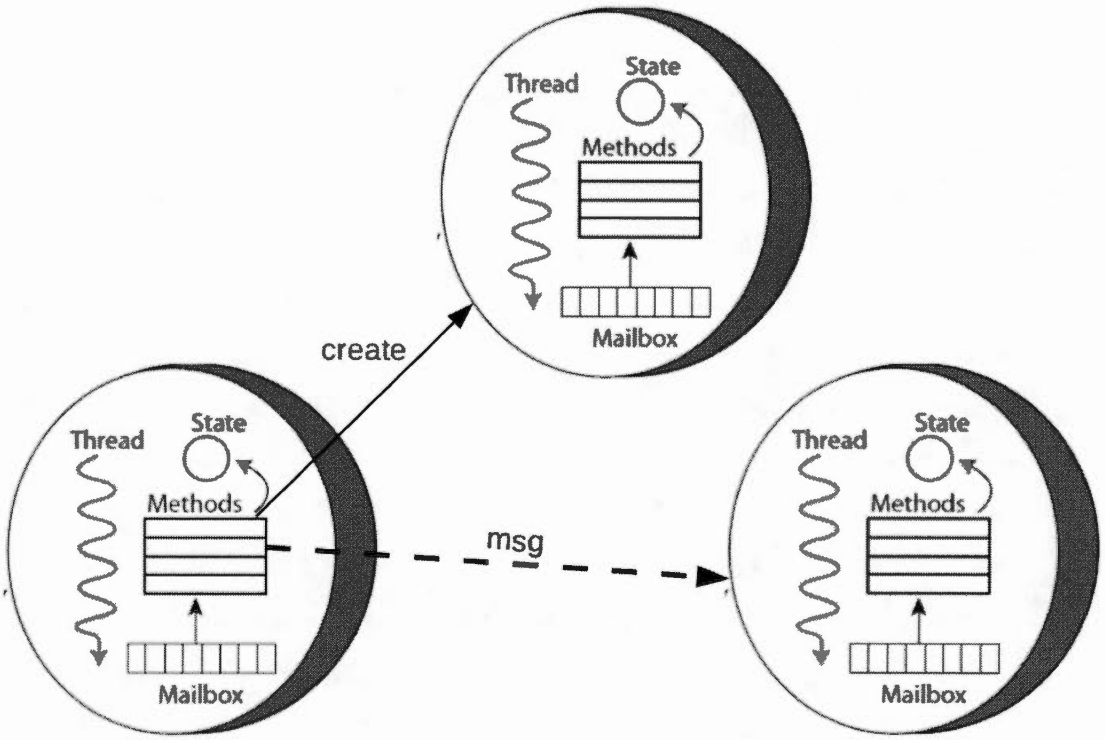

Figure 1.1: Caractéristiques d'un acteur — figure tirée de (Karmani et Agha, 2011)

— Encapsulation de l'état et exécution atomique des messages ;

— Équité ;

— Transparence de localisation ;

Toutes les implémentations du modèle ne respectent pas forcément chacune des propriétés. Des compromis sont faits pour simplifier l'implémentation et garder de bonnes performances.

— Encapsulation et exécution atomique :

Les acteurs ne peuvent pas partager d'état. Comme chaque acteur s'exécute de façon concurrente, si plusieurs acteurs partageaient un état et qu'ils essayaient de le modifier en même temps, cela pourrait créer des situations de compétition. De plus, si un acteur est en train de traiter un message qui va modifier son état, l'exécution du traitement ne doit pas être interrompue par la réception d'un nouveau message, sinon l'acteur pourrait se retrouver dans un état incohérent. L'exécution atomique évite ce problème : un seul message à la fois est traité, sans que l'arrivée d'un autre message interrompe le traitement en cours.

— Équité :

Chaque acteur doit pouvoir progresser s'il a des calculs à faire, et chaque message doit éventuellement être livré à son destinataire, sauf si celui-ci est définitivement désactivé.

— Transparence de localisation :

La transparence de localisation correspond à l'utilisation de noms logiques et uniques pour représenter des adresses physiques. Dans le cadre du modèle acteur, cela veut dire que le nom d'un acteur ne représente pas sa localisation actuelle. En effet, les acteurs communiquent par messages avec d'autres acteurs qui peuvent être n'importe où — sur le même coeur, sur le même processeur, sur une autre machine du réseau, etc. La transparence de localisation pour le nommage des acteurs permet d'abstraire la localisation d'un acteur, et donc le programmeur n'a pas à savoir où se trouvent physiquement les divers acteurs qu'il utilise.

Cette abstraction permet aussi d'apporter une certaine flexibilité, puisque cela

permet de migrer l'exécution des acteurs d'une machine vers une autre. En utilisant cette propriété, le système peut, par exemple, gérer la répartition de la charge de travail ou la tolérance aux pannes.

1.2 Implémentations du modèle acteur

Il existe de nombreuses implémentations, souvent très différentes, du modèle acteur : des langages spécialisés (Armstrong et al., 1993; Thomas, 2014), des extensions de langages et des bibliothèques (Odersky, Venner et Spoon, 2010). Ces implémentations du modèle acteur apparaissent dans différents paradigmes de programmation, les plus populaires étant la programmation fonctionnelle et la programmation orientée objet.

Une des implémentations parmi les plus connues est celle du langage Erlang (Armstrong et al., 1993; Larson, 2009), un langage fonctionnel typé dynamiquement développé chez Ericsson pour la mise en oeuvre de commutateurs téléphoniques robustes.

Plus récemment, Valim a développé le langage Elixir (Thomas, 2014; Elixir, 2017), lui aussi un langage fonctionnel, qui repose sur la machine virtuelle d'Erlang (BEAM), mais qui propose une syntaxe plus simple et plus flexible, inspirée de celle de Ruby.

Le langage Scala (Odersky, Venner et Spoon, 2010), dès sa première version, proposait lui aussi des constructions pour définir et utiliser des acteurs. Depuis, les acteurs d'Akka se sont popularisés. Nous les décrivons plus bas, mais dans le contexte du langage Java.

En pratique, le modèle acteur théorique définit aussi une certaine manière de programmer. Il faut pouvoir décomposer le problème à résoudre en terme de composants communiquant par envoi de messages asynchrones, ce qui ne s'adapte pas forcément à tous les problèmes. De plus, le fonctionnement du modèle est dynamique puisqu'un acteur peut changer de comportement après la réception d'un message, et que l'ordre d'arrivée des messages est indéterminé.

En programmation orientée objet, les objets communiquent par envoi de messages (appels de méthode) et permettent de regrouper un comportement et un état. On peut

directement voir que le modèle objet et le modèle acteur sont susceptibles de bien s'intégrer ensemble. On peut alors considérer un acteur comme étant un objet, mais qui envoie et reçoit des messages de façon asynchrone, et ne partage pas son état directement avec les autres objets. Les messages qu'un acteur peut traiter correspondent alors à ses méthodes (c'est-à-dire à son comportement). Il suffit donc de transformer les appels de méthode (synchrones) en envois de messages (asynchrones). Cela permet de réutiliser plusieurs avantages de la programmation à objets comme l'encapsulation et l'abstraction.

Toutefois, le comportement d'un acteur est dynamique, il peut recevoir n'importe quel type de message, contrairement aux objets classiques.

1.3 Exemple utilisé pour comparer Akka, Celluloid et CelluloNit

Dans la suite de ce mémoire, nous nous concentrons sur l'implémentation du modèle acteur dans des langages de programmation à objets. Dans les deux prochaines sections, nous présentons deux implémentations : Akka, pour Java, et Celluloid, pour Ruby. Dans le chapitre suivant, nous présentons ensuite notre propre implémentation, CelluloNit, pour le langage Nit. Pour comparer ces implémentations, nous allons utiliser un exemple simple qui nous permettra d'illustrer les fonctionnalités d'Akka (Sect. 1.4) et de Celluloid (Sect. 1.5), puis celles de CelluloNit (Chap. 2).

Pour cet exemple, nous considérons simplement une classe A possédant les éléments suivants — voir Fig. 1.2 :

- Un attribut x, de type **Integer** ;
- Une méthode *foo*, prenant un paramètre v de type **Integer**, qui ajoute cette valeur à l'attribut x ;
- Une méthode *bar*, sans paramètre, qui retourne la valeur de l'attribut x de A.

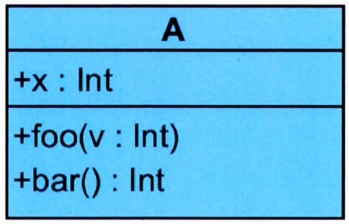

Figure 1.2: Diagramme UML décrivant la classe `A` utilisée dans les exemples.

1.4 Akka

Akka est une implémentation du modèle acteur pour la machine virtuelle Java proposant une API en Java et une en Scala. Dans cette section, nous présentons l'API Java, puisque le language est plus populaire.

Akka est un projet de grande envergure, avec plus de 20 000 *commits* et 461 contributeurs. Ainsi, Akka possède un grand nombre de fonctionnalités et d'extensions qui permettent de s'adapter à différents cas d'utilisation. Dans cette section, nous présenterons les fonctionnalités principales d'Akka — trop aller dans les détails serait inutile dans le cadre de ce mémoire.

1.4.1 Définition d'un acteur

Le Programme 1.1 présente le code pour transformer notre classe `A` d'exemple en acteur Akka, selon les bonnes pratiques suggérées par la documentation du *framework*. Un acteur Akka doit hériter de la classe `AbstractActor` (`extends AbstractActor`). Il doit aussi définir son comportement lors de la réception de messages en redéfinissant (`@Override`) la méthode `createReceive()`. Pour ce faire, Akka fournit une méthode `receiveBuilder()` avec une interface coulante (*fluent interface* (Fowler, 2011)) — plus précisément, un *Expression Builder* (Fowler, 2013). C'est donc l'objet ainsi construit qui va déterminer les messages que l'acteur peut recevoir et comment il les traite — voir plus bas pour des détails additionnels.

Programme 1.1 Définition d'un acteur pour la classe `A` selon les bonnes pratiques officielles Akka.

```
1  public class A extends AbstractActor {
2     private int x;
3
4     public A(int x){
5        this.x = x;
6     }
7
8     static Props props(int x){
9        return Props.create(A.class, () -> new A(x));
10    }
11
12    public static class FooMsg {
13       private int v;
14
15       public FooMsg(int v) {
16          this.v = v;
17       }
18    }
19
20    public void foo(int v) {
21      this.x += v;
22    }
23
24    public static class BarMsg {
25    }
26
27    public int bar() {
28      return this.x;
29    }
30
31    @Override
32    public Receive createReceive() {
33       return receiveBuilder()
34         .match( FooMsg.class, fm -> {
35             this.foo(fm.v);
36          } )
37         .match( BarMsg.class, bm -> {
38             getSender().tell(this.bar(), getSelf());
39          } )
40         .matchAny( msg -> {
41             System.out.println("dont know this message");
42          } )
43         .build();
44    }
45 }
```

La classe `Props` est une classe de configuration permettant de passer des paramètres lors de l'initialisation de l'acteur. Dans notre cas, la classe `A` a besoin qu'on initialise son attribut `x`. La bonne pratique conseillée par Akka est de définir une méthode statique `props` qui prend en paramètre les valeurs nécessaires à l'initialisation de l'objet, et qui retourne une instance de la classe `Props`. Ici, on passe à la méthode statique `Props.create()` le type de la classe qu'on veut instancier (`A.class`) ainsi qu'une lambda-expression qui instancie `A` en appelant le constructeur avec les bons paramètres.

Les classes statiques `FooMsg` et `BarMsg` déclarent les messages que l'acteur peut recevoir. Elles sont utilisées dans la méthode de réception et traitement des messages, créée par l'intermédiaire de `createReceive()`. On utilise des appels à `.match()` pour décrire, via une forme de *pattern-matching* basée entre autres sur des noms de classe, le type du message et le traitement à exécuter pour un message de ce type — traitement décrit à l'aide d'une lambda-expression. Quant à `.matchAny()`, on l'utilise pour décrire le cas par défaut, i.e., lorsque le message reçu n'est reconnu par aucune des autres clauses. Dans cet exemple, l'implémentation du traitement des messages se trouve donc dans les lambda-expressions apparaissant dans la méthode générée par le `receiveBuilder`, lesquelles expressions utilisent les méthodes `public void foo(int v)` et `public int bar()`. Le traitement du message pour la méthode `foo` est simple, puisqu'on appelle simplement cette méthode avec l'argument approprié, provenant de l'objet `fm`, de classe `FooMsg`. Par contre, le traitement pour la méthode `bar` est un peu plus complexe, puisqu'on doit retourner, de façon asynchrone, un résultat à l'acteur appelant : voir plus loin pour des explications à ce sujet (p. 16).

1.4.2 Utilisation d'un acteur

Nous décrivons maintenant comment créer et utiliser des acteurs lorsque la classe appropriée est définie. Pour ce faire, nous utilisons le Programme 1.2.

Programme 1.2 Création et envoi des messages pour des appels à `foo` et `bar` à un acteur de type `A` en Akka.

```
1 // Création du système d'acteurs.
2 final ActorSystem system =
3    ActorSystem.create("Simplesystem");
4
5 // Création d'un acteur pour A avec une valeur initiale de 0.
6 final ActorRef aActor =
7    system.actorOf(A.props(0), "a");
8
9 // Envoi d'un message pour la méthode foo(10).
10 aActor.tell(new FooMsg(10), getSelf());
11
12 // Envoi d'un message pour la méthode bar().
13 Timeout timeout =
14    new Timeout(Duration.create(5, "seconds"));
15 Future<Object> future =
16    Patterns.ask(aActor, new BarMsg(), timeout);
17 Integer result = (Integer) Await.result(future, timeout.duration());
18 System.Out.println(result);
19
20 // Terminaison du système d'acteurs.
21 aActor.tell(akka.actor.PoisonPill.getInstance(), ActorRef.noSender());
22 system.shutdown();
```

Création du système d'acteurs

En Akka, tout est acteur, et tous les acteurs doivent faire partie d'un *système d'acteurs*. Un système d'acteur est une structure qui alloue de 1 à N threads, où N est un paramètre de configuration — par défaut, N = nombre de processeurs disponibles. En général, on crée un seul système d'acteurs par application. Dans un système d'acteurs Akka, les acteurs forment une hiérarchie de supervision, où chaque acteur est supervisé par au moins un parent. Nous n'entrerons pas dans les détails de la supervision d'Akka. Toutefois, il est important de comprendre que la création d'un système d'acteurs Akka avec `ActorSystem.create()` initialise la hiérarchie d'acteurs de l'application. Notamment, trois acteurs nécessaires au bon fonctionnement du système sont crées :

— Le gardien utilisateur `/user`. Il est le parent de tous les acteurs créés par le programme. Si le gardien utilisateur signale une erreur, le gardien racine le termine, ce qui arrête le système d'acteurs au complet.
— Le gardien système `/system`. Il surveille le gardien utilisateur et les acteurs systèmes. Il permet d'arrêter proprement le système en gardant le système de journalisation actif pendant l'arrêt, ce dernier étant lui aussi réalisé par des acteurs.
— Le gardien racine `/`, qui supervise tous les acteurs spéciaux — `/user`, `/system`, `/deadLetters`, `/temp`, `/remote` — les arrêtant à la moindre exception.

Instanciation d'un acteur

Une fois que l'on possède une référence vers un système d'acteurs, on peut commencer à créer des acteurs de haut niveau. Pour cela, on utilise la méthode usine `actorOf` disponible dans les classes `ActorSystem` et `ActorContext` — celle d'`ActorContext` est utilisée lorsque l'on crée un acteur enfant depuis l'intérieur d'un acteur existant. La méthode `actorOf()` prend en paramètre une instance de la classe de configuration `Props`, et un nom (optionnel) pour l'acteur. L'appel `actorOf()` retourne une instance de type `ActorRef` qui est le seul moyen d'interagir avec un acteur. Une instance d'`ActorRef`

est immuable, sérialisable et contient des informations qui permettent de l'utiliser à travers le réseau de façon transparente. Les acteurs sont automatiquement démarrés de façon asynchrone lors de leur création. En Akka, un acteur n'est pas directement relié à un *thread* système, le *runtime* se charge de gérer la répartition du travail sur un ensemble de *threads* — par défaut, le nombre maximum supporté par la machine sur laquelle le programme s'exécute — qu'il contrôle.

Envoi de messages

Deux méthodes permettent d'envoyer des messages à un acteur :

- `tell` : envoie un message asynchrone et retourne immédiatement ;
- `ask` : envoie un message asynchrone et retourne un `Future` représentant une réponse éventuelle.

La méthode `tell` est celle recommandée, puisqu'elle permet d'envoyer le message et de retourner de façon immédiate, sans se soucier d'attendre une réponse éventuelle. Elle prend en paramètre le message à envoyer et la référence de l'envoyeur. Cette dernière est obtenue avec la méthode `getSelf()`, car l'utilisation de `this` doit être évitée du fait qu'elle expose l'état interne d'un acteur. Avec `getSelf()`, on envoie alors l'`ActorRef`, qui encapsule correctement la référence vers l'acteur. Lorsqu'un acteur reçoit un message, il peut alors accéder à la référence de l'expéditeur via la méthode `getSender()`.

Quant à la méthode `ask`, elle permet d'obtenir, possiblement avec une attente, le résultat d'un message par l'intermédiaire d'un `Future`. Elle prend en paramètre l'acteur à qui le message est destiné, le message à envoyer, et une échéance (*timeout*) après laquelle une exception sera signalée. Pour récupérer la valeur d'un `Future` en Akka, on utilise `Await.result` qui prend en paramètre le `Future` en question et une durée. Le `Future` retourné par `ask` est typé avec `Object`, d'où la conversion en `Integer` nécessaire à la récupération du résultat.

Arrêt d'un acteur

Il est de la responsabilité du programmeur d'arrêter les acteurs en Akka. Comme ce sont des entités asynchrones autonomes, les acteurs ne peuvent pas décider d'eux-même de s'arrêter et libérer les ressources qu'ils consomment. Pour arrêter un acteur, il existe plusieurs moyens :

— Utiliser la méthode `stop` d'`ActorSystem` ou `ActorContext`.

— Envoyer un message de classe `akka.actor.PoisonPill` ;

— Utiliser la méthode `gracefulstop`.

La méthode `stop` prend en paramètre une instance d'`ActorRef`. L'acteur à qui on demande de s'arrêter termine le traitement de son message courant avant de s'arrêter. S'il existe d'autres messages dans la *mailbox*, ils ne seront pas traités. L'arrêt s'effectue de façon asynchrone, en deux étapes. Tout d'abord, l'acteur termine tous ses enfants, et attend leur notification d'arrêt. Ensuite, il se termine lui-même — il libère la *mailbox*, puis publie le message `Terminated` pour indiquer à ses superviseurs qu'il s'est arrêté. Si l'un de ses enfants ne répond pas — par exemple, parce qu'il est occupé à traiter un long message et ne reçoit pas le message d'arrêt — alors tous le processus sont bloqués. Après l'arrêt complet d'un acteur, la méthode `postStop` est appelée, permettant de libérer d'éventuelles ressources toujours en sa possession.

L'envoi d'un message `akka.actor.PoisonPill` stoppera l'acteur lors de son traitement par l'acteur. Un tel message est donc mis dans la file de la *mailbox* comme n'importe quel autre message, et sera traité après tous les messages déjà reçus par l'acteur.

Finalement, la méthode `gracefulStop` permet d'attendre la terminaison, et ainsi composer une terminaison ordonnée de plusieurs acteurs. Lorsque `gracefulStop` retourne avec succès, la méthode `postStop` de l'acteur arrêté a été exécutée.

1.4.3 Autres fonctionnalités d'Akka

Dans les sous-sections précédentes, nous nous sommes concentrés sur la description des éléments les plus importants d'Akka dans le cadre de ce mémoire.

Akka est un *framework* qui offre de nombreuses possibilités. Nous avons notamment omis la partie supervision, qui permet aux applications Akka de s'auto-régénérer selon des stratégies de supervision configurables. La structure des acteurs Akka permet notamment d'avoir la possibilité de les redémarrer sans avoir à relancer complètement l'application en cours d'éxécution. Il est aussi possible de configurer les *mailboxs* des acteurs pour modifier la stratégie de gestion des messages reçus. De même, les acteurs Akka sont des machines à état, et on peut changer dynamiquement leur comportement, c'est-à-dire, leur boucle de réception et traitement de messages. Avec Akka viennent aussi un grand nombre de configurations possibles permettant de gérer la mise à l'échelle d'applications, pour qu'elles fonctionnent de façon distribuée, notamment en assurant un certain équilibre de la charge de travail. Il existe aussi un version *typée* d'Akka, les `TypedActor`, mais dont l'API est au moins aussi verbeuse que la version non typée — deux interfaces et une classe pour définir un acteur typé. Et de nombreux autres modules d'Akka apportent des fonctionnalités diverses, que nous ne décrivons pas ici.

1.5 Celluloid

Celluloid (Celluloid, 2017) est une implémentation du modèle acteur en Ruby (Thomas, Hunt et Fowler, 2013), qui s'éxécute avec JRuby sur la machine virtuelle Java. C'est un projet plus modeste qu'Akka, mais c'est celui dont nous nous sommes inspirés le plus pour CelluloNit, l'API de Celluloid étant plus proche de l'objet. Dans cette section, nous présentons les fonctionnalités principales de Celluloid, comme nous l'avons fait dans la section précédente avec Akka.

Programme 1.3 Définition d'un acteur Celluloid.

```
1 class A
2    include Celluloid
3
4    def initialize(x)
5      @x = x
6    end
7
8    def foo(v)
9      @x += v
10   end
11
12   def bar
13     @x
14   end
15 end
```

1.5.1 Définition d'un acteur

La définition d'un acteur en Celluloid est illustrée dans le Programme 1.3. Pour ce faire, il suffit de définir une classe Ruby normale qui inclut le module `Celluloid` (`include Celluloid`).

Les instances de cette classe deviennent alors des acteurs, appellées `cells` en Celluloid. Il n'est pas nécessaire de coder la boucle de réception des messages comme en Akka, puisque *les messages des acteurs correspondent directement aux méthodes.* Cette propriété est particulièrement intéressante et nous l'avons préservée dans CelluloNit.

1.5.2 Utilisation d'un acteur

Le Programme 1.4 illustre la création d'un objet acteur `a` de la classe `A`, ainsi que des appels à ses méthodes `foo` et `bar`, tant des appels synchrones que des appels asynchrones.

Comme l'illustrent les deux premiers appels, on peut utiliser un objet acteur comme un objet normal, en faisant des appels classiques à ses méthodes. Ces appels sont synchrones : l'appelant attend leur complétion avant de continuer son exécution.

Si l'on souhaite envoyer un message *asynchrone* (et non bloquant) à l'acteur, on utilise

Programme 1.4 Utilisation d'un acteur Celluloid.

```
1 # Création d'un acteur a.
2 a = A.new(0)
3
4 # Appels synchrones.
5 a.foo(10)
6 r = a.bar
7 print r # Affiche 10
8
9 # Appels asynchrones.
10 a.async.foo(10)
11
12 f = a.future.bar
13 x = f.value
14 print x # Affiche 20
15
16 # Terminaison de l'acteur a.
17 a.terminate
```

la méthode **async**, sur laquelle on appelle la méthode désirée. L'appelant continue alors son exécution, sans attendre la réponse de l'appel de méthode. La méthode est ensuite exécutée par l'objet concurrent qui reçoit le message, sur un autre fil d'exécution, et donc de façon concurrente. Les méthodes éxécutées par l'intermédiaire d'**async** ont la même signature que celles de la classe qui inclut Celluloid.

Les appels asynchrones ne signalent jamais d'exception. Si l'exécution du message produit une exception, le receveur *plante* et les appels asynchrones suivants ne signaleront pas d'exception non plus.

Si on souhaite envoyer un message asynchrone à un acteur **et** en récupérer le résultat, on peut utiliser la méthode `future` sur une classe ayant inclus Celluloid. L'appel retourne alors immédiatement un objet de type `Celluloid::Future`. Pour obtenir le résultat, il suffit alors d'appeler la méthode `value` sur l'instance du `Future`. Le cas échéant, ce dernier appel bloque tant que la valeur n'a pas été produite par l'exécution asynchrone du message envoyé. Dans le cas de l'utilisation de `Future`, si une exception est signalée lors de l'exécution du message, elle est signalée à nouveau par l'appel à `future.value`.

En Celluloid, un acteur est associé à un *thread* système. Ainsi, un appel à `A.new(0)` crée

un *thread* système en arrière-plan. Contrairement à des objets réguliers, les acteurs ne sont pas collectés par le ramasse-miettes, même lorsqu'ils ne sont plus référencés. Il faut donc les arrêter explicitement, si l'on ne veut pas qu'ils continuent à exister indéfiniment. Pour cela, on utilise la méthode `terminate` qui envoie un message asynchrone à l'acteur, lui demandant de se terminer proprement.

1.5.3 Autres fonctionnalités

Celluloid fournit d'autres mécanismes intéressants, nous en décrivons quelques-uns ici, qui nous semblent les plus importants.

Registre des acteurs

La méthode `Celluloid::Actor[]` enregistre un acteur sous un nom symbolique. Cela permet d'y accéder par son nom plutôt que par une référence.

Groupes de threads

Lors de la création d'un acteur — donc lors de l'instanciation d'une classe incluant `Celluloid` —, on peut choisir d'utiliser la méthode de classe `pool` à la place de `new`. Cette méthode retourne un proxy qui délègue les appels à un *groupe* de `cells`, de la taille du nombre de processeurs/coeurs disponibles sur la machine par défaut — la taille du pool est configurable. C'est-à-dire que l'on obtient une référence qui nous permet d'envoyer des messages à une *mailbox* partagée par plusieurs acteurs — où un acteur = un *thread*. Les appels synchrones restent synchrones, mais les appels asynchrones sont distribués entre les différents *threads* du *groupe*. Cela permet d'effectuer des calculs en parallèle, par exemple, en utilisant un nombre *threads* approprié pour la machine. La création d'un *groupe* est configurable, par exemple, on peut indiquer le nombre de *threads* à utiliser. Une note intéressante est qu'un *groupe* est automatiquement terminé lors de sa collection par le ramasse-miettes, donc il n'est pas nécessaire de terminer explicitement les *groupes*.

Supervision des acteurs et exceptions

Une exception pendant le traitement d'un message fait *planter* l'acteur qui le traitait. Si un acteur doit être notifié de l'arrêt d'un autre acteur, il peut utiliser la méthode `link`, qui prend en paramètre une référence vers l'acteur à surveiller. Il est alors possible d'enregistrer un *callback* à l'aide de la méthode `:trap_exit`, laquelle sera appelée lors de la terminaison des acteurs surveillés. On peut aussi utiliser la méthode `new_link`, qui permet d'initialiser un acteur et de le surveiller en même temps. Sans la définition d'un *callback* permettant d'être notifié de l'arrêt d'un acteur surveillé, tous les acteurs liés sont arrêtés à leur tour — l'exception est propagée dans la hiérarchie des acteurs. Si nécessaire, la méthode `unlink` supprime le lien entre acteurs.

Si on veut redémarrer des acteurs terminés à cause d'une exception, on peut utiliser la méthode `supervise` à la place de `new` pour la création d'un acteur. Elle possède la même signature que `new`, mais retourne un superviseur en plus de créer l'acteur. Le superviseur se charge de redémarrer les acteurs qui *plantent*, avec les mêmes arguments que lors de leur première création.

1.6 Récapitulatif et discussion

Akka et Celluloid implémentent tous deux le modèle acteur, mais chacun à sa façon. Akka est basé sur Java et Scala, qui sont tous deux des langages statiques, et ils doivent donc proposer une interface de programmation statique. De son côté, Celluloid est écrit en Ruby, un langage enitèrement dynamique, lui donnant plus de liberté pour l'interface de programmation fournie.

En Akka, lorsqu'il définit des acteurs, le programmeur est obligé de programmer explicitement les tests pour identifier le type des messages reçus, dans le but de déterminer le comportement à adopter. De plus, s'il veut rendre son code plus clair et compréhensible, les bonnes pratiques d'Akka font en sorte qu'il doit lui-même déclarer une classe statique, pour chacun des types de message. Concrètement, du code acteur écrit en Akka est donc assez éloigné d'un code objet classique, et est beaucoup plus « verbeux ».

En Celluloid, l'inclusion du module `Celluloid` modifie complètement le comportement d'une classe. L'aspect dynamique de Ruby fait en sorte qu'il suffit de connaitre les nouveaux services fournis par la classe incluant le module `Celluloid` pour pouvoir l'utiliser. De plus, les services d'une classe incluant `Celluloid` sont peu modifiés. Bien sûr, cela vient avec le désavantage de ne pas être certain de la justesse du code tant que l'on a pas exécuté le programme, puisqu'aucune analyse statique n'est effectuée. L'aspect le plus intéressant de l'interface fournie par Celluloid est que l'utilisation d'acteurs n'est pas trop éloignée de l'utilisation d'objets classiques : ce sont les méthodes exposées par un acteur qui correspondent aux différents messages qu'il peut recevoir, et le programmeur n'est pas obligé d'implémenter la boucle de réception des messages lui-même.

En conclusion, Akka est plus complexe et verbeux à utiliser que Celluloid, même s'il ne faut pas oublier qu'Akka fournit plus de fonctionnalités et est conçu pour être plus robuste. Notre objectif initial étant la facilité d'utilisation, nous avons préféré nous inspirer du modèle de Celluloid pour concevoir notre solution, CelluloNit, dont nous présentons l'API dans le prochain chapitre.

CHAPITRE II

API DES ACTEURS CELLULONIT

Dans ce chapitre, nous présentons l'API des acteurs CelluloNit. Notamment, nous présentons un petit exemple pour en illustrer l'utilisation et l'effet des diverses opérations.

L'objectif principal de l'API des acteurs CelluloNit est de rendre la programmation parallèle et concurrente facile à utiliser par le programmeur. Elle est aussi conçue pour que l'on puisse passer de la programmation séquentielle à la programmation parallèle et concurrente rapidement, et ce sans apporter de nombreuses modifications au programme original. Pour cela, l'API CelluloNit est fortement inspirée de celle de Celluloid, qui correspond plus à un modèle de programmation orientée objet classique que l'API Akka. En outre, contrairement à Celluloid, l'API Nit bénéficie du typage statique qui facilite la programmation puisqu'on peut détecter de nombreuses erreurs plus tôt, lors de la compilation.

2.1 Principaux éléments de l'API des acteurs CelluloNit

Les principaux éléments de l'API des acteurs CelluloNit sont les suivants :

- `actor` : Annotation utilisée dans une classe pour en faire un acteur ;
- `async` : Méthode pour que l'appel de méthode qui suit soit effectué de façon aynchrone ;
- `Future[T]` : Retourne éventuellement une valeur de type `T` ;

— `join` : Méthode pour obtenir — en bloquant si nécessaire — la valeur associée à un objet `Future` ;

— `terminate` : Méthode pour arrêter l'exécution d'un acteur ;

— `terminate_now` : Idem ;

— `wait_termination` : Méthode pour attendre qu'un acteur se termine ;

— `active_actors` : Attribut qui permet, via la méthode `wait`, d'attendre la terminaison des acteurs actifs.

Nous décrivons plus en détail ces divers éléments dans les sections qui suivent, en commençant par décrire brièvement le langage Nit.

2.2 Le language Nit

Nit est un langage de programmation à objets statiquement typé. Il se veut facile à prendre en main et possède une syntaxe élégante proche de Ruby — avec possiblement des déclarations de type en plus. Son typage implicite et adaptatif participe grandement à cette élégance. Le typage implicite fait en sorte que le programmeur peut, dans de nombreux cas, ne pas déclarer le type d'une variable qu'il utilise, alors que le typage adaptatif permet que le type associé à une variable change, ce que le compilateur détecte selon les diverses affectations effectuées sur la variable.

Un programme Nit est composé d'un ensemble de module (partiellement ordonné par les imports) qui définissent ou ajoutent du comportement — voir Fig. 2.1. Un module est composé de plusieurs classes et d'un programme `main` optionnel, comme en Ruby. Lorsqu'un module en importe un autre, il peut :

— Déclarer de nouvelles classes ;

— Sous-classer les classes importées ;

— Raffiner les classes importées.

Le raffinement de classe de Nit, exprimé avec le mot clé `redef`, offre la possibilité de *réouvrir* une classe déjà définie, ce qui permet de modifier son comportement de diverses

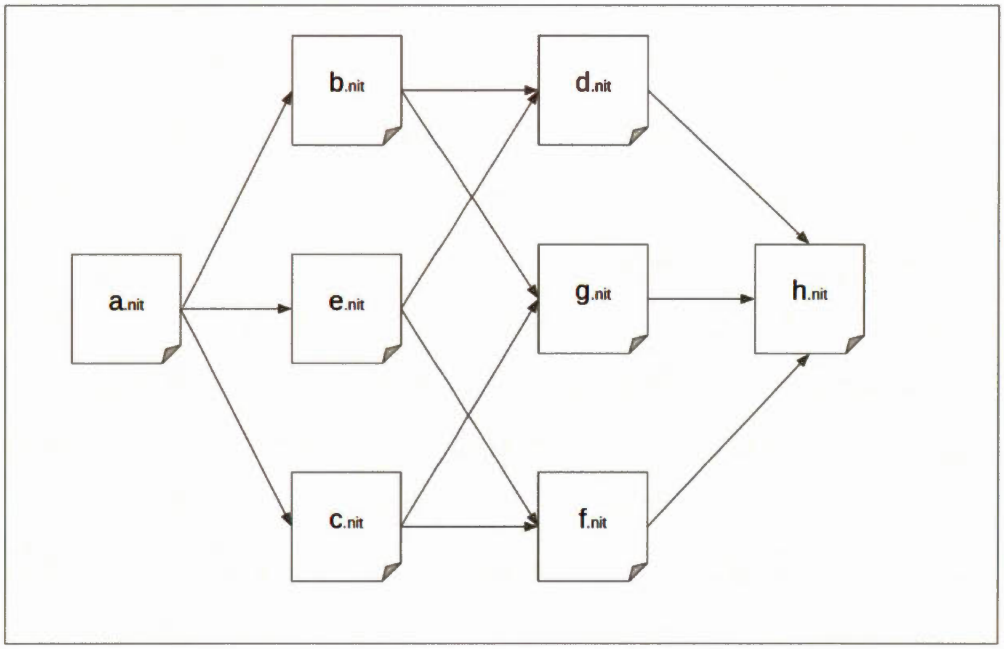

Figure 2.1: Organisation générale d'un programme Nit composé de plusieurs modules.

façons :

- En ajoutant des super-classes ;
- En ajoutant des propriétés — attributs ou méthodes ;
- En modifiant des méthodes existantes, i.e., en changeant leur implémentation.

Dans ce dernier cas, on peut utiliser le mot-clé `super`, pour faire appel à l'implémentation précédente de la méthode redéfinie.

En Nit, il existe deux types d'annotations : celles définies par le langage et celles définies par le programmeur. Le deuxième type permet à un développeur de bibliothèque Nit de définir une nouvelle annotation dans un module. Par la suite, le développeur de la bibliothèque peut mettre en oeuvre une phase spécifique du compilateur pour gérer cette annotation de la façon requise pour implémenter sa sémantique.

2.3 Définition d'un acteur

Le module `actors.nit`, inclus dans la bibliothèque standard de Nit, introduit une annotation `actor` utilisable dans la définition d'une classe. Il suffit donc d'annoter une classe avec `actor` pour que les diverses instances de cette classe deviennent automatiquement des acteurs, rendant ainsi disponibles les autres opérations de notre API.

Reprenons notre exemple avec la classe `A` :

```
1 class A
2     actor
3
4     var x: Int
5
6     fun foo(v: Int) do self.x += v
7     fun bar: Int do return self.x
8 end
```

Dans cet exemple, la seule différence par rapport à un code Nit sans acteur est l'ajout de l'annotation `actor`. Les instances de la classe `A` seront donc des acteurs.

2.4 Utilisation des méthodes `async` et `join`

Maintenant, voyons comment on peut utiliser les instances de la classe `A` comme des acteurs. L'annotation `actor` donne accès à une nouvelle méthode `async` dans les classes annotées, laquelle méthode permet ensuite d'envoyer des messages à un acteur de façon *asynchrone*. Plus spécifiquement, l'envoi d'un message précédé de `async` devient une opération *non bloquante*, i.e., elle retourne à l'appelant de façon immédiate. L'expéditeur du message peut donc continuer à effectuer ses propres traitements indépendants, puisque les messages sont traités par l'acteur de façon concurrente, sur son propre fil d'exécution. Signalons que les divers messages envoyés à l'acteur sont traités dans l'ordre de réception, avec les contraintes suivantes :

- Si un objet `a` envoie de façon asynchrone un premier message `m1` puis un deuxième message `m2`, le message `m1` sera traité *avant* `m2` ;
- Si un objet `a1` envoie un message `m1` et un autre objet `a2` envoie un message `m2`.

l'ordre de traitement est *indéterminé.*

Toujours dans l'objectif de garder l'API acteur de Nit la plus élégante et simple possible, la méthode **async** d'une classe annotée avec **actor** fait en sorte que les appels asynchrones des méthodes de la classe annotée ont presque le même comportement que les méthodes originales sous réserve des règles suivantes :

- Si une méthode n'a pas de type de retour, la signature de la version asynchrone ne change pas ;
- Si une méthode possède un type de retour, la signature de la version asynchrone est modifiée au niveau de l'acteur. Dans ce cas, la méthode de la classe de l'acteur retournera un **Future** du type de retour de la méthode de la classe annotée.

Dans le cadre de cet exemple, **a.async.bar** retourne donc un résultat de type **Future[Int]** puisque **a.bar** retourne un **Int**, tel qu'illustré par le segment de code suivant :

```
1 var a = new A(0)
2 a.async.foo(10) # Appel asynchrone sans retour de résultat
3
4 var future = a.async.bar # Appel asynchrone retournant un Future[Int]
5 var value = future.join  # Appel bloquant sur le Future
6 print value # Affiche 10
```

Un **Future** représente la promesse d'un résultat. Récupérer un **Future** permet de choisir de récupérer son résultat au moment où on le souhaite, quand on en a besoin... *si celui-ci est disponible.* Donc, pour un **Future**, la méthode **join** est bloquante tant que le résultat n'a pas été calculé.

Exemple illustrant les objets ***Future*** *: Parallélisme récursif*

Illustrons ce mécanisme avec un exemple plus concret, soit un acteur avec une méthode pour calculer la somme des éléments d'un tableau. Le Programme 2.1 présente le code d'un tel exemple.

Remarque : Pour simplifier l'exemple, nous avons ignoré les cas avec un nombre impair

Programme 2.1 Programme CelluloNit pour effectuer la somme des éléments d'un tableau avec du parallélisme récursif.

```
1  class SommeTableau
2      actor
3
4      var seuil: Int
5
6      fun somme_recursive(tab: Array[Int]): Int do
7          if tab.length <= seuil then
8              var r = 0
9              for x in tab do r += x
10             return r
11         else
12             var mid = tab.length / 2
13             var child = new SommeTableau(seuil)
14             var s1 = child.async.somme_recursive(tab.sub(0, mid))
15             var s2 = somme_recursive(tab.sub(mid, tab.length))
16             return s1.join + s2
17         end
18     end
19 end
20
21 var tab = [1,2,3,4,5,6,7,8,9,10]
22 var s = new SommeTableau(5)
23 var r = s.async.somme_recursive(tab)
24 print r.join
```

d'éléments.

Dans cet exemple, on définit une classe `SommeTableau` avec une méthode `somme_recursive` qui fait la somme des éléments d'un tableau `tab`. Pour un tableau de grande taille, il est intéressant de paralléliser le traitement. Dans le cadre de cet exemple, une approche diviser-pour-régner illustre bien le parallélisme et les `Futures`. Lorsque la taille du tableau est suffisamment petite, la somme est faite de façon séquentielle et itérative (boucle `for`). Par contre, si la taille du tableau est supérieure à un certain seuil, la méthode `somme_recursive` va créer un nouvel acteur pour effectuer la somme d'une des moitiés du tableau, et ce en effectuant un appel récursif de façon asynchrone. Pendant ce temps, un autre appel est fait récursivement, pour effectuer la somme de la moité restante. Dans le cadre de cet exemple, il est nécessaire d'utiliser un `Future` pour récupérer la valeur calculée par l'appel récursif asynchrone.

Remarques sur l'utilisation combinée d'appels synchrones et asynchrones

Il est à noter qu'en présence d'appels synchrones combinés à des appels asynchrones, on peut se retrouver face à des situations de compétition – c'est-à-dire si on choisit d'instancier un objet d'une classe acteur et de l'utiliser à la fois comme un objet normal **et** comme un acteur. En effet, pour garder une interface la plus simple possible avec le moins de modifications au code écrit par le programmeur, nous avons choisi de ne pas interdire les appels normaux sur les instances de la classe annotée. Donc, dans notre exemple où la méthode `a` modifie l'attribut `x`, si on écrivait `a.foo(10)`, ce serait alors un appel purement *synchrone* sur l'objet `a`. Par contre, si juste avant on avait écrit `a.async.foo(10)`, on aurait alors un appel *asynchrone*, et donc il est possible que l'on ait une situation de compétition quant à la modification de l'attribut `a.x`.

D'autre part, on peut utiliser une classe annotée `actor` dans un programme sans se soucier du fait qu'il s'agit d'un acteur. L'instanciation des objets requis par la mise en oeuvre — proxy, *mailbox*, *thread* système associé : voir Chapitre 3 — se fait *de façon paresseuse*, donc uniquement lorsque l'on utilise la méthode `async` de la classe annotée.

Ainsi, on peut tout à fait à la fois utiliser des instances de la classe `A` comme des acteurs, et d'autres instances comme des objets normaux dans la même application, sans surcoût pour les utilisations simplement synchrones.

2.5 Cycle de vie d'un acteur

Chaque acteur Nit est lié à un *thread* système, et les ressources que ce *thread* occupe ne sont pas libérées automatiquement par le ramasse-miettes. Ainsi, le cycle de vie d'un acteur doit être géré à la main par le progammeur. Afin de libérer les ressources liées à un acteur, l'API Nit propose plusieurs solutions, illustrées ci-bas en prenant en compte la classe `A` présentée précédemment (avec les méthodes `foo` et `bar`) :

- `a.async.terminate` envoie un message spécifique qui ordonne de libérer le *thread* système sous-jacent. Ce message est envoyé comme n'importe quel autre message, i.e., il est mis à la suite des autres messages déjà reçus par l'acteur. Ainsi, si l'acteur a de nombreux messages en attente à traiter (dans sa *mailbox*), il les traitera tous avant de s'arrêter.
- `a.async.terminate_now` fait la même chose que le message `terminate`, à ceci près que le message est inséré pour être traité immédiatement *après* le message en cours, donc en court-circuitant les appels en attente.
- `a.async.wait_termination` est un appel qui bloque l'appelant tant que l'acteur n'a pas terminé. Cela permet par exemple d'être certain qu'un acteur a terminé son travail avant de continuer l'exécution du programme.

Dans l'implémentation actuelle, même si on peut libérer le *thread* système qui traite les messages reçus par l'acteur, les autres ressources (e.g., *mailbox*) ne sont pas libérées pour autant. Ainsi, on peut toujours envoyer des messages à un acteur terminé, mais ceux-ci ne seront jamais traités — ils ne feront qu'occuper de la mémoire au fur et à mesure que la *mailbox* se remplit. En revanche, si l'instance de la classe annotée utilisée pour communiquer avec l'acteur via `async` n'est plus référencée plus tard dans le programme, et que l'acteur correspondant a terminé, alors toutes les ressources seront libérées — la

mailbox, l'instance de la classe annotée, etc.

2.6 Arrêt du système d'acteurs

Pour le modèle d'acteurs de Nit, nous nous sommes posé la question de l'arrêt d'un programme utilisant des acteurs. Si on laisse entièrement la responsabilité de la terminaison au programmeur, la gestion de l'arrêt peut s'avérer difficile. En effet, le programmeur est obligé d'ajouter des mécanismes de synchronisation uniquement pour gérer l'arrêt du programme. Cela serait contraire à notre objectif de rendre la programmation avec les acteurs la plus simple possible.

Par conséquent, nous avons décidé de fournir un mécanisme automatique qui se charge de terminer le programme seulement lorsqu'on est certain que tout le travail est terminé. Pour cela, on détermine qu'il n'y a plus rien à faire lorsque tous les acteurs du programme sont inactifs — un acteur est considéré inactif s'il n'a plus de messages dans sa *mailbox* et qu'il n'est pas en train de traiter un message. Ce mécanisme permet à un programmeur d'écrire un programme dont le `main` ne fait que créer des acteurs et fait des appels asynchrones, sans que son programme ne se termine prématurément. Si un tel mécanisme n'était pas disponible, à la fin du `main` le programme se terminerait et libèrerait les ressources, sans se soucier des acteurs travaillant sur leurs propres *threads* en arrière-plan.

Par comparaison, notons qu'en Akka, il faut explicitement arrêter le système à la main, sinon il fonctionne indéfiniment. Et en Celluloid, à la fin du `main` d'un programme, les acteurs du systèmes sont arrêtés brutalement, ce qui force le programmeur à gérer lui-même l'arrêt, mais cette fois en ajoutant un mécanisme de synchronisation pour empêcher le programme de se terminer.

2.7 Synchronisation d'acteurs

Dans notre API, nous avons aussi introduit une variable globale `active_actors` qui permet de faire de la synchronisation globale au niveau des acteurs. C'est cette même

variable qui est utilisée pour gérer l'arrêt du système. Sa particularité est qu'elle possède une méthode `wait` qui bloque tant que des acteurs sont encore actifs. Ainsi, si le programmeur a besoin de lancer une série de calculs en parallèle, puis d'attendre qu'ils soient tous terminés avant de continuer l'exécution de son programme, il peut utiliser l'instruction `active_actors.wait` pour créer un point de synchronisation global.

2.8 Héritage et redéfinition

L'un des avantages des acteurs Nit est le fait qu'ils sont des objets normaux, *augmentés par le processus de compilation* — voir chapitre 3. Il est donc normal de considérer jusqu'à quel point on peut leur appliquer les concepts objets classiques. En l'occurrence, on s'est intéressé à l'héritage et la redéfinition d'acteurs.

Pour le moment, le modèle d'acteurs de Nit permet le raffinement (de classes) d'acteurs, *mais ne permet pas l'héritage.*

Par exemple, supposons qu'on veuille ajouter une méthode `baz(v: Int): Int` qui combine les deux méthodes déjà présentes dans notre classe `A`. Il suffit alors d'écrire le code suivant :

```
1 redef class A
2     fun baz(v: Int): Int do
3         self.x += v
4         return self.x
5     end
6 end
```

Comme la classe `A` était déjà un acteur, on va juste rajouter le message correspondant à la nouvelle méthode `baz` en raffinant, lors de la compilation, le module comprenant le code précédent.

Quelques question se posent quant à l'API fournie au programmeur pour redéfinir des acteurs ou sous-classer des acteurs en Nit. D'une part, est-ce qu'un programmeur qui raffine une classe annotée `actor` veut aussi raffiner l'acteur correspondant ? D'autre part, si un programmeur fait une sous-classe d'une classe annotée `actor`, veut-il réellement spécialiser l'acteur ou la classe correspondante ? Une réponse possible à ces questions

serait de générer le code acteur pour ce genre de cas seulement si l'annotation `actor` est répétée lors de la redéfinition ou de la spécialisation d'une telle classe. Néanmoins, la sous-classe hériterait quand même de la méthode `async` de la super-classe annotée `actor`, cette solution ne serait donc pas parfaite.

CHAPITRE III

MISE EN ŒUVRE DES ACTEURS CELLULONIT

Dans ce chapitre, nous présentons certains choix et techniques mis en oeuvre pour réaliser l'API des acteurs CelluloNit présentée dans le chapitre précédent. L'implémentation du modèle acteur Nit repose à la fois sur l'utilisation d'un module de la bibliothèque standard du langage et sur un support au niveau du compilateur sous la forme de deux phases, une première faisant de l'*injection de modèle* (Section 3.3), l'autre effectuant de la génération de code (Section 3.5).

3.1 Explications préliminaires

Depuis le début, notre objectif est de fournir une API simple. Ainsi, nous voulions en fournir une similaire à Celluloid, mais statiquement typée. Cela veut dire que le programmeur utilisant CelluloNit doit avoir un moyen simple et rapide de transformer ses classes en acteur, en ayant le moins de code supplémentaire à écrire. Pour cela, nous avons introduit l'annotation `actor`, que le programmeur utilise sur ses classes pour qu'elles deviennent des acteurs. Cette annotation a pour effet de fournir une nouvelle méthode aux classes annotées, `async`, qui permet de faire de l'envoi de messages asynchrones. Néanmoins, fournir l'envoi de messages asynchrones via `async` implique que le code d'implémentation d'un acteur et de la communication avec un acteur est caché au programmeur. Nous avons donc commencé par définir un modèle objet solide, pas plus intrusif pour les classes annotées que l'ajout d'`async`, mais qui permette de faire fonctionner des acteurs. Plusieurs essais et astuces d'implémentation plus tard, nous avons

figé ce modèle, et "remonté" ses composantes abstraites dans un module `actors.nit`, dans la bibliothèque standard, qui nous sert aussi à introduire l'annotation `actor`. Nous détaillons le modèle objet des acteurs CelluloNit plus loin.

Cependant, Nit ne possède pas de mécanisme équivalent à l'inclusion d'un module dans une classe en Ruby. Nous avons donc réfléchi aux différentes options à notre disposition pour que cet ajout soit le plus transparent possible pour le programmeur. Nous avons conclu que, pour que cela fonctionne, il fallait obligatoirement modifier le code à la compilation. Pour cela, on doit modifier le compilateur Nit.

Le compilateur Nit fonctionne, comme la plupart des compilateurs modernes, en différentes étapes :

- Analyse lexicale : découpe le code source en jetons ;
- Analyse syntaxique : construit l'arbre syntaxique abstrait ;
- Analyse sémantique : complète l'arbre syntaxique abstrait avec, notamment, des informations de type et vérifie l'intégrité sémantique du programme ;
- Génération de code : génère du code C, lequel est ensuite compilé à l'aide de `gcc` pour obtenir le code exécutable.

De plus, la phase d'analyse sémantique est modulaire, c'est-à-dire qu'elle est découpée en différentes phases, jouant chacune un rôle distinct. Par exemple, la phase `check_annotation` permet de vérifier que les annotations utilisées dans un programme existent bel et bien, qu'elles soient des annotations primitives — définies dans la grammaire du langage — ou déclarées par le programmeur.

La figure 3.1 présente le processus de compilation d'un programme Nit standard.

Lorsque l'on a commencé ce projet, Nit possédait déjà différents outils qui généraient ou modifiaient du code existant lors de la compilation — par ex., sérialisation automatique, gestion d'annotations. Ces outils apparaissent principalement sous deux formes :

- Un outil autonome qui importe le *frontend* du compilateur Nit et se sert des

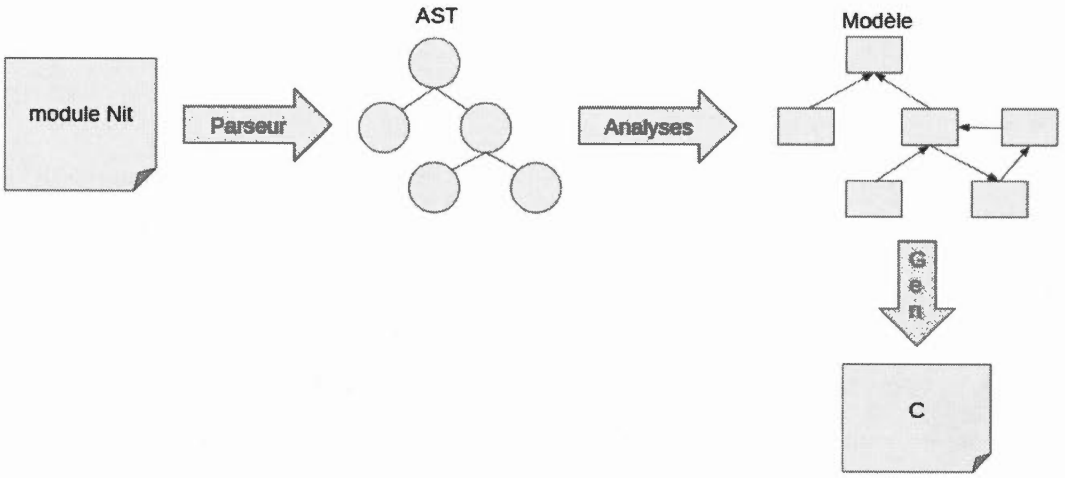

Figure 3.1: Les différentes phases du processus de compilation en Nit.

analyses de celui-ci pour générer un module de support contenant toutes les classes et/ou redéfinitions de classes qui répondent à ses objectifs.

— Une ou plusieurs phases ajoutées directement au *frontend* du compilateur Nit, qui effectuent leur travail à la compilation d'un programme.

Dans le premier cas, il suffit ensuite d'importer le module de support généré pour profiter des fonctionnalités ajoutées par l'outil par raffinement de classe. Dans le second cas, il n'y a rien à faire, car les modifications sont faites directement sur le programme lors de sa compilation. En général, les outils qui sont autonomes sont ainsi car ils génèrent beaucoup de code, et l'API du compilateur Nit qui permet d'introduire du nouveau code ou de manipuler du code existant à la compilation n'est pas idéale. Il est donc plus intéressant de générer le code en question dans un module à part afin de l'intéger ensuite au code existant avant de le compiler avec le compilateur Nit. Néanmoins, si la quantité de modification/génération de code est moindre, ajouter une phase au *frontend* du compilateur reste une meilleure solution, car cela évite au programmeur de devoir gérer plusieurs outils pour compiler son programme.

Ainsi, notre première implémentation était un outil qui générait un module de support introduisant la méthode `async` et tous les autres éléments nécessaires à la transforma-

tion d'une classe annotée en acteur. Par contre, l'interface que l'on voulait proposer avait quelque chose de nouveau par rapport aux outils déjà existants. En effet, nous apportons avec la méthode **async** des services supplémentaires pour le programmeur voulant écrire des acteurs CelluloNit. Cela signifie que pour les utiliser, le programmeur doit posséder le module de support qui les introduit. Or, tant qu'il n'avait pas écrit son module avec ses classes annotées et utilisé l'outil pour générer le module de support, il ne pouvait pas utiliser les nouveaux services que nous lui proposions. Cela était problématique, car cela forçait non seulement le programmeur à utiliser un outil pour générer le code nécessaire avant de pouvoir compiler ses programmes, mais il devait en plus ne pas utiliser les nouveaux services fournis par l'outil avant d'avoir déjà écrit le module sans eux. En effet, comme l'outil utilise le *frontend* du compilateur pour analyser le module possèdant les classes annotées, si le programmeur utilisait **async** avant d'avoir généré le module de support, l'outil plantait en soulevant des erreurs. Cela était dû au fait qu'au moment de l'analyse du code du module avec les classes annotées, **async** n'existait pas encore, car la classe annotée n'avait pas encore été raffinée par le module de support. Au final, cela forçait le programmeur à effectuer de nombreuses étapes et modifications intermédiaires avant d'obtenir un programme fonctionnel. On était alors loin de l'élégance et la simplicité d'utilisation de Celluloid.

C'est à ce moment que nous avons eu l'idée de *court-circuiter* le compilateur et ce, en ajoutant une phase *d'injection de modèle*. Son but est de faire croire au compilateur, lors de la première analyse du code du programmeur, que la méthode **async** et ses méthodes existent déjà. Pour cela, on introduit dans le modèle du programme compilé les entités appropriées, pour que tout le modèle soit statiquement validable, et pouvoir passer à notre phase de génération de code.

Cet ajout permettait donc au programmeur de pouvoir écrire du code acteur — en utilisant **async** et ses méthodes — et de générer le module de support, le tout en une seule étape. Malgré cela, la solution n'était pas encore satisfaisante, car cela prenait toujours deux compilations pour un programme — une pour générer le module de support, et une pour compiler le programme en ayant importé le module de support.

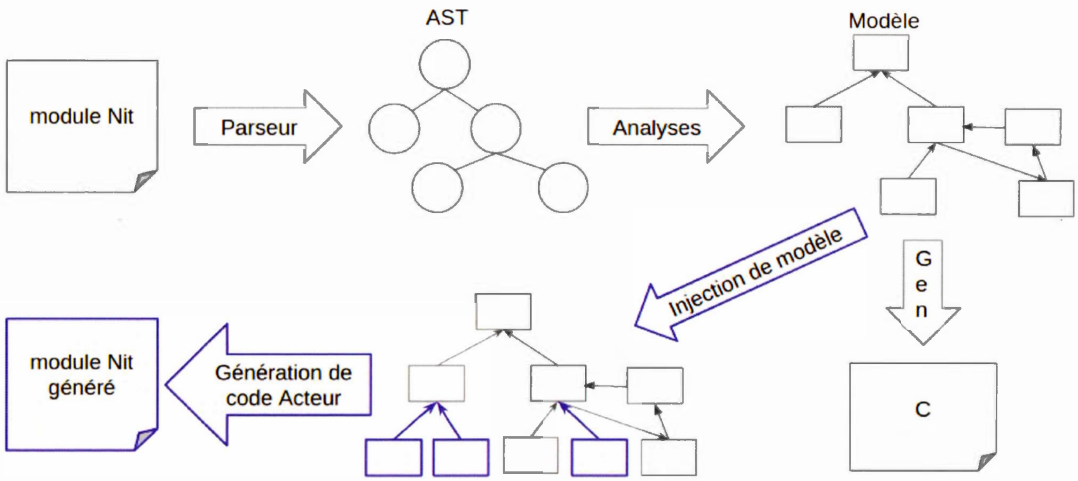

Figure 3.2: Modifications du processus de compilation de Nit apportées par l'ajout des phases d'acteurs.

La dernière étape consistait à trouver un moyen de tout faire en une seule compilation. Pour cela, le compilateur a été modifié pour, au moment de la compilation, injecter un nouveau module dans la hiérarchie de modules du programme. Ensuite, nous avons simplement ajouté au compilateur nos phases d'injection de modèle et de génération de code afin de compiler les programmes CelluloNit en une seule fois.

Dans les sections suivantes, nous présentons plus en détail la mise en oeuvre des différentes phases de compilation ainsi que le modèle objet derrière les acteurs CelluloNit.

3.2 Processus de compilation

La figure 3.2 illustre le processus de compilation — avec les nouvelles parties en bleu — lorsqu'on compile un programme CelluloNit. Les phases d'injection de modèle et de génération de code s'exécutent seulement sur les modules contenant des classes annotées `actor` — donc qui importent le module `actors.nit`.

Les classes du module sont d'abord analysées normalement. Ensuite, la phase d'injection

de modèle injecte des nouvelles propriétés aux classes annotées. Enfin, la phase de génération de code ajoute le code d'implémentation des acteurs, sous la forme d'un nouveau module Nit qui ajoute des classes et redéfinit les classes du module annoté original. Ce nouveau module est ensuite lui-même analysé, et ajouté à la hiérarchie de modules du programme. La compilation se poursuit ensuite comme dans le cas normal, le code C généré prenant maintenant en compte le code Nit généré par la phase de génération de code de CelluloNit.

La figure 3.3 montre une vue d'ensemble d'un module `m` déclarant la classe `A` de notre exemple, avec les modifications apportées par les phases d'injection de modèle et de génération de code acteur. Nous détaillons chaque partie de cette vue d'ensemble plus loin dans ce chapitre.

3.3 Phase d'injection de modèle

La figure 3.4 montre en bleu les modifications effectuées à notre classe d'exemple `A` par la phase d'injection de modèle de CelluloNit. La phase d'injection de modèle est la première étape de la compilation d'un programme CelluloNit. Comme expliqué dans la section 3.1, elle se charge de *court-ciircuiter* le compilateur.

En effet, le compilateur Nit a besoin de connaître toutes les informations d'un programme pour le compiler. Si le programme contient une erreur sémantique, comme essayer d'utiliser une méthode non déclarée par exemple, le compilateur peut alors la détecter, le signaler au programmeur, et s'arrêter sans terminer la compilation du programme incorrect.

En l'occurrence, les ajouts faits à la compilation par la phase de génération de code se font assez tard dans le processus de compilation. La méthode `async` et ses propres méthodes ne sont donc pas censées exister au moment de la construction et de l'analyse d'une classe annotée `actor`. De ce fait, sans la phase d'injection de modèle, l'utilisation de la méthode `async` dans les classes annotées est une erreur, puisque le module qui l'ajoute par raffinement n'existe pas encore.

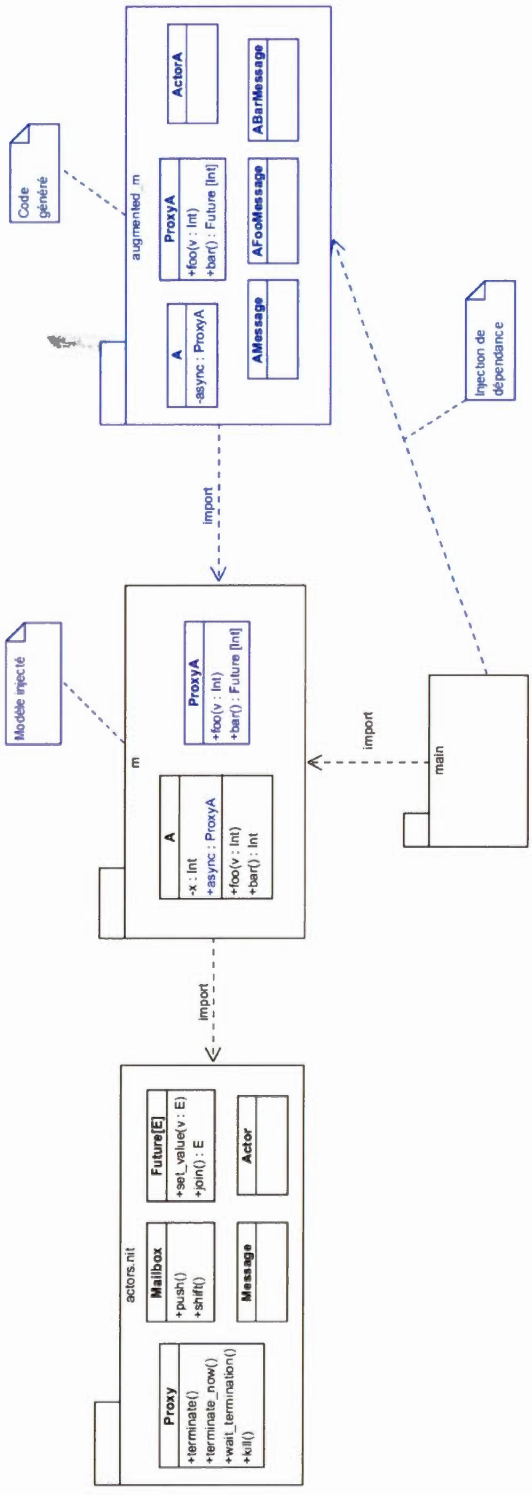

Figure 3.3: Vue d'ensemble des éléments de CelluloNit dans le cas de l'exemple de la classe A.

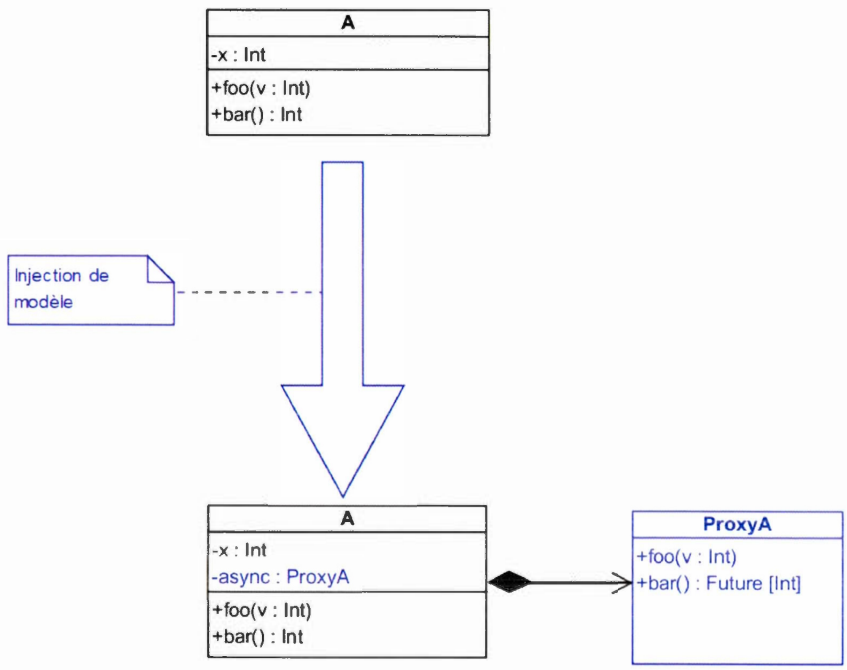

Figure 3.4: Modifications apportées à la classe A par la phase d'injection de modèle.

Pour *court-circuiter* le compilateur, la phase d'injection de modèle ajoute virtuellement des nouvelles informations au programme en cours de compilation. En l'occurence, elle ajoute dans le modèle la méthode `async` aux classes annotées `actor`, ainsi que la classe de `async` et ses méthodes. Par ajouter virtuellement, nous voulons dire ajouter des entités du modèle, sans code d'implémentation associé.

Cette phase intervient sur les classes annotées juste après que leur modèle soit construit, et n'a aucun effet sur les classes non annotées. En se plaçant à ce moment de la compilation, on profite du fait que l'on connaît toutes les propriétés du modèle des classes annotées. Si l'on se plaçait plus tard, l'analyse du code d'implémentation des méthodes des classes serait déjà effectuée, et la compilation serait interrompue par des erreurs lors de l'utilisation de `async` et de ses méthodes.

Plus précisément, la phase d'injection de modèle effectue les ajouts suivants pour chaque classe annotée `actor` :

- La classe d'`async` correspondante, dont le nom est dérivé de la classe annotée ;
- La méthode `async` dans la classe annotée, typée par la classe qui vient d'être introduite ;
- Les méthodes spécifiques d'`async`, basées sur celles de la classe annotée.

À ce stade, on ne fait qu'indiquer au compilateur que tous ces éléments existent dans le modèle, même s'il n'y a aucun code d'implémentation qui leur est relié. Pour les méthodes de la classe de proxy, on reprend simplement la signature de celles de la classe annotée, en modifiant le type de retour pour le transformer en `Future` s'il y a lieu. L'injection de ces entités dans le modèle suffit à valider statiquement le code du programmeur ; le code d'implémentation de celles-ci sera fourni plus tard par la phase de génération de code acteurs.

À ce stade, on a augmenté virtuellement — en ajoutant seulement des éléments au niveau du modèle — le programme CelluloNit, et le compilateur pourrait donc, même sans la phase de génération de code, le compiler jusqu'au bout et générer l'excécutable.

Néanmoins, tous les appels de méthodes ajoutées par la phase d'injection de modèle planteraient à l'exécution. En effet, le compilateur Nit autorise à compiler du code contenant des méthodes abstraites, mais si elles sont appelées à l'exécution sans avoir de code d'implémentation, le programme quitte avec une erreur indiquant qu'un appel sur une méthode abstraite a été fait.

Cette stratégie ne comporte aucun risque particulier au niveau du programmeur qui utilise les acteurs CelluloNit. Les ajouts faits au niveau du modèle sont statiquement valides et ne font pas apparaître de nouvelles erreurs lors de la compilation.

3.4 Modèle objet des acteurs

Dans cette section, nous présentons le modèle objet complet derrière les acteurs CelluloNit.

La figure 3.5 présente le modèle objet associé à la classe `A` présentée précédemment, après sa compilation, dont nous reproduisons le code ci-bas :

```
1 class A
2     actor
3
4     var x: Int
5
6     fun foo(v: Int) do self.x += v
7     fun bar: Int do return self.x
8 end
```

Les classes en bleu dans la figure 3.5 sont déclarées dans le module de la bibliothèque standard `actors.nit`. Ce module regroupe les super-classes de toutes les classes de CelluloNit, elles factorisent le comportement commun de toutes les classes générées. Nous avons préféré les regrouper dans ce module, qui est aussi le module qui introduit l'annotation `actor`, plutôt que de générer ce code commun à chaque compilation. En l'occurence, les classes les plus importantes de ce module sont les suivantes :

— `Proxy` est la super-classe de toutes les classes de proxy générées. Elle implémente les méthodes de haut niveau dont tous les proxys hériteront. Le rôle des proxy

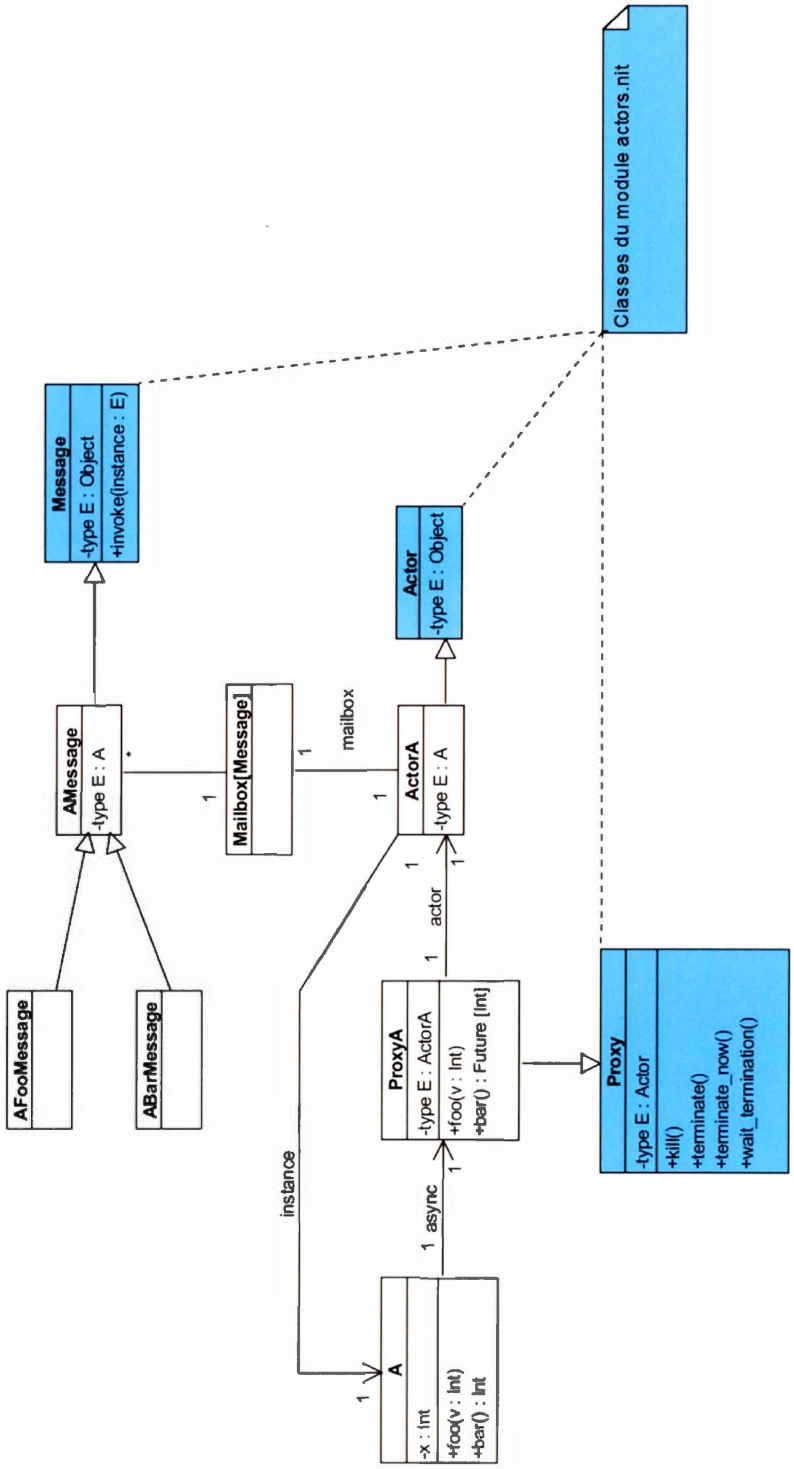

Figure 3.5: Modèle objet complet de la classe A annotée, après le passage des phases d'injection de modèle et de génération de code.

est de transformer un appel asynchrone en envoi de message asynchrone. Nous en parlons plus en détails plus loin.

— `Mailbox` implémente le code de la *mailbox* des acteurs. Celle-ci n'est ni modifiée, ni héritée dans les phases d'injection de modèle et de génération de code d'acteurs. Son implémentation dans le module `actors.nit` suffit déjà pour qu'elle puisse être utilisée en l'état.

— `Actor` est la super-classe de tous les acteurs générés et implémente tout le code nécessaire au fonctionnement d'un acteur, notamment, la boucle de réception et de traitement des messages. Les sous-classes générées vont seulement redéfinir un type virtuel que nous décrivons plus loin.

— `Message` est la classe utilisée pour les messages mis dans la *mailbox* (générique) d'un acteur. C'est la super-classe de toutes les classes de message générées lors des envois de messages, qui introduit notamment un type virtuel et une méthode `invoke` appropriée — voir plus loin.

Pour écrire des programmes CelluloNit, il suffit de se servir d'`async` et de ses méthodes, toutes les autres classes formant une hiérarchie parallèle, dont le programmeur n'a pas à se soucier.

Avec le modèle de Celluloid, inclure le module `Celluloid` dans une classe modifie à la fois l'interface et le comportement de celle-ci. Les objets instanciés via un constructeur d'une classe incluant le module `Celluloid` sont des `Cell`, une encapsulation de la classe originale. Les `Cell` possèdent la même interface que les classes originales, avec quelques ajouts. Un appel de méthode sur une `Cell` est en fait un envoi de message *asynchrone* à un acteur, mais tout le mécanisme est caché au programmeur. Le programmeur perd la possibilité d'utiliser sa classe originale, mais en contrepartie gagne l'accès à un acteur.

Le modèle de Nit conserve la classe originale tout en ajoutant les fonctionalités d'acteur. L'accès à ces nouvelles fonctionnalités d'acteur se fait via l'utilisation de `async` — le but étant de maintenir le plus possible la simplicité de Celluloid, tout en permettant que la classe originale soit utilisable de façon synchrone.

Pour cela, les deux plus gros problèmes auxquels nous avons été confrontés ont été les suivants :

— Transformer un appel objet classique en envoi de message à un acteur ;
— Faire en sorte que tout le modèle puisse être validé statiquement... sinon, le compilateur Nit ne pourrait traiter correctement le programme.

Le diagramme de séquence de la Figure 3.6 montre le processus d'exécution lors de l'envoi d'un message asynchrone `foo` à une instance d'acteur de la classe `A`.

Plus précisément, cet exemple illustre les interactions qui surviennent entre les divers objets associés à un acteur `a` de la classe `A` (Fig. 3.5) pour un appel `a.async.foo(10)`. Nous expliquons plus en détail dans les sections suivantes les différents composants et leurs interactions.

3.4.1 Les messages

Pour pouvoir exécuter de façon asynchrone un appel à une méthode en déléguant son traitement à l'acteur sous-jacent, on peut représenter l'appel par un *message* qui contient les informations appropriées, i.e., la méthode à exécuter et les arguments. Il est donc nécessaire d'avoir un mécanisme pour réifier l'exécution d'un appel de méthode ou d'un bout de code arbitraire. La plupart des langages objets possèdent ce genre de mécanismes. Les plus utilisés sont les fermetures de code (*code closure*) et la réflexivité. Une fermeture est composée d'un pointeur vers le code à exécuter (typiquement, une fonction) et d'un contexte d'exécution — les variables et références nécessaires à l'exécution de la fonction. Ces éléments permettent d'appeler la méthode concernée au moment approprié de l'exécution. Quant à la réflexivité, elle permet à un programme en cours d'exécution d'examiner, et possiblement modifier, le modèle du programme. On peut alors exécuter une méthode sur un objet en ayant une référence à cet objet, et en connaissant le nom et les arguments de la méthode que l'on veut appeler.

Nit ne possède ni la réflexivité de Ruby, ni la possibilité de passer un bloc de code d'un endroit de l'exécution à un autre — donc il n'y a pas de fermetures de code en Nit (pas

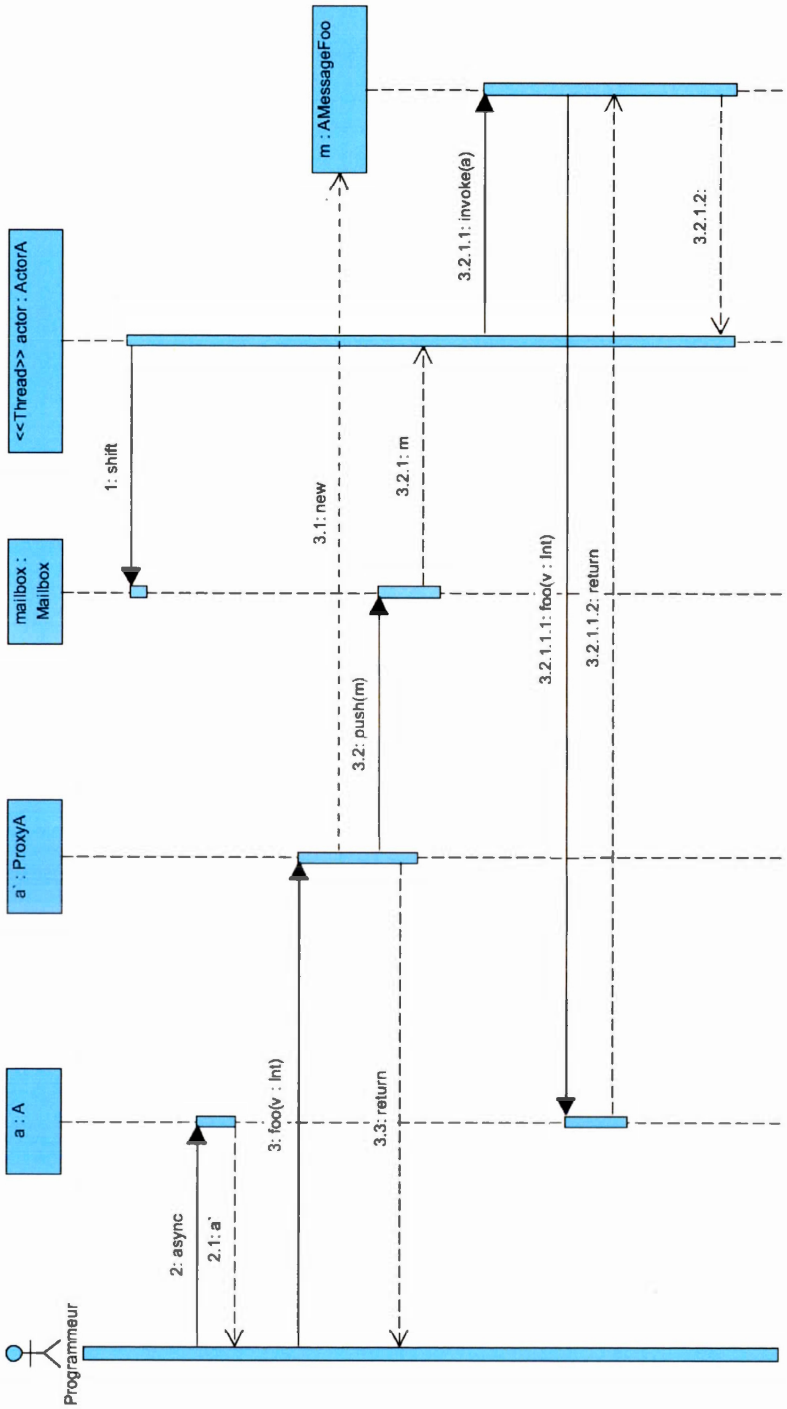

Figure 3.6: Diagramme de séquence d'un envoi de message asynchrone `foo` à un acteur de la classe `A`. Plus précisément, cet exemple illustre un appel `a.async.foo(10)`.

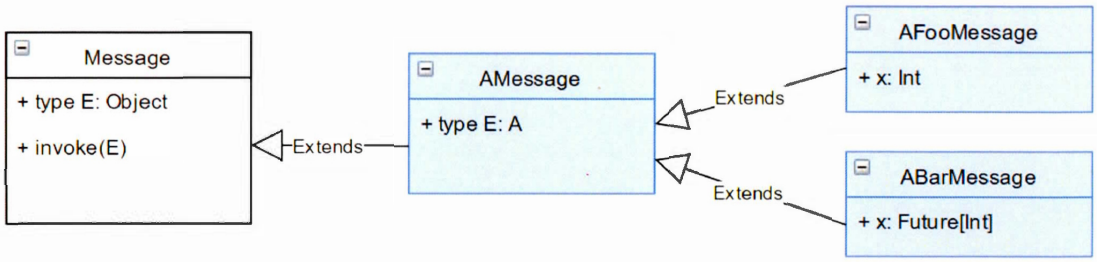

Figure 3.7: Hiérarchie de classes de messages pour la classe `A`, avec les méthodes `foo` et `bar`, qui réfèrent à l'attribut `x`.

de *closures*). Il faut donc trouver un autre moyen pour transformer un appel direct en envoi asynchrone de message.

Voici donc la classe qui abstrait le concept des messages en Nit, introduite dans le module `actors.nit` :

```
1 abstract class Message
2   type E: Object
3
4   fun invoke(instance: E) is abstract
5 end
```

Le mot-clé `type` déclare un type virtuel dans la classe `Message`. Un type virtuel peut être utilisé dans la classe et les sous-classes. Les sous-classes peuvent redéfinir le type virtuel avec un type plus spécifique. Dans notre cas, ce type sera spécialisé pour indiquer la classe sur laquelle on veut exécuter ce type de message — via la méthode `invoke`.

La figure 3.7 montre les classes de message générées pour la classe `A` annotée de notre exemple. On peut voir qu'il y a une classe générale `AMessage` dont le but est uniquement de redéfinir le type virtuel `E` pour qu'il soit maintenant typé par `A` plutôt que par `Object` — puisque les `AMessage` s'appliquent (sont exécutés) sur des objets de classe `A`.

Dans ce modèle, un message correspond directement à une méthode de la classe de

base. C'est pour cela que l'on obtient deux sous-classes de `AMessage` — `AFooMessage` et `ABarMessage` — qui correspondent respectivement aux méthodes `foo` et `bar` de la classe de base `A`.

De plus, les paramètres des méthodes deviennent des attributs des classes de message. Cela permet de pouvoir les utiliser dans la méthode `invoke`, lors de l'appel à la méthode concernée de la classe de base. Aussi, si la méthode doit retourner une valeur, un autre attribut est ajouté sous la forme d'un future. La valeur du future est assignée par le retour de l'appel à la méthode sur l'instance de la classe de base.

Ainsi, les implémentations des classes de message pour la classe de base `A` sont présentées dans le Programme 3.1.

Programme 3.1 Classes des messages pour les méthodes `foo` et `bar` de la classe `A`.

```
1  class AFooMessage
2    super AMessage
3
4    var x: Int
5
6    redef fun invoke(instance) do instance.foo(x)
7  end
8
9  class ABarMessage
10   super AMessage
11
12   var x = new Future[Int]
13
14   redef fun invoke(instance) do x.set_value(instance.bar)
15 end
```

Les instances de `AFooMessage` et `ABarMessage` permettent donc d'appeler respectivement les méthodes `foo` et `bar` sur une instance de la classe `A`, et ce uniquement en leur donnant accès à l'instance via leur méthode `invoke`. Tout ceci est alors statiquement juste et vérifiable par le compilateur.

3.4.2 Le proxy

On a maintenant une manière de créer des appels de méthodes sur des instances de la classe `A` par l'intermédiaire des classes de message. Il reste à déterminer la manière d'envoyer les messages et de les traiter.

Le rôle de la transformation en envoi de messages à l'acteur est assigné à une classe de proxy. La classe de proxy possède aussi les différentes méthodes permettant d'arrêter l'exécution d'un acteur et ce, de différentes manières. Toutes ces méthodes sont regroupées dans la classe `Proxy` introduite dans le module `actors.nit`, qui est la superclasse de tous les autres proxys de CelluloNit. Ce sont les instances de cette classe qui possèdent une référence vers l'acteur travaillant sur son propre *thread*. La méthode `async` introduite dans la classe annotée est une référence vers une classe de proxy. La classe de proxy possède à peu de choses près les mêmes méthodes que la classe annotée. Néanmoins, on ne peut pas faire en sorte qu'elle en soit une sous-classe, car elle ne possède pas exactement les même caractéristiques. La phase d'injection de modèle se charge d'introduire la classe de proxy correspondant à une classe annotée, ainsi que ses méthodes.

Dans la classe de proxy, on duplique les signatures des méthodes de la classe annotée avec une règle supplémentaire : une méthode qui retourne une valeur dans la classe annotée retourne dans la classe de proxy un `Future` de ce type.

L'implémentation des méthodes de la classe de proxy — apportée par la phase de génération de code — crée un message correspondant à la méthode de la classe annotée correspondante, puis envoie le message à l'acteur via sa *mailbox*. Comme la création et l'envoi du message dans la *mailbox* de l'acteur ne sont pas des actions bloquantes, cela rend le processus d'envoi de message asynchrone, puisque la méthode du proxy retourne immédiatement après avoir poussé le message dans la *mailbox*. Si la méthode de la classe annotée renvoie une valeur, la méthode du proxy retourne alors un `Future` pour cette valeur, ce qui fait en sorte de rendre l'appel non bloquant. Bien sûr, si l'utilisateur a

besoin de la valeur de retour du message qu'il vient d'envoyer, il doit alors obtenir la valeur du future avec un appel à `join`, appel qui bloquera tant que la valeur ne sera pas disponible.

Si on reprend l'exemple, l'implémentation de la classe de proxy `ProxyA` ressemble à ce qui est présenté dans le programme 3.2 — on ignore le constructeur dans cet exemple, qui n'est pas utile à notre explication.

Programme 3.2 Classe pour le proxy de la classe `A` avec les méthodes `foo` et `bar`.

```
1  class ProxyA
2    super Proxy
3
4    redef type E: ActorA
5
6    fun foo(v: Int) do
7      var msg = new AFooMessage(v)
8      actor.mailbox.push(msg)
9    end
10
11   fun bar: Future[Int] do
12     var msg = new ABarMessage
13     actor.mailbox.push(msg)
14     return msg.ret
15   end
16 end
```

On voit ici qu'on redéfinit un type virtuel `E`. Une classe annotée est associée à une classe de proxy spécifique — `ProxyA` pour `A` — et une classe de proxy spécifique est associée à une classe d'acteur spécifique — `ActorA` pour `ProxyA`. Dans son introduction dans le module `actors.nit`, le type virtuel défini par la classe `Proxy` générale est borné par `Actor`.

Un appel effectué via le proxy, introduit comme méthode `async` dans la classe annotée, transforme les appels synchrones en envoi de messages asynchrones. On peut ainsi modifier des programmes séquentiels en programmes parallèles simplement en ajoutant l'annotation `actor` aux classes concernées, puis en infixant les appels avec `.async`. Le seul comportement éventuel à modifier est celui lorsque l'on fait appel à des méthodes qui retournent une valeur, ces appels retournant maintenant un `Future`. De plus, la

méthode `async` introduite dans la classe annotée est instancié paresseusement. Cela permet d'utiliser les instances de classes annotées `actor` aussi bien en tant qu'acteur qu'en tant qu'objet Nit classique, sans utiliser de ressources supplémentaires dans le deuxième cas.

3.4.3 L'acteur

On possède maintenant un moyen de transformer des appels de méthode en envoi de message par l'intermédiaire des classes de messages. De plus, on peut aussi faire en sorte que ces envois soient asynchrones via les classes de proxy. Il ne manque plus que le mécanisme permettant d'exécuter ces messages de façon asynchrone.

Un acteur CelluloNit est relativement simple : à chaque acteur est associé un *thread* système qui exécute une boucle dans laquelle il traite les messages reçus, et ce dans leur ordre d'arrivée. Pour cela, il possède une *mailbox* pour le stockage des messages reçus, et une référence à une instance dont il se sert pour les exécuter.

La *mailbox* est réalisée simplement par une file synchronisée. Toutes les *mailboxes* de CelluloNit partagent la même implémentation, introduite par la classe `Mailbox` du module `actors.nit`. Cependant, on ne veut pas que l'acteur exécute sa boucle de traitement lorsqu'il n'a aucun message à traiter. Donc, la méthode `shift` de la *mailbox* est bloquante lorsqu'elle est vide, ce qui fait en sorte que le *thread* associé à l'acteur ne consomme pas de ressources lorsqu'aucun message n'est présent dans sa *mailbox*. Lorsque la *mailbox* est vide et qu'on y ajoute un message avec la méthode `push`, un signal est envoyé qui réveille le *thread* en attente afin qu'il puisse traiter le ou les messages reçus. La classe `Mailbox` étant entièrement générique, il n'est pas nécessaire qu'elle connaisse le type spécifique des messages qu'elle reçoit. La *mailbox* endosse aussi la responsabilité de mettre à jour le statut d'un acteur, donc c'est elle qui prévient le contexte d'exécution de l'activité ou de l'inactivité de l'acteur auquel elle appartient, ce qui permet au contexte d'exécution de détecter correctement l'inactivité des acteurs.

La boucle principale de l'acteur consiste donc uniquement à récupérer le prochain mes-

sage à traiter, via un appel à `shift` sur sa mailbox, puis appeler la méthode `invoke` du message vue précédemment, en lui passant en argument la référence vers l'instance de l'objet sur lequel le message doit être appelé.

Le module `actors.nit` définit une classe abstraite `Actor` qui implémente toutes les méthodes dont un acteur Nit a besoin. En plus de la boucle principale, cette classe définit des méthodes pour arrêter l'acteur, de façon normale ou *abrupte*.

L'acteur ne nécessite presque pas de modification pour fonctionner et être valide statiquement : les acteurs spécifiques ont seulement besoin d'hériter de la classe `Actor`, et de redéfinir le type virtuel de l'instance vers laquelle ils gardent une référence pour l'invocation de messages.

Ainsi, la classe `ActorA` ressemble à ce qui suit :

```
1 class ActorA
2   super Actor
3
4   redef type E: A
5 end
```

Tous les messages qui vont arriver dans la *mailbox* d'un `ActorA` seront des instances de la classe `AMessage`. Ces messages font des appels sur des instances de la classe `A` dont `ActorA` possède une référence, qu'il leur passe via leur méthode `invoke`.

3.4.4 Récapitulatif

On peut maintenant comprendre plus facilement la figure 3.5 présentée au début de cette section. Après compilation, notre classe `A` d'exemple possède une méthode `async`, qui retourne une référence à une instance de type `ProxyA`. Cette référence est initialisée paresseusement, et instancie elle même son attribut de type `ActorA`. en lui passant une référence à l'instance de `A` sur laquelle les messages seront exécutés. Dès sa création, l'instance d'`ActorA` commence à attendre de recevoir des messages via sa *mailbox*. Comme l'instance de `ProxyA` associée à notre instance de classe `A` est la seule à pouvoir lui envoyer des messages, tous ces messages sont bien des sous-classes d'`AMessage`, qui

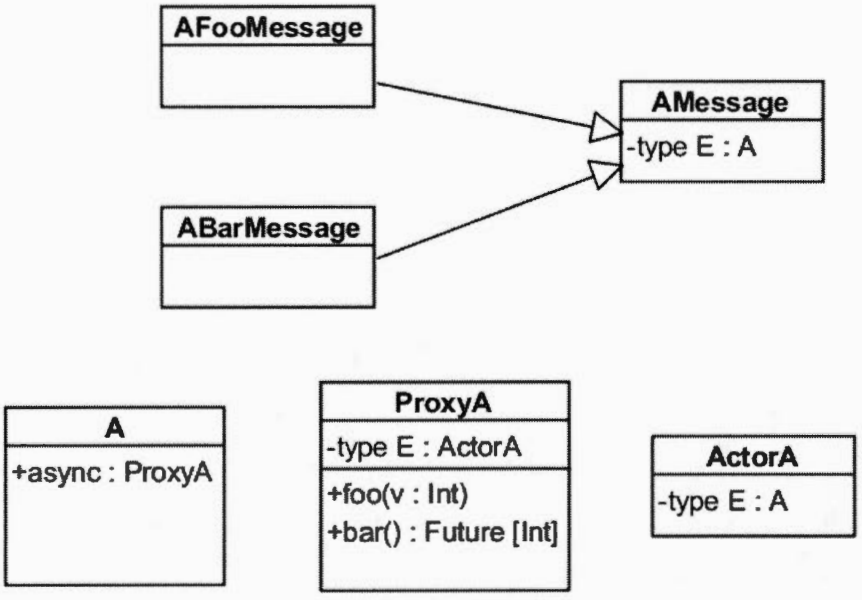

Figure 3.8: Classes générées par la phase de génération de code de CelluloNit.

prennent en paramètre de leur méthode `invoke` une instance de type `A` pour invoquer la bonne méthode sur cette instance. Tout cela grâce à l'utilisation de trois types virtuels, un dans `Proxy`, un dans `Message`, et un dans `Actor`, qui sont redéfinis par leurs sous-classes respectives. Le modèle est entièrement valide statiquement, et donc le programmeur sera prévenu s'il fait des erreurs de types, lors de la compilation du programme.

3.5 Phase de génération du code acteurs

La phase de génération du code acteurs est le dernier composant de CelluloNit, elle est exécutée à la toute fin, lorsque les modules contenant des classes annotées `actor` ont été entièrement analysés. Cette phase est responsable de fournir le code d'implémentation des propriétés et classes dont nous avons discuté plus tôt, dans la section 3.4. Pour cela, elle crée un sous-module du module contenant les classes annotées, qui raffine des classes existantes et en introduit de nouvelles.

La figure 3.8 présente les classes générées — y compris les redéfinitions de types virtuels — pour la classe `A` de notre exemple. Nous avons volontairement omis les relations entre les classes `A`, `ProxyA` et `ActorA` pour se concentrer sur les éléments générés les plus significatifs.

Dans l'ordre :

- Une classe acteur ;
- Une classe de message générale à la classe annotée, puis toutes les classes de message spécifiques représentant les différentes méthodes de celle-ci ;
- Une classe de proxy et l'implémentation de ses méthodes ;
- Une redéfinition de la classe annotée pour lui ajouter l'attribut `async` ainsi que le mécanisme d'initialisation paresseuse de celui-ci.

Une fois que le code a été généré pour toutes les classes annotées du module de base, on l'écrit dans un nouveau sous-module Nit. Ensuite, tous les modules qui dépendent du module de base se verront injecter en plus une dépendance vers le sous-module généré.

Signalons que le sous-module généré est conservé sous forme de fichier, dans le même répertoire que le module Nit contenant les fichiers des classes annotées. On pourrait ne pas générer de fichiers pour ce sous-module, mais cela est utile pour d'éventuelles séances de débogage.

Un autre détail intéressant est que les classes de proxy injectées par la phase d'injection de modèle ainsi que leurs propriétés doivent être redéfinies lors de la génération de code. Ceci est dû au fait que pour le compilateur, elles existent déjà. Si on ne les redéfinissait pas, le compilateur signalerait une erreur indiquant que ces classes et propriétés sont déjà définies.

En combinant le modèle objet introduit dans le module de la bibliothèque standard `actors.nit`, la phase d'injection et la phase de génération de code, on peut maintenant écrire des programmes en Nit avec des acteurs, qui reçoivent et exécutent des messages de façon asynchrone.

3.6 Récapitulatif et discussion

Avec les messages, le proxy permettant de faire l'envoi de message asynchrone, et l'acteur qui s'occupe de récupérer et traiter les messages, on a maintenant un système d'acteurs fonctionnel.

De plus, il est facile de passer d'un programme séquentiel à un programme concurrent avec ce modèle, puisqu'il suffit d'importer le module `actors.nit` et d'annoter les classes que l'on veut utiliser comme des acteurs. Programmer avec ce modèle reste donc assez près la programmation objet classique. On garde aussi tous les avantages du typage statique de Nit. Et en plus, le modèle est facile à comprendre. En effet, l'utilisation des types virtuels permet de ne pas avoir à faire de conversions un peu partout.

Néanmoins, il est possible de briser le modèle et d'obtenir des comportements non déterministes. Le fait de vouloir garder le code utilisateur le plus intact possible nous oblige à ne pas trop modifier la classe annotée. Ainsi, lors de la création paresseuse du proxy et de l'acteur au premier accès à l'attribut `async`, c'est une référence à `self` qui est passée via le proxy à l'acteur pour l'exécution des messages. Ainsi, un appel synchrone à une instance de `A` peut interférer avec un message envoyé à l'acteur, puisque les deux vont s'exécuter sur le même objet.

Cela signifie que la séquence de code suivante donne un résultat non-déterministe (toujours dans le cadre de notre exemple avec la classe `A`) :

```
1 var a = new A(0)
2 a.async.foo(99)
3 a.foo(1)
4 print a.x  # Peut imprimer 1... ou 100!?
```

De plus, on ne force pas le contenu des messages à être immuable. On peut donc envoyer par référence des objets du programme à plusieurs acteurs différents et faire face à des situations de compétition (*race condition*). Enfin, si on arrête un acteur via le proxy et qu'on essaie ensuite de lui envoyer des messages, ils ne seront pas exécutés puisque l'acteur ainsi que le *thread* système traitant les messages sont arrêtés. Aussi, le proxy

n'est pas libéré puisque l'instance de la classe annotée possède toujours une référence vers lui.

CHAPITRE IV

EXPÉRIMENTATIONS SUR DES PROGRAMMES *BENCHMARKS* ET RÉSULTATS

Dans ce chapitre, nous présentons une évaluation de notre solution, le modèle d'acteurs de Nit, sous forme de quelques expérimentations, faites avec différents programmes *benchmarks*. Nous évaluons notamment les performances par rapport aux deux autres implémentations présentées plus tôt dans ce mémoire, Akka (Sect. 1.4) et Celluloid (Sect. 1.5). Mais dans un premier temps, nous discutons des aspects que nous tentons d'évaluer et présentons les programmes *benchmarks*.

4.1 Paramètres évalués

Les différents programmes *benchmarks* réalisés avec notre modèle d'acteurs de Nit visent à évaluer la performance du modèle selon différents aspects :

- Coût des communications :
  Le modèle d'acteurs reposant sur l'envoi de messages, il est important de pouvoir comparer le coût des communications
- Coût des synchronisations :
  Il est fortement déconseillé d'utiliser des mécanismes de synchronisation tels que des barrières ou des verrous lorsque l'on utilise des acteurs. Néanmoins, l'utilisation de ces mécanismes est, dans certains cas, inévitable, et donc il est important d'évaluer l'impact de leur utilisation.
- Exploitation du parallélisme :

On utilise les acteurs pour la concurrence, mais aussi pour le parallélisme, donc il est intéressant d'examiner le gain de performance d'une version parallèle d'un programme par rapport à sa version séquentielle.

4.2 Description des benchmarks

Nous présentons maintenant les différents programmes *benchmarks* utilisés pour faire nos mesures de performance. N'ayant pas trouvé de *benchmarks* spécifiques pour les acteurs, nous avons choisi d'utiliser des programmes venant d'un site de *benchmarks* de parallélisme, `benchmarksgame` (benchmarksgame, 2017). Ces tests ont été développés, à la base, pour comparer les implémentations parallèles de différents langages de programmation. Nous avons choisi quatres programmes parmi ceux de `benchmarksgame`, les plus pertinents pour notre évaluation :

— Mandelbrot
— Thread-ring
— Chameneos-redux
— Fannkuch-redux

Mandelbrot

Le but de ce benchmark est de générer l'ensemble de Mandelbrot sur l'intervalle complexe $[-1.5 - i, 0.5 + i]$ et d'en ressortir une image au format bitmap portable. La bitmap doit être de taille $N \times N$, où N correspond au premier argument fourni au programme. Le nombre d'itérations maximum pour déterminer l'appartenance d'un nombre à l'ensemble est fixé à 50. Tous les programmes testés utilisent le même algorithme.

Les calculs nécessaires pour déterminer si un nombre appartient ou non à l'ensemble de Mandelbrot étant indépendants, i.e., c'est un problème avec *parallélisme embarrassant*, il est donc facile à paralléliser avec du parallélisme *fork/join*. Cependant, répartir la charge de travail de façon équitable est plus compliqué. En effet, au centre de l'image, on aura un plus grand nombre de points qui appartiennent à l'ensemble, ce qui veut

dire qu'on devra exécuter toutes les 50 itérations pour ces points. Il sera donc préférable d'utiliser une répartition *dynamique* de la charge de travail, plutôt que statique — les tâches associées à un *thread* seront donc déterminées en cours d'exécution, et non au début de l'exécution du programme.

Soulignons que ce type de *benchmark* est *CPU-bound*, c'est-à-dire qu'un processeur plus rapide effectuera les calculs en moins de temps.

Thread-ring

Le *Thread-ring* consiste à créer un certain nombre de *threads*, connectés entre eux en anneau — le premier au deuxième, le deuxième au troisième, etc., puis le dernier au premier. Chaque *thread* possède son propre identifiant qui représente sa position dans l'anneau. Le but du programme est de passer un jeton (un *token*) d'un *thread* à l'autre autour de l'anneau, et ce N fois, où N est le premier argument fourni au programme. La sortie du programme est simplement l'identifiant du *thread* qui a reçu le jeton en dernier, ce qui permet de vérifier que l'exécution s'est bien déroulée.

Ce *benchmark* a pour but de mesurer le coût de communication entre *threads*, puisqu'il n'y a pas vraiment de parallélisme — un seul *thread* à la fois s'exécute, celui ayant le jeton en sa possession. Dans notre cas, on remplace les *threads* par des acteurs, ce qui permet de mesurer la performance de l'envoi de message entre acteurs.

Chameneos-redux

Ce *benchmark* consiste à faire rencontrer des créatures *Chameneos* de différentes couleurs à un point de rendez-vous. Lorsqu'elles se rencontrent, leurs couleurs se complémentent puis les créatures essaient de rencontrer à nouveau une autre créature au même point de rendez-vous. Le point de rendez-vous est représenté par un objet partagé et synchronisé. Pour se rencontrer, deux créatures doivent obtenir l'accès à cet objet. Pendant l'exécution, toutes les créatures du programme sont soit à la recherche d'un rendez-vous, soit en train de rencontrer une autre créature. Une créature ne peut

pas se rencontrer elle même.

Le programme s'exécute en deux phases : la première où l'on ne crée que trois créatures, puis la seconde on l'on crée 10 créatures.

En temps normal, chaque créature est un *thread.* Dans notre cas, chaque créature est plutôt un acteur, auquel est associé un *thread.* Le nombre total de rencontres à chaque phase de l'exécution du programme est déterminé par N, le premier argument fourni au programme.

En sortie, le programme donne le nombre de rendez-vous effectués par chaque créature, puis le nombre total de rendez-vous — qui doit être $2N$.

Ce benchmark mesure plus la concurrence/synchronisation que le parallélisme, puisque chaque créature doit attendre qu'une autre soit présente au point de rencontre pour effectuer un rendez-vous.

Fannkuch-redux

L'algorithme de ce *benchmark* est le suivant :

- Prendre une permutation de $\{1, \ldots, n\}$, par exemple $[4, 2, 1, 5, 3]$.
- Prendre le premier élément, ici 4, et inverser l'ordre des 4 premiers éléments : $[5, 1, 2, 4, 3]$.
- Répéter jusqu'à ce que le premier élément soit un 1 (changer l'ordre ne changerait plus rien) : $[3, 4, 2, 1, 5]$, $[2, 4, 3, 1, 5]$, $[4, 2, 3, 1, 5]$, $[1, 3, 2, 4, 5]$.
- Compter le nombre d'inversions nécessaires pour que le 1 soit en première position — ici 5.
- Faire cette série d'opérations pour toutes les $n!$ permutations, et déterminer le nombre maximum d'inversions.

Le nombre d'inversions pour chaque permutation peut être calculé indépendamment, il suffit donc de les répartir sur plusieurs acteurs pour paralléliser le problème.

Ce *benchmark* représente encore un problème *CPU-bound*, pour lequel on préfèrera une répartition dynamique des tâches, ne sachant pas à l'avance combien d'inversions sont nécessaires pour une permutation donnée

4.3 Résultats

Tous les résultats de cette section ont été obtenus en lançant les programmes sur une machine possédant les caractéristiques suivantes :

- Un processeur AMD Ryzen 5 1500X avec 4 coeurs physiques, 8 *threads* et fonctionnant à une fréquence d'horloge de 3.5 GHz.
- Système d'exploitation : Ubuntu 16.04 lts 64 bits.
- 16 GB de *RAM*.
- Un SSD, impliquant des accès disque rapides.

De plus, les programmes Akka sont exécutés avec la JVM 1.8.0_144, alors que les programmes Celluloid sont exécutés avec JRuby 9.1.6.0 — version conseillée sur le wiki de Celluloid. Quant aux programmes Nit, ils sont exécutés avec la version *v0.8-1907-gb636f2c* de Nit. Les programmes Akka et Celluloid sont exécutés sans options particulières, et les programmes Nit sont compilés avec l'option `--global`.

Mandelbrot

Dans la figure 4.1, on peut voir que, pour de petites valeurs de N, CelluloNit est le plus rapide. Cela est dû au temps de démarrage d'Akka et de Celluloid — dû au temps de démarrage de la JVM, donc aussi de JRuby. De son côté, CelluloNit ne nécessite aucun temps de démarrage. Celluloid sort rapidement à l'extérieur des limites du graphe, dont nous avons ajusté l'échelle de l'axe des ordonnées pour le rendre plus lisible. JRuby est plus performant que MRI, mais cela reste du Ruby, qui est plus lent que Java et plus lent que Nit qui lui est compilé. CelluloNit est plus rapide qu'Akka au début, puis se fait rattraper ; et plus on augmente la taille de l'image générée, plus l'écart se creuse. Mandelbrot est un benchmark qui comporte de nombreuses opérations sur des tableaux,

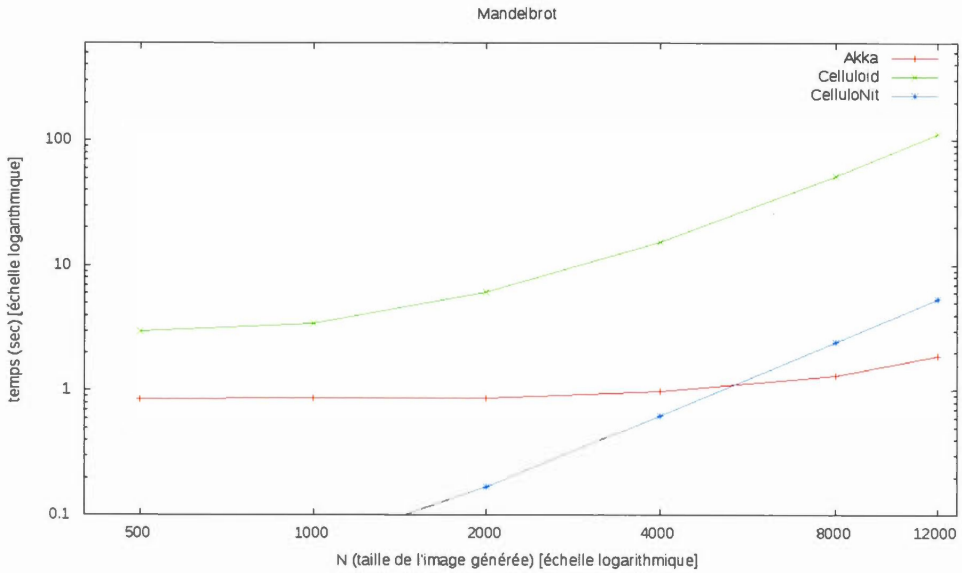

Figure 4.1: Résultats pour le programme *benchmark* Mandelbrot.

qui sont des types primitifs en Java, mais pas en Nit. Ainsi, toutes ces opérations sur des tableaux sont plus lentes en Nit, ce qui explique cet écart de performance.

Thread-ring

Le benchmark Thread-ring permet de tester le coût des communications entre acteurs — envoi et réception de messages. Dans la figure 4.2, nous avons encore une fois ajusté l'echelle de l'axe des ordonnées, Celluloid devenant grandement plus lent qu'Akka et CelluloNit. Akka est le plus rapide, d'assez loin — environ quatre fois plus rapide que CelluloNit pour la plus grande valeur de N, en incluant le temps de démarrage. La rapidité d'Akka par rapport à Celluloid et CelluloNit s'explique par une différence fondamentale au niveau des implémentations. Alors que les acteurs CelluloNit et Celluloid correspondent chacun à un *thread* système, les acteurs d'Akka sont beaucoup plus légers, le *runtime* se charge de répartir la charge de travail sur un ensemble restreint de *threads* système. Ainsi, Celluloid et CelluloNit créent 503 *threads* systèmes pour ce *benchmark*, alors qu'Akka n'en crée que 8 — le nombre de coeurs disponibles sur la machine de

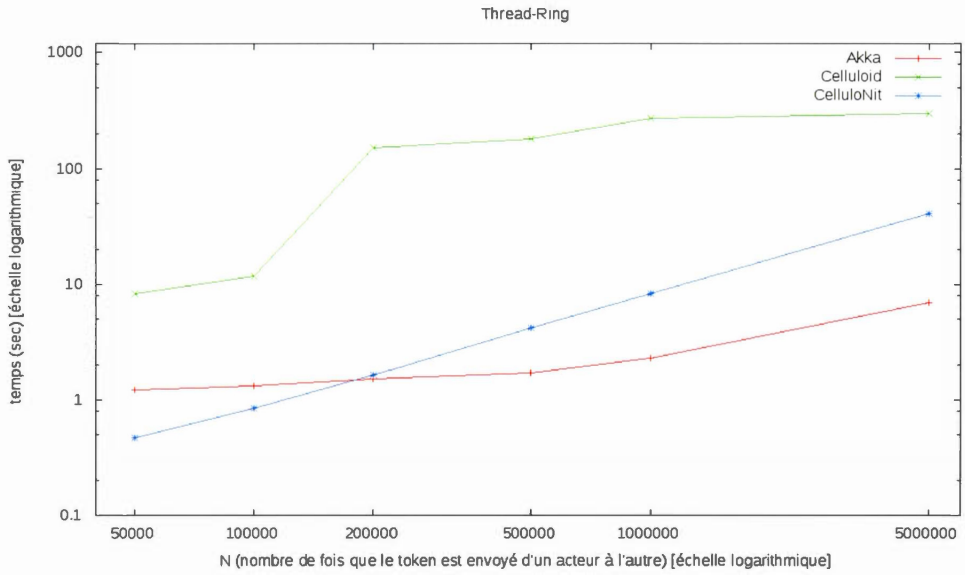

Figure 4.2: Résultats pour le programme *benchmark* Thread-ring.

benchmark. À cause de cela, les programmes Celluloid et CelluloNit passent beaucoup de temps à exécuter des changements de contexte. De plus, le ramasse-miettes de Nit est moins efficace que celui de la JVM, ce qui contribue à creuser l'écart entre Akka et CelluloNit. Pour ce qui est de Celluloid, il reste beaucoup plus lent que les deux autres — l'implémentation de l'envoi et réception de messages en est la cause principale.

Chameneos-redux

Encore une fois, la figure 4.3 montre que pour des petites valeurs, CelluloNit est plus performant, du fait qu'il n'y a pas de surcoûts au démarrage. Dans ce *benchmark*, on exécute surtout des opérations de synchronisation. CelluloNit et Celluloid utilisent les `Future` pour obtenir des réponses aux messages envoyés, tandis qu'en Akka, on récupère explicitement l'identité de l'expéditeur pour lui retourner la réponse. Or, l'utilisation des `Future` ajoute des opérations de synchronisation supplémentaires, qui sont assez coûteuses. Ainsi, on peut voir que c'est Akka qui s'en sort le mieux, puis CelluloNit, suivi par Celluloid. Encore une fois, les performances de JRuby placent Celluloid en

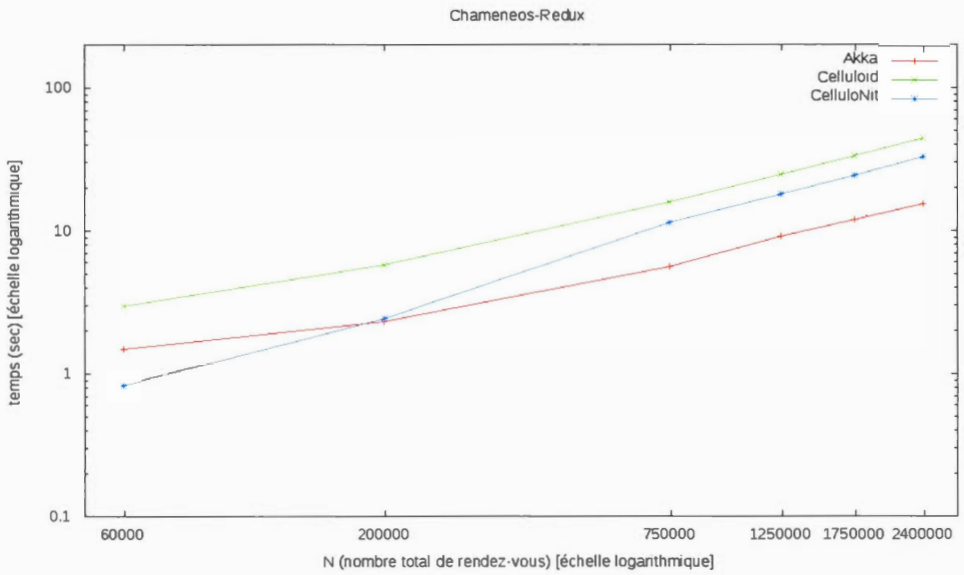

Figure 4.3: Résultats pour le programme *benchmark* Chameneos-redux.

dernière position, même si l'écart est moins flagrant. Cela est dû au fait que, cette fois, il n'y a pas de calculs intensifs, mais surtout de la synchronisation.

Fannkuch-redux

Fannkuch-redux est encore une fois un *benchmark* intensif en calculs sur des tableaux. Logiquement, la figure 4.4 montre donc les même tendances que pour Mandelbrot. La principale différence est que la complexité algorithmique de ce benchmark est $O(n!)$, c'est pour cela que l'on voit une énorme différence dans le temps d'exécution entre les premières valeurs de N et les dernières. Néanmoins, on peut quand même voir que le temps d'exécution de Celluloid commence à augmenter de façon significative lorsque N=10, alors qu'il commence à augmenter pour Akka et CelluloNit lorsque N=11. Pour les valeurs plus petites de N, CelluloNit reste devant, toujours grâce au fait que le démarrage est quasi-instantané. Encore une fois, les différences de performance s'expliquent dans la nature des opérations et la performance des langages concernés — Java, Nit et Ruby.

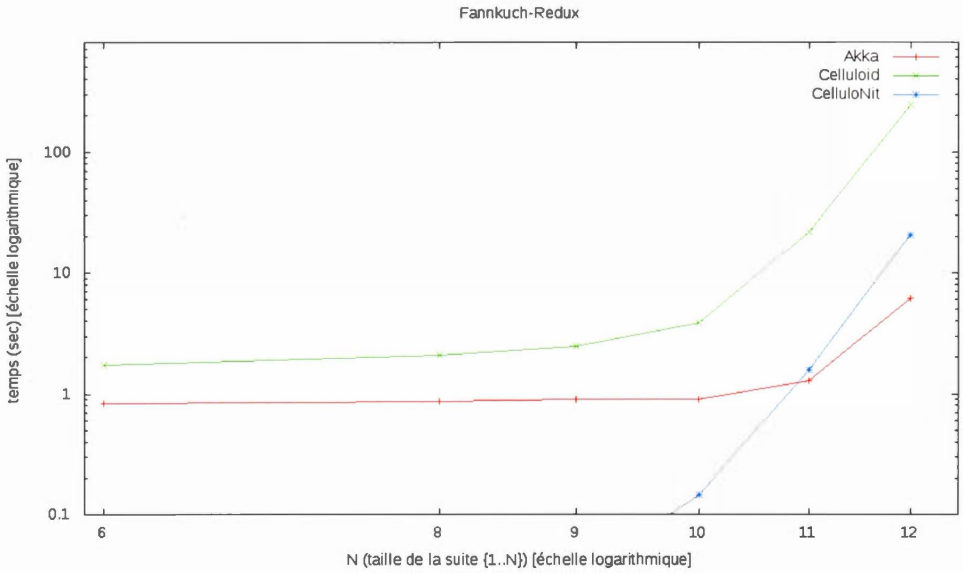

Figure 4.4: Résultats pour le programme *benchmark* Fannkuch-redux.

4.4 Récapitulatif et discussion

Dans cette section, notre but était de comparer notre implémentation du modèle acteur, CelluloNit, à deux autres implémentations que nous avons étudiées, Celluloid et Akka. Pour les benchmarks *CPU-bound* présentés, ce sont surtout les performances des langages qui jouent au niveau de l'écart de performance. Cependant, l'implémentation de CelluloNit ne semble pas ajouter de surcoût significatif. Au niveau de la synchronisation, CelluloNit se place entre Akka et Celluloid en terme de performance. Pour ce qui est de l'envoi et de la réception des messages, le fait que CelluloNit crée un *thread* système par acteur est handicapant, mais la performance reste raisonnable — largement meilleure que Celluloid —. même si Akka est beaucoup plus efficace. Ces résultats nous montrent que les performances de CelluloNit sont raisonnables, puisque les goulots de performance semblent relever d'autres aspects de Nit, comme les tableaux par exemple.

CONCLUSION

Dans ce mémoire, nous avons présenté le modèle acteur ainsi que quelques-unes de ses implémentations. Nous avons débuté en présentant Akka, écrite dans un langage statique (Java), puis Celluloid, écrite dans un langage dynamique (Ruby). Ensuite, nous avons présenté CelluloNit, notre propre implémentation du modèle acteur pour le langage Nit. Cette dernière présente une API proche de Celluloid, comme son nom le laisse entendre. L'objectif initial étant la simplicité d'utilisation par le programmeur, nous croyons l'avoir atteint. En effet, CelluloNit fournit un moyen simple et élégant de programmer avec le modèle acteur. De plus, et ce, contrairement à Celluloid, un programme CelluloNit est vérifié statiquement par le compilateur. Aussi, passer d'un programme séquentiel à un programme parallèle est facile, ne nécessitant que peu de modification au code original. Bien sûr, l'implémentation comporte certaines limites, notamment, il n'y a pas de supervision des acteurs, contrairement à Celluloid. Par contre, nous avons contribué en proposant une interface de programmation différente qui gère de façon intelligente l'arrêt du système.

De plus, nous avons comparé les performances de CelluloNit avec Akka et Celluloid. Notre implémentation se place entre Akka et Celluloid, avec des performances raisonnables.

Ces dernières années, d'autres implémentations du modèle acteur ont vu le jour, certaines essayant d'en imiter des existantes — par ex., Pykka (Akka en Python) (Pykka, 2017), Akka.NET (Akka en C#) (Akka.net, 2017) —, d'autres proposant des améliorations — par ex., TAkka (Akka typé) (He, Wadler et Trinder, 2014) — et d'autres proposant des solutions différentes — par ex., ACT++ (acteurs en C++).

Dans le contexte de travaux futurs, d'autres éléments pourraient être ajoutés à Cellu-

loNit, notamment :

— Gestion et supervision des acteurs : Cela nécessiterait la mise en place d'un système de gestion d'erreurs en Nit, non disponible pour le moment.

— Héritage d'acteurs : Des langages comme ACT++ ont proposé des pistes pour rendre cela possible.

— Utilisation de *pools* d'acteurs : L'implémentation actuelle nécessiterait probablement des modifications conséquentes.

— Distribution des applications : Il faudrait revoir la couche de communication de l'implémentation.

BIBLIOGRAPHIE

Agha, G. 1986. *ACTORS : A Model of Concurrent Computation in Distributed Systems.* The MIT Press Series in Artificial Intelligence.

Akka. 2017. Akka documentation. <`http://doc.akka.io/docs/akka/current/java/index-actors.html?_ga=2.72814163.1368779821.1504258515-1404886620.1464666596`>. Consulté le 1er septembre 2017.

Akka.net. 2017. Akka.net documentation. <`http://getakka.net/`>. Consulté le 1er septembre 2017.

Armstrong, J., R. Virding, C. Wikstrom et M. Williams. 1993. *Concurrent Programming in ERLANG (Second Edition).* Prentice-Hall.

benchmarksgame. 2017. The computer language benchmarks game. <`http://benchmarksgame.alioth.debian.org`>. Consulté le 20 août 2017.

Butenhof, D. 1997. *Programming with POSIX Threads.* Addison-Wesley.

Cavé, V., J. Zhao, J. Shirako et V. Sarkar. 2011. « Habanero-Java : The new adventures of old X10 ». In *Proc. of the 9th International Conference on Principles and Practice of Programming in Java.* Coll. « PPPJ '11 », p. 51–61.

Celluloid. 2017. Celluloid. <`https://celluloid.io/`>. Consulté le 1er septembre 2017.

Chamberlain, B., D. Callahan et H. Zima. 2007. « Parallel programmability and the Chapel language ». *International Journal of High Performance Computing Applications*, vol. 21, no. 3, p. 291–312.

Chandra, R., L. Dagum, D. Kohr, D. Maydan, J. McDonald et R. Menon. 2001. *Parallel Programming in OpenMP.* Morgan Kaufmann Publishers.

Charles, P., C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun et V. Sarkar. 2005. « X10 : an object-oriented approach to non-uniform cluster computing ». In *OOPSLA '05 : Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, p. 519–538, New York, NY, USA. ACM.

De Wael, M., S. Marr, B. De Fraine, T. Van Cutsem et W. De Meuter. 2015. « Partitioned global address space languages ». *ACM Comput. Surv.*, vol. 47, no. 4, p. 62 :1–62 :27.

Eiffel. 2017. Concurrent programming with SCOOP. <https://www.eiffel.org/doc/solutions/Concurrent%20programming%20with%20SCOOP>. Consulté le 21 août 2017.

Elixir. 2017. Elixir. <https://elixir-lang.org/>. Consulté le 19 août 2017.

Fowler, M. 2011. *Domain-Specific Languages.* Addison-Wesley.

———. 2013. ExpressionBuilder. <https://martinfowler.com/bliki/ExpressionBuilder.html>. Consulté le 21 août 2017.

He, J., P. Wadler et P. Trinder. 2014. « Typecasting actors : from akka to takka ». In *Proceedings of the Fifth Annual Scala Workshop*, p. 23–33. ACM.

Hewitt, C. 1977. « Viewing control structures as patterns of passing messages ». *Artificial Intelligence*, vol. 8, p. 323–363.

Hewitt, C., P. Bishop et R. Steiger. 1973. « A universal modular actor formalism for artificial intelligence ». In *Advance Papers of the Conference.* T. 3, p. 235. Stanford Research Institute.

Karmani, R. K., et G. Agha. 2011. *Actors.* Coll. « Encyclopedia of Parallel Computing », p. 1–11. Springer.

Larson, J. 2009. « Erlang for concurrent programming ». *Communications of The ACM*, vol. 52, no. 3, p. 48–56.

Odersky, M., B. Venner et L. Spoon. 2010. *Programming in Scala.* Walnut Creek.

Pacheco, P. 1997. *Parallel Programming with MPI.* Morgan Kaufman Publ.

Privat, J. 2006. « De l'expressivité à l'efficacité : une approche modulaire des langages à objets ». Thèse de Doctorat, Université de Montpellier.

Pykka. 2017. Pykka. <https://www.pykka.org/en/latest/>. Consulté le 1er septembre 2017.

Quinn, M. 2003. *Parallel Programming in C with MPI and OpenMP.* McGraw-Hill.

Thomas, D. 2014. *Programming Elixir—Functional |> Concurrent |> Pragmatic |> Fun.* Pragmatic Bookshelf.

Thomas, D., A. Hunt et C. Fowler. 2013. *Programming Ruby 1.9 & 2.0 : The Pragmatic Programmer's Guide.* Addison-Wesley.

Tilkov, S. 2016. « Vaughn Vernon on reactive programming with the actor model ». *IEEE Software*, vol. 33, no. 3, p. 109–112.

Torshizi, F., J. Ostroff, R. Paige et M. Chechik. 2009. « The SCOOP concurrency model in Java-like languages ». In *Communicating Process Architectures 2009.*