Rapport de Magistère

Spécification et vérification de systèmes temporisés réactifs en logique linéaire

Magistère MMFAI, École Normale Supérieure

Antoine KREMER

20 septembre 2001

Introduction

Le magistère MMFAI m'a permis de construire un cursus à la fois ciblé sur mon objectif de thèse et varié pour me permettre d'acquerir des connaissances et une pratique large des sujets concourants à ma formation de base.
La première année du magistère a été consacrée au passage d'une licence et d'une maîtrise d'informatique qui s'est conclue par un stage au LIS (laboratoire d'Informatique et Systèmes) à Jussieu sous la direction de Bertrand Granado. La deuxième année quant à elle, m'a permis d'effectuer un stage de 6 mois aux Etats-Unis à Duke University sur les *algorithme en mémoire externe* sous la direction de Jeff Vitter. Ce stage n'a pas donné lieu à la rédaction d'un rapport. Pour la fin de la deuxième année, j'ai choisi de suivre différents cours dont celui de Logique Informatique, qui, avec le cours d'interprétation abstraite de première année, m'a conforté dans ma décision de faire un DEA SPL. La troisième année a été consacrée au DEA, qui a été sanctionné par la soutenance d'un mémoire portant sur un stage réaliser au LIX (Laboratoire d'Informatique de l'École Polytechnique) sous la direction de Jean-Pierre Jouannaud. Une thèse au LRI (Laboratoire de Recherche en Informatique, Paris XI) donnera suite à ce stage.

Le premier chapitre se compose d'un *Curriculum Vitæ*. Le second chapitre, quant à lui, brossera un bref descriptif de mon sujet de thèse, thème développé plus avant lors du rapport de DEA qui tiens lieu de troisième chapitre. Enfin, le dernier chapitre présentera le travail effectué lors du stage de maîtrise.

Je tiens à remercier tout particulièrement les différents professeurs du magistère, mon directeur de thèse Jean-Pierre Jouannaud ainsi que mon tuteur, Louis Granboullan pour leurs conseils, leur disponibilité et leur gentillesse.

Table des matières

Première partie

Curriculum Vitæ

Deuxième partie

Une courte présentation du sujet de recherche

Une approche logique de la spécification et de la vérification des systèmes temporisés réactifs

Contexte

Les systèmes réactifs temps réels sont généralement spécifiés dans une logique temporisée classique. L'algorithmique est ensuite basée, d'une façon ou d'une autre, sur des techniques d'automates. Okada, Kanovitch et Scedrov ont eu l'idée d'utiliser une logique linéaire temporisée, qui permet de raisonner en termes de consommation de ressources. Ils ont montrés d'une part que le fragment linéaire considéré (fragment monadique de Horn) permettait d'écrire des spécifications de systèmes temporisés très naturellement, et d'autre part qu'il y avait de bonnes propriétés de décidabilité (problèmes de sureté, d'ordonnacement, et c), comme les logiques temporelles classiques.

Dans ce cadre, les systèmes temps-réels à états finis peuvent être spécifiés comme implications linéaires entre conjonction s de longueurs finies (et fixées). En traitant le temps comme un ordre linéaire dense, les problèmes de sureté peuvent être vu comme des problèmes de prouvabilité (décidables en PSPACE pour cette classe de problèmes). Un des avantages de cette méthode est que soit elle etablie la propriété de sureté, soit elle renvoie un *run* non-sur.

Cette thèse s'inscrit dans le cadre du projet *Fatalis*, qui est un langage de spécifications couplé à un prototype de démonstrateur permettant de vérifier des propriétés des systèmes temps réels(pour les problèmes de sureté, d'ordonnancement, etc).

Méthodes et objectifs

Le but de cette thèse est d'explorer les différentes méthodes et outils que peut apporter ce cadre logique unique.

Les perspectives offertes par cette approche sont nombreuses :

- Mise en place d'une intéraction entre l'assistant de preuve Coq et *Fatalis*.
- Possibilités de spécifications et de vérification pour certaines classes de systèmes temps-réels avec un nombre d'agents variable.
- Utilisation du paradigme "preuve en logique linèaire - processus (temps réels)".
- Interprétations des autres connecteurs de la logique linéaire (connecteurs additifs) du point de vue des spécifications.
- Applications possibles aux systèmes temps-réels avec un nombre de transitions infini.

- Applications possibles aux protocoles de communication sécurisés avec *time-stamp*.

Troisième partie

Rapport de DEA : Spécification et vérification de systèmes temporisés réactifs en logique linéaire : Paramétrisation de preuves

Chapitre 1

Introduction

Les systèmes réactifs temps-réel sont généralement spécifiés dans une logique temporisée classique. L'algorithmique est ensuite basée, d'une façon ou d'une autre, sur des techniques d'automates. Kanovitch, Okada et Scedrov [1] ont eu l'idée d'utiliser une logique linéaire temporisée, qui permet de raisonner en termes de consommation de ressources. Ils ont montré d'une part que le fragment linéaire considéré permettait d'écrire des spécifications de systèmes temporels très naturelles, et d'autre part qu'il avait de bonnes propriétés de décidabilité, (en PSPACE pour les problèmes de sureté, ordonnancement, etc.), comme les logiques temporelles classiques.

Un premier stage éffectué l'an dernier par Vincent Cremet a permis l'élaboration d'un langage de spécification basé sur ces travaux, et d'un prototype de démonstrateur permettant de vérifier des propriétés de sureté, d'ordonnancement, etc., de systèmes logiciels réalisant une telle spécification. Cependant ce vérificateur n'utilisait qu'une formalisation en termes de graphe de ces problèmes, qui ne permettait pas la vérification de propriétés paramètrées par des variables temporelles.

Dans ce rapport, nous commencerons par présenter le formalisme logique ainsi que les outils théoriques nécéssaires à la formalisation de systèmes réactifs temps-réels avec une logique linéaire.

Ensuite, nous presenterons un algorithme donnant une condition nécéssaire sur la non-sûreté de systèmes temps-réels, cependant, seul le cas du temps discret sera traité dans ce rapport.

Enfin, nous mettrons en exergue les prolongements possibles de ce stage tels qu'ils sont envisagé pour le futur, ainsi que les diverses difficultés d'ordre techniques rencontrées.

Chapitre 2

Logique linéaire, alarmes et spécifications

D'après des idées de Okada, Kanovitch et Scedrov [1], il a été montré que le fait de spécifier un système temporisé réactif avec un fragment multiplicatif de la logique linaire permettait, en traitant le temps comme un ordre linéaire dense, d'exprimer des problèmes de *sureté* comme une instance d'un problème de prouvabilité.

On peut spécifier des propriétés temporelles qualitative telle que *sometimes, always, never* en considérant seulement des formules où toutes les variables temps sont quantifiées.

Ces propriétés peuvent être vue en termes de logiques temporelles où toutes les variables temps sont liées par l'intermédiaire d'opérateurs modaux ou de quantificateurs.

En revanche, lorsqu'il est nécéssaire de manipuler des propriétés temporelles *quantitatives*, où il est fait référence à des délai de temps *explicites*, le cadre des logiques temporelles s'avère bien moins adapté.

Exemple :

- <<tel évènement doit impérativement se produire avant B unités de temps>>
- <<tel évènement ne doit pas se produire avant B unités de temps>>

Pour pouvoir manipuler ce type de propiétés dans une logique temporelle, on doit pouvoir représenter l'état courant d'un système temps-réel et, pour cela avoir à disposition une représentation *explicite* du temps.

2.1 Spécification dans une logique linéaire

L'idée est ici de marquer chaque évènement avec son instant d'éxécution, par exemple P(e,t) pour "l'évènement e a lieu à l'instant t"

Le problème des prédicats binaires, qui causent souvent l'indécidabilité du système peut être ici contourné en scindant le prédicat *binaire* en deux prédicats *unaires.*

Ainsi notre prédicat P se décompose en un prédicat *Time* et un prédicat *Q(e)* pour indiquer qu'un évènement est survenu dans le système.

Ainsi on aura : $P(e,t) = Q(e)\ AND\ Time(t)$

Pour décrire un sytème temps-réel, il est également nécéssaire de pouvoir spécifier des *transitions* d'états (temporisées ou non).

Une manière simple de faire serait d'utiliser une notion d'implication. Ainsi un évènement *e1* se produisant à l'instant *t1* qui est suivi d'une action *e2* à l'instant *t2* pourra être spécifié de la manière suivante :

(1) $Q(e1)\ AND\ Time(t1)\ IMPLIES\ Q(e2)\ AND\ Time(t2)$.

Il est bien entendu indispensable de s'attarder maintenant sur l'interpretation à donner à ces connecteurs logiques. Une vision en termes de logique propositionelle classique ou intuitioniste s'avère ici totalement inadaptée, et ce pour la même raison qu'il est possible de prouver $A \Rightarrow A \wedge A$. En effet dans de tels cadres logique, pour reprendre l'exemple (1) ,en interprétant AND par $\wedge$ et IMPLIES par $\Rightarrow$, de prouver par exemple que : $Q(e1) \wedge Time(t1) \Rightarrow Q(e2) \wedge Time(t1) \wedge Time(t2)$.

Dans ces conditions, l'utilisation d'un cadre logique prenant en compte l'utilisation des ressources est indispensable. La logique linéaire s'impose donc comme un cadre logique adéquat pour la modélisation de tels comportements. En effet, dans ce formalisme le connecteur $\wedge$ est décomposé en deux connecteurs linéaires : $\otimes$ et & . Le connecteur $\otimes$ pouvant être vu comme désignant *l'existence concurrente.*

L'implication linéaire quand à elle, est considérée dans *l'interprétation capitaliste* des connecteurs lineaires comme l'opérateur permettant d'obtenir la conclusion "en échange" de la prémisse (ie :la prémisse est consommée).

Dans un système temps-réel, on représentera alors les *agents* du système comme un prédicat unaire sur leur états, l'information sur le temps étant quant à elle contenue dans un prédicat unaire *Time* sur un ordre linéaire dense R contenant les entiers naturels.

Il est donc désormais possible de spécifier des évènements retardés (telle action à l'instant t1 est suivie d'une autre action à l'instant t2), mais il nous est toujours impossible de gèrer des *propriétés quantitatives* sur le temps. C'est dans ce but que [1], introduit l'idée d'alarmes pour faire respecter certaines contraintes de temps au système.

2.2 Les alarmes

Dans le but de faire respecter au système des contraintes de temps, on introduit la notion d'alarmes *indépendantes* attachées à chacune de ces contraintes. Les alarmes seront représentées par des prédicats unaires sur $R \bigcup \{+\infty\}$ ($+\infty$ est la valeur choisie pour éteindre l'alarme). Notons également que dans la suite du rapport, pour des raisons de simplicté, toutes les alarmes seront considérées comme strictes.

Dans cette approche, on distingue deux types d'alarmes :

2.2.1 Les contraintes hautes

Les contraintes de temps hautes, permettent de spécifier qu'un évènement e_1 doit être suivi d'un autre évènement e_2 *avant* que l'alarme arrive à son terme. En d'autres termes, lorsque e_1 se produit, l'alarme associée à cette contrainte est reglée pour se déclencher un certain nombre d'unités (fixé) de temps après le temps courant. Dans la suite, nous appelerons *borne* de l'alarme, l'intervalle de temps entre lequel une alarme est reglée et celui où elle se déclenche.

2.2.2 Les contraintes basses

Les contraintes basses permettent de spécifier qu'un évènement *ne doit pas se produire avant* que l'alarme ne soit arrivée à son terme. La notion de *borne* est identique à celle définie pour les alarmes hautes.

2.3 Spécifications

Un système temps réels est composé d' *agents Ag1,...,Agn* pouvant prendre différents états, d' alarmes hautes $Hi_1, ..., Hi_k$ de bornes respectivces $hi_1, ..., hi_k$. De même, on notera $low_1, ..., low_l$ les bornes respectives des alarmes basses $Low_1, ..., Low_l$ du système. Bien évidemment $hi_1, ..., hi_k, low_1, ..., low_k \in R$.

Dans la suite de ce rapport, nous noterons :
$Etats(e) \equiv Ag1(e_1) \otimes Ag2(e_2) \otimes ... \otimes Agn(e_n)$
$Alarmes(al) \equiv (Hi_1(al_1) \otimes ... \otimes Hi_k(al_k)) \otimes (Low_1(al_{k+1}) \otimes ... \otimes Low_l(al_{k+l}))$

Nous considérerons également un prédicat binaire $<$ sur $R \bigcup \infty$ qui a la signification habituelle d'ordre strict. Pour cela nous ajoutons à la logique les axiomes $(x < y) \vdash 1$ pour toutes les instances vraies de $(x < y)$ $(x, y \in R \bigcup \infty)$.

Une spécification S est alors l'ensemble formé d'un axiome d'écoulement du temps et d'un nombre fini d'axiomes de transition d'états.

2.3.1 Ecoulement du temps

Cet axiome simule l'écoulement du temps en prenant garde à ne pas dépasser les dates limites de toutes les alarmes hautes.
$Time(s) \otimes Hi_1(a_1) \otimes ... \otimes Hi_k(a_k) \otimes (s < t) \otimes Less(a, t) \vdash$
$Time(t) \otimes Hi_1(a_1) \otimes ... \otimes Hi_k(a_k)$
où $Less(a, t) = (a_1 < t) \otimes ... \otimes (a_k < t)$

2.3.2 Transitions d'état

Les transitions d'état, c'est à dire les axiomes ne modifiant que les états des agents du système et les valeurs des alarmes, peuvent être en deux types :

Transitions non-temporisées

Ces transitions sont les plus simples, elle ne modifient pas les valeurs des alarmes, ces axiomes sont donc de la forme : $Etats(e_1) \vdash Etats(e_2)$.

Transitions temporisées

Ces transitions modifient non seulement les états des *agents* du système, mais également les valeurs des alarmes.

$Time(t) \otimes Etats(e) \otimes SHi(w) \otimes SLow(x) \otimes RHi(y) \otimes Less(y, +\infty) \otimes RLow(z) \otimes Lbound(z, t) \vdash Time(t) \otimes Etats(e^{'}) \otimes SHi(x|t + hi) \otimes SLo(t + low) \otimes RHi(\infty) \otimes RLo(\infty)$

où :

- SLow repésente les alarmes basses allant être enclenché.
- RLow représente les alarmes basses qui doivent être eteintes.
- $Lbound(z, t) \equiv (z_1 < t) \otimes ... \otimes (z_l < t)$.
- $x||t + hi = t + hi$ si $x = \infty$ et x sinon.

On defini de manière identique SHi et RHi.

2.4 L'outil de vérification *fatalis*

Un stage de DEA de l'année précédente a permis l'élaboration d'un outil de spécification et de vérification des systèmes temps-réels. Ce logiciel, utilisant une représentation en termes de graphes pour les éxécutions permet de résoudre le problème de sureté qui est alors vu comme un problème d'accessibilité dans un graphe. Ce logiciel, également basé sur les traveaux de [1], repose sur le théorème d'élimination du temps, expliqué ci-dessous.

2.4.1 Théorème d'élimination du temps pour le problème de sureté

Lorsqu'on a spécifié un système dans une logique linéaire, la question qui nous importe est alors de savoir si telle configuration initiale codée dans ce formalisme, peut aboutir à telle configuration finale. Ce problème de sûreté peut alors être vu comme un problème de *satisfiabilité* dans la logique linéaire.

Ainsi, on voudra prouver :
$Time(t_0) \otimes Etats(e_0) \otimes Alarmes(al_0) \vdash \exists t, e, al\ Time(t) \otimes Etats(e) \otimes Alarmes(al)$
Ce problème, de *prouvabilité paramètrique* est résoluble en espace polynomial en le nombre d'axiomes du sytème et les valeurs des bornes des alarmes.

L'idée est d'éliminer l'information explicite du temps de la spécification et de se ramener ainsi à une recherche finie de preuves en ne gardant sur les alarmes que les informations suivantes :

- Pour chaque alarme *haute* A à allumer : soit A est déjà allumée auquel cas elle garde sa valeur, soit elle est éteinte et elle est initialisée à 0.
- Chaque alarme *basse* à allumer est initialisée à 0.
- Pour chaque alarme *haute* A à eteindre, on doit avoir $al(A) \neq Off$ et $al^{'}(A) = Off$.
- Pour chaque alarme A de type *basse* à éteindre, on doit avoir $al(A) = Top$ et $al^{'}(A) = Off$.

On introduit alors la règle long_tick qui permet de simuler l'écoulement d'une unité de temps.

long_tick

$Etats(e) \otimes Alarmes(al) \vdash Etats(e^{'}) \otimes Alarmes(al^{'}$
si :

- $e = e^{'}$
- chaque alarme *haute* allumée est incrémentée d'une unité et vérifie : $al(A) \leq borne(A) - 1$
- chaque alarme *basse* allumée est incrémentée d'une unité si elle n'atteint pas la valeur de sa borne. Dans le cas contraire, on fixe l'alarme sur la constante spéciale *Top*.

Les règles de transition, sont, elles modifiées de la manière suivante :

transition d'états

A partir d'une α-transition $Etats(e) \otimes Alarmes(al) \vdash Etats(e^{'}) \otimes Alarmes(al^{'})$
si :

- Pour chaque alarme *haute* A à allumer : soit A est déjà allumée auquel cas elle garde sa valeur, soit elle est éteinte et elle est initialisée à 0.
- Chaque alarme *basse* à allumer est initialisée à 0.
- Pour chaque alarme *haute* A à éteindre, on doit avoir $al(A) \neq Off$ et $al^{'}(A) = Off$.

– Pour chaque alarme A de type *basse* à éteindre, on doit avoir $al(A) = Top$ et $al^{'}(A) = Off$.

Si on considère un état global du sytème dans la spécification réduite : $Etats(e) \otimes Alarms(al)$, on s'aperçoit que :

– le nombre de configuration possible pour e est fini (chaque agent du système a un nombre fini d'états).

– chaque alarme *basse* (stricte) de borne B peut prendre B+3 valeurs, en comptant les valeurs spéciales *Off* et *Top*.

– chaque alarme *haute* (stricte) de borne B a B+1 configurations possibles ($0 \leq al \leq B$ ou al = *Off*).

L'espace de recherche est donc désormais fini, le problème est décidable (PSPACE en fait). Il est montré dans [1] et [2] que cette transformation est correcte et complète.

Chapitre 3

Paramètrisation de preuves de run non-sûr

3.1 Paramétrisation de preuve de sureté en logique linéaire : Le cas du temps discret

Une fois le problème de sureté résolu pour des valeurs concrètes des bornes, une extension intéressante est de paramétriser la spécification en considérant les valeurs des bornes des alarmes comme des variables non nulles.

En effet, à partir de la preuve de sureté d'une éxécution du système, nous allons construire la preuve correspondante de cette éxécution lorsque le système est paramétré par les valeurs des bornes de ses alarmes.

Pour cela, à partir de la preuve *non-temporisée* d'un run concret (ie :après élimination du temps), nous allons extraire des contraintes sur les bornes des alarmes.

Soit la spécification non-temporisée $S = ST \bigcup \{long\_tick\}$, où ST est l'ensemble des axiomes ayant trait aux transitions d'états et long_tick est la règle d'écoulement unitaire du temps décrite dans le théorème d'élimination du temps.

Soit *Init* et *Reach*, deux états du système tels que $Init \vdash Reach$ soit prouvable dans la logique linéaire intuitioniste enrichie de l'ensemble d'axiomes S.

Posons la définition suivante : On appelera long_long_tick(n) le schéma d'axiome suivant : $Etats(e) \otimes Alarmes(al) \vdash Etats(e^{'}) \otimes Alarmes(al^{'})$
si :

- $e = e^{'}$.
- Toutes les alarmes hautes allumées sont incrémentées de n.
- Chaque alarme basse allumée dans *al* est incrémentée de n, sauf si elle dépasse sa borne, auxquel cas elle passe à la valeur spéciale *Top*.
- Pour chaque alarme haute allumée A dans *al*, on a $valeur(A) + n \leq borne(A) - 1$.

Cette règle simule en fait l'application répétée de plusieurs règles *long_tick*.

Posons maintenant $S^* = ST \bigcup \{long\_long\_tick(\_)\}$

On prouve aisément la propriété suivante :

Propriété : Les deux affirmations suivantes sont équivalentes :

- $Init \vdash Reach$ est prouvable dans $LL + S$
- $Init \vdash Reach$ est prouvable dans $LL + S^*$

Nous pouvons désormais penser à la paramètrisation de nos preuve de *run* non-sûr.

En effet, supposons que $Init \vdash Reach$ soit prouvable dans LL+S. D'après la propriété précedente, on a que $Init \vdash Reach$ est prouvable dans $LL + S^*$.

S'il on paramètrise la preuve de $Init \vdash Reach$ dans $LL + S^*$ en remplacant les valeurs concrètes des bornes des alarmes par des variables, il est inutile de garder les valeurs concrètes sur la "taille" des *long_long_tick(n)*, en effet l'une des conditions qui doit être satisfaite pour l'application d'un *long_long_tick* est que les alarmes allumées et incrémentées ne franchissent pas leur borne.

On remplace donc, chaque application d'une règle long_long_tick(_) par l'application d'une règle $long\_long\_tick(\alpha_i)$, les α_i devant être syntaxiquement distincts à chaque nouveau remplacement.

On obtient alors une paramètrisation totale de la preuve concrète initiale. On peut également noter que l'arbre preuve paramètrée n'a pas de branche infinie.

3.2 Génération des contraintes à partir d'une preuve paramètrée

L'idée sous jacente du théorème d'élimination du temps est que l'applicabilité d'une règle long_tick (et donc, *a fortiori* d'une règle long_long_tick) est conditionnée par des conditions intervenant au niveau *méta* de la règle (cf déf. long_tick et long_long_tick).

L'algorithme qui suit, a pour ambition de récupérer ces informations, par un parcours de l'arbre (fini) de la preuve paramètrisée. Dans toute la suite, on considère que les paramètres des bornes des alarmes sont strictement positifs. Ce choix se justifie en pratique car l'utilité d'une alarme qui sonne dès qu'on l'enclenche est assez limité.

Durant le parcours de l'arbre de preuve, à chaque utilisation de l'axiome long_long_tick(α_i) pour prouver $Etats(e) \otimes Alarmes(al) \vdash Etats(e) \otimes > Etats(e) \otimes Alarme(al^{'})$ les contraintes suivantes sont générées :

- de chaque alarme *haute* Hi_i de borne hi_i allumée de valeur x dans le membre gauche, on extrait la contrainte : $\alpha_i + x \leq hi_i - 1$.
- de chaque alarme *basse* Low_i de borne low_i allumée de valeur x passant à *Top* après un *long_long_tick*, on extrait la contrainte suivante :$low_i \leq x + \alpha_i$.

- de chaque alarme *basse* Low_i de borne low_i allumée de valeur *Top*, on a deja une contrainte de la forme $low_i < x$, on ajoute alors la contrainte $low_i < x + \alpha_i$.

On pourrait se demander pourquoi cet algorithme ne construit pas de contrainte pour les alarmes *basse* qui ne sont pas arrivées à leur terme, mais le simple fait que la valeur de cette alarme soit différente de *Top* nous assure que l'alarme n'a pas dépassé sa borne.

Notons que nos spécifications n'utilisent que des tenseurs, cela nous evite d'avoir à gérer deux ensembles d'inégalités distincts, comme cela pourrait être le cas avec l'utilisation de connecteurs additifs comme $\oplus$.

3.3 Extraction d'une condition suffisante pour le problème de sûreté

Une fois l'ensemble de contraintes généré comme il a été décrit dans la section précédente, on peut, en ajoutant ces contraintes comme hypothèses du systèm

Pour que la preuve paramètrée obtenue en section précédente soit bien une preuve, il faut s'assurer que chacune des *méta* conditions associée à l'application d'une règle long_long_tick soient vérifiées. Ces conditions sont :

- les bornes des alarmes hautes ne dépassent pas leur date d'échéance apres incrémentation.
- pour les alarmes basses allumées qui passe à la valeur spéciale *Top*, on doit s'asssurer que la borne de l'alarme est plus petite que la somme des valeurs des long_long_tick appliqués à cet alarme.

Ainsi, en ajoutant en hypothèse l'ensemble C de contraintes générées lors de l'éxécution de l'algorithme présenté ci-dessus on obtient le résultat suivant :

$C \Rightarrow Init \vdash Reach$.

Le chapitre suivant va détailler, sur l'exemple classique du système train-passage à niveau, l'obtention de cette condition suffisante.

Chapitre 4

Variation sur un thème : Le passage à niveaux

Pour illustrer l'algorithme présenté au chapitre précédent, continuons avec l'exemple du controleur de passage à niveaux.

Dans ce cas precis, nous allons considérer un système très simple de passage à niveaux et détailler toutes les étapes permettant d'obtenir une condition suffisante pour un système non-sûr.

- Un seul train (un agent du système donc) avec trois états : *safe, crossing* et *approaching*
- Une barrière pouvant prendre 3 états distincts : *up, down* et *moving_down* (abgrégé dans la suite *mv_d*)
- Trois alarmes de type hautes (strictes) Hi_1, Hi_2 et Hi_3 de bornes respectives Ug_1, Ug_2 et Ug_3.
- Une alarme de type basse (stricte) *Low* de borne *Lc*.

Cet exemple n'est bien-sûr pas très réaliste en ce sens qu'on ne se préoccupe pas de ce qui se produit après que le train soit passé (pas de règle pour l'ajout d'un nouveau train, pas d'état *moving_up* pour le passage, etc...)

Cet exemple est donc destiné à modeliser l'arrivée d'un unique train à l'approche.

Les axiomes modélisant le système sont donc crées en conséquence.

4.1 Les axiomes : le détail

Informellement, la spécification du système que nous voulons décrire serait la suivante :

- Il s'écoule au moins *Lc* unités de temps entre le moment où le train entre en zone d'approche et le moment où il commence à traverser mais il ne peut pas traverser tant que *Lc* unités de temps ne sont pas écoulées.

- Il s'écoule au plus *Ug1* unités de temps entre le moment où le train passe de l'état *safe* à l'état *approaching* et le moment où la barrière commence à descendre.
- Il s'ecoule au plus *Ug2* unités de temps entre le moment où la barrière est dans l'état *moving_down* et le moment où elle est dans l'état *down*.

4.2 Les axiomes, plus formellement

Une fois les spécifications décrites en langage naturel, ou formalisées dans l'outil de vérification *fatalis*, il faut les traduire en termes de logique linéaire : Dans la suite on notera : $(x|y) = x$ si $x < +\infty$, y sinon.

- t1 : $\forall t, l$
  $(Time\ \ t) \otimes (Tr\ \ safe) \otimes (Hi_1\ \ hi1) \otimes (Low\ \ low) \vdash$
  $(Time\ \ t) \otimes (Tr\ \ appr) \otimes (Hi_1\ \ (hi_1|t + Ug1) \otimes (Low\ \ t + Lc)$
- t2 : $\forall l$
  $(Time\ \ t) \otimes (Tr\ \ approaching) \otimes (Low\ \ l) \otimes (l < t) \vdash (Tr\ \ crossing) \otimes (Low\ \ \infty)$
- t3 : $\forall t$
  $(Tr\ \ crossing) \vdash (Tr\ \ safe)$
- g1 :
  $(Time\ \ t) \otimes (Gate\ \ e) \otimes (Hi_1\ \ x) \otimes (Hi_2\ \ y) \otimes (x < +\infty) \vdash$
  $(Time\ \ t) \otimes (Gate\ \ mv\_down) \otimes (Hi_1\ \ \infty) \otimes (Hi_2\ \ (y|t + Ug_2))$
- g2 :
  $(Gate\ \ mv\_down) \otimes (Hi_2\ \ x) \otimes (x < +\infty) \vdash (Gate\ \ down) \otimes (Hi_2\ \ \infty)$
- g3 :
  $(Tr\ \ far) \otimes (Gate\ \ down) \vdash (Tr\ \ far) \otimes (Gate\ \ up)$
- time_progress :
  $(Time\ \ s) \otimes (Hi_{1,2,3}\ \ x\ \ y\ \ z) \otimes (x \leq t) \otimes (y \leq t) \otimes (z \leq t) \vdash (Time\ \ t) \otimes (Hi_{1,2,3}\ \ x\ \ y\ \ z)$

Bien que la spécification précédente soit conforme à la réalité physique, il est notable que les preuves réalisées à partir de ces axiomes ne sont pas très lisibles. En effet, la logique linéaire prend en compte les ressources, ainsi tous les prédicats présents dans un membre gauche d'axiome doivent également être présents dans le membre droit. Ainsi, si un agent du système ne change pas d'état, il peut être omis dans l'axiome.

Par contre, le fait de garder dans la spécification chacun des agents, même ceux qui restent inchangés par l'application de la règle, permet de construire des preuves plus lisibles beaucoup plus facilement. Une telle spécification est la suivante :

Dans les axiomes qui suivent, on note : $Alarmes(h_1, h_2, h_3, l) \equiv Hi_1(h_1) \otimes Hi_2(h_2) \otimes Hi_3(h_1) \otimes Low(l)$
et $Etats(v, g) \equiv Tr(v) \otimes Gate(g)$

- t1 :
  $(Time\ \ t) \otimes (Etats\ \ safe\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low) \vdash$
  $(Time\ \ t) \otimes (Etats\ \ appr\ \ g) \otimes (Alarmes\ \ hi_1|t + Ug1\ \ hi_2\ \ hi_3\ \ t + Lc)$
- t2 :
  $(Time\ \ t) \otimes (Etats\ \ appr\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low) \otimes (low < t) \vdash$
  $(Time\ \ t) \otimes (Etats\ \ cross\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ \infty)$
- t3 :
  $(Time\ \ t) \otimes (Etats\ \ cross\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low) \vdash$
  $(Time\ \ t) \otimes (Etats\ \ safe\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low)$
- g1 :
  $(Time\ \ t) \otimes (Etats\ \ tr\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low) \otimes (hi_1 < +\infty) \vdash$
  $(Time\ \ t) \otimes (Etats\ \ tr\ \ mv\_down) \otimes (Alarmes\ \ \infty\ \ (hi_2|t+Ug2)\ \ hi_3\ \ low)$
- g2 :
  $(Time\ \ t) \otimes (Etats\ \ tr\ \ mv\_down) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low) \otimes (hi_2 < +\infty) \vdash$
  $(Time\ \ t) \otimes (Etats\ \ tr\ \ down) \otimes (Alarmes\ \ hi_1\ \ \infty\ \ hi_3\ \ low)$
- g3 :
  $(Time\ \ t) \otimes (Etats\ \ safe\ \ down) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low) \vdash$
  $(Time\ \ t) \otimes (Etats\ \ safe\ \ up) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low)$
- time_progress :
  $(Time\ \ t) \otimes (Etats\ \ tr\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low) \otimes (LESS\ \ s\ \ hi_1\ \ hi_2\ \ hi_3) \vdash$
  $(Time\ \ s) \otimes (Etats\ \ tr\ \ g) \otimes (Alarmes\ \ hi_1\ \ hi_2\ \ hi_3\ \ low)$
  où :
  $(LESS\ \ s\ \ hi_1\ \ hi_2\ \ hi_3) \equiv (hi_1 \leq s) \otimes (hi_2 \leq s) \otimes (hi_3 \leq s)$

4.3 Un exemple de preuve paramétrée

Prenons un exemple avec des valeurs concrètes des bornes des alarmes et essayons de parametriser la preuve d'un *run* non-sûr. Jusqu'à la fin du chapitre, nous considérerons que $Ug_1 = 3, Ug_2 = 2, Ug_3 = 1$ et $Lc = 2$. On pose également :
$Init \equiv (State\ \ safe\ \ up) \otimes (Alarms\ \ Off\ \ Off\ \ Off\ \ Off)$
$Reach \equiv \exists g, h_1, h_2, h_3, l(State\ \ crossing\ \ g) \otimes (Alarms\ \ h_1, h_2, h_3, l)$

4.3.1 Obtention d'une preuve d'un run non-sûr

Par l'intermédiaire de l'outil *fatalis*, on extraie la preuve (sans l'information sur le temps) d'une éxécution non-sûre :

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{Init \vdash Init}\ Ax}{Init \vdash (State\ \ approaching\ \ up) \otimes (Alarms\ \ 0\ \ Off\ \ Off\ \ 0)}\ Cut+t1}{Init \vdash (State\ \ approaching\ \ up) \otimes (Alarms\ \ 1\ \ Off\ \ Off\ \ 1)}\ Cut+long\_tick}{Init \vdash (State\ \ approaching\ \ moving\_down) \otimes (Alarms\ \ Off\ \ 0\ \ Off\ \ 1)}\ Cut+g1}{Init \vdash (State\ \ approaching\ \ moving\_down) \otimes (Alarms\ \ Off\ \ 3\ \ Off\ \ Top)}\ Cut+long\_tick}{Init \vdash Reach}\ Cut+t2
$$

Puis on paramétrise la preuve en remplacant toutes les éxécutions d'une ou plusieurs règles *long_tick* par une règle *long_long_tick* (qui simule l'application de plusieurs *long_tick* succéssifs) comme décrit dans le chapitre précédent, on ajoute en même temps les contraintes générées aux hypothèses.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\overline{Init \vdash Init}\ Ax}{Init \vdash (State\ \ appr\ \ up) \otimes (Alarms\ \ 0\ \ Off\ \ Off\ \ 0)}\ Cut+t1}{\Delta_1 \otimes Init \vdash (State\ \ appr\ \ up) \otimes (Alarms\ \ \alpha\ \ Off\ \ Off\ \ \alpha)}\ Cut+l\_l\_tick}{\Delta_1 \otimes Init \vdash (State\ \ appr\ \ mv\_d) \otimes (Alarms\ \ Off\ \ 0\ \ Off\ \ \alpha)}\ Cut+g1}{\Delta_2 \otimes Init \vdash (State\ \ appr\ \ mv\_d) \otimes (Alarms\ \ Off\ \ \beta\ \ Off\ \ Top)}\ Cut+l\_l\_tick}{\Delta_2 \otimes Init \vdash Reach}\ \exists right+Cut+t2}{\exists \alpha, \beta \quad Delta_2 \otimes Init \vdash Reach}\ \exists left
$$

où :

$$
\begin{aligned}
\Delta_1 &= (\alpha \leq x-1) && (4.1)\\
\Delta_2 &= (z < \alpha+\beta) \otimes (\beta \leq y-1) \otimes \Delta_1 && (4.2)\\
& && (4.3)
\end{aligned}
$$

On a ensuite : $z < x+y-2 \Rightarrow \exists \alpha, \beta(z < \alpha+\beta) \otimes (\beta \leq y) \otimes (\alpha \leq x)$.

D'où le résultat : $z < x+y-2 \Rightarrow Init \vdash Reach$

Et on peut egalement prouver que $\exists \alpha, \beta(z < \alpha+\beta) \otimes (\beta \leq y) \otimes (\alpha \leq x) \Rightarrow z < x+y-2$

La solution est donc optimale.

Une implantation de cet algorithme par un interfaçage entre l'assistant de preuve Coq et le logiciel de vérification *fatalis* est actuellement en cours. C'ette implantation est basée sur un codage de la logique linéaire dans CCI [3].

Chapitre 5

Conclusion

Ce stage a permi d'étudier les possibilités offertes par l'utilisation de la logique linéaire dans le domaine de la spécification et de la vérification de systèmes temps-réels, dans le cas d'un problème de sureté en temps discret.

L'algorithme présenté permet d'une part, de nous donner une condition nécéssaire relativement simple à la non-sureté d'un système temps-réels dans le cas du temps discret. D'autre part, de laisser espèrer de telles propriétés dans le cas de systèmes avec temps dense.

Une automatisation en coq de l'algorithme présenté en chap. 2 est en cours de développement, celle ci s'articulant autour du logiciel *fatalis*. La

Les extensions possibles de ce stage de DEA sont nombreuses. Il est déjà possible d'essayer d'étendre l'algorithme présenté en chapitre 2 au cas d'un temps linéaire dense. D'autre part seule le problème de sureté est pour l'instant traité, il serait intéréssant de voir ce que peut apporter la logique linéaire dans d'autres types de problèmes comme le rendez-vous, etc... Et ce par l'intermédiaire de l'utilisation de connecteurs additifs (pour coder l'alternative) ou encore de quantificateurs existentiels. Mieux comprendre les intéractions entre les spécifications en logique linéaire et les modèles tels que les automates temporisés...

D'un point de vue plus pratique, une autre possibilité est d'améliorer les algorithmes de parcours de graphe actuellement utilisés dans *fatalis* et intégrer l'assistant de preuve *Coq* de manière plus naturelle dans le système.

Quatrième partie

Rapport de maîtrise : Conception d'un outil graphique de génération de réseaux connexionnistes

Chapitre 6

Introduction

Ce stage a été réalisé au L.I.S (Laboratoire d'Instruments et Systèmes) sous la direction de Bertrand GRANADO.
Son objectif est de concevoir un logiciel permettant de réaliser graphiquement des réseaux de neurones par l'intermédiaire d'une interface au langage Aspirin, langage dédié à la description des réseaux de neurones de type perceptron multi-couches (designes par PMC dans la suite du rapport). Dans ce rapport, nous commencerons par brosser un bref tableau sur les réseaux de neurones et leur utilité pratique. Ensuite, nous étudierons en détail les outils utilisés pour mettre en oeuvre ce logiciel ainsi que les choix technologiques effectués pour résoudre les problèmes posés. Enfin, nous mettrons en exergue les prolongements possibles de ce projet ainsi que les difficultés recontrées.

Chapitre 7

Réseaux de Neurones

Le principe des réseaux de neurones ou réseaux connexionnistes est calqué sur le fonctionnement de leur proche parent biologique. En effet, le fonctionnement d'un ensemble de neurones "biologiques" est basé sur une transmission de potentiels. Un neurone perçoit une excitation et émet en réponse un potentiel P (plusieurs en fait). Ce potentiel est, ensuite, retransmis et modulé en fréquence par d'autres neurones au terme d'un processus chimique et physique que nous ne détaillerons pas ici.
Les neurones biologiques se comportent donc comme autant de micro-calculateurs sans grande puissance de calcul individuelle mais dont l'intégration au sein d'un réseau organisé permet la résolution de problèmes non triviaux.
Les réseaux de neurones formels sont régis par le même principe : chaque neurone est representé, à un instant donné, par une valeur caractéristique appelée *état*. L'évolution de cette valeur au cours du temps est fonction des états des neurones adjacents, c'est à dire les neurones lui étant connectés. Ces connexions ont souvent appelées *synapses* par analogie au modèle biologique, elles possédent des valeurs appelées *poids synaptique*. Un réseau de neurones formels peut donc être représenté par un graphe valué. Les réseaux connexionnistes sont donc des algorithmes non plus programmés mais entraînés pour la réalisation d'une tâche. Il y a deux phases principales dans la "vie" d'un réseau de neurones : l'apprentissage et la phase de reconnaissance.

La phase d'apprentissage se limite au calcul de la fonction d'évaluation du réseau, l'apprentissage peut être supervisé ou non, ie : on peut lancer l'apprentissage sur un jeu de données types, ou bien laisser le réseau "apprendre sur le tas". En fait, plus que le calcul de la fonction d'évaluation du réseau, la phase d'apprentissage réalise une approximation de celle-ci, ce qui induit nécéssairement une notion d'erreur. Cette notion d'erreur est souvent utilisée pour réaliser, outre le calcul initial, une classification des entrées. Lors de cette phase le réseau règle ses paramètres, notamment les poids synaptiques, dans l'optique de minimiser l'erreur totale du réseau.

L'apprentissage est une étape critique pour le réseau de neurones, en effet, d'un trop court apprentissage, ou d'un apprentissage trop peu diversifié résultera un réseau *sous-entraîné*, c'est à dire que le réseau ne sera pas assez sélectif, l'erreur dans le calcul de la fonction sera trop grande pour pouvoir envisager un traitement utlerieur des données. Inversement, d'un apprentissage trop ciblé résultera un réseau *sur-entraîné*, lors de la phase de reconnaissance, le réseau ne pourra reconnaître qu'un nombre restreint d'entrée.

La phase de reconnaissance, quant à elle consiste aux calculs des états des neurones du réseau. Le calcul de l'état d'un neurone donné s'effectue en deux temps : dans un premier temps, on détermine une valeur interne au neurone appelée *potentiel post-synaptique*, fonction des état de tous les neurones amont connectés au neurone en question. Cette fonction est souvent une somme pondérée des états des neurones amont par les poids synaptiques, soit un calcul de distance (Euclide, Manhattan, Mahalanobis...). Ensuite, l'état du neurone est calculé comme fonction du potentiel post-synaptique (une sygmoïdale en règle générale). Ce calcul se propage ensuite pour tous les neurones du réseau.

A partir d'ici, nous ne ferons plus la différence entre réseau de neurones et réseau de neurones formels, la confusion avec les neurones biologiques étant impossible dans le cadre de ce stage.

On distingue deux familles principales de réseaux de neurones : les Perceptrons Multi-Couches (PMC) qui sont l'objet du stage, et les RBF qui pourraient être utilisés dans une extension du projet.
La principale différence entre ces deux types de réseaux est la topologie des connexions.
Pour plus de détails sur le fonctionnement des réseaux de neurones, le lecteur interèssé pourra se reporter à [8].

7.1 Les perceptrons multi-couches (PMC)

Les PMC sont des réseaux connexionnistes constitués de trois types de couches interconnectées, d'une part, la couche d'entrée et la couche de sortie (généralement unique) et les couches cachées d'autre part. Ces couches peuvent être indifféremment mono-dimensionnelles ou bi-dimensionnelles. La particularité des *PMC* est de pouvoir définir une topologie pour les connexions entre les différentes couches ; une topologie est invariante pour une connexion entre deux couches données ie : toutes les connexions entre ces deux couches sont faites suivant le même schéma de base.
On peut remarquer deux types principaux de topologies de connexion entre les couches : les connexion complètes et les connexions locales.

Les connexions complètes sont les plus simples à définir, une connexion entre une couche A et une couche B est dite complète *ssi* tous les neurones de la couche A sont connectés à tous les neurones de la couche B.

Les connexions locales quant à elles, doivent être définies avec plus de soin. Considérons à des fins d'explications deux couches A et B reliées entre elles (supposons que la couche B est la plus proche de la couche de sortie)
Ce type de connexion est définie par trois valeurs :

1. la localité de la connexion ou *Tessellation* qui est le nombre de neurones de la couche A connectés à *un* neurone de la couche B (des entiers pour une couche mono-dimensionnelle et un couple d'entiers pour une couche bi-dimensionnelle). C'est conceptuellement une fenêtre de connexion, tous les neurones de la couche A contenus dans cette fenêtre sont connectés à un neurone de la couche B.
2. le recouvrement ou *Overlap* qui représente l'enchevêtrement de la fenêtre de connection.
3. l'*Offset* qui représente le décalage de la fenêtre de connexion par rapport aux "bords" de la couche.

Nous retrouverons ces notions lors du court descriptif du langage *Aspirin*.

7.2 Les réseaux à fonction à base radiale (RBF)

Bien que ne faisant pas à proprement parlé partie du stage, nous ne pouvions passer sous silence l'existence des RBF. D'une part, car il forment une famille de réseaux connexionnistes couramment utilisée et d'autre part, parcequ'ils pourraient constituer une suite intéressante au projet.
La principale différence (et la plus visible) entre les PMC et les RBF est la structure même du réseau, en effet, les RBF ne sont constitués que de trois couches, la couche d'entrée, une couche cachée et une couche de sortie. La couche d'entrée et la couche de sortie sont toutes deux reliées à la couche cachée suivant un schéma de connexion complet.

7.3 Les applications des réseaux connexionnistes

Les réseaux de neurones sont couramment utilisés dans de nombreux domaines où les entrées varient sensiblement d'une phase de reconnaissance à l'autre.

La première application des réseaux de neurones venant à l'esprit est sans doute la reconnaissance de caractères. Un des exemples les plus célèbres de ce type d'application est le réseau *LeNet* développé par les laboratoires ATT pour le traitement automatique des chèques bancaires ; ce réseau constitue à l'heure actuelle l'un des plus gros PMC jamais réalisé (4365 neurones, 1920 connexions complètes et 96522 connexions locales). Notons au passage que les réseaux de grande taille tendent à être délaissés au profit de réseaux plus petits, avec une phase d'apprentissage moins longue et plus facile à stabiliser.
Un autre exemple célèbre de ce type d'application est l'ordinateur de poche *Apple Newton*, capable de reconnaître l'écriture de l'utilisateur après une courte phase d'apprentissage au cours de laquelle celui-ci doit écrire sur l'écran chacune des lettres de l'alphabet.

On peut également envisager l'utilisation des réseaux de neurones pour la reconnaissance de formes, la reconnaissance vocale ou même la génération de syllabes (un autre exemple célèbre est le réseau *NetTalk*) ou le contrôle de production, dans le cas où la matière à contrôler présente de caractéristiques non linéaires (du pétrole, par exemple).

Chapitre 8

Les outils utilisés

Lors de ce stage, nous avons été amenés à utiliser différents outils de développement. Dans un premier temps, nous avons dû apprendre la syntaxe du langage de description des réseaux de *Aspirin* neurones pour pouvoir générer automatiquement du code dans ce langage.
Ensuite, nous avons été confrontés au choix de la librairie graphique. De concert avec Bertrand GRANADO, nous avons choisi la librairie multi-plateforme Qt, ce choix sera discuté dans la section *Problèmes rencontrés* [1]. Ce logiciel a été développé conjointement sur plateforme i386 intel et Sun-Sparc. D'autres outils ont été utilisés dans le cadre de ce projet (GNU-make, g++) mais nous ne les décrirons pas dans le cadre de ce rapport ; pour de plus amples informations le lecteur pourra se reporter à [7].

8.1 Le langage de description des PMC Aspirin

Aspirin/Migraines est un langage déclaratif désigné à décrire des réseaux de neurones de type PMC arbitrairement complexes. Contrairement à ce que pourrait laisser penser le nom du langage, sa syntaxe est assez simple, mais la richesse des options fournies permet de décrire des réseaux de neurones arbitrairement complexes. Nous présenterons, ici, les grandes lignes de la syntaxe de ce langage, ce qui nous permettra, ensuite, de justifier les structures de classes choisies pour la génération de code. Cette partie ne se veut pas un descriptif complet des options du langage *Aspirin*, le lecteur soucieux de mieux connaître ce langage pourra se reporter à [5].
Dans ce langage, les réseaux de neurones sont représentés par des objets ou des groupes d'objets nommés *Blackbox* et les couches par des objets nommés *Layer*'s.

[1] *Voir* 10

8.1.1 Les blackboxes

Conceptuellement, une *Blackbox* est soit un réseau, soit un *sous-réseau*. Un fichier *Aspirin* est composé de la description d'une ou de plusieurs *Blackbox* interconnectées. On définit une blackbox par le mot clef `DefineBlackBox`.
Une déclaration type de *BlackBox* est la suivante :

```
DefineBlackBox <name>
{
    OutputLayer-> <layer-name>
    Components->
        {
            <list-of-component-layers>
        }
}
```

La première ligne de la déclaration concerne la couche de sortie de la *Blackbox*. La seconde donne la liste des couches de neurones de ce réseau (ou sous réseau). Ce sont les deux seules lignes indispensables dans la déclaration d'une BlackBox. Il existe cependant de nombreuses autres options pour, par exemple, filtrer les entrées, mais leur description n'est pas indispensable à la compréhension de la structure générale(*cf* [5] *pour plus de détails*).

8.1.2 Les layers

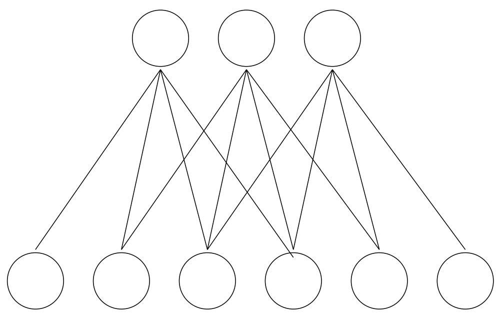
Les *Layers* constituent l'autre élément de base d'une description d'un réseau de neurones dans le langage Aspirin, elles représentent les couches d'un réseau de neurones. Leur description dans le langage *Aspirin* nécéssite quatre champs : le type de noeud (la fonction permettant d'obtenir l'état du neurone à partir des potentiels post-synaptiques), le nom (unique!) de la couche, la taille de la couche (de la forme `[a x b]` où a et b sont des valeurs entières) et des informations sur la manière dont la couche est reliée aux autres couches.
Une déclaration type d'une couche en tant que `component` d'une black box est la suivante :

```
<type de noeud> <arguments optionnels> <nom> <taille>
    <arguments optionnels> {
        <specification de la connexion>
    }
```

Il existe cinq types de noeuds prédéfinis dans le langage Aspirin, et une possibilité de définir ses propres fonctions en langage C. Les trois principaux noeuds prédéfinis sont des noeuds classiques de type sygmoïdal, les différences ayant trait à l'amplitude de la réponse. La dernière définition à donner pour déclarer un réseau de neurones est la topologie des connexions.

8.1.3 La Tessellation

La *Tessellation* dans le langage Aspirin décrit la topologie des connexions entre les couches. Tout comme nous l'avons expliqué dans la section 7.1 la tessellation est décrite par trois paramètres, la localité, le recouvrement (ou Overlap) et le décalage (ou Offset).
Encore une fois, la description de cette topologie dans le langage *Aspirin* est quasi triviale. Ainsi le code représentant le réseau suivant serait :



```
DefineBlackBox Exemple
{
 OutputLayer-> LAYER
 InputSize->[6x1]
 Components->
  {
   PdpNode LAYER 3
    {
     InputsFrom->INPUTS (with a [4x1] Tessellation
                         using a 3 XOverlap)
    }
  }
}
```

8.2 La librairie graphique Qt

Lors du début du stage, il a fallu choisir une librairie graphique pour réaliser l'interface graphique. De concert avec Bertrand Granado, nous avons choisi la librairie semi-libre Qt, en particulier, pour sa compatibilité avec les environnements de types Windows. Ce choix s'est avéré à l'usage plutôt malheureux. [2]

[2] *Voir* 10

Cette librairie C++ se compose de classes héritant toutes d'un "super-objet" *QObject*. Dans sa version pour environnement *Unix* la librairie est en fait une surcouche de la Xlib. Les objets mis à disposition par la librairie sont pour la plupart des *Widgets* (acronyme pour *WInDow gadGET*). Leur particularité est d'offrir un système assez complet de *signals* et de *slots* permettant la communication inter-classe et, au-delà, une communication enre les différentes widgets. En effet, chaque objet de la librairie émet des signaux lorsqu'un évenement se produit sur cet objet. Chaque objet Qt possède également des *slots* ou prises, auxquels il est possible (et même indispensable !) de connecter des signaux. Les *slots* associés à une classe sont, en fait, de simple méthodes, qui peuvent être reimplémentée ou enrichies.

Un programme Qt se présente donc comme une chaîne de connexions, un évenement produit un signal, receptionné par un *slot* pouvant entraîner lui même l'émission d'un nouveau signal, ou encore l'appel d'une méthode.

C'est donc au programmeur de *connecter* les signaux et les slots dans toutes ses classes. Il peut également définir de nouveaux signaux et de nouveaux slots dans ses propres classes.

L'exemple suivant résume ce mécanisme :

```
class Foo : public QObject
{
    Q_OBJECT
  public :
    Foo() ;
    int value() const  return val ;
  public slots :
    void setValue( int ) ;
  signals :
    void valueChanged( int ) ;
  private :
    int val ;
} ;
```

Toutes les classes qui contiennent des signaux et des slots doivent contenir le mot clef `Q_OBJECT` qui permet au Meta Object Compiler (moc pour les intimes), un outil fourni avec la librairie, de *linker* les signaux aux slots.
Une implémentation possible de `Foo : :setValue( int )` pourrait être :

```
void Foo : :setValue( int v )
{
  if ( v != val ){
    val = v ;
    emit valueChanged(v) ;
```

```
  }
}
```

La ligne `emit signal( argument )` emet le signal valueChanged de l'objet. Mais il reste encore à connecter les signaux et les slots :

```
Foo a, b ;
connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int))) ;
b.setValue( 11 ) ;
a.setValue( 79 ) ;
b.value() ; // b vaut maintenant 79 !
```

Ainsi, tous les évènements de l'interface sont gérés par ce mécanisme de signaux et slots.
Ce système est assez souple pour les opérations de bases, mais devient vite très lourd à utiliser quand la taille de l'interface devient plus conséquente. Et, surtout, il décuple la taille du code source ; en effet a chaque évènement on doit associer un ou plusieurs slots, c'est à dire, une ou plusieurs méthodes dans la classe...
De plus, ce sytème de connexion, bien que permettant de réaliser des opérations assez fines, est très mal documenté (*Voir* [6]). La lecture du code source de la librairie n'est pas plus instructive et relève du stakanovisme ; de la vocation multi-plateforme de la librairie résulte un code source quasiment illisible.
L'utilisation de cette librairie a été le plus gros frein au développement du projet. De nombreux bugs non résolus sont encore présents dans la librairie et, en particulier, un majeur sur architecture Sun-Sparc, qui empêche malheureusement le programme de fonctionner sur machines Sun. Ce défaut en incombe à une incompatibilité entre le compilateur gcc et la librairie Qt. [3]

[3] *cf* 10

Chapitre 9

La structure du programme

Lors du développement de ce programme, nous avons été amenés à partitionner le travail, ainsi le programme général peut se décomposer en trois parties distinctes : la production de code, l'interface et l'implementation d'une librairie pour la prédiction en temps d'éxécution des réseaux de neurones.

Il a d'abord fallu trouver une représentation des réseaux de neurones et la manière de produire du code Aspirin, ensuite, réaliser de l'interface graphique elle même. Et, enfin, programmer le calcul du graphe de précèdence du réseau.

9.1 La représentation des réseaux de neurones et la production de code

Pour représenter les réseaux de neurones et la production de code, nous avons choisi le langage C++. En effet, l'orientation objet de ce langage permet de représenter un réseau très simplement et de définir des classes pour chacun des composants d'un réseau. De plus, l'appel de méthodes permet d'avoir un accès simple à la production de code. La structure de donnée retenue est calquée sur la méthode de déclaration des réseaux de neurones dans le langage *Aspirin*. En effet, la structure très "imbriquée" des déclarations *Aspirin*[1], à savoir, les black box composées de layers, elles-même composées d' "inputsfrom" qui représentent les couches du réseau qui ont une connection *vers* la layer, ainsi que la topologie de ces connections, s'adapte particulierement bien à un langage orienté objet. Ainsi, on retrouvera dans le code C++ les objets Layer, Blackbox et inputsfrom. Nous avons également introduit un objet *project* représentant un ensemble de black box.
Chacun de ces objets est constitué principalement de la liste des objets qu'il contient, et une méthode produisant le code correspondant.

[1] *Voir* 8.1

La production de code, grâce à cette structure, est alors triviale. En effet, de simples parcours de listes donnent le résultat voulu. Tous ces objets contiennent également divers champs pour les options du langage.

9.2 L'interface

L'interface graphique, réalisée grâce à la librairie graphique Qt[2] , permet de gérer la production de code, d'éditer les différents champs des objets. Cette interface est composée de trois fenêtres :

1. La fenêtre principale
2. Une fenêtre d'édition pour les Black box
3. Une fenêtre d'édition pour les Layer

La fenêtre principale est le centre névralgique de l'interface, elle est constituée de deux menus déroulant. Le premier représente la liste des blackbox du réseau, toutes les opération d'édition agissent sur la blackbox courante c'est à dire celle qui est selectionnée.

Ces options de base sont :
- l'ajout d'une nouvelle black box, la nouvelle black box est alors ajoutée à la fin de la liste.
- la suppression de la black box courante
- l'édition qui ouvre la fenêtre d'édition des blackbox.

La seconde liste représente la liste des layers correspondant à la black box courante. À chaque changement de black box la liste de layer correspondante est mise à jour instantanement. Il est possible d'effectuer les mêmes opérations sur cette liste que sur la liste de blackbox. Il est de plus possible de faire de la layer courante la couche de sortie de la black box. Par défaut, la couche de sortie d'une black box est la première couche insérée dans la liste de layer.

Les deux autres fenêtres d'édition servent à éditer les différentes options des black box et des layer. Nous ne détaillerons pas ici la signification de ces options, ni le moyen de les modifier. Pour plus d'informations sur ce sujet, on pourra se reporter au tutoriel d'utilisation fourni avec le logiciel et au manuel de l'utilisateur *Aspirin* (*cf* citeAspirin).

La conception de l'interface a été la phase la plus complexe, non pas que la syntaxe et l'utilisation des objets Qt soit particulièrement difficile, mais nous

[2] *Voir* 8.2

avons été confrontés à une série de bugs de Qt résultant d'une incompatibilité entre le compilateur GCC et la librairie graphique[3].

9.3 Prédiction des performances d'un réseau de type PMC sur une architecture donnée

L'un des autres objectifs de ce stage, outre la réalisation de l'interface graphique et la production de code, était l'implémentation d'une méthode de prédiction des performances pour les PMC, méthode développée par Bertrand GRANADO au cours de sa thèse(*Voir* [4]). Couplée à une méthodologie d'évaluation des architectures, également développée dans [4], cette méthode de prédiction permet d'évaluer précisemment le temps d'éxécution d'un réseau de neurones sur une architecture donnée. Nous ne détaillerons pas ici la méthodologie d'évaluation des architectures.
Nous pouvons cependant relever trois étapes dans celle-ci :

1. détermination des primitives du modèle neuronal : c'est à dire déterminer le calcul le plus représentatif du réseau de neurone.
2. détermination d'un schéma de placement du réseau sur l'architecture cible.
3. détermination du modèle analytique de l'architecture cible pour le modèle neuronal.

La méthode de prédiction, développée sous la forme d'une librairie C, découle des trois étapes précedente.

9.3.1 Présentation de la méthode de prédiction

A partir d'un réseau de neurones (qui n'est finalement rien d'autre qu'un graphe !), nous cherchons à réaliser le graphe de précédence des primitives du réseau. Ainsi, grâce au modèle analytique déterminé au cours des trois premières étapes, il est possible de prédire le temps d'execution du réseau sur *une* architecture donnée, le graphe de précédence nous donnant l'ordonnancemment des calculs. Une fois ce graphe obtenu, il suffit d'affecter à chaque noeud un temps d'éx'ecution déduit du modèle analytique, et il est possible de prédire le temps déxécution total du réseau sur une architecture donnée.

Le calcul du graphe de précédence se fait suivant les règles suivantes :
Dans un premier temps, il nous faut déterminer le graphe des couches du réseau(ça c'est trivial), ensuite, on créee, à partir de ce graphe, *l'ensemble des noeuds représentant les primitives associées au réseau.*
Il y a deux types de noeuds dans cet ensemble, qui sont définis de la manière suivante :

[3] *Voir* 10

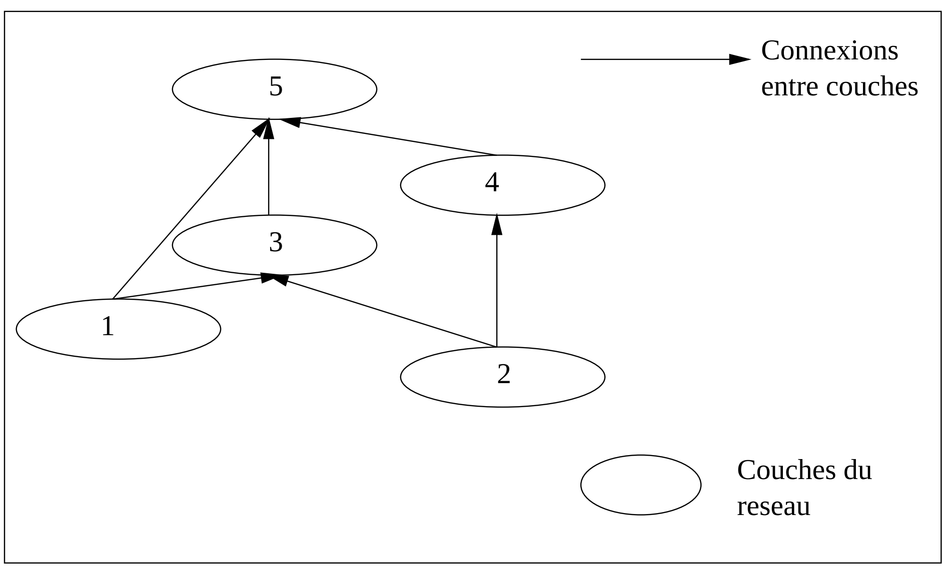
1. Un noeud connexion est crée pour tout couple de couche (a, b) interconnectées.
   Ce noeud correspond à la primitive associée au calcul des potentiels post-synaptiques de la couche (b).
2. Un noeud couche est crée pour toute couche (a), hormis celle d'entrée. Ce noeud correspond à la primitive associée au calcul des état des neurones de la couche (a).

A partir de cet ensemble il est alors possible de construire trivialement le graphe de précédence du réseau.
En effet, il suffit de rétablir les connexions entre les couches, ce qui peut se formaliser de la manière suivante :

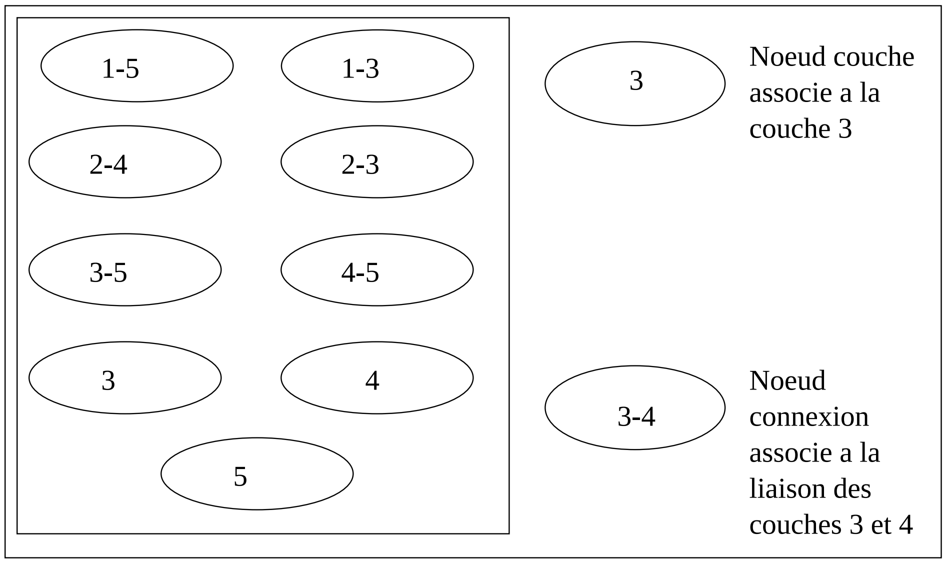
1. un arc d'un noeud connexion (a, b) vers un noeud couche (c) est crée *ssi* la couche dans le perceptron est la couche terminale de la connexion, ie : si $b = c$.
2. un arc d'un noeud couche (a) vers un noeud connexion (b, c) est crée *ssi* la couche est la couche initiale de cette connexion dans le perceptron, ie : si $a = b$.

Prenons un exemple concret pour illustré les règles que nous venons d'énoncer :

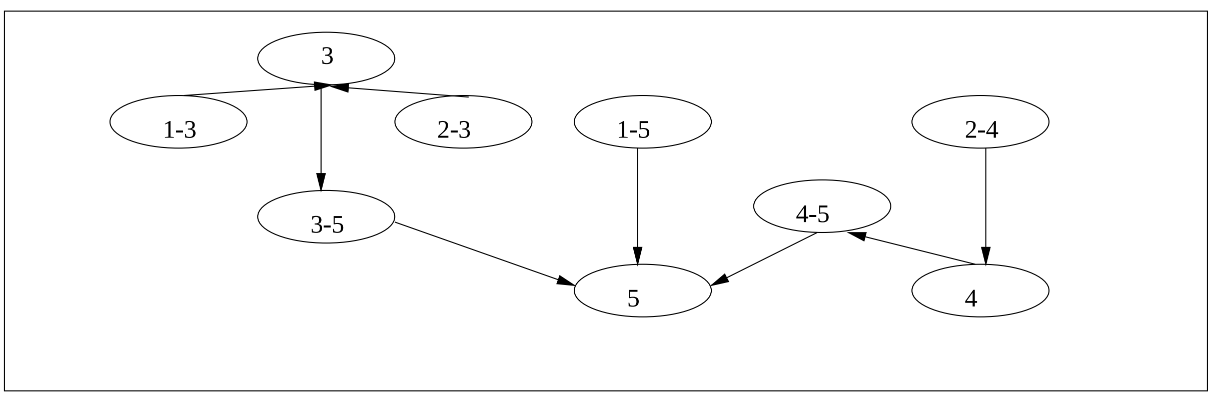
Réseau de neurones à traiter :



Création de l'ensemble des noeuds couche et des noeuds connexion :



Création du graphe de précédence :



9.3.2 Structures de donnée retenues et algorithmes mis en oeuvre pour réaliser le graphe de précédence

Pour représenter ces graphes, nous avons utilisé de simples matrices d'adjacence. A chaque couche est assigné un numéro (commençant à 0) représentant la ligne et la colonne dans la matrice.
S'il existe une connexion d'une couche A vers une couche B, alors l'élément de la matrice situé à la $indice_A^{ieme}$ ligne et à la $indice_B^{ieme}$ colonne est mis à la valeur 1 (les autres coefficients de la matrice sont, bien entendu, à la valeur 0).
Il reste ensuite à constuire l'ensemble des noeuds couche et des noeuds connexion, celui-ci est représenté par une simple liste chaînée. Ce choîx peut paraitre peu judicieux pour la représentation d'un ensemble, mais, cet ensemble étant *en*

pratique toujours “petit”, [4] la structure de liste ne pose pas de problème .... Encore faut-il construire cet ensemble. Dejà, il contient toutes les couches du réseau n’étant pas une couche d’entrée. Ces couches sont aisément reconnaissables car la colonne d’indice correspondant à cette couche dans la matrice d’adjacence ne comporte que des zéros (aucunue couche n’a de connexion vers une couche d’entrée). Les noeuds connexion sont encore plus faciles à reconnaître puisque qu’il en est crée un pour chaque connexion entre deux couches du réseau.

Une fois cette étape réalisée, il suffit de construire les arcs entre les éléments de l’ensemble suivant les règles définies précédemment. Il faut noter, cependant, le que la nouvelle matrice d’adjacence du graphe de précédence est indicée de la même manière que les éléments de la liste.

[4] A titre d’exemple, le réseau LeNeT, le plus gros réseau de neurones actuel, comporte à peine une dizaine de couches. De plus les grands réseaux tendent à être abandonné à cause de leur trop grande instabilité lors de la phase d’apprentissage.

Chapitre 10

Problèmes rencontrés

Lors de ce stage, nous avons été grandement ralenti par de nombreux problèmes dus à la librairie graphique Qt. En effet, cette librairie de par son manque de documentation et son manque de stabilité.

Nous avons été confrontés, lors de la réalisation de ce projet, à de nombreux bugs de la librairie, Qt. Nous avons pu en identifier certains, par expemple, il semble qu'il y ait un problème d'allocation mémoire sur architecture Sun-Sparc-64bits. Dans un premier, temps nous avons pensé que l'erreur pouvait provenir de la production de code, mais des tests de cette partie du code ont prouvés qu'il n'en était rien. Un examen de la mémoire grâce au debugger *GDB* a permis de cerner le problème avec plus de précision : il en résulte que le seg-fault du programme est dû à une erreur d'allocation mémoire de la librairie Qt. En effet, lors d'un click sur un des boutons de l'interface, il y a une animation, gérée en interne par la librairie par la méthode du *double-buffering*. [1] C'est lors de ce processus que l'erreur d'allocation mémoire a lieu; en effet, la librairie essaie de libérer un espace mémoire qu'elle n'a pas allouée.

Nous avons également rencontré beaucoup d'autres bugs dans cette librairie. Ils ont considérablement ralenti l'avancement du projet et, dans une moindre mesure, l'ergonomie de l'interface, rendant son utilisation beaucoup moins intuitive et (surtout) alourdissant le code car il a fallu avoir recours à un certain nombre d"'astuces" pour pouvoir réaliser certaines widgets.

Lors de la phase finale du projet (réalisation de la dernière fenêtre), nous avons encore été confrontés à un bug "aléatoire" qui empêche cette fenêtre de fonctionner, et par conséquent, malheureusement, l'interface aussi ...
Ces bugs sont d'autant plus étonnants qu'un grand nombre d'entre eux ne sont pas répertoriés par le distributeur. Cependant, la lecture de forums[2] de discussion nous ont appris qu'il y a une incompatibilité entre le compilateur GCC et la librairie Qt. En effet, les optimisations automatiques que GCC réalise

[1]une image de l'animation est calculée pendant qu'une autre est affichée.

[2]Notamment le message d'un développeur du projet GCC, ancien élève de l'École. informatique.lang messge 173

rigoureusement lors de la phase de compilation de la librairie génèrent une librairie complètement buggée (même sans aucune optimisation !!!). Ainsi, il semble que Qt fait des appels de méthodes sur des objets qui ont déjà été détruits. Nous avons essayés pour contourner ces bugs, de *re-implémenter* à la main les widgets défectueuses, mais en vain. Ainsi, ces bugs sont totalement aléatoires et empêchent le programme général de fonctionner, bien que la plus grande partie soit totalement fonctionnelle.

Chapitre 11

Conclusion

Il est dommage que le projet soit bloqué par les bugs de Qt, cependant, le constat général est loin d'être totalement négatif. En effet, l'interface a été conçue pour devenir, au final, une plateforme complète de développement des réseaux de neurones. En faisant abstraction de l'incompatibilité quasi "génétique" entre Qt et gcc (incompatibilité qui sera sans doute résolue dans les prochaines versions de Qt), l'interface présente une bonne ergonomie. De plus, les développements possibles à ce projet sont nombreux :

1. Gestion des options de compilation Aspirin par l'interface.
2. Réalisation d'un parser pour pouvoir ouvrir un fichier *Aspirin* quelconque dans l'interface [1]
3. Gestion de la phase d'apprentissage (grâce au programme *Migraines*) et visualisation, en temps réel, de la convergence de l'erreur globale du réseau.
4. Réalisation d'une surcouche au langage Aspirin pour la gestion des réseaux de type RBF[2]

[1] En effet, l'interface actuelle ne permet que la sauvegarde du code généré dans un fichier, mais pas sa réouverture dans l'interface

[2] *Voir* 7.2

Bibliographie

[1] Mitsuhiro Okada, Max Kanovitch, Andre Scedrov.
Specifying Real-Time Finite-State Systems in Linear Logic

[2] Vincent Cremet. Rapport de Stage de DEA.
Spécification et vérification de systémes temps-réels avec la logique linéaire

[3] James Power, Caroline Webster.
Working with Linear Logic in Coq

[4] Bertrand GRANADO. *Architecture des systèmes électroniques pour les réseaux de neurones - Conception d'une rétine connexionniste.* Thèse de doctorat, Université de Paris-Sud, 1998

[5] *Aspirin USER MANUAL*

[6] Documentation en ligne de la librairie Qt : `http ://www.troll.no/`

[7] Site Web du compilateur GCC : `http ://www.gnu.org/`

[8] *NEURAL NETWORKS - Advances and Applications* E.GELENBE Editor