Université Paris 7 - Denis Diderot

D.E.A. de Programmation : Sémantique, Preuves et Langages

Rapport de stage

Spécification et vérification de systèmes temps-réel avec la logique linéaire

Vincent Cremet

Responsable du D.E.A. : Guy Cousineau

Responsables du stage : J.-P. Jouannaud et M. Okada

Laboratoire d'accueil : LRI, Université Paris-Sud

Année 1999–2000

Table des matières

Remerciements

Je tiens tout particulièrement à remercier les deux responsables de mon stage, Jean-Pierre Jouannaud et Mitsuhiro Okada pour le suivi qu'ils ont apporté à mon travail, m'incitant ainsi à perséverer dans les inévitables périodes de découragement.

Mitsuhiro Okada en particulier a tenu à garder avec moi un dialogue constant, ce qui était pour lui une manière de rester à l'écoute de mes problèmes et d'y répondre, et pour moi un moyen de toujours être au fait de ses attentes. Cet échange productif a pu aussi bien prendre place lors des conférences qu'il a données au LRI concernant les recherches à la base de mes travaux, que lors de conversations plus informelles dans notre bureau.

Jean-Pierre Jouannaud, malgré un emploi du temps toujours très chargé, a pris le temps de superviser mon travail et m'a toujours beaucoup soutenu dans mes démarches de recherche de thèse. Christine Paulin a aussi été très attentive à mon parcours, je les en remercie tous les deux.

Ils ont aussi toujours témoigné beaucoup de gentillesse et d'affection à mon égard me faisant découvrir un milieu de la recherche très sérieux mais aussi très humain.

Bien sûr je suis reconnaissant au LRI de m'avoir accueilli pendant ce stage et particulièrement à l'équipe Demons à laquelle j'étais rattachée , et dont chaque membre est imprégné de l'esprit sympathique qui la caractérise. Lors de ma présoutenance au laboratoire ils ont été très nombreux à intervenir et à me faire bénéficier de leurs conseils et de leurs critiques constructives dans le but d'améliorer ma présentation.

Enfin, je n'oublie surtout pas le responsable du DEA, Guy Cousineau, ainsi que l'ensemble des professeurs de cette formation dont les cours ont toujours été captivants par leur contenu scientifique et leur présentation enthousiaste.

Introduction

Le but de ce stage de DEA a été de donner une première implémentation à un travail jusqu'ici théorique mené conjointement par Max I. Kanovich, Mitsuhiro Okada et Andre Scedrov [KOS97] [OKA00]. Cette nouvelle théorie, basées sur la logique linéaire [GIR87], se présente comme une alternative à celle des automates temporisés [AD94] pour modéliser des systèmes réactifs temps-réel, pour spécifier et vérifier certaines de leurs propriétés. Rappelons que la complexité inhérente à ce domaine provient de la nécessité de prendre en considération un temps continu, représentable par les nombres rationnels, par opposition à un temps discret pour lequel des théories éprouvées existent déjà, qui ne permettent malheureusement pas de modéliser des systèmes formés de sous-systèmes évoluant dans le temps de manière asynchrone.

La spécification du système temps réel se fait dans un langage proche de la description humaine des propriétés du système et nous l'appelerons donc le "langage naturel".

On peut se représenter une spécification dans ce langage comme un graphe orienté fini dont les sommets sont des transitions et dont les arêtes sont étiquetées par des contraintes de temps, contrairement à un automate temporisé classique où les sommets représentent des états du système. Par exemple, une contrainte "$< 1/3$" sur une arête reliant une transition α à une transition β signifie que si la transition α a lieu, alors la transition β doit se produire moins de $1/3$ unité de temps plus tard.

Ensuite on traduit cette spécification décrite en langage naturel en un ensemble de clauses de la logique linéaire qui nous serviront par la suite d'axiomes. Cela nous permet de définir le segment initial d'une exécution valide du système comme une preuve en logique linéaire dans un environnement composé des axiomes.

A partir de là, on peut spécifier, par une formule de la logique linéaire, un certain nombre de propriétés standards et vouloir les vérifier : sûreté, accessibilité générale, ordonnancement, rendez-vous, vivacité.

Les avantages de cette théorie par rapport à celle des automates temporisés sont de fournir un langage de spécification de haut niveau, le langage naturel, par opposition à une spécification "orientée modèle", de permettre de modéliser certains comportements dynamiques comme l'ajout de nouveaux agents et le changement des contraintes de temps en cours d'exécution et enfin de donner un cadre complétement formel, la logique linéaire, pour représenter à la fois le système, la propriété à vérifier et la preuve de cette propriété. Ce nouveau formalisme s'accompagne aussi d'une nouvelle méthodologie pour la spécification. L'idée est de fournir un environnement interactif de vérification qui combine l'utilisation de procédures automatiques pour les problèmes décidables et la preuve assistée pour des problèmes plus complexes. L'utilisation de l'assistant de preuve Coq a été jugée pertinente. Par exemple, on peut essayer différentes valeurs pour une borne supérieure et constater que si les valeurs restent dans un certain intervalle le système reste sûr. Ensuite, on peut

prouver manuellement que pour toute valeur comprise dans cet intervalle, le système reste sûr.

Le travail réalisé pendant ce stage a consisté à assimiler les concepts théoriques, à les décortiquer pour en extraire un contenu opérationnel, et finalement concrétiser cette démarche en programmant dans le langage Ocaml un vérificateur automatique de propriétés de sûreté. On peut donc dire que la partie visible de mon travail se compose d'une implémentation en Ocaml et de ce rapport qui fait le lien entre la théorie et l'implémentation, qui donne une présentation formelle des idées originales exprimées par les trois auteurs de la théorie, et notamment par Mitsuhiro Okada qui a suivi de près mes travaux.

C'est la première fois qu'une implémentation de ce formalisme est réussie, plusieurs tentatives antérieurs ayant buté sur la difficulté d'extraire des algorithmes effectifs des preuves mathématiques abstraites.

Insistons encore sur le fait que le travail réalisé pendant le stage n'a pas été qu'un travail d'implémentation car le concept étant encore neuf, il a fallu en coopération avec M. Okada clarifier et rendre plus cohérents certains points, transformer des preuves abstraites de théorème en des algorithmes effectifs. J'ai notamment proposé une extension du langage naturel, plus expressive et plus cohérente, qui garde la propriété d'être traductible dans le formalisme sous-jacent.

Le lecteur s'étonnera peut-être que la logique linéaire, qui apparaît pourtant dans le titre de ce document soit si peu présente dans son contenu. En fait dans cette permière étape de l'implémentation dans laquelle on ne considère que le problème de sûreté et pour seul comportement dynamique celui de changer les contraintes de temps, on peut travailler directement sur le graphe de transition étiqueté en masquant les concepts de la logique linéaire. Par contre pour les autres problèmes décidables et pour prendre en compte l'ajout de nouveaux agents, l'utilisation de la logique linéaire se révélera indispensable et constituera le cœur des implémentations futures.

Plan :

Le contenu de ce chapitre a été organisé pour à la fois faire comprendre les concepts théoriques mis-en-oeuvre et pour présenter l'implémentation qui a été réalisée. Chaque chapitre reflète cette ambition et seul un chapitre, le chapitre 7, est réellement centré sur l'implémentation.

Le chapitre 1 présente les concepts généraux concernant les systèmes temps-réel et l'utilisation conjointe des alarmes et de la logique linéaire pour les spécifier. On y décrit en particulier un système ferroviaire très simple qui va nous servir d'exemple et nous accompagner tout au long de l'exposé.

Dans le chapitre 2 on décrit le langage naturel et sa traduction en une spécification plus facilement exploitable appelée "spécification compacte".

Le chapitre 3 explique comment modéliser des comportements dynamiques dans notre formalisme.

Le chapitre 4 replace le problème de sûreté, celui dont il sera principalement question dans ce rapport étant donné qu'il est le seul qui ait été implémenté, parmi l'ensemble des problèmes décidables classiques. On y aborde des questions de complexité.

Le chapitre 5 touche au coeur de l'algorithme de décision d'un problème de sûreté. Il justifie en effet la réduction du problème à un graphe de transition fini après élimination

du temps.

Le chapitre 6 présente un exemple de spécification dans Coq d'un système et d'une propriété de sûreté plus générale, car paramétrée. On y donne aussi l'idée d'une preuve formelle.

Nous l'avions annoncé, le chapitre 7 est dédié spécifiquement au programme de vérification. On y décrit son utilisation, son architecture, avec un découpage en modules qui est mis en relation avec les parties théoriques de ce rapport, ainsi qu'un ensemble de tests.

Enfin le dernier chapitre, le chapitre 8, contient une proposition d'extension du langage de spécification que j'ai formalisée.

Chapitre 1

Logique linéaire et alarmes

1.1 Principe général

Spécifier un système temps réel c'est fixer quelles seront les évolutions au cours du temps de l'état de ce système.

Il convient donc de définir tout d'abord l'état d'un système.

Un système peut être décomposé en plusieurs *agents* qui pourront chacun être dans un état donné. Un *état total du système* à un instant donné sera alors déterminé par l'état pris par chacun de ses agents au même instant. Par exemple si nous avons 3 agents Ag_1, Ag_2 et Ag_3 dont les états respectifs sont E_1, E_2 et E_3, un état total du système peut être représenté sous la forme $Ag_1(e_1) \otimes Ag_1(e_2) \otimes Ag_1(e_3)$ avec e_1, e_2 et e_3 appartenant respectivement à E_1, E_2 et E_3.

Ensuite, on peut distinguer deux sortes d'évolution du système : soit son état reste stationnaire pendant un temps donné, soit il change d'état de façon instantanée. On appelera le premier type d'évolution une *transition continue* (ou temporelle) et le second une *transition discrète* (ou d'état).

Habituellement on représente les transitions discrètes comme un graphe de transition dont les sommets sont les états du systèmes et dans lequel deux états sont reliés si le système peut évoluer de l'un à l'autre.

Il est assez naturel de rajouter le temps courant à l'état du système comme si c'était un agent de manière à obtenir une *configuration* du système. En reprenant notre notation précédente une configuration d'un système sera alors de la forme $Temps(t) \otimes Ag_1(e_1) \otimes Ag_1(e_2) \otimes Ag_1(e_3)$ dans laquelle t est un nombre rationnel positif. Remarquons qu'au lieu des rationnels on aurait pu prendre n'importe quel ensemble dense et contenant les entiers (comme les réels par exemple). Cela correspond bien à notre représentation intuitive du temps comme une entité indéfiniment divisible et non bornée.

Ensuite, il faut définir des primitives pour exprimer des contraintes de temps sur un système (réactif) temps réel. Le choix adopté ici est d'utiliser des bornes inférieurs et des bornes supérieurs comme nous allons l'expliquer ci-dessous, mais il existe d'autres approches qui font intervenir d'autres concepts, comme la notion d'intervalle de temps par exemple.

Dans notre approche, il y a donc deux sortes de directives que l'on veut pouvoir utiliser pour restreindre les évolutions possibles de ce système à celles qui sont effectivement

permises dans la réalité.

La première consiste à **obliger** un évènement à se produire dans un temps donné, nous appelerons ce type de contrainte une *contrainte haute* :
"si tel évènement se produit, alors tel autre événement doit se produire dans un laps de temps donné"
La seconde consiste à **interdire** l'apparition d'un évènement pendant un certain temps, nous appelerons ce type de contrainte une *contrainte basse* :
"si tel évènement se produit, alors tel autre événement ne doit pas se produire pendant un laps de temps donné"

1.2 Des alarmes pour se rappeler des contraintes

Notre approche est basée sur l'utilisation d'alarmes indépendantes qui serviront chacune à gérer le respect d'une contrainte de temps, de type haute ou basse. Ces alarmes peuvent soit être éteintes, soit fixées à une valeur stationnaire qui correspond à leur date d'échéance.

Considérons une contrainte haute (si l'évènement e_1 a lieu, alors l'événement e_2 doit se produire dans un laps de temps C).
Lorsque e_1 se produit on fixe l'alarme correspondant à cette contrainte à une valeur égale à la date limite, c'est-à-dire la somme du temps courant et de C. On interdit en outre au temps qui s'écoule de franchir la date limite, ainsi la contrainte haute sera respectée. Le temps qui est assimilé à un agent devra attendre que cette alarme soit désactivée lorsque l'évènement e_2 se produit.

Considérons maintenant une contrainte basse (si l'évènement e_1 a lieu, alors l'événement e_2 ne doit pas se produire pendant un laps de temps C).
Lorsque e_1 se produit on fixe l'alarme correspondant à cette contrainte à une valeur égale à la date d'échéance de l'interdiction, c'est-à-dire la somme du temps courant et de C. On ne permet ensuite l'apparition de l'évènement e_2 que lorsque la valeur du temps courant aura dépassé celle de l'alarme.

L'utilisation d'alarmes attachées à chaque contrainte de temps est assez naturel, comme nous l'avons montré. Il nous faut maintenant définir ce que l'on a jusqu'ici appelé *évènement*.
Dans notre approche on donne un nom à des groupes de transitions entre état du système, et l'on appelle évènement le déclenchement d'une transition d'un de ces groupes.
C'est une différence avec l'approche basée sur les automates temporisés qui considère comme évènement l'entrée dans un état donné. Notre approche s'intéresse à détecter et contraindre le déclenchement de transitions alors que celle des automates temporisées s'intéresse à détecter et contraindre l'entrée dans certains états.

Nous ajoutons maintenant le dernier ingrédient à la notion de configuration pour obtenir une configuration complète du système : la valeur courante des alarmes. Par convention on utilise la lettre U (upper-bound) pour désigner une alarme attachée à une contrainte haute et la lettre L (lower-bound) pour désigner une alarme attachée à une contrainte basse. Une configuration du système sera donc finalement une conjonction d'atomes de la forme :

$$Temps(t),\ Ag_1(e_1),\ Ag_1(e_2),\ Ag_1(e_3),\ U_\alpha(u),\ L_\beta(v)$$

1.3 De l'utilisation de la logique linéaire

Nous allons maintenant voir comment on peut utiliser la logique linéaire pour représenter facilement une configuration et les transitions nommées. En fait, on utilise de manière standard l'implication linéaire entre produits tensoriels linéaires pour repésenter l'exécution concurrente. On peut aussi voir cette approche comme de la réécriture de multi-ensembles, l'avantage de la logique linéaire est cependant de pouvoir introduire des quantificateurs, notamment existentiels, pour spécifier de façon purement formelle des propriétés au lieu de les faire intervenir au niveau méta.

On utilise des prédicats unaires pour représenter chaque paramètre d'une configuration : le temps, les agents et les alarmes. On relie ensuite les atomes correspondant à chacune de ces entités par un produit tensoriel correspondant au "et" multiplicatif de la logique linéaire.

Une configuration du système sera donc représenté par la formule :

$$Temps(t) \otimes Ag_1(e_1) \otimes Ag_1(e_2) \otimes Ag_1(e_3) \otimes U_\alpha(u) \otimes L_\beta(v)$$

L'implication de la logique linéaire entre deux formules de la forme présentée ci-dessus dans dans lesquelles certaines valeurs sont remplacées par des

Nous avons dit que la brique de base dans notre approche était des groupes nommés de transitions d'état.

Pour représenter un groupe de transitions d'état de manière compacte, on va utiliser une implication linéaire entre deux produits tensoriels qui pourront contenir des variables libres.

Par exemple

$$Ag_1(e_1) \otimes Ag_2(e_2) \otimes Ag_3(x) \multimap Ag_1(e_1') \otimes Ag_2(y) \otimes Ag_3(e_3)$$

va représenter l'ensemble des transitions d'état qui peuvent être obtenues en instanciant les variables x et y par des constantes. Attention, comme la logique linéaire prend en compte la consommation des ressources, tous les prédicats présents dans le membre gauche d'une implication doivent aussi être présents dans le membre droit. Pour la même raison, si on veut spécifier qu'un agent ne change pas d'état, on peut l'omettre dans l'implication.

Maintenant voyons comment on peut allumer puis éteindre une alarme de type *obligation* :

$$\alpha : Time(t) \otimes Ag_1(e_1) \otimes U(v) \multimap Time(t) \otimes Ag_1(e_1') \otimes U(v|t + C)$$

$$\beta : Ag_2(x) \otimes U(v) \otimes (v < +\infty) \multimap Ag_2(e2) \otimes U(+\infty)$$

($(x|y) = x$ si $x < +\infty$, y sinon. On utilise cet opérateur car on ne veut pas qu'une date limite puisse être repoussée)

On procède de manière similaire avec une alarme de type *interdiction* :

$$\gamma : Time(t) \otimes Ag_1(e_1) \otimes L(v) \multimap Time(t) \otimes Ag_1(e_1') \otimes L(t + C)$$

$$\delta : Time(t) \otimes Ag_2(e_2) \otimes L(v) \otimes (v < t) \multimap Time(t) \otimes Ag_2(e_2') \otimes L(+\infty)$$

1.4 L'exemple du train

Le mieux pour comprendre l'utilisation du formalisme est de l'appliquer sur un exemple. On a choisi l'exemple très classique d'un système composé d'un train et d'une barrière qui lorsqu'elle est abaissée empêche la circulation sur un passage à niveau.

Voici une description textuelle du système :

"Le système est composé d'un train et d'une barrière. Le train peut être dans l'un des états *loin*, *approchant*, *traversant*. La barrière peut être dans l'un des états *basse*, *montante*, *haute*, *descendante*.

Au départ le train est loin et la barrière est levée.

Quand le train commence à approcher, moins de H_0 secondes plus tard la barrière commence à se baisser et il va pouvoir traverser la voie mais pas avant que B_0 secondes ne se soient écoulées.

Entre le moment où la barrière a commencé à se baisser et où elle atteint effectivement la position basse, il s'écoule au plus H_1 secondes.

Dès que le train a traversé, la barrière commence à remonter et moins de H_2 secondes plus tard elle est en position haute."

Voici comment on peut spécifier formellement ce système à l'aide de la logique linéaire (on utilise les identificateurs anglais qui sont plus courts).

La position de départ est spécifiée par la formule :

$$Time(0) \otimes Train(far) \otimes Gate(up) \otimes L_0(+\infty) \otimes U_0(+\infty) \otimes U_1(+\infty) \otimes U_2(+\infty)$$

Les alarmes L_{α_3}, U_{α_1}, U_{α_2} et U_{α_5} correspondent respectivement aux contraintes de temps impliquant les bornes B_0, H_0, H_1 et H_2. Au départ elles sont éteintes ce qui se traduit par le fait qu'elles sont initialisées à $+\infty$.

TRANSITION α_0 :

$$Time(t) \otimes Train(far) \otimes L_{\alpha_3}(x) \otimes U_{\alpha_1}(y) \multimap Time(t) \otimes Train(approach) \otimes L_{\alpha_3}(t+B_0) \otimes U_{\alpha_1}(y|t+H_0)$$

TRANSITION α_1 :

$$Time(t) \otimes Gate(x) \otimes U_{\alpha_1}(y) \otimes (y < +\infty) \otimes U_{\alpha_2}(z) \multimap Time(t) \otimes Gate(mv\_down) \otimes U_{\alpha_1}(+\infty) \otimes U_{\alpha_2}(z|t+H_1)$$

TRANSITION α_2 :

$$Gate(mv\_down) \otimes U_{\alpha_2}(y) \otimes (y < +\infty) \multimap Gate(down) \otimes U_{\alpha_2}(+\infty)$$

TRANSITION α_3 :

$$Time(t) \otimes Train(approach) \otimes L_{\alpha_3}(x) \otimes (x < t) \multimap Time(t) \otimes Train(cross) \otimes L_{\alpha_3}(+\infty)$$

TRANSITION α_4 :

$$Time(t) \otimes Train(crossing) \otimes Gate(x) \otimes U_{\alpha_5}(y) \multimap Time(t) \otimes Train(far) \otimes Gate(mv_up) \otimes U_{\alpha_5}(y|t+H_3)$$

TRANSITION α_5 :

$$Gate(mv\_up) \otimes U_{\alpha_5}(y) \otimes (y < +\infty) \multimap Gate(up) \otimes U_{\alpha_5}(+\infty)$$

On ne doit pas oublier la formule de progression du temps :

$$Time(s) \otimes (s < t) \otimes U_{\alpha_1}(x) \otimes U_{\alpha_2}(y) \otimes U_{\alpha_5}(z) \otimes (t < x) \otimes (t < y) \otimes (t < z)$$

$$\multimap$$

$$Time(t) \otimes U_{\alpha_1}(x) \otimes U_{\alpha_1}(y) \otimes U_{\alpha_1}(z)$$

Le temps ne peut s'écouler entre une valeur s et une valeur t que s'il ne franchit pas l'une des dates limites associées aux alarmes hautes. C'est une manière astucieuse de s'assurer que les contraintes hautes seront toujours respectées. Pour pouvoir franchir les barrières que représentent ces alarmes, le temps devra attendre qu'elles soient éteintes.

Pour une meilleure compréhension de la manière dont on peut utiliser la logique linéaire pour spécifier un système temps-réel le lecteur est invité à lire [KOS97].

Chapitre 2

Spécifier en langage naturel

Dans le chapitre précédent nous avons présenté un formalisme qui en combinant l'utilisation d'alarmes et de la logique linéaire permettait de spécifier de façon compacte et en se basant sur des concepts simples un système temps réel.

En fait, il est possible de définir un langage plus proche encore de l'expression humaine d'un système temps réel. Pour cette raison, mais un peu par abus de langage, nous l'appelerons par la suite *langage naturel*. Avec la possibilité, présentée ultérieurement, de modéliser des comportements dynamique, cette caractéristique constitue un des points les plus appréciables de cette nouvelle approche.

Une spécification dans ce langage naturel pourra être automatiquement traduite en une spécification basée sur le formalisme précédent. C'est donc une surcouche pour l'utilisateur qui simplifie la définition d'une spécification, principalement en masquant l'utilisation des alarmes et de la logique linéaire, et en permettant de définir des contraintes globales sur l'évolution du système et non pas seulement localement au niveau des transitions. Bien sûr en gagnant en simplicité on perd en expressivité, mais c'est aussi un point intéressant car nous verrons dans un autre chapitre que les problèmes de sûreté pour des spécifications exprimées dans ce langage deviennent décidables.

Ce langage est issu de la théorie développée par Kanovich, Okada et Scedrov et nous allons présenter ici les points intéressants concernant sa syntaxe et son implémentation.

2.1 Spécification en langage naturel

Spécifier un système en langage naturel recquiert de définir quatre choses :
- les agents composant le système
- un nombre fini de contraintes
- un nombre fini d'actions
- la configuration initiale

Dans la suite on se sert de l'exemple classique de simulation ferroviaire.

2.1.1 Les agents

On doit indiquer pour chaque agent le nombre fini des états qui lui sont associés :

```
System = {
  Train := far | approaching | crossing;
  Gate := down | mv_up | up | mv_down
}
```

2.1.2 Les contraintes

Les contraintes portent sur les états du systèmes. Par exemple :

```
Constraints = {
c1 : interval (<Train=approaching> -- <Train=crossing>) > 10
}
```

spécifie qu'entre l'instant où le train entre dans l'état *approaching* et le moment où il entre dans l'état *crossing*, il s'écoule plus de 10 unités de temps. Ici on a une contrainte basse, mais on pourrait aussi avoir des contraintes hautes. Les arguments de *interval* sont deux états partiels. On remarque enfin qu'on donne un nom à une contrainte. Nous verrons par la suite qu'à chaque contrainte correspond une alarme qui prendra son nom et qui sera initialisée ou éteinte lors du déclenchement de certaines transitions (sur notre exemple les transitions qui ont pour effet d'amener le système dans l'état où le train approche ou dans celui où il commence à traverser la voie).

2.1.3 Les actions

Une action est caractérisée par son nom, un état partiel de départ, un état partiel d'arrivée et par un nombre fini de contraintes sur d'autres actions. Le système ne peut changer d'état qu'en déclenchant une de ses actions.

Dans l'exemple suivant :

```
Actions = {

a0: from <Train=far> to
    ->  Train := approaching
    cause g1 within strict 10/3


a1: from any to
    -> Gate := mv_down
    cause g2 within 7.1749923
...

}
```

la première contrainte a pour nom $a1$, l'état partiel de départ est $Train(far)$, l'état partiel d'arrivée est $Train(approaching)$ et nous avons un contrainte qui stipule que moins de $10/3$ unités plus tard, l'action $g1$ devra se déclencher.

2.1.4 La configuration initiale

Evidemment on donne aussi la configuration initiale du système constituée de l'état total initial et de la date initiale :

```
Initial_event = <Train=far,Gate=up>

Initial_time = 0
```

2.1.5 Les états dangereux

Lorsqu'on s'intéresse à un problème de sûreté, on peut aussi lister la liste des états totaux du système qui sont jugés dangereux :

```
Final_events = {
        <Train=crossing,Gate=up>;
        <Train=crossing,Gate=mv_up>;
        <Train=crossing,Gate=mv_down>
}
```

2.2 Spécification compacte

Après avoir défini une spécification en langage naturel, nous aimerions lui associer une sémantique en terme de graphe de transition qui représentera l'ensemble des trajectoires du système.

On va en fait réaliser ce travail en deux étapes en traduisant tout d'abord une spécification faite en langage naturel en une spécification compacte, puis en associant un graphe de transition à une spécification compacte. Après avoir défini une spécification compacte nous aborderons la première phase de ce processus dans la suite de ce chapitre. La deuxième phase sera explicitée dans le chapitre 5.

Nous allons extraire de l'étude du train faite dans le chapitre précédent une notion très compacte d'une spécification qui ne fait plus intervenir la logique linéaire.

On garde l'idée de décrire un système par ses agents et leurs états, et de modéliser l'évolution du système au moyen d'alarmes et de groupes nommés de transitions définis au moyen de couples d'états partiels du système. Mais nous allons essayer de ne retenir comme information que le strict nécessaire à partir duquel on pourra si on le désire reconstruire une spécification en langage naturel, c'est pourquoi nous appelerons ce type de spécification une *spécification compacte*.

Au fond que se passe-t-il réellement lors du déclenchement d'une transition ? Toute simplement on va vouloir qu'un certain nombre d'alarmes soient initialisées et qu'un certain nombre d'autres alarmes soient éteintes. Un certain nombre de conditions sont attachées à ces initialisations et extinctions, mais elles sont toujours les mêmes pour un type d'alarme donnée, il n'est donc pas nécessaire de les expliciter à chaque fois. Par exemple on ne pourra initialiser une alarme de type haute si elle l'est déjà, ce qui reviendrait à repousser une date limite. De la même façon on ne pourra éteindre une alarme de type basse que si sa date d'échéance a été dépassée.

La spécification d'un système se résume donc à ce qui suit :

Une *spécification* d'un système est définie par la donnée d'un ensemble fini d'*agents*, d'un ensemble fini d'*alarmes* et d'un ensemble fini d'*actions*.

Un *agent* Ag est défini par son nom et par un ensemble fini d'états qui lui sont propres.

Un *état partiel des agents* du système est une valuation partielle des agents par un de leurs états.

Un *état total des agents* est une valuation totale des agents par un de leurs états.

Une *alarme* est définie par son nom α, son type (U pour une contrainte haute, L pour une contrainte basse), sa borne, qui est un rationnel positif, et par l'indication de la nature de cette borne, c'est-à-dire stricte ou large, exemple $U_\alpha(t_0)$

Enfin une *configuration* du système est la donnée d'un état total des agents e, d'une valuation al des alarmes par des éléments de $\mathbb{Q} \cup \{+\infty\}$ et d'un temps t qui représente

le temps courant (la valeur $+\infty$ signifie que l'alarme est éteinte, une valeur fixe c signifie que l'échéance de cette alarme est fixée à l'instant c). Par la suite, on notera souvent une configuration sous la forme : $Temps(t) \otimes Alarmes(al) \otimes Etat(e)$.

Une *action*, quant à elle, est définie par son nom, un état partiel des agents de départ, un état partiel des agents d'arrivée, une liste d'alarmes à allumer, et par une liste d'alarmes à éteindre. La signification d'une action est de permettre de passer d'un configuration à une autre, de facon instantanée, en modifiant l'état des agents du système et en activant ou désactivant certaines alarmes. Précisons encore qu'une action ne peut être déclenchée que sous certaines conditions liées au contexte courant de l'exécution.

2.3 Traduction en une spécification compacte

On va donner ici les grandes lignes de la traduction d'une spécification faite en langage naturelle en une spécification compacte, ce qui est un moyen de donner à notre langage une sémantique relative (il restera encore à donner la sémantique d'une spécification compacte, ce qui sera fait dans un prochain chapitre). Pour les détails de cette traduction on peut toujours se reporter au code de mon programme (fichier `langage_naturel.ml` et `spec_compacte.ml`).

La traduction se fait en plusieurs étapes :
- on commence par ignorer les contraintes et extraire, des informations sur les actions, le graphe de leurs dépendances ;
- à partir de celui-ci on génère ensuite un premier ensemble d'alarmes et de transitions ;
- puis on intègre à ces transitions les alarmes correpondant à la liste des contraintes.

2.3.1 Extraction du graphe de dépendance

La section *actions* d'une spécification en langage naturel peut être représentée graphiquement comme un graphe de dépendance entre actions.

Par exemple si une action α entraîne le déclenchement d'une action β dans un temps déterminé C, alors dans le graphe de dépendance une arête aura pour action de départ α et action d'arrivée β. Celle-ci sera étiqueté par "$< C$" pour préciser le type de dépendance (si le déclenchement de l'action α avait eu pour conséquence de geler le déclenchement de l'action β pendant un temps D on aurait eu comme étiquette "$> D$").

2.3.2 Génération des alarmes et des transitions

On part maintenant du graphe de dépendance pour générer un ensemble d'alarmes et de transitions.

Pour générer les alarmes, on regarde dans le graphe quelles actions sont sommets d'arrivée d'une arête. Par exemple si l'action β est sommet d'arrivée d'une arête étiquetée par "$> C$" comme dans notre exemple précédent, alors on génère une action de nom β, de type U, de borne C et dont le régime est stricte (à cause de l'inégalité stricte). De cette façon on peut remplacer chaque étiquette dans le graphe de dépendance par une alarme.

Ensuite, à chaque action correspond une transition qui porte le même nom et qui hérite de son état partiel de départ et de son état partiel d'arrivée. Pour déterminer quelles alarmes seront allumées ou éteintes par cette transition (qui rappelons-le correpond à un sommet du graphe et non pas à une arête), on opère de la façon suivante : pour chaque arête entrante de l'action on éteind l'alarme associée à cette arête, et pour chaque arête sortante on allume l'alarme associée à cette arête.

Mais nous n'avons pas encore pris en compte la section *contrainte* de notre spécification en langage naturel.

2.3.3 Ajout des alarmes de contrainte

En fait à chaque contrainte va correspondre une alarme possédant les mêmes caractéristiques (nom, type, borne, régime de la borne). Il reste maintenant à déterminer quelles transitions vont allumer ces alarmes et quelles transitions vont les éteindre.

L'idée est ici fort simple mais la mise en place très pénible : étant donné une contrainte qui a pour état partiel de départ e_1 et pour état partiel d'arrivée e_2, à chaque fois qu'une des transitions fait entrer dans l'état e_1 on lui ajoute l'alarme associée à la contrainte dans sa liste des alarmes à allumer. A chaque fois qu'une des transitions fait entrer dans l'état e_2 on lui ajoute l'alarme associée à la contrainte dans sa liste des alarmes à éteindre.

La difficulté est qu'une transition est en fait une façon compacte de représenter un ensemble d'instances de transitions et il faut donc faire le tri parmi celles-ci pour déterminer lesquelles répondent aux critères précédents. On est donc obligé d'instancier les transitions mais juste ce qu'il faut de manière à garder une forme compacte.

Remarque :

Contrairement à l'approche classique basée sur les automates temporisés dans laquelle un système est représenté comme un graphe de transition entre les états du systèmes, ici une spécification en langage naturel peut être vu comme un graphe de dépendance entre des transitions nommées du système.

Ainsi une phrase du langage naturel sera de la forme : si la transition α est déclenchée, alors la transition β se déclenchera moins de 10 secondes plus tard. Alors qu'avec un automate temporisé on écrirait plutôt des phrases du type : si je suis dans l'état e, alors je passerai forcément dans l'état e' moins de 10 secondes plus tard. Les deux approches peuvent donc apparaître comme duales.

Chapitre 3

Modélisation de comportements dynamiques

Nous avons vu jusqu'ici que cette approche basée sur la logique linéaire pouvait faire au moins aussi bien que les automates temporisés dans la spécification de systèmes temps réel et la vérification automatique de certaines propriétés standards.

C'était un premier point à vérifier avant de se concentrer sur certains aspects qui sont abordables avec cette approche mais qui ne le sont pas avec l'approche classique basée sur les automates temporisés.

Nous allons montrer dans ce chapitre que l'un des principaux avantages de l'approche est de permettre de modéliser des comportements dynamiques. Dans ce domaine nous nous sommes concentré sur deux phénomènes : la prise en compte d'un nombre potentiellement infini d'agents qui peuvent rejoindre le système à tout moment, et la possibilité de changer la valeur d'une alarme en cours d'exécution du système. En fait, avec la logique linéaire il est très facile de spécifier ce genre de comportements.

3.1 Changement d'une contrainte de temps

Permettre le changement d'une contrainte de temps est très simple. Par exemple si nous voulons laisser à tout moment la possibilité pour une date limite dépendant d'une alarme U_α d'être repoussée d'une valeur C nous pouvons rajouter l'axiome :

$$U_\alpha(v) \otimes (v < +\infty) \multimap U_\alpha(v + C)$$

Nous pourrions aussi changer la valeur d'une alarme en lui assignant une nouvelle valeur par rapport au temps courant, et nous écririons :

$$Time(t) \otimes U_\alpha(v) \otimes (v < +\infty) \multimap Time(t) \otimes U_\alpha(t + C)$$

Dans les exemples ci-dessus, le changement de la valeur de l'alarme peut avoir lieu à tout moment, mais il est aussi possible de ne permettre ce changement que dans le cas où une certaine transition est effectuée. Si par exemple on veut que la transition $\alpha \multimap \beta$ entraîne l'avancée de l'alarme L_γ d'une valeur C, nous remplacerions cette transition par :

$$\alpha \otimes L_\gamma(v) \otimes (v < +\infty) \multimap \beta \otimes L_\gamma(v + C)$$

Laisser à l'utilisateur la possibilité de spécifier son système ne remet pas en cause la décidabilité du problème de sûreté tel qu'il a été décrit précédemment.

A titre de démonstration il a été implémenté dans le programme cette possibilité de repousser une date limite.

Voici la syntaxe utilisée dans une spécification en langage naturel :

```
Timer_changes = {
  Extended_deadline(g1,1)
}
```

Comme dans le langage naturel on ne fait pas explicitement mention des alarmes, on utilise le nom d'une action ou d'une contrainte qui , rappelons-le, sont en correspondance directe avec les alarmes. Dans l'exemple ci-dessus, on spécifie simplement qu'à tout moment la contrainte haute sur la transition $g1$ peut être repousser d'une unité.

3.2 Ajout de nouveaux agents

Permettre d'ajouter en cours d'exécution de nouveaux agents sans en restreindre le nombre est aussi très naturel dans le formalisme de la logique linéaire. Par contre, il est nécessaire de modifier notre précédent formalisme de deux manières : d'une part chaque agent sera désormais représenté par un prédicat binaire qui contiendra le numéro de l'agent en plus de son état de contrôle courant, d'autre part, pour rendre les exemples intéressants il est nécessaire de pouvoir paramétrer les règles par des entiers, afin qu'elles puissent s'appliquer à n'importe quel agent ayant rejoint le système.

Pour des raisons techniques on doit aussi ajouter un prédicat qui sera chargé de compter le nombre d'agents dans le système.

La manière la plus simple de spécifier l'ajout d'un nouveau train dans notre système ferroviaire sera alors :

$$Compteur(n) \multimap Train(n+1, far) \otimes Compteur(n+1)$$

Bien sûr, comme pour le changement d'alarme, on peut vouloir ne permettre cette entrée en lisse d'un nouvel agent que sous certaines conditions.

Par exemple après une certaines date t_1 :

$$Compteur(n) \otimes Time(t) \otimes (t > t_1) \multimap Train(n+1, far) \otimes Compteur(n+1) \otimes Time(t)$$

Ou lorsque une transition $\alpha \multimap \beta$ se produit :

$$\alpha \otimes Compteur(n) \multimap \beta \otimes Train(n+1, far) \otimes Compteur(n+1)$$

Bien entendu, le nombre d'agent étant potentiellement infini, même après avoir éliminé le temps on garde un graphe de transition avec un nombre infini de sommets. Au mieux on obtient alors une procédure de semi-décision.

Chapitre 4

Propriétés décidables

Une fois qu'on a défini formellement un système temps-réel, il existe plusieurs problèmes classiques qu'on peut vouloir résoudre. Le plus simple, le problème de sûreté, est le seul qui ait été implémenté dans notre vérificateur à l'heure actuelle et c'est donc celui qui sera le plus abordé dans ce rapport. On présente cependant briévement d'autres problèmes qui pourront faire l'objet d'une implémentation future.

4.1 Problème de sûreté

Étant donnés une configuration initiale $Init \equiv Temps(t_0) \otimes Alarmes(al_0) \otimes Etat(e_0)$ et un état partiel des agents a, on se demande s'il existe une évolution du système à partir de la configuration initiale qui permette d'atteindre une configuration compatible avec a.

Si l'on reprend notre formalisme de la logique linéaire, on peut spécifier entièrement notre propriété sans faire appel à des méta-quantifications existentielles grâce à la formule :

$$Init \quad \vdash \quad \exists t \, \exists al \, \exists b \, Temps(t) \otimes Alarmes(al) \otimes P(a) \otimes \bar{P}(b)$$

Le plus souvent, on utilise pour l'état partiel cible a un état jugé dangereux, par exemple $Train(crossing) \otimes Gate(up)$ dans notre simulation ferroviaire, d'où le nom du problème, mais en fait on a affaire à un problème d'accessibilité dans un graphe de transition. Le seul problème est que ce graphe est, à cause du temps, indéfini et qu'il est donc impossible de l'explorer de façon naïve. C'est tout le propos du chapitre suivant que de montrer qu'on peut en fait se ramener à un graphe fini en éliminant astucieusement l'information concernant le temps.

4.2 Autres problèmes décidables

4.2.1 Problème général d'accessibilité

Ce problème est similaire au problème de sûreté tout en étant plus symétrique. En effet on remplace la configuration initiale totalement déterminée dans le membre gauche de l'implication par une formule de même type que celle du membre droit.

Il généralise bien le problème de sûreté (ou d'accessibilité) dans le sens où on pourrait répondre au premier problème grâce à un algorithme qui déciderait du second.

Voici sa définition formelle en logique linéaire :

$$\exists t \, \exists al \, \exists b \, Temps(t) \otimes Alarmes(al) \otimes P(a) \otimes \bar{P}(b) \quad \vdash \quad \exists t \, \exists al \, \exists d \, Temps(t) \otimes Alarmes(al) \otimes R(c) \otimes \bar{R}(d)$$

où a et c sont les donneés du problèmes.

4.2.2 Problème de rendez-vous

Encore une fois ce problème est analogue au problème de sûreté. La différence est que maintenant on fixe le temps t de la configuration d'arrivée qui était auparavant quantifié existentiellement :

$$Init \quad \vdash \quad \exists al \, \exists b Temps(t) \otimes Alarmes(al) \otimes P(a) \otimes \bar{P}(b)$$

où $Init$, t et a sont les données du problème.

4.2.3 Problème d'ordonnancement

Étant donnés une configuration initiale et une liste de transitions d'état, on se demande s'il est possible d'intercaler entre ces dernières des transitions temporelles de manière à ce que l'ensemble devienne une exécution valide du système.

4.2.4 Problème d'appartenance

C'est le mème problème que précédemment à l'exception que maintenant on peut aussi intercaler des transitions d'état et ne plus se limiter aux seules transitions temporelles.

4.2.5 Propriété de vivacité

La propriété de vivacité exprime le fait que le système ne peut pas rester bloqué dans une configuration ou évoluer indéfiniment sans que le temps courant ne dépasse une certaine constante (c'est le paradoxe de Zeno, aussi connu comme celui d'Achille et de la tortue).

Plus formellement, pour toute configuration accessible c_1 du système et tout entier n, il doit exister une configuration c_2 accessible depuis c_1 et telle que le temps courant qui lui est associé soit supérieur à n.

4.3 Complexité

Le problème de sûreté, le problème général d'accessibilité et le problème d'ordonnancement sont PSPACE. Les autres problèmes sont eux-aussi décidables mais on connaît moins bien leur complexité qui est certainement exponentielle.

Dans ce rapport nous nous contenterons de donner un algorithme de résolution du problème de sûreté (voir chapitre suivant) et de montrer comment on pourrait rendre ce dernier PSPACE par rapport à la place utilisée pour stocker la relation de transition.

Enfin, curieusement, le fait de fixer le temps dans la configuration cible pour le problème de rendez-vous augmente sa complexité par rapport au problème de sûreté.

Chapitre 5

Elimination du temps pour le problème de sûreté

Dans ce chapitre nous allons nous intéresser à définir la sémantique d'une spécification compacte en terme de graphe de transition.

Nous reformulons aussi la notion de propriété de sûreté dans ce cadre nouveau faisant intervenir les graphes de transition, alors que dans le chapitre 3 nous nous placions dans le cadre de la logique linéaire. En fait grâce à un théorème de complétude de la logique linéaire vis-à-vis de la sémantique que nous allons décrire, établi par M. Okada [OKA00], on peut voir que ces deux formulations sont équivalentes.

Ensuite on énonce le théorème qui établit la complétude et la correction de l'algorithme qui a été implémenté dans le vérificateur automatique et qui décide de la vérification d'une propriété de sûreté.

5.1 Sémantique d'une spécification compacte

On va définir un graphe de transition étiqueté qui représentera les évolutions possibles du système. La signification de ce graphe est que tout chemin fini à partir d'une configuration initiale représente un segment initial valide d'une exécution du système.

Ce graphe de transition doit prendre en compte les deux possibilités qu'a le système pour évoluer, c'est-à-dire **de manière discrète** en déclenchant une des transitions nommées, qui sont instantannées, ou **de manière continue** en procédant à un écoulement du temps qui laisse les valeurs des agents et des alarmes inchangées.

On introduit donc deux types d'étiquettes : $Trans(\alpha)$ pour le déclenchement d'une transition nommée α et $Tick(\tau)$ pour l'écoulement d'un temps τ.

5.1.1 Transition temporelle

On aura une transition de la forme :

$$Temps(t) \otimes Alarmes(al) \otimes Etat(e) \quad \xrightarrow{Tick(\tau)} \quad Temps(t') \otimes Alarmes(al') \otimes Etat(e')$$

si :

- $t' = t + \tau$
- $al' = al$
- $e' = e$
- pour chaque valeur v d'une alarme allumée de type U on a $t' < v$ (ou $t' \leq v$ selon le régime de l'alarme)

5.1.2 Transition d'état

On aura une transition de la forme :

$$Temps(t) \otimes Alarmes(al) \otimes Etat(e) \quad \xrightarrow{Trans(\alpha)} \quad Temps(t') \otimes Alarmes(al') \otimes Etat(e')$$

avec

$$\alpha \equiv (e_{depart}, e_{arrivee}, alarmes\_a\_allumer, alarmes\_a\_eteindre)$$

si :
- $t' = t$
- e et e_{depart} d'une part et e' et $e_{arrivee}$ d'autre part sont compatibles
- pour chaque agent Ag qui n'apparaît pas dans $e_{arrivee}$ on a $e'(Ag) = e(Ag)$
- pour chaque alarme A de type U à allumer, soit A est déjà allumée et dans ce cas elle garde sa valeur, soit elle est éteinte et dans ce cas on lui assigne une valeur égale au temps courant t auquel on ajoute la valeur de la borne de l'alarme
- à chaque alarme A de type L à allumer on assigne une valeur égale au temps courant t auquel on ajoute la valeur de la borne de l'alarme
- pour chaque alarme A de type U à éteindre, on doit avoir $al(A) < \infty$ et $al'(A) = +\infty$
- pour chaque alarme A de type L à éteindre, on doit avoir $al(A) > t$ (ou $al(A) \geq t$ selon le régime de l'alarme A) et $al'(A) = +\infty$

Nous venons ainsi d'expliquer comment on peut représenter les comportements d'un système temps-réel à travers un graphe de transition dans lequel le temps induit un nombre infini de sommets. Pour cette raison nous l'appelerons G_∞.

5.2 Problème de sûreté

Lorsque un système temps-réel est spécifié de cette manière, on peut se demander si à partir de la configuration initiale on peut aboutir en suivant un chemin dans le graphe de transition à une configuration dans laquelle la partie *Etat* est fixée.

Plus formellement, étant donné un état total des agents e, existe-t-il un temps t et une valuation des alarmes al tels qu'on ait :

$$Temps(t_0) \otimes Alarmes(al_0) \otimes Etat(e_0) \quad \xrightarrow[G_\infty]{}{}^{*} \quad Temps(t) \otimes Alarmes(al) \otimes Etat(e) \ ?$$

Cette question est en fait un problème d'accessibilité dans un graphe orienté qui, parce que le nombre de sommets accessibles est potentiellement infini, ne se ramène pas à une procédure de parcours de graphe classique.

Nous allons montrer qu'on peut pourtant décider de cette question en éliminant le temps dans le graphe associé au système et en se ramenant de cette manière à un graphe fini dans lequel la recherche exhaustive d'un chemin sera possible.

5.3 Graphe de transition fini associé à une spécification

On commence par supposer sans perte de généralité que les bornes des alarmes sont des entiers, sinon on prend leur pgcd qui constituera une unité dans laquelle elles s'exprimeront toutes sous forme d'entiers. On voit que ce procédé, pour très sensible qu'il soit aux valeurs effectives des bornes, ne met en jeu qu'un étirement proportionnel du temps et des valeurs des alarmes et qu'il ne modifie en rien le problème de sûreté.

Les notions de tic long et de tic courts qui vont être introduites ont été abordées par M. Okada lors de ses conférences au LRI [OKA00].

Pour aboutir à un graphe de transition fini, que nous appelerons G par opposition à G_∞, nous avons dit que nous allions éliminer le temps. C'est pourquoi un sommet de ce graphe sera maintenant de la forme :

$$Etat(e) \otimes ALARMES(AL) \otimes Q(q)$$

où le prédicat $Temps$ a disparu. Pour la différencier de la notion de configuration précédente, nous appelerons une telle entité une *configuration réduite*.

Nous voyons que l'information du temps courant ($Temps(t)$) a été remplacée par un prédicat Q. Chaque valeur q de Q fixe la position relative des alarmes entre elles et vis-à-vis du temps courant. Le nombre d'alarmes étant fini, q ne peut prendre qu'un nombre fini de valeurs.

Nous allons définir une fonction $\hat{(.)}$ qui associe à toute configuration sa configuration réduite. Bien entendu cette fonction ne sera pas injective puisque le nombre de configurations réduites est fini.

Auparavant, la signification de la partie $Temps(t) \otimes Alarmes(al)$ d'une configuration était que le temps courant est t et que pour chaque alarme A si $al(A) = +\infty$ alors A est éteinte, et si $al(A) = c < +\infty$ alors l'échéance de cette dernière interviendra à l'instant c.

Désormais on remplace ces informations dans une configuration réduite par la partie $ALARMES(AL) \otimes Q(q)$ qui ne garde qu'une partie des informations.

AL est une valuation des alarmes qui représente la partie entière du temps écoulé depuis le démarrage de l'alarme. Plus formellement, la valeur d'une alarme de type U est soit Off si elle est éteinte, soit un entier égal à la partie entière du temps écoulé depuis qu'elle a été allumée. Pour une alarme de type L, c'est la même chose sauf qu'on rajoute une valeur Top que pourra prendre l'alarme lorsqu'elle aura dépassé sa date limite.

Potentiellement le domaine des valeurs pour une alarme est infini, ce qui contredit note affirmation sur la finitude du graphe que nous voulons construire. Nous allons cependant voir lorsque nous définirons la relation de transition de ce graphe que celle-ci a pour propriété importante de garder chaque alarme dans l'intervalle défini par sa borne. Donc même si avec cette présentation le nombre de sommets du graphe reste infini, en fait seul un nombre fini de sommets sont accessibles.

Le prédicat Q a lui pour rôle de garder l'information sur la date d'activation des alarmes, mais pas de façon absolue. D'une part on ne s'intéresse qu'à la partie décimale de ces instants et d'autre part on ne s'intéresse pas tant à leur valeur qu'à leur position relative, l'une par rapport à l'autre, et aussi par rapport au temps courant qui sert de référence.

Soyons plus précis : pour obtenir la valeur de l'argument du prédicat Q à partir de $Temps(t) \otimes Alarmes(al)$, il suffit de placer sur un cercle unitaire la partie décimale correspondant à l'instant d'activation de chaque alarme (on extrait cette information de la valeur al des alarmes et de leur borne). Il faut aussi placer la partie décimale de la valeur courante du temps. Ensuite en partant de la position représentant le temps et en tournant dans le sens contraire des aiguilles d'une montre, on énumère les alarmes rencontrées.

Par exemple, à partir du cercle unitaire de la figure 5.1, on obtient comme valeur pour Q : $(A_4, >, A_0, >, A_1, =, A_3)$. La signification de cette expression est donc que toutes les alarmes qui n'y apparaissent pas (par exemple A_2) sont éteintes, et que les parties décimales des autres sont rangées dans cette ordre en prenant comme origine la valeur décimale

du temps courant.

Définissons maintenant la relation de transition associé au graphe que nous tentons de construire.

Désormais, la transition continue correpondant à l'écoulement du temps est partagée en deux types de transitions, l'une ("tic long") correspondant à un tour complet de notre horloge unitaire, l'autre ("tic court(i)") correpondant à un tour d'horloge inférieur à une unité mais qui permet de franchir au moins un nombre entier i d'alarmes sur le cercle unitaire. On imagine bien qu'on va pouvoir simuler un écoulement quelconque du temps par un nombre fini de tics longs suivi d'un tic court.

Les transitions discrètes dues au déclenchement d'une transition nommée sont assez similaires à ce qu'elles étaient pour G_∞, il faut simplement modifier la valeur du prédicat Q pour supprimer les alarmes qu'on aura éteintes et intégrer celles qui auront été allumées.

Les étiquettes utilisées qui correspondent à ces trois types de transition sont désormais $Long--tick$, $Short--tick(i)$ et $Trans(\alpha)$ comme précédemment.

Pour simplifier les conditions de chaque type de transition, on va considérer que toutes les alarmes sont strictes. La prise en compte d'alarmes dont les bornes sont larges ne pose aucune difficulté théorique supplémentaire mais nécessite de faire très attention aux cas limites.

5.3.1 L'horloge unitaire

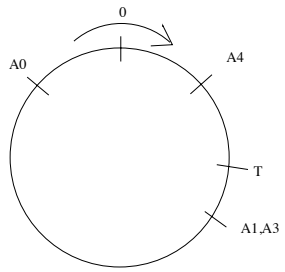

FIG. 5.1 – horloge

On associe à chaque configuration de l'horloge unitaire une valeur q. Celle-ci reflète l'ordre induit sur les alarmes par leur position sur l'horloge unitaire à partir du temps courant T. Par exemple avec la configuration dessinée dans la figure précédente on obtient $q = (>, A_4, >, A_0, >, A_1, =, A_3)$.

De même à chaque valeur q on peut associer une classe d'équivalence de cercles unitaires par l'opération inverse (deux cercles sont dits équivalents si l'on peut passer de l'un à l'autre en déplaçant les curseurs des alarmes et du temps sur le cercle tout en préservant leur ordre). Les deux représentations sont donc équivalentes modulo l'équivalence sur les cercles unitaires.

5.3.2 Tic long

On aura :

$$Etat(e) \otimes Alarmes(al) \otimes Q(q) \quad \xrightarrow{Long-tick} \quad Etat(e') \otimes Alarmes(al') \otimes Q(q')$$

si :

- $e' = e$
- chaque alarme allumée de type U est incrémentée d'une unité
- chaque alarme allumée de type L est incrémentée d'une unité si ce faisant elle n'atteint pas sa borne, dans ce dernier cas sa valeur est fixée à la constante spéciale TOP
- pour chaque alarme allumée A dans al de type U dans on a $al(A) < borne(A) - 1$
- $q' = q$

5.3.3 Tic court

On aura :

$$Etat(e) \otimes Alarmes(al) \otimes Q(q) \quad \xrightarrow{Short-tick(i)} \quad Etat(e') \otimes Alarmes(al') \otimes Q(q')$$

avec i compris entre 1 et le nombre de curseurs différents sur le cercle unitaire correspondant à q, si :

- $e' = e$
- chaque alarme allumée de type U est incrémentée d'une unité
- chaque alarme allumée de type L est incrémentée d'une unité si ce faisant elle n'atteint pas sa borne, dans ce dernier cas sa valeur est fixée à la constante spéciale TOP
- pour chaque alarme allumée A dans al de type U dans on a $al(A) < borne(A) - 1$
- pour obtenir q' on a déplacé le curseur du temps de i sections sur le cercle unitaire associé à q (dans le sens des aiguilles d'une montre)

5.3.4 Transition d'état

On aura :

$$Etat(e) \otimes Alarmes(al) \otimes Q(q) \quad \xrightarrow{Trans(\alpha)} \quad Etat(e') \otimes Alarmes(al') \otimes Q(q')$$

avec

$$\alpha \equiv (e_{depart}, e_{arrivee}, \mathbb{A}, \mathbb{E})$$

si :

- $t' = t$
- e et e_{depart} d'une part et e' et $e_{arrivee}$ d'autre part sont compatibles
- un agent qui apparaît dans e mais pas dans $e_{arrivee}$ garde sa valeur dans e'
- pour chaque alarme A de type U à allumer, soit A est déjà allumée et dans ce cas elle garde sa valeur, soit elle est éteinte et dans ce cas on l'initialise à 0
- chaque alarme A de type L à allumer est initialisée à O
- pour chaque alarme A de type U à éteindre, on doit avoir $al(A) \neq Off$ et $al'(A) = Off$
- pour chaque alarme A de type L à éteindre, on doit avoir $al(A) = Top$ et $al'(A) = Off$
- pour obtenir q' on efface du cercle unitaire représentant q les alarmes ayant été éteintes et on rajoute les alarmes qui ont été allumées à l'endroit où se trouve le temps courant t

5.4 Théorème d'élimination du temps

Ce théorème exprime la correction et la complétude de d'élimination du temps dans le graphe de transition initiale.

Théorème 1 (Elimination du temps) *Pour toute configuration init et toute valuation totale des agents e, les deux affirmations suivantes sont équivalents :*

1. *il existe un instant t et une valuation des alarmes al tels que :*

$$init \quad \xrightarrow[G_\infty]{}^{*} \quad Temps(t) \otimes Alarmes(al) \otimes Etat(e)$$

2. *il existe une valeur q et une valuation des alarmes AL tels que :*

$$\hat{init} \quad \xrightarrow[G]{}^{*} \quad Etat(e) \otimes ALARMES(AL) \otimes Q(q)$$

Preuve :

Ce théorème est en fait un corrolaire des deux lemmes de correction et de complétude qui suivent.

Lemme 1 (Correction) *Pour toute configuration c, état des agents e, valuation des alarmes AL et valeur q, si*

$$\hat{c} \quad \xrightarrow[G]{} \quad Etat(e) \otimes ALARMES(AL) \otimes Q(q)$$

alors il existe un instant t et une valuation des alarmes al tels que :

$$c \quad \xrightarrow[G_\infty]{} \quad Temps(t) \otimes Alarmes(al) \otimes Etat(e)$$

Idée de la preuve :

La preuve se fait par cas sur le type de transition utilisée :

- **tic long :** c'est le cas le plus simple, la transition correspondante est étiquetée par $Tick(1)$. On prend pour t la valeur du temps dans c incrémentée d'une unité, et pour al sa valeur dans c.
- **tic court :** la transition correspondante est étiqueté par $Tick(\tau)$. On repère sur le cercle unitaire de q entre quelles alarmes est situé le curseur de temps et on donne à t la valeur correspondant au milieu de cette intervalle. On prend pour al sa valeur dans c.
- **transition d'état α :** la transition correpondante est étiquetée par $Trans(\alpha)$. Comme elle est instantanée le temps ne change pas donc on prend pour t la valeur du temps dans c. Pour al, Les alarmes qui sont éteintes par la transition sont mises à $+\infty$ et les autres sont fixées au temps courant de c auquel on ajoute la valeur de leur borne.

Au niveau de l'implémentation, cette preuve est utilisée pour reconstruire une exécution réelle du système (avec indication des valeurs du temps et des alarmes) à partir d'un chemin dans le graphe de transition fini aboutissant à un état dangereux.

Lemme 2 (Complétude) *Pour toutes configurations c_1 et c_2, si $c_1 \xrightarrow[G_\infty]{} c_2$ alors $\hat{c_1} \xrightarrow[G]{}^{*} \hat{c_2}$*

Idée de la preuve :

La preuve se fait par cas sur le type de la transition utilisée.

Remarques :

Dans les conférences qu'il a données au LRI pendant l'été 2000 [OKA00], Mitsuhiro Okada a proposé une traduction systématique des règles de transition incluant le temps en des règles de transition dans lequel il est absent en se basant sur les notions de tic long et de tic court pour simuler la progression du temps.

Auparavant il n'existait qu'un théorème donnant la complexité du problème de sûreté [KOS97] et de la preuve duquel il était assez difficile d'extraire un algorithme.

Le théorème 1 justifie l'élimination du temps dans la procédure de décision concernant le problème de sûreté. Par contre, la façon dont il est formulé en terme de graphes de transition n'est pas généralisable aux autres problèmes, comme le problème générale d'accessibilité, qui requiert une formulation plus générale mettant d'avantage en évidence l'utilisation de la logique linéaire. Parmi les problèmes présentés au chapitre 4, le problème de rendez-vous nécessite la forme la plus générale.

5.5 Calcul du nombre de sommets du graphe de transition fini

Rappelons qu'une configuration réduite est de la forme :

$$Etat(e) \otimes ALARMES(AL) \otimes Q(q)$$

Pour compter le nombre de configurations réduites, on doit additionner le nombre de valeurs que peut prendre chaque agent, chaque alarme et le nombre de combinaisons possibles pour q.

- Le nombre de valeurs que peut prendre un agent est égal à son nombre d'états
- Le nombre de valeurs que peut prendre une alarme ayant une borne B est $B+2$ pour une alarme de type U ($0 \leq U \leq B$ ou $U = Off$) et $B+3$ pour une alarme de type L (on rajoute une valeur possible correspondant à la valeur Top). On a ici considéré que l'alarme U était de type strict et l'alarme L était de type large. Dans les autres cas, il faut retrancher 1 aux chiffres trouvés. On a aussi fait l'hypothèse que les alarmes ne dépassaient pas leurs bornes au cours de l'évolution du système, ce qui se vérifie aisément en observant la forme des transitions permises.
- une valeur pour Q est déterminée par l'ensemble des alarmes éteintes et par la position des autres sur le cercle unitaire. Si l'on appelle $f(n)$ le nombre de façons de placer n curseurs sur le cercle unitaire à isomorphisme d'ordre près et si N est le nombre d'alarmes on doit donc considérer le nombre $\sum_{k=1}^{N+1} C_{N+1}^{k} f(k)$ (on prend $N+1$ et non pas seulement N car on doit aussi placer le curseur du temps sur le cercle unitaire). Pour calculer f, on considère une autre fonction g telle que $g(n,p)$ soit le nombre de façons de placer n curseurs sur le cercle unitaire sachant que déjà p y sont présents. On a donc $f(n) = g(n,0)$ et g est définie par récurrence par :
  - $g(0,k) = 0$
  - $g(n+1,0) = g(n,1)$
  - $g(n+1,p+1) = (p+1).(g(n,p+1) + g(n,p+2))$

On peut déjà deviner que le nombre de configurations réduites sera très important même pour des systèmes très simples. Nous en avons la preuve en examinant les résulats de tests effectués avec plusieurs valeurs des paramètres. En fait, l'explosion combinatoire provient de deux choses :

- des bornes des alarmes qui sont démultipliées lorsqu'elles sont exprimées dans l'unité de leur pgcd

– du nombre de combinaisons possibles pour la valeur de Q qui est exponentiel par rapport au nombre d'alarmes

Heureusement, toutes les configurations réduites ne sont pas accessibles.

Remarque : si l'on choisit un temps discret, le calcul du nombre de combinaisons pour q est bien plus simple ; en effet, si l'on considère le cercle unitaire, les curseurs du temps et de toutes les alarmes allumées se trouveront toujours concentrée à l'origine. Un cercle unitaire d'une configuration est donc déterminé par le sous-ensemble des alarmes allumées, si bien que le nombre de combinaisons devient 2^N si N est le nombre d'alarmes.

5.6 Procédure de parcours du graphe

Le théorème 1 nous dit qu'on peut se ramener pour décider du problème de sûreté à un graphe de transition fini. On peut maintenant appliquer une procédure de parcours de graphe fini classique en largeur ou en profondeur. La procédure implémentée dans le vérificateur est actuellement basée sur un algorithme de parcours en largeur où la seule stratégie accélératrice utilisée est celle consistant à interdire un tic long après un tic court (en effet toute séquence de tics loncs et courts entremêlés peut être simulée par une séquence de tics longs suivie d'un seul tic court).

Cependant, on peut envisager une procédure de parcours du graphe qui privilégierait par ordre de priorité les transitions d'état, puis les tics courts, puis les tics longs. Ainsi la première configuration dangereuse détectée, dans le cas où le système serait non sûr, aurait aussi la propriété d'être parmi les premières accessibles en terme de temps (pas forcément la première, mais dans la même tranche de temps unitaire que la première).

5.7 Représentation du graphe en mémoire

Comme nous l'avons vu le nombre de configurations réduites est très vite très grand, et même si le nombre des configurations accessibles est sensiblement plus petit on ne peut tout de même pas envisager de stocker en mémoire la relation de transition sous forme d'une liste de couples de configurations.

Dans l'implémentation qui a été réalisée, la relation de transition est représentée par une fonction qui calcule la liste des successeurs d'une configuration réduite donnée.

Pour expliquer la manière dont elle s'exécute nous devons introduire la notion de *règle* qui a pour but de représenter de manière compacte l'ensemble des instances de transitions correpondantes à un type de transition donnée. On aura par exemple une règle pour le tic long, une règle par tic court et une règle par transition d'état.

Une règle est une structure de donnée qui regroupe :

– un couple de configurations réduites, représentant une transition, dans lequel certaines valeurs du membre gauche (temps, alarmes ou q) sont remplacées par des variables et dans lequel certaines valeurs du membre droit sont remplacés par des termes

– une liste de préconditions paramétrées par les variables et qui peuvent aussi être vu comme des termes de type booléen. Ces préconditions sont exactement celles que nous avons utilisées pour définir notre graphe de transition fini

Les règles sont générées à partir de la spécification compacte.

Voici comment la fonction successeur calcule les successeurs d'une configuration réduite donnée à partir de la base de règles :

pour chaque règle

1. elle commence par essayer d'unifier le membre gauche de la règle avec la configuration réduite
2. si elle réussit, elle applique la substitution à la liste de préconditions pour tester si celles-ci sont vérifiées
3. si tel est le cas elle applique la substitution au membre droit pour obtenir une configuration réduite successeur

Le but était d'obtenir un algorithme PSPACE par rapport à la taille nécessaire pour stocker la relation de transition. Pour y arriver il sera nécessaire d'optimiser la procédure de parcours du graphe en se basant sur la preuve du fait que $NPSPACE = PSPACE$.

5.8 Temps discret

On peut traiter le temps continu, mais on peut aussi se limiter au temps discret. Il suffit de l'indiquer au programme qui empêchera simplement l'utilisation des tics courts dans la procédure de parcours du graphe de transition fini.

Chapitre 6

Exemple de vérification manuelle dans Coq

Nous avons présenté dans le chapitre précédent un algorithme qui décide de la sûreté d'un système. Pour des valeurs fixées de tous les paramètres de la spécification (configuration initiale, bornes des alarmes, etc), le programme renvoie soit une réponse positive, nous indiquant que le système est sûr, soit une réponse négative accompagnée de la trace d'une exécution qui conduit à un état dangereux.

Cependant on aimerait parfois prouver des propriétés plus générales pour lesquelles notre programme serait trop limité. On pourrait par exemple vouloir démontrer ou réfuter une propriété du système qui ne soit plus ne'cessairement décidable. Ou bien, même en restant dans le cadre d'un problème de sûreté, plutôt que de fixer la valeur des paramètres du système on préférerait avoir des hypothèses plus faibles comme l'appartenance à un intervalle ou l'existence de relations arithmétiques entre ces paramètres.

Reprenons notre exemple du train et de la barrière. Rappelons que le train met un temps supérieur à une constante C_1 pour atteindre le passage à niveau une fois qu'il a commencé à approcher, que moins de C_2 unités après avoir entamé son approche la barrière commence à se baisser et qu'enfin après avoir commence sa descente cette dernière met moins de C_3 unités pour se retrouver dans la position basse.

Après avoir vérifié la sûreté du système (le train ne doit pas passer le passage à niveau sans que la barrière ne soit abaissée) pour plusieurs valeurs de C_1, C_2 et C_3 on peut remarquer et ensuite vouloir prouver que si $C_1 > C_2 + C_3$ alors le système sera sûr et que si $C_1 \leq C_2 + C_3$ alors il existe la possibité pour le système d'atteindre un état dangereux.

Remarquons que même si les paramètres n'ont plus de valeurs fixées il est encore possible de spécifier formellement un tel système dans notre formalisme basé sur la logique linéaire, il suffit d'utiliser des variables dans notre environnement de preuve.

Pour illustrer cette possibilité nous avons décidé de prouver la seconde des deux propriétés du système du train énoncées ci-dessus, c'est-à-dire de faire une démonstration formelle que si $C_1 \leq C_2 + C_3$ alors il existe une exécution dangereuse du système.

Comme assistant de preuve nous avons choisi Coq, une implémentation du calcul des constructions inductives qui est une logique d'ordre supérieure.

Nous aurions pu formaliser la logique dans Coq puis s'appuyer strictement sur la correspondance entre preuve de la logique linéaire et exécution du système dans l'idée de prouver la propriété visée. C'est d'ailleurs ce qui a été fait dans un premier temps en utilisant avantageusement les prédicats inductifs pour simuler les règles d'inférence de la logique linéaire

et en se servant de la gestion des variables faite par Coq plutôt que d'implémenter une surcouche pour permettre de quantifier sur les formules de la logique linéaire.

En fait, le problème étant suffisamment simple (par exemple on n'introduit pas de comportements dynamiques comme l'intervention de nouveau agents ou de changements de la valeur des alarmes), on a préféré représenter le système dans Coq sous forme d'un graphe de transition étiqueté, ce qui rend la spécification ainsi que la preuve beaucoup plus compacte.

6.1 Spécification d'un système dans Coq

Le listing intégral de la spécification suivi de l'énoncé de la propriété à prouver se trouve dans les annexes. La preuve formelle de la propriété au moyen de tactiques Coq manquant de lisibilité nous nous contenterons de donner dans la section suivante une preuve en langage mathématique.

Voyons maintenant quels choix ont été réalisés pour modéliser dans Coq la spécification du système et la propriété. On se sert essentiellement de définitions inductives pour définir les structures de données (de type *Set*) et les prédicats (de type *Prop*).

6.1.1 Les rationnels

Pour représenter les valeurs relatives au temps on utilise les nombres rationnels, parfois complétés de la constante $+\infty$. On a donc besoin de définir formellement les rationnels, ainsi que l'égalité, la relation d'ordre naturel et l'addition sur cet ensemble. On donne aussi quelques règles de syntaxes pour les opérations.

```
Load rationnel.
```

6.1.2 Les agents du système

On représente les agents du système à l'aide de définitions inductives dont les constructeurs correspondent aux états du système. Par exemple pour le train on écrit :

```
Inductive Train : Set :=
| far : Train
| approaching : Train
| crossing : Train.
```

6.1.3 Une configuration

Une configuration du système regroupe les informations concernant le temps courant, l'état des agents et la valeur courante des alarmes. De nouveau on utilise une définition inductive pour spécifier une structure de donnée qui renferme toutes ces informations.

```
Inductive configuration : Set :=
  config : r -> Train -> Gate ->  r_infini ->  r_infini ->  r_infini ->
             r_infini -> configuration.
```

6.1.4 La configuration initiale

On définit une configuration initiale en passant au constructeur *config* les valeurs désirées :

```
Definition init := (config (Rat (0) (1) (lt_0_Sn (0)))
                           far up Infini Infini Infini Infini).
```

6.1.5 La relation de transition

Nous avons choisi de représenter notre système temps-réel sous la forme d'un graphe de transition dont les sommets seraient l'ensemble des configurations du système. Il nous reste maintenant à définir la relation de transition qui correspond aux évolutions du système.

Rappelons que dans notre modèle, le système ne peut évoluer que de façon continue en faisant s'écouler le temps, ou de manière discrète en déclenchant une transition nommée.

On va donc utiliser un prédicat inductif binaire *transition* qui prend en argument deux configurations. Un des constructeurs, nommé "tick", correspond à l'avancé du temps, les autres correspondent aux transitions nommées dont ils portent le nom.

Dans l'implémentation de notre vérificateur automatique de propriétés de sûreté, nous avions représenté de manière compacte la relation de transition grâce à un ensemble de termes qui instanciés révélaient l'ensemble des transitions possibles. Chacun de ces termes était accompagné d'un ensemble de conditions dont la satisfaction pour une instance donnée dépendait de la substitution associée.

Cette approche est également très naturelle pour représenter les transitions sous forme de constructeurs. En effet on peut représenter un ensemble d'instances de transitions en utilisant des variables et modéliser les conditions comme des propositions Coq dépendant de ces variables.

Cette manière de faire est visible dans le type des constructeurs *tick* et $a1$ reproduits ci-dessous et représentant respectivement l'avancée du temps et la transition $a1$ de notre exemple du train.

```
Inductive transition [B,H0,H1,H2 : r] : configuration -> configuration -> Prop :=

| tick : (t, s : r) (v_Train : Train) (v_Gate : Gate)
         (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (lt_rat s t) -> (le_ri (Fini t) v_U_g1) ->
 (le_ri (Fini t) v_U_g2) -> (lt_ri (Fini t) v_U_g3) ->
 (transition B H0 H1 H2 (config s (v_Train) (v_Gate) v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t (v_Train) (v_Gate) v_U_g1 v_U_g2 v_U_g3 v_L_c))

| a1 : (t : r)  (v_Gate : Gate) (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (transition B H0 H1 H2 (config t far v_Gate v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t approaching v_Gate (v_U_g1 || (t ++ H0)) v_U_g2 v_U_g3 (Fini (t ++ B))))

| ...
| ...
.
```

6.1.6 Un chemin

On définit aussi le prédicat d'accessibilité entre deux configurations qui n'est rien d'autre que la fermeture reflexive et transitive de la relation de transition et qui se prête donc naturellement à une définition par induction.

```
Inductive chemin [B,H0,H1,H2 : r] : configuration -> configuration -> Prop :=
  refl_chemin : (a : configuration) (chemin  B H0 H1 H2 a a)
| trans_chemin : (a,b,c:configuration) (transition B H0 H1 H2 b c) ->
  (chemin  B H0 H1 H2 a b) -> (chemin  B H0 H1 H2 a c).
```

6.2 Remarques

6.2.1 Paramétrisation de la spécification

Pour exprimer une propriété plus générale de notre système nous avons voulu paramétrer notre spécification par les valeurs des bornes des alarmes. La relation de transition dépendant de ces valeurs il est naturel de les retrouver dans les paramètres de la définition inductive correpondante. La relation *chemin* qui dépend de la relation *transition* est aussi paramétrée par ces grandeurs.

6.2.2 Traduction automatique

En examinant la façon dont on a traduit une spécification dans le formalisme de Coq on s'aperçoit que chaque étape pourrait être automatisée.

En fait, l'exemple de traduction du système du train qui figure en annexe a été généré par une procédure Ocaml de traduction qui fait partie du programme *okada*.

6.3 Exemple de preuve formelle de sûreté

6.3.1 La propriété

Enfin, on exprime la propriété de sûreté comme un lemme dont la traduction en langage naturel serait :

Pour toutes bornes B, $H0$, $H1$ et $H2$, si $B < H0 + H1$ alors il existe un temps t et une valuation al des alarmes tels qu'il existe un chemin qui relie, dans le graphe des évolutions du système, la configuration initiale à une configuration dans laquelle le temps courant est t, la valeur des alarmes est donné par al, le train est dans l'état où il traverse le passage a niveau et la barrière est en train de descendre.

```
Lemma cas_non_sur :
(B,H0,H1,H2 : r) (lt_rat B (H0 ++ H1)) ->
(EX t : r |
(EX v_U_g1 : r_infini |
(EX v_U_g2 : r_infini |
(EX v_U_g3 : r_infini |
(EX v_L_c : r_infini |

(chemin B H0 H1 H2 init (config t crossing mv_down v_U_g1 v_U_g2 v_U_g3 v_L_c))

)))))).
```

6.3.2 La preuve

La preuve formelle dans Coq au moyen de tactiques est en cours de développement et est basée sur l'idée suivante : il s'agit d'exhiber un chemin qui parte de la configuration initiale et qui aboutisse à une configuration dans laquelle le train traverse alors que la barrière est encore en train de s'abaisser.

L'exécution présentée sur la figure 7.1 répond bien à cette attente et on peut montrer que c'est une exécution valide :

1. à l'instant 0 on déclenche la transition $a1$, le système se retrouve dans l'état $Train(approach).Gate(up)$ et l'on devra déclencher la transition $g1$ avant (au sens large) la date $H0$

2. ensuite on effectue une avancée du temps de $H0$ unités, ce qui est permis car la borne HO est large
3. à l'instant $H0$ on déclenche la transition $g1$, le système se retrouve dans l'état $Train(approcah).Gate(mv\_down)$ et l'on devra déclencher la transition $g2$ avant (au sens large) la date $H0 + H1$
4. on effectue de nouveau une avancée du temps qui nous amène à la date limite (mais permise) $H0 + H1$
5. comme $H0+H1$ est strictement supérieur à B, l'interdiction concernant la transition $a2$ ne tient plus et on peut la déclencher, atteignant par là même l'état dangereux $Train(crossing).Gate(mv_down)$.

Remarques :

Pour simplifier la preuve, on a choisit de prendre les bornes $H0$ et $H1$ larges et la borne B stricte. Dans les autres cas, nous aurions sûrement dû utiliser pour finir la preuve le lemme suivant qui exprime la densité des nombres rationnels :

$$\forall p, q \in \mathbb{Q},\ p < q \Rightarrow \exists r \in \mathbb{Q}.\, p < r < q$$

La preuve mathématique du fait que le système était non sûr sous les hypothèses $B < H0 + H1$ s'est révélée très simple, même si sa formalisation en Coq risque d'être longue et fastidieuse. En fait, prouver que le système est non sûr est bien plus facile que de montrer qu'il est sûr. En effet, il suffit dans un cas d'exhiber une exécution dangereuse alors que dans l'autre cas, s'il n'existe pas une structure logique sous-jacente de la propriété il est nécessaire de prendre en compte tous les cas, qui peuvent être très nombreux.

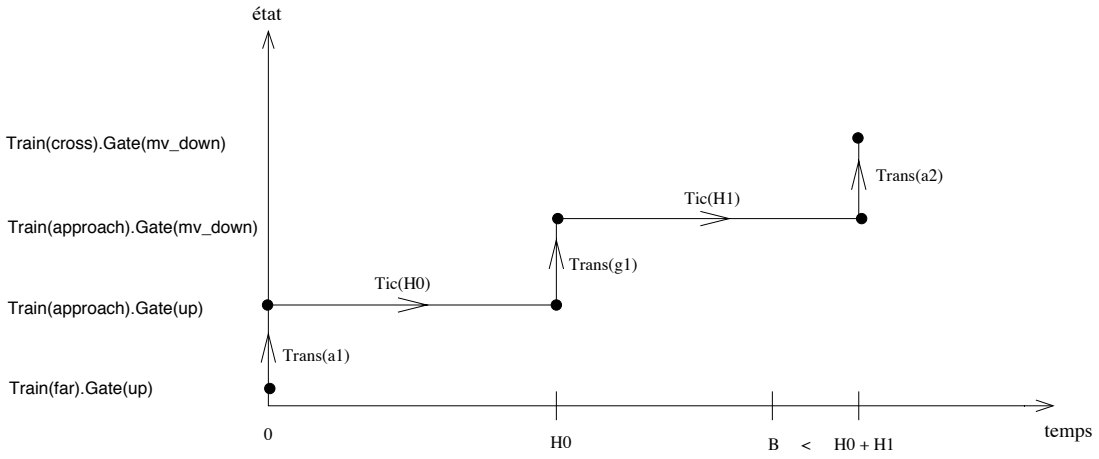

FIG. 6.1 – exécution dangereuse

Chapitre 7

Spécifications, architecture et tests du vérificateur

Dans ce chapitre nous allons présenter le programme écrit en Ocaml qui fait la synthèse de tous les concepts présentés jusqu'à présent. Sa fonction principale est de vérifier des propriétés de sûreté sur des systèmes temps-réel spécifiés en langage naturel, mais comme nous le verrons il est aussi capable de traduire la spécification dans le formalisme de la logique linéaire et de générer une spécification faite en terme de graphe de transition dans Coq.

Dans un premier temps on donne les spécifications du programme en précisant les données qu'il prend en entrée et expliquant comment doivent être comprises les données retournées en sortie.

Enfin, on commente l'architecture globale du programme en mettant en relation chaque module avec les concepts qui ont été introduits dans les chapitres précédents.

7.1 Spécifications du programme

Le programme prend **en entrée** :
- un fichier contenant une spécification du système en langage naturel et un ensemble d'états jugés dangereux (Cf. 2.1). Un exemple de fichier de spécification est fourni dans les annexes (le_train.spec, Cf. A.1)
- l'indication de la nature du temps : discret ou continu

Ces deux informations sont passées au programme sur la ligne de commande :

```
prog exemples/le_train.spec temps_continu
```

En sortie le programme renvoie sur la sortie standard :
- la spécification en logique linéaire traduite de celle en langage naturel au format ascii. C'est en fait un ensemble de formules qu'on décompose en plusieurs groupes : la formule de départ, les sous-formules d'arrivée correspondant aux états dangereux, la formule de progression du temps et les formules de transition
- le nombre de sommets du graphe de transition fini associé à la spécification
- enfin, dans le cas où il existe une exécution du système qui permette d'atteindre un état dangereux, le programme renvoie une trace de cette exécution. Dans le cas contraire, il conclut à là sûreté du système

La trace d'une exécution est de la forme :

```
0) {Depart} [0] : Train(far) . Gate(up)
```

```
1) {Transition} [0] : Train(approaching) . Gate(up)
2) {Temps} [19/6] : Train(approaching) . Gate(up)
3) {Transition} [19/6] : Train(approaching) . Gate(mv_down)
4) {Temps} [121/12] : Train(approaching) . Gate(mv_down)
5) {Transition} [121/12] : Train(crossing) . Gate(mv_down)
```

Chaque ligne représente la partie *temps* (entre crochets) et la partie *etat* d'une configuration (après les :). La première colonne est seulement le rang de la configuration sur le chemin. La deuxième colonne donne des indications sur le type de la transition qui a conduit à cette configuration (temporelle ou d'état). On aurait pu indiquer la valeur des alarmes, mais rappelons-nous qu'au niveau utilisateur l'utilisation des alarmes est transparente.

Pour qu'il n'y ait pas d'ambiguité, commentons les première lignes : au départ le système se trouve au temps 0 dans l'état $Train(far).Gate(up)$. Il effectue une transition discrète et se retrouve dans l'état $Train(approaching).Gate(up)$. Ensuite le temps s'écoule sans changement jusqu'à la date 19/6, etc.

On donne en annexe la sortie standard du programme correspondant au fichier de spécification mentionné plus haut (Cf. A.2)

Enfin, le programme génère un fichier Coq dans lequel est représenté sous forme d'un graphe de transition le système et la propriété de sûreté. Encore une fois on peut retrouver ce fichier dans les annexes (Cf. A.3)

7.2 Architecture du programme

L'implémentation fait environ 3000 lignes de code Ocaml organisées en plusieurs fichiers ou modules. Le style de programmation utilisé est exclusivement fonctionnel, à l'exception de la procédure de recherche d'un chemin dans le graphe de transition fini qui fait intervenir des traits impératifs.

Le découpage en modules du programmes est visible sur la figure 7.1. Les flèches représentent les dépendances entre modules et permettent de suivre pas-à-pas les différentes phases de traitement de l'information. Nous donnons ci-dessous les fonctions de chaque module en faisant à chaque fois référence aux chapitres ou aux sections qui lui correspondent :

- **analyseurs lexical et syntaxique :** ils sont générés par ocamllex et ocamlyacc à partir d'une description de la syntaxe et de la grammaire d'une spécification en langage naturel
- **langage naturel :** ce module renferme la définition du type de donnée représentant une spécification en langage naturel ainsi qu'un test de validité d'une spécification pour aider au débogage (Cf. 2.1)
- **spécification compacte** : contient la définition informatique d'une spécification compacte et la fonction de traduction d'une spécification faite en langage naturel en une spécification compacte (Cf. 2.2 et 2.3).
- **logique linéaire :** contient le type de donnée représentant une formule de la logique linéaire, la fonction traduisant une spécification compacte en une liste de formules et une fonction de formatage ascii de ces formules (Cf. 1)
- **spécification Coq :** est chargé de générer un fichier qui renferme la traduction dans Coq d'une spécification compacte en terme de graphe de transition (Cf. 6)
- **spécification entière** : contient une fonction qui calcule le pgcd des bornes des alarmes d'une spécification compacte et génère une spécification equivalente dans laquelle les bornes des alarmes sont maintenant des entiers. Afin de reconstruire une exécution réelle (Cf. module *interprétation*), on garde aussi en mémoire la valeur de l'unité (Cf. 5.3)

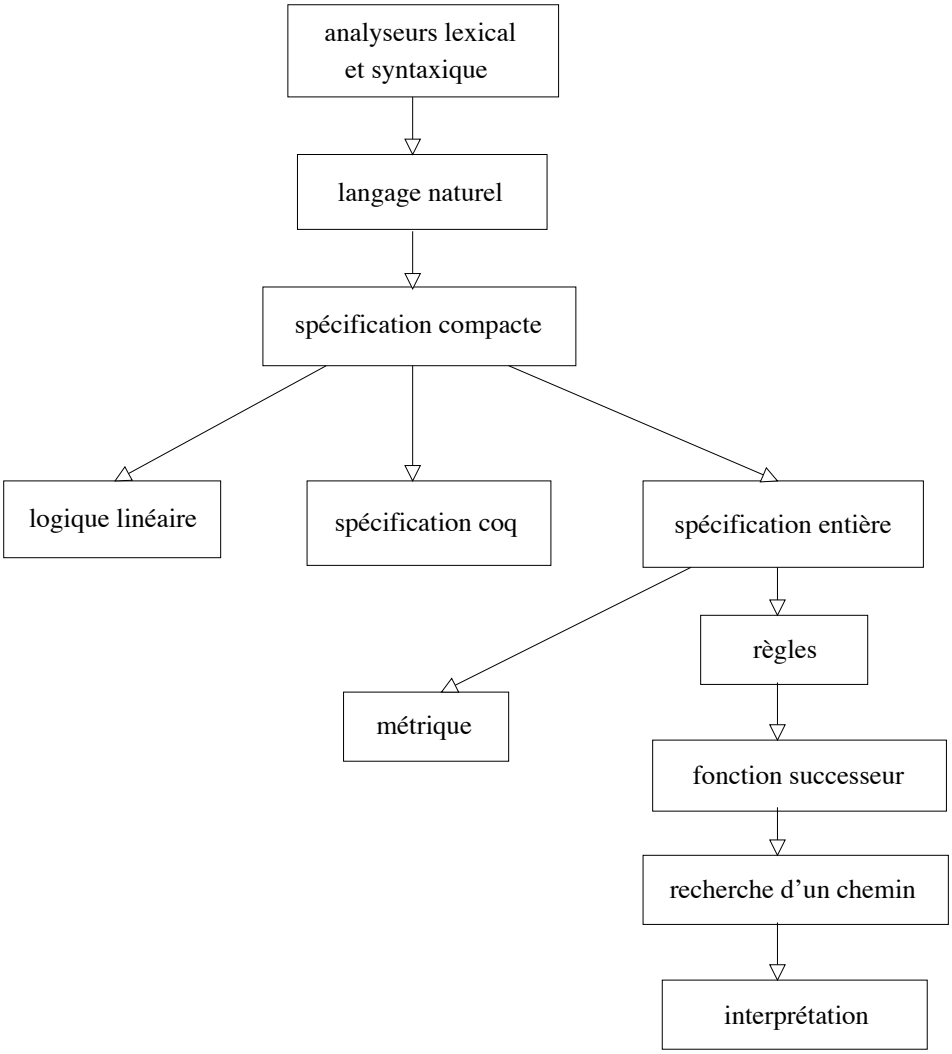

FIG. 7.1 – modules

- **métrique :** est charger de compter le nombre total de configurations réduites (Cf. 5.5)
- **règles :** renferme la définition d'une règle et une fonction qui les génère à partir d'une spécification compacte dans laquelle les bornes sont des entiers (spécification entière) (Cf. 5.7)
- **fonction successeur :** contient la fonction qui génère les successeurs d'une configuration réduite donnée à partir d'une base de règles. Le module contient donc aussi des fonctions d'unification et d'évaluation d'un terme ou d'une précondition par une substitution (Cf. 5.7)
- **recherche d'un chemin :** renferme l'algorithme de recherche d'un chemin entre la configuration réduite initiale et une configuration réduite dangereuse dans le graphe de transition fini (Cf. 5.6)
- **interpretation :** contient la fonction qui reconstruit une exécution réelle du système à partir d'un chemin à l'interieur du graphe de transition fini. L'algorithme correspond trait pour trait à la preuve du lemme 1 (Cf. 5.4)

7.3 Tests et analyses

Pour tous ces tests nous allons nous servir exclusivement de l'exemple du train tel qu'il a éte décrit dans le chapitre 1 (Cf. 1.4). Pour obtenir des résultats différents nous ferons varier la valeur des paramètres B_0, H_0, H_1 et H_2, ainsi que la nature du temps, discret ou continu.

Rappelons briévement la signification des différentes bornes :

- il s'écoule au moins B_0 unités entre le moment où le train commence à approcher et celui où il commence à traverser
- la barrière commence à descendre moins de H_0 unités après que le train ait commencé à approcher
- moins de H_1 unités plus tard la barrière est complètement abaissée
- la barrière met moins de H_2 unités pour atteindre la position haute à partir du moment où elle a commencé à remonter

7.3.1 Cas sûr / cas non sûr

On commence par montrer les deux réponses possibles du programme selon que le système est sûr ou on sûr.

Cas non sûr

	B_0	H_0	H_1	H_2
nature de la borne	stricte	stricte	stricte	large
valeur de la borne	10	7	3.1	2

Ici, comme $B_0 < H_0 + H_1$, il va y avoir possibilité pour le système de commencer à traverser le passage à niveau alors que la barrière est encore en train de s'abaisser, c'est ce que répond le programme au bout de cinq minutes :

```
Nombre de sommets : 22611561984

Nombre de sommets visites : 13367

Le systeme n'est PAS SUR :
```

```
0) {Depart} [0] : Train(far) . Gate(up)
1) {Transition} [0] : Train(approaching) . Gate(up)
2) {Temps} [139/20] : Train(approaching) . Gate(up)
3) {Transition} [139/20] : Train(approaching) . Gate(mv_down)
4) {Temps} [401/40] : Train(approaching) . Gate(mv_down)
5) {Transition} [401/40] : Train(crossing) . Gate(mv_down)
```

Cas sûr

Simplement en augmentant un peu le délai d'arrivée du train (de 10 à 10.1) on obtient un système sûr :

	B_0	H_0	H_1	H_2
nature de la borne	stricte	stricte	stricte	large
valeur de la borne	10.1	7	3.1	2

Au bout de deux heures le programme répond :

```
Nombre de sommets : 22831091712

Nombre de sommets visites : 53072

Le systeme est SUR
```

Quand le système est sûr, le nombre de sommets visités à la fin de la vérification correspond au nombre de sommets accessibles. On voit ici que ce nombre est bien moins important que le nombre total de sommets du graphe.

On remarque aussi, et ceci peut apparaître comme une sorte de paradoxe, que le programme est plus efficace lorsque le système est non sûr (il termine plus vite). Cela provient du fait que pour prouver que le système est non sûr il suffit de trouver un chemin entre la configuration initiale et une configuration dangereuse alors que prouver la sûreté nécessite d'avoir parcouru complètement le graphe des configurations avant de se prononcer.

7.3.2 Explosion du nombre de sommets

On va essayer de voir quel est l'influence de la précision des bornes sur le nombre de sommets du graphe de transition.

On considère que toutes les bornes sont des bornes larges.

Voici les résultats :

B_0	H_0	H_1	H_2	nombre de sommets
10	7	3	2	9 486 720
10	7	10/3	2	310 321 152
10.1	7.2	3.3	2.7	33 977 961 360
10.18	7.29	3.32	2.71	298 599 443 513 280

Dans la première ligne tous les bornes sont entières et le nombre de sommets est acceptable. Dans la deuxième ligne, on transforme la valeur entière 3 en le nombre rationnel 10/3 et le nombre de sommets augmente considérablement. Dans les troisièmes et dernières lignes on augmente les valeurs d'une, puis de deux décimales et l'on assiste à une explosion combinatoire.

Ce phénomène a pour origine la réduction des bornes à une même unité , par le calcul de leur pgcd, qui augmente considérablement leur valeur.

On aurait observer un phénomène analogue si l'on avait augmenter le nombre des bornes plutôt que leur précision comme cela est expliquer dans la section 5.5.

7.3.3 Comparaison temps discret / temps continu

Rappelons qu'on peut indiquer au programme qu'on se restreint à un temps discret. On va comparer ici le nombre de sommets du graphe dans les deux cas, discret et continu, lorsque les bornes ont les valeurs suivantes :

	B_0	H_0	H_1	H_2
nature de la borne	large	large	large	large
valeur de la borne	10	7	3	2

Voici le résultat :

	nombre de sommets
temps discret	414 720
temps continu	9 486 720

Comme attendu, dans le cas discret on a un graphe des transitions qui est beaucoup plus petit, les dates d'échéance des alarmes étant forcément des entiers.

7.3.4 Repousser une date limite

Considérons les valeurs suivantes pour les bornes :

	B_0	H_0	H_1	H_2
nature de la borne	stricte	stricte	stricte	stricte
valeur de la borne	10	10/3	6	2

Comme $B_0 \geq H_0 + H_1$, le système est sûr, comme le vérifie rapidement le programme :

```
Nombre de sommets : 18973440

Nombre de sommets visites : 528

Le systeme est SUR
```

Ajoutons mettant dans la section *Timer_changes* du fichier de spécification la possibilité pour la barrière de se lever 1 unités plus tard que prévu :

```
Extended_deadline(g1,1)
```

Désormais le programme découvre une exécution malheureuse, celle où le système, d'entrée de jeu, prend l'opportunité qui lui est donnée de repousser la date limite d'abaissement de la barrière :

```
Nombre de sommets : 212041368

Nombre de sommets visites : 3310

Le systeme n'est PAS SUR :

0) {Depart} [0] : Train(far) . Gate(up)
1) {Transition} [0] : Train(approaching) . Gate(up)
2) {Extended_timer} [0] : Train(approaching) . Gate(up)
3) {Temps} [25/6] : Train(approaching) . Gate(up)
```

```
4) {Transition} [25/6] : Train(approaching) . Gate(mv_down)
5) {Temps} [121/12] : Train(approaching) . Gate(mv_down)
6) {Transition} [121/12] : Train(crossing) . Gate(mv_down)
```

Remarques :

Dans tous les exemples qu'on a pris, lorsque $B_0 \geq H_0 + H_1$ on a vu que le système était sûr, on peut donc conjecturer que c'est une condition suffisante pour obtenir un système sûr et vouloir le prouver formellement avec Coq. Cette démarche qui combine la vérification automatique et la vérification assistée semble assez intéressante, même si l'exemple que nous considérons est un peu trop simple pour vraiment sans rendre compte.

Chapitre 8

Proposition d'extension du langage naturel

On se propose ici de raffiner le langage concu par Mitsuhiro Okada, Max I. Kanovich et Andre Scedrov pour décrire les systèmes temps réel. L'idée est de le rendre plus expressif et plus cohérent, ce qui devrait en faciliter l'utilisation.

La description d'un système dans ce langage se fonde sur les notions très simples de *transitions nommées*, *signaux* et *directives*. Intuitivement le système ne peut changer d'état qu'en effectuant une des transitions nommées si les contraintes qui pèsent sur elle sont satisfaites. Lors de chaque transition sont émis trois types de signaux. L'utilisateur peut alors associer à chacun de ces signaux un ensemble de directives qui influeront sur l'évolution du système.

On souhaite aussi utiliser par la suite le même formalisme logique, celui de la logique linéaire, auquel on ajoutera seulement un prédicat pour indiquer si une transition est verrouillée ou non.

8.1 Description d'un système

Un *système* est composé d'un ensemble fini d'*agents*. A chaque agent on associe un ensemble fini d'états. L'état du système dans son ensemble est donné par une valuation de chacun de ses agents par un de ses états.

On appelle *évènement partiel* ou plus simplement *évènement* une valuation partielle des agents et *évènement total* une valuation totale des agents.

Deux évènements sont dits *compatibles* si un agent représenté dans les deux évènements l'est avec le même état.

Une *configuration* du système est la donnée d'un temps t et d'un évènement total a. La première phase de description d'un système consiste à se donner une configuration initiale.

8.1.1 Transition nommée

La seconde phase de description du système consiste à se donner une liste de *transitions nommées*.

On définit une *transition* comme un couple d'évènements partiels. En fait, une transition $a \to b$ représente l'ensemble fini de ses instances dont chacune est un couple d'évènements totaux $c \to d$ tel que d'une part a et c, et d'autre part b et d soient compatibles.

A chaque instant, une transition peut être verrouillée ou non. Une transition verrouillée ne pourra pas être déclenchée, mais une transition déverrouillée ne sera pas forcément

déclenchable suivant les contraintes de temps qui pèsent sur elle. On indiquera aussi pour chaque transition nommée si elle est verrouillée ou non au départ.

Par la suite, le système ne pourra passer d'une configuration à une autre que de deux facons différentes :

- soit de manière continue, en faisant avancer le temps (dans ce cas seule la partie temps de la configuration change),
- soit de manière discrète, en déclenchant une des transitions nommées (dans ce cas seule la partie évènement de la configuration change).

8.1.2 Signaux

Le déclenchement d'une transition, dont l'effet principal est de passer d'une configuration à une autre, provoque l'émission de *signaux* de trois types différents :

$Trans(n)$	signifie que c'est la transition de nom n qui a été déclenchée
$Enter(a)$	signifie que la configuration d'arrivée est compatible avec l'évènement partiel a mais que la configuration initiale ne l'est pas
$Exit(a)$	signifie que la configuration de départ est compatible avec l'évènement partiel a mais que la configuration d'arrivée ne l'est pas

8.1.3 Directives

On définit ensuite quatre types de *directives* grâce auxquelles on pourra influer sur l'évolution du système.

$Cause(c)$	où c est une combinaison booléenne positive de couples *(signal,temps)*, force le système à déclencher un certains nombre de transitions avant un certain temps de manière à ce que "la combinaison soit satisfaite" (voir explications ci-dessous)
$Froze(s,t)$	empêche pendant un temps t le déclenchement des transitions qui aboutiraient à émettre le signal s
$Lock(n)$	verrouille la transition de nom n
$Unlock(n)$	déverrouille la transition de nom n

Donnons deux exemple pour mieux comprendre la directive *Cause* :

- $Cause((s1,t1)and(s2,t2))$ forcera le système à évoluer de telle sorte que le signal s1 soit émis avant t1 et le signal s2 avant t2.
- $Cause((s1,t1)or(s2,t2))$ forcera le système à évoluer de telle sorte que le signal s1 soit émis avant t1 ou le signal s2 avant t2.

Remarquons aussi que l'apparition de $(Trans(n),t)$ dans une directive Cause entraîne automatiquement le déverrouillage de la transition de nom n.

L'utilisateur peut alors se servir de ces signaux pour décider d'appliquer certaines directives quand certains signaux sont recus.

La dernière phase de la description est donc de se donner une liste d'*actions* qui associe à certains signaux une liste de directives.

8.2 Syntaxe formelle du langage

Les unités syntaxiques de base sont TIME, NOM_TRANSITION, EVENT et AGENT.

TIME est un nombre rationnel. NOM_TRANSITION est un identificateur. Un évènement partiel EVENT a la forme : `(Train=far,Gate=up)`. Un agent AGENT a la forme : `Train := far | approaching | crossing`

```
SPEC ::=
  System = { liste d'AGENTs }
  Initial_event = EVENT
  Initial_time = TIME
  Transitions = { liste de TRANSITIONs }
  Actions = { liste d'ACTIONs }

TRANSITION ::=
  NOM_TRANSITION(LOCK) : EVENT -> EVENT

LOCK ::= on | off

ACTION ::= SIGNAL -> liste de DIRECTIVESs

SIGNAL ::=
  Trans(NOM_TRANSITION)
| Enter(EVENT)
| Exit(EVENT)

DIRECTIVE ::=
  Cause(COMBINAISON)
| Froze(SIGNAL,TIME)
| Lock(NOM_TRANSITION)
| Unlock(NOM_TRANSITION)
```

où COMBINAISON est une combinaison linéaire positive de couples (SIGNAL,TIME).

8.3 Exemple de description d'un système temps réel

Le plus simple pour comprendre est encore de reformuler l'exemple du train :

```
System = {
  Train := far | approaching | crossing;
  Gate := down | mv_up | up | mv_down
}

Initial_event = (Train=far,Gate=up)
Initial_time = 0

Transitions = {

t1(on) : (Train=far) -> (Train=approaching)
t2(on) : (Train=approaching) -> (Train=crossing)
t3(on) : (Train=crossing) -> (Train=far,Gate=mv_up)
```

```
g1(off) : () -> (Gate=mv_down)
g2(off) : (Gate=mv_down) -> (Gate=down)
g3(off) : (Gate=mv_up) -> (Gate=up)

}

Actions = {

Enter(Train=approaching) -> Froze(Enter(Train=crossing),10.1)
Trans(a1) -> Cause(Trans(g1),2.5)
Trans(g1) -> Cause(Trans(g2),9)
Trans(a3) -> Cause(Trans(g3),2)

}
```

8.4 Traduction des extensions du langage

Par rapport à la traduction d'une spécification en langage naturel classique il faut tenir compte de la possibilité pour une transition nommée d'être verrouillée, et de la possibilité d'obliger une combinaison booléenne de contraintes hautes grâce à la directive *Cause*.

Pour le verrouillage, il suffit de rajouter aux prédicats déjà définis (temps, agents, alarmes) un prédicat unaire de type booléen par transition nommée.

Pour la directive *Cause*, le seul cas nouveau est celui où l'on se trouve en présence d'une disjonction entre couples (signal,temps). Sinon on commence par mettre notre combinaison booléenne en forme normale disjonctive (conjonction de disjonctions), puis on traite indépendamment chacune de ces disjonctions de la manière que nous allons présenter.

Nous nous limitons, pour expliquer la traduction, à une disjonction entre deux couples (signal,temps) $(Cause((s_1, t_1)\, or\, (s_2, t_2)))$, la généralisation à un nombre quelconque de couples étant immédiate. On suppose aussi que s_1 et s_2 sont du type $Trans(\alpha)$ pour simplifier.

On considère donc l'action

$$s_0 \longrightarrow Cause((Trans(\alpha_1), t_1)\, or\, (Trans(\alpha_2), t_2))$$

On commence par ajouter, en tant qu'alarmes à allumer, les alarmes U_{α_1} et U_{α_2} à toutes les transitions qui déclenchent le signal s_0.

Et on juxtapose aux transitions α_1 et α_2 les règles

$$U_{\alpha_1}(v_1) \otimes U_{\alpha_2}(v_2) \otimes (v_1 < +\infty) \multimap U_{\alpha_1}(+\infty) \otimes U_{\alpha_2}(+\infty)$$

$$U_{\alpha_1}(v_1) \otimes U_{\alpha_2}(v_2) \otimes (v_2 < +\infty) \multimap U_{\alpha_1}(+\infty) \otimes U_{\alpha_2}(+\infty)$$

En effet, si l'une des alarmes est éteinte, il faut aussi éteindre l'autre pour respecter la signification de la disjonction.

Conclusion

Traditionnellement les systèmes temps réel sont spécifiés à l'aide d'automates temporisés introduits par Alur et Dill et leurs propriétés sont exprimées par des formules de logique temporisée. La satisfaction d'une formule par un modèle est alors décidable en PSPACE par des techniques d'automate.

L'objectif de ce DEA a été d'explorer une nouvelle approche pour spécifier et vérifier les systèmes temps réel basée sur l'utilisation naturelle de la logique linéaire pour définir à la fois le système et ses propriétés, le problème de vérification se ramène alors à celui de la prouvabilité de la propriété dans le formalisme de la logique linéaire à partir d'un ensemble d'axiomes représentant le système.

L'un des avantages de cette approche par rapport à celle existante est de permettre de modéliser des systèmes qui ne soient plus statiques mais évoluent dans le temps de facon dynamique. On a par exemple la possibilité d'accroître le nombre d'agents ou de changer les contraintes de temps en cours d'exécution.

D'autre part ce nouveau formalisme permet d'offrir une interface de spécification de haut-niveau grâce au langage naturel qui masque l'utilisation sous-jacente des alarmes et de la logique linéaire.

Enfin il s'accompagne d'une nouvelle méthode interactive de spécification de systèmes temps-réel qui combine la vérification automatique de certaines propriétés avec la preuve manuelle, mais formelle, de propriétés plus générales avec un assistant de preuves.

L'originalité de ce travail est d'avoir donné un début d'implémentation à des résultats très récents restés à ce jour théoriques, en écrivant un programme ocaml de résolution d'un problème de sûreté et en établissant une preuve formelle d'une propriété plus générale en utilisant l'assistant de preuve Coq. De plus l'auteur a proposé une extension du langage naturel en regroupant des idées développées par Okada lors de ses conférences au LRI. Ces derniers points constituant la partie originale du travail, ils ont été largement commentés dans ce rapport.

Certes, le travail réalisé n'est qu'une première étape dans l'implémentation de ce formalisme, qui n'exploite pas encore toute la richesse de la correspondance entre les évolutions d'un système et la logique linéaire. Pour autant, cette démarche implacable de concrétisation qu'est la programmation a permis de bien isoler les concepts fondamentaux et de mieux comprendre certaines subtilités. En outre, le prototype obtenu permet déjà de se faire une bonne idée de ce à quoi pourraît ressembler les implémentations futures.

Pour finir nous allons présenter quelques prolongements intéressants qu'on pourrait

donner aux travaux réalisés pendant ce stage.

Tout d'abord, on peut envisager d'implémenter d'autres problèmes décidables tels que ceux qui n'ont été que brièvement présentés dans le chapitre 4, c'est-à-dire le problème général d'accessibilité, d'ordonnancement, d'appartenance, de rendez-vous ou de vivavité.

Ensuite on pourrait continuer d'enrichir le langage naturel, sur le modèle de ce qui a été fait dans le chapitre 8, de manière à le rendre plus complet et plus expressif.

On peut aussi remarquer que, bien que fondamentalement différente, cette approche déductive semble avoir quelques liens avec celle basée sur les automates temporisés, comme par exemple une complexité commune de certains problèmes en PSPACE. L'enjeu serait alors d'explorer plus finement cette relation pour voir s'il ne serait pas possible de définir une nouvelle notion d'automates temporisés qui bénéficient des apports de notre approche en terme de systèmes évolutifs et dynamiques.

Enfin lorsqu'on évoque les traits dynamiques de ce nouveau formalisme basé sur la logique linéaire, on est tenté d'y voir un terrain fertile pour la modélisation du code mobile. Encore une fois, la pertinence de cette conjecture mérite d'être approfondie.

Les expériences réalisées montrent que la taille des sous-graphes parcourus lors d'une preuve effective de sûreté est souvent faible, et peu en rapport avec la taille calculée de ces graphes qui est elle extrêmement sensible aux valeurs effectives des alarmes. Cela suggèrent que des systèmes qui ne diffèrent que par des valeurs d'alarmes différentes puissent néanmoins avoir des comportements identiques. Mettre à jour une équivalence de comportement expliquant ce phénomène est un autre problème intéressant dont la résolution pourraît drastiquement réduire la taille théorique des graphes manipulés.

Bibliographie

[AD94] R.Alur, D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183-235, 1994.

[GIR87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50 :1-102, 1987.

[KOS97] M. Kanovich, M. Okada, A. Scedrov. Specifying Real–Time Finite State Systems in Linear Logic, 1997.

[OKA00] M. Okada. Conférences au LRI sur la vérification de systèmes temps-réel avec la logique linéaire, Université Paris-Sud, 2000.

Première partie

Annexes

Annexe A

Exemple d'exécution du programme

A.1 Spécification du système du train

```
System = {
  Train := far | approaching | crossing;
  Gate := down | mv_up | up | mv_down
}

Constraints = {
c : interval (<Train=approaching> -- <Train=crossing>) > 10
}

Actions = {

a1: from <Train=far> to
    ->  Train := approaching
    cause g1 within strict 10/3

g1: from any to
    -> Gate := mv_down
    cause g2 within 7

g2: from <Gate=mv_down> to
    -> Gate := down

a2: from <Train=approaching> to
    -> Train := crossing

a3: from <Train=crossing> to
    -> Train := far
    -> Gate := mv_up
    cause g3 within strict 2

g3: from <Gate=mv_up> to
    -> Gate := up
}

Initial_event = <Train=far,Gate=up>

Initial_time = 0

Final_events = {
```

```
        <Train=crossing,Gate=up>;
        <Train=crossing,Gate=mv_up>;
        <Train=crossing,Gate=mv_down>
}
```

A.2 Résultats sur la sortie standard

```
26/7/2000   15:31   exemples/le_train.spec

FORMULE DE DEPART
-----------------

Time(0) . Train(far) . Gate(up) . Lc(+oo) . Ug1(+oo) . Ug2(+oo) . Ug3(+oo)

SOUS-FORMULES D'ARRIVEE
-----------------------

Train(crossing) . Gate(up)
Train(crossing) . Gate(mv_up)
Train(crossing) . Gate(mv_down)

FORMULE DE PROGRESSION DU TEMPS
-------------------------------

Time(s) . (s < t) . Ug1(u_{g1}) . Ug2(u_{g2}) . Ug3(u_{g3}) .
          (t < u_{g1}) . (t <= u_{g2}) . (t < u_{g3})
-o
Time(t) . Ug1(u_{g1}) . Ug2(u_{g2}) . Ug3(u_{g3})

FORMULES DE TRANSITION
----------------------

[0]
Train(far) . Time(t) . Ug1(u_{g1}) . Lc(l_{c})
-o
Train(approaching) . Time(t) . Ug1(u_{g1} | t + 10/3) . Lc(t + 10)

[1]
Gate(var_Gate) . Time(t) . Ug2(u_{g2}) . Ug1(u_{g1}) . (u_{g1} < +oo)
-o
Gate(mv_down) . Time(t) . Ug2(u_{g2} | t + 7) . Ug1(+oo)

[2]
Gate(mv_down) . Ug2(u_{g2}) . (u_{g2} < +oo)
-o
Gate(down) . Ug2(+oo)

[3]
Train(approaching) . Time(t) . Lc(l_{c}) . (l_{c} < t)
-o
Train(crossing) . Time(t) . Lc(+oo)

[4]
Train(crossing) . Gate(var_Gate) . Time(t) . Ug3(u_{g3})
-o
```

```
Train(far) . Gate(mv_up) . Time(t) . Ug3(u_{g3} | t + 2)

[5]
Gate(mv_up) . Ug3(u_{g3}) . (u_{g3} < +oo)
-o
Gate(up) . Ug3(+oo)

RESULTAT DE LA PROCEDURE DE DECISION
------------------------------------

Nombre de sommets potentiels : 256681656

Nombre de sommets visites : 1368

Le systeme n'est PAS SUR :

0) {Depart} [0] : Train(far) . Gate(up)
1) {Transition} [0] : Train(approaching) . Gate(up)
2) {Temps} [19/6] : Train(approaching) . Gate(up)
3) {Transition} [19/6] : Train(approaching) . Gate(mv_down)
4) {Temps} [121/12] : Train(approaching) . Gate(mv_down)
5) {Transition} [121/12] : Train(crossing) . Gate(mv_down)
```

A.3 Fichier Coq généré

```
Load rationnel.

Inductive Train : Set :=
| far : Train
| approaching : Train
| crossing : Train.

Inductive Gate : Set :=
| down : Gate
| mv_up : Gate
| up : Gate
| mv_down : Gate.

Inductive configuration : Set :=
  config : r -> Train -> Gate ->  r_infini ->  r_infini ->  r_infini ->
             r_infini -> configuration.

Definition init := (config (Rat (0) (1) (lt_O_Sn (0))) far up Infini Infini Infini Infini).

Inductive transition [B,H0,H1,H2 : r] : configuration -> configuration -> Prop :=
| tick : (t, s : r) (v_Train : Train) (v_Gate : Gate)
         (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (lt_rat s t) -> (le_ri (Fini t) v_U_g1) ->
 (le_ri (Fini t) v_U_g2) -> (lt_ri (Fini t) v_U_g3) ->
 (transition B H0 H1 H2 (config s (v_Train) (v_Gate) v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t (v_Train) (v_Gate) v_U_g1 v_U_g2 v_U_g3 v_L_c))
| a1 : (t : r)  (v_Gate : Gate) (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (transition B H0 H1 H2 (config t far v_Gate v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t approaching v_Gate (v_U_g1 || (t ++ H0)) v_U_g2 v_U_g3 (Fini (t ++ B))))
| g1 : (t : r)  (v_Train : Train) (v_Gate : Gate) (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
```

```
 (lt_ri v_U_g1 Infini) -> (transition B H0 H1 H2
 (config t v_Train v_Gate v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t v_Train mv_down Infini (v_U_g2 || (t ++ H1)) v_U_g3 v_L_c))
| g2 : (t : r)  (v_Train : Train) (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (lt_ri v_U_g2 Infini) -> (transition B H0 H1 H2
 (config t v_Train mv_down v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t v_Train down v_U_g1 Infini v_U_g3 v_L_c))
| a2 : (t : r)  (v_Gate : Gate) (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (lt_ri (Fini t) v_L_c) -> (transition B H0 H1 H2
 (config t approaching v_Gate v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t crossing v_Gate v_U_g1 v_U_g2 v_U_g3 Infini))
| a3 : (t : r)  (v_Gate : Gate) (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (transition B H0 H1 H2 (config t crossing v_Gate v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t far mv_up v_U_g1 v_U_g2 (v_U_g3 || (t ++ H2)) v_L_c))
| g3 : (t : r)  (v_Train : Train) (v_U_g1, v_U_g2, v_U_g3, v_L_c : r_infini)
 (lt_ri v_U_g3 Infini) -> (transition B H0 H1 H2
 (config t v_Train mv_up v_U_g1 v_U_g2 v_U_g3 v_L_c)
 (config t v_Train up v_U_g1 v_U_g2 Infini v_L_c)).

Inductive chemin [B,H0,H1,H2 : r] : configuration -> configuration -> Prop :=
  refl_chemin : (a : configuration) (chemin  B H0 H1 H2 a a)
| trans_chemin : (a,b,c:configuration) (transition B H0 H1 H2 b c) ->
  (chemin  B H0 H1 H2 a b) -> (chemin  B H0 H1 H2 a c).

Lemma cas_non_sur :
(B,H0,H1,H2 : r) (lt_rat B (H0 ++ H1)) ->
(EX t : r |
(EX v_U_g1 : r_infini |
(EX v_U_g2 : r_infini |
(EX v_U_g3 : r_infini |
(EX v_L_c : r_infini |

(chemin B H0 H1 H2 init (config t crossing mv_down v_U_g1 v_U_g2 v_U_g3 v_L_c))

))))).

Intros.
```