Méthode B et temps réel : étude de l'intégration du calcul des durées

Samuel Colin

14 décembre 2006

Table des matières

1 Introduction **4**

2 Calcul des durées **5**
2.1 Calcul des durées 5
2.1.1 Calcul des durées "pur" 5
2.1.2 Calcul des durées avec itération 10
2.1.3 Support de preuve 11

3 B **13**
3.1 B 13
3.1.1 Présentation rapide 13
3.1.2 Outillage 15

4 B & temps réel **16**
4.1 B temps-réel 16
4.1.1 Expression de spécifications temporelles en B 16
4.1.2 Substitutions généralisées et calcul de formules 17
4.1.3 Obligations de preuves supplémentaires 20
4.2 Perspectives 23
4.3 Conclusion 23

Abstract

Ce document traite principalement de l'intégration du formalisme logique du *"calcul des durées"*(DC) dans un cycle de développement en B.

Dans la première section, nous présenterons succinctement le formalisme de DC, puis ses éventuels supports d'aide à la preuve. Puis après avoir présenté rapidement B, nous verrons comment, par l'adjonction du calcul des durées, obtenir une méthode formelle étendue permettant de prendre en compte aisément et efficacement les exigences relevant du temps réel.

Chapitre 1

Introduction

Les méthodes formelles, en particulier la méthode B, ont désormais acquis une solide réputation au sein des entreprises soucieuses d'obtenir un code certifié correct et conforme aux spécifications. Le domaine du logiciel embarqué fait grandement appel à ce genre de méthodes, et les a déjà utilisées avec succès pour nombre de projets ([Sabatier et al.00, Behm et al.99] par exemple).

Mais parfois la nature intrinsèque de ces méthodes n'est pas suffisamment expressive pour permettre de spécifier certains aspects rencontrés. En effet, dans beaucoup de projets (sinon tous) intégrant du logiciel embarqué, le principal écueil de ces méthodes est de ne pas pouvoir traiter aisément de problèmes temporels, i.e. des problèmes pouvant aller du simple ordonnancement de tâches (problème pour lequel une logique temporelle simple peut suffire) à des problèmes faisant intervenir des fenêtres de temps précises et limitées (dans ce cas une logique d'intervalle peut faire l'affaire). Et cela peut même aller plus loin lorsque s'y ajoutent des contraintes de durée sur des événements, auquel cas faire appel au calcul des durées (DC) s'avère utile. Ainsi il est intéressant de se demander si le calcul des durées peut être ajouté facilement à une méthode formelle (ici B), plutôt que de devoir recréer une méthode formelle pour ce seul formalisme.

Chapitre 2

Calcul des durées

Contents

2.1 Calcul des durées 5
2.1.1 Calcul des durées "pur" 5
2.1.2 Calcul des durées avec itération 10
2.1.3 Support de preuve 11

2.1 Calcul des durées

2.1.1 Calcul des durées "pur"

Historique

La recherche sur une logique temporelle plus puissante que la logique temporelle d'intervalle "classique" (cf [Dutertre95]) a été initiée par le projet ProCos[1] du programme ESPRIT[2], dans les groupes de travail BRA[3] 3104 et 7071.

Cette initiative a abouti en 1990 à l'article «A calculus of durations» ([Zhou et al.91]), qui établissait les bases du calcul des durées. Ont suivi des études plus poussées sur le cadre d'utilisation de DC, traitant par exemple des problèmes de complétude ou de décidabilité (voir par ex. [Hansen et al.97]), puis des déclinaisons de DC, comme DC avec des intervalles de temps infinis [Zhou et al.95], ou DC d'ordre supérieur [Zhou et al.99], pour ne citer qu'eux (voir par ex [Dca]).

Il existe également des exemples d'utilisation du calcul des durées ([Naijun99]), de développement de logiciels temps-réel, ainsi que des systèmes d'aide à la preuve pour DC ([Heilmann8a, Heilmann99]). Aujourd'hui, l'institution la plus active dans ce domaine est sans doute l'IIST[4]. Son site [Dcb] regroupe de nombreux

[1] Provably correct systems
[2] European Strategic Program for Research in Information Technology
[3] Basic Research Action
[4] International Institute for Software Technology, affilié à l'Université des Nations Unies

liens en rapport avec DC.

Modélisation de problèmes temps-réel

[Siewe et al.01] présente un exemple de mise en oeuvre d'un problème temps-réel. Bien que le cas traité soit assez particulier, il recouvre parfaitement toutes les étapes de n'importe quel autre problème temps-réel. La mise en oeuvre se décompose comme suit :

- Les variables du problème sont définies
- Les spécifications du problème sont mises sous forme d'une formule de durée *Req*
- Des décisions de conception sont prises et mises aussi sous la forme d'une formule de DC *Des*, de telle façon que $Des \Rightarrow Req$
- La conception ayant eu lieu dans un univers continu, une étape de discrétisation peut être requise, auquel cas il faut trouver une formule de DC *Cont* telle que $\mathcal{A} \vdash Cont \Rightarrow Des$, $\mathcal{A}$ étant des hypothèses relatives au comportement de l'environnement et aux relations entre les variables continues et discrètes
- Enfin il faut écrire un programme qui vérifie les contraintes discrètes indiquées dans *Cont*.

Ici l'étape de programmation est la dernière, et le langage utilisé dans [Siewe et al.01] est simple.

L'idée ici est donc d'exploiter le fait que l'étape de programmation se passe après l'étape de spécification en B, la programmation étant conjointe à la preuve, pour obtenir une simplification de toutes ces étapes. Avant de présenter B, voici donc une présentation de DC (incomplète pour des raisons de place) illustrée par quelques exemples.

Identificateurs utilisés

- Lettres propositionnelles temporelles : $X, X_1, X_2, \ldots$ interprétées comme des fonctions des intervalles vers les booléens
- Variables d'état : $P, P_1, P_2, \ldots$ interprétées comme des fonctions du temps vers les booléens
- Variables globales : $x, y, z, \ldots$ interprétées comme des constantes réelles (sauf si elles sont liées par un quantificateur existentiel ou universel, auquel cas ce sont des variables réelles)
- Symboles de fonctions : $f, f_1, \ldots$ Leur signification est standard (et usuellement , ne sont utilisés que les opérateurs arithmétiques de base $+, -, etc$)
- Symboles de relations : $R, R_1, \ldots$ Là encore, leur signification est standard, et sont surtout utilisées les relations d'ordonnancement $<, >, =, \ldots$

Syntaxe abstraite

États $S ::= \quad 0 \quad | \quad 1 \quad | \quad P \quad | \quad S_1 \wedge S_2 \quad | \quad \neg S$

Les états sont interprétés comme des fonctions du temps vers les booléens. Ils permettent de décrire des relations entre les états et les événements d'un système. Les constantes 0 et 1 peuvent être vues comme les constantes booléennes *false* et *true* respectivement. Mais comme les états, lorsque l'on passe au niveau des termes, sont liés par une intégrale, il est plus naturel ici d'utiliser des constantes numériques.

Par exemple, $Fuite(t) \widehat{=} Gaz(t) \wedge \neg Feu(t)$, signifie que l'événement Fuite (de gaz) est vrai lorsque l'événement Gaz est vrai et l'événement Feu n'est pas vrai.

Termes $\theta ::= \quad x \quad | \quad \ell \quad | \quad \int S \quad | \quad f(\theta_1, \ldots, \theta_n)$

Les termes sont interprétés comme des fonctions des intervalles vers les réels. Ils permettent d'exprimer les durées des événements.

Par exemple, $\int Gaz <= 10$ signifie que l'événement Gaz ne peut être vrai qu'au plus 10 unités de temps.

Formules $Atom ::= \quad \mathbf{true} \quad | \quad X \quad | \quad R(\theta_1, \ldots, \theta_n)$

$\phi ::= \quad Atom \quad | \quad \neg\phi \quad | \quad \phi \wedge \psi \quad | \quad (\phi ^\frown \psi) \quad | \quad \exists x . \phi$

Les connecteurs logiques au niveau des états et des formules n'entrent jamais en conflit (comme le montre la syntaxe). Si les premiers n'ont avant tout qu'un but descriptif, les seconds permettent eux de raisonner véritablement sur les formules (avec le système de déduction adéquat). Ces formules possèdent au moins la puissance de la logique classique. Le connecteur supplémentaire $^\frown$, appelé *chop* car son interprétation "coupe" un intervalle de temps en deux, donne la possibilité d'exprimer les variations des états au cours du temps.

Par exemple, $(\ell = 10)^\frown(\ell = 5)$ indique un intervalle de 10 unités de temps suivi par un intervalle de 5 unités de temps.

Notations supplémentaires

$$\begin{array}{lcl} \Diamond\phi & \widehat{=} & \mathbf{true}^\frown(\phi^\frown\mathbf{true}) \\ \Box\phi & \widehat{=} & \neg\Diamond(\neg\phi) \end{array}$$

Ces deux connecteurs sont à rapprocher des logiques temporelles classiques. Si usuellement $\Diamond$ signifie "à un moment" en LTL[5], ici il signifie "dans un sous-intervalle de l'intervalle considéré". De la même façon, $\Box$ signifie classiquement "à tout moment la formule est vraie", en DC il signifie "pour tout sous-intervalle considéré".

$$\begin{array}{lcl} \lceil\rceil & \widehat{=} & \ell = 0 \\ \lceil S \rceil & \widehat{=} & \int S = \ell \wedge \ell > 0 \end{array}$$

[5] Linear Temporal Logic, une logique temporelle simple. Voir par ex. [My96]

Des raccourcis syntaxiques, ces deux formules étant souvent utilisées. La première formule a surtout l'avantage d'être aisément identifiable dans une formule complexe. La seconde formule indique que l'on est en train de considérer la durée de l'événement S, celle-ci n'étant pas nulle.

Par exemple :

$$\lceil\lceil \mathrm{Fumée} \rceil\rceil \Rightarrow \lceil\lceil \mathrm{Feu} \rceil\rceil \text{ et } \Box(\lceil\lceil \mathrm{Fumée} \rceil\rceil \Rightarrow \lceil\lceil \mathrm{Feu} \rceil\rceil)$$

La première formule indique que sur tout l'intervalle considéré, si l'événement *Fumée* est présent, alors l'événement *Feu* l'est aussi, et les deux événements durent toute l'intervalle de temps dans ce cas.

La seconde formule est un peu plus subtile : dans l'intervalle de temps considéré, pour chaque sous-intervalle où il y a *Fumée*, alors dans ce même sous-intervalle, *Feu* est présent et ces deux événements durent autant.

Quelques indications supplémentaires (voir [Hansen et al.97] pour plus de détails) :

- Une formule *rigide* ne change pas au cours du temps. Sinon elle est dite *flexible*
- Une formule *chop free* ne contient pas de $\frown$

Système de preuve

Axiomes DC étant basé sur la logique temporelle d'intervalle [Dutertre95] , celle-ci se voit augmentée des axiomes de durée :

DCA1 : $\int 0 = 0$
La durée d'un événement ne survenant jamais est nulle

DCA2 : $\int 1 = \ell$
Un événement arrivant à chaque instant a la durée de l'intervalle de temps

DCA3 : $\int S \geq 0$
La durée d'un événement ne peut pas être négative

DCA4 : $\int S_1 + \int S_2 = \int(S_1 \wedge S_2) + \int(S_1 \vee S_2)$
Un axiome formel, qui permet de calculer la durée de l'union ("$\vee$") ou de l'intersection ("$\wedge$") de deux événements

DCA5 : $((\int S = x)\frown(\int S = y)) \Rightarrow (\int S = x + y)$
La durée de deux événements semblables consécutifs est la somme des durées de ces événements

DCA6 : $\int S_1 = \int S_2$ si $S_1 \Leftrightarrow S_2$
Deux événements ont la même durée s'ils sont "indiscernables" d'un point de vue logique

Règles de déduction

Modus ponens (MP) :	si ϕ et $\phi \Rightarrow \psi$ alors ψ
Generalization (G) :	si ϕ alors $\forall x.\phi$(généralisation)
Quantification (Q) :	$\forall x.\phi(x) \Rightarrow \phi(\theta)$ si θ est libre pour x dans $\phi(x)$ et $\begin{cases} \text{soit } \theta \text{ est rigide} \\ \text{soit } \phi(x) \text{ est chop free} \end{cases}$
Necessitation (N) :	si ϕ alors $\neg(\neg\phi \frown \psi)$
	si ϕ alors $\neg(\psi \frown \neg\phi)$
Monotony (M) :	si $\phi \Rightarrow \psi$ alors $\phi \frown \varphi \Rightarrow \psi \frown \varphi$
	si $\phi \Rightarrow \psi$ alors $\varphi \frown \phi \Rightarrow \varphi \frown \psi$

Les trois premières règles se retrouvent dans tous les systèmes de règles pour le calcul des prédicats. La règle N indique que si une formule est vraie, alors sa négation est fausse pour un préfixe (respectivement un postfixe) quelconque de l'intervalle considéré. La règle M indique que si une formule est conséquence d'une autre autre, alors l'implication est conservée même si l'une et l'autre sont suivies (respectivement précédées) par une autre formule sur l'intervalle considéré.

Soit $H(X)$ une formule contenant la lettre propositionnelle temporelle X, et $S_1, \ldots, S_n$ des expressions d'états, avec $S_1 \vee \ldots \vee S_n = 1$.

IR1 :	si $H(\lceil\rceil)$ et $H(X) \vdash H(X \vee (X \frown \lceil S_1 \rceil) \vee \ldots \vee (X \frown \lceil S_n \rceil))$ alors $H(\mathbf{true})$
IR2 :	si $H(\lceil\rceil)$ et $H(X) \vdash H(X \vee (\lceil S_1 \rceil \frown X) \vee \ldots \vee (\lceil S_n \rceil \frown X))$ alors $H(\mathbf{true})$

IR1 et IR2 sont des règles de récurrence utilisant le connecteur $\frown$.

Calculs et hypothèses supplémentaires À ce système de déduction il faut rajouter le calcul des prédicats du premier ordre (pour pouvoir utiliser les connecteurs des formules, cela va de soi), ainsi que les axiomes pour les nombres réels et les opérations associées (addition, multiplication,etc). On y rajoute également tous les axiomes pertinents concernant les symboles de fonctions et de relations introduits.

Ensuite, est faite une hypothèse forte (mais fort peu handicapante en pratique) sur les fonctions : elles doivent être finiment variables. Exemple d'une fonction non finiment variable :

$$f(t) = \begin{cases} 1 \text{ si t est rationnel} \\ 0 \text{ si t est irrationnel} \end{cases}$$

Toute fonction définie dans DC doit donc varier un nombre fini de fois dans un intervalle. Cette hypothèse est nécessaire pour démontrer certains résultats de correction et de complétude, et n'est pas gênante, ce comportement n'apparaissant jamais en pratique pour les systèmes modélisés.

Une dernière remarque concernera le domaine d'interprétation : nous avons considéré ici un domaine d'interprétation continu ($\mathbb{R}$), mais il est possible d'utiliser un domaine d'interprétation discret (comme $\mathbb{N}$). Dans ce cas, certaines formules ne sont plus valides, alors que d'autres le deviennent. D'après [Hansen et al.97], il est difficile de caractériser la façon dont les formules sont valides ou non selon qu'on se trouve dans un domaine temporel discret ou continu. Cela s'explique par les différences de propriétés arithmétiques qu'il peut y avoir entre un domaine continu ($\mathbb{R}$) et un domaine discret ($\mathbb{N}$).

De plus, si l'on a choisi un domaine continu, il est possible après coup de faire une hypothèse "semi-discrète", par exemple en introduisant en hypothèse une formule de durées indiquant que les variations d'états prennent un temps minimum.

2.1.2 Calcul des durées avec itération

Ajouts à DC

Le calcul des durées simple est souvent présenté comme une façon de spécifier des problèmes simples, mais lorsqu'on considère une hybridation avec un langage de programmation, on utilise plus volontiers une extension de DC, DC*, i.e. le calcul des durées avec itération.

Le connecteur d'itération sert à indiquer qu'une formule se répète un nombre arbitraire de fois. Dans un langage de programmation, cela correspond à une boucle "while" gardée par une condition non numérique. Sémantiquement, ce connecteur * est à rapprocher du $^\frown$. La différence est qu'au lieu de couper un intervalle en deux, il coupe l'intervalle en un nombre arbitraire (mais fini) de sous-intervalles, sur chacun desquels la formule ϕ doit être vérifiée, si la formule était ϕ^*. Peu d'ajouts à DC sont faits :

- Un connecteur logique *
- Trois axiomes :
  - DC_1^* $\ell = 0 \Rightarrow \phi^*$
    Même si l'intervalle est de longueur nulle, ϕ^* est valide, le connecteur * considérant un nombre arbitraire de sous-intervalles, donc même l'absence de sous-intervalle
  - DC_2^* $(\phi^{*\frown}\phi) \Rightarrow \phi^*$
    Le connecteur * est absorbant
  - DC_3^* $((\phi^* \wedge \psi)^\frown \mathbf{true}) \Rightarrow$
    $((\psi \wedge \ell = 0)^\frown \mathbf{true}) \vee ((((\phi^* \wedge \neg\psi)^\frown \phi) \wedge \psi)^\frown \mathbf{true})$
    Un axiome formel
- Une règle de déduction :
  ω : si $\forall k < \omega, H((\lceil\lceil S \rceil\rceil \vee \lceil\lceil \neg S \rceil\rceil)^k)$ alors $H(\mathbf{true})$

Ainsi que quelques notations (usuelles) supplémentaires :

$$\phi^+ \;\widehat{=}\; \phi^\frown(\phi^*)$$
$$\phi^0 \;\widehat{=}\; \ell = 0$$
$$\phi^k \;\widehat{=}\; \underbrace{\phi^\frown \ldots ^\frown\phi}_{k \text{ fois}} \text{ pour } k > 0$$

Sous-ensemble décidable

Il a été démontré dans [Guelev et al.99] qu'un sous-ensemble de DC* était décidable :

$$\phi ::= \ \ell = 0 \mid \lceil\lceil S \rceil\rceil \mid a \leq \ell \mid \ell \leq a \mid (\phi \vee \phi) \mid (\phi \wedge \phi) \mid (\phi ^\frown \phi) \mid \phi^*$$

Les formules de ce sous-ensemble sont appelées *simples*. Nous appellerons dans la suite ce sous-ensemble SDC* Ce sous-ensemble est intéressant, puisque la procédure de décision est simple à écrire (un exemple en est donné dans [Guelev et al.99] pour une sous-classe des formules simples). De plus, de la même façon que dans [Siewe et al.01], nous allons faire en sorte que les formules temporelles correspondant aux programmes fassent partie de SDC*, ce qui garantira la prouvabilité des formules obtenues.

2.1.3 Support de preuve

Les outils présentés sont tous référencés sur [Dcb], pour les lecteurs intéressés. Avant de décrire l'outillage plus avant, il est intéressant de noter qu'il y a deux manières d'ajouter le calcul des durées à un outil d'aide à la preuve.

La première façon est de plonger le formalisme de DC dans le formalisme de l'outil, en ajoutant des axiomes et les règles d'inférence comme tactiques de base (comme avec le système de preuve indiqué dans la section 2.1.1), ce qui revient à faire un plongement léger.

L'autre façon est de décrire DC selon sa sémantique, c'est-à-dire la façon dont les formules de DC peuvent être interprétées (plongement profond).

En dehors des deux systèmes présentés succinctement ici, il existe d'autres systèmes de preuve pour DC, comme par exemple [Xia98] et [Xiaoguang et al.96].

PVS

L'un des premiers outils de preuve à proposer une librairie pour le calcul des durées fut PVS[6], avec une version nommée *PC/DC*. Ici la méthode utilisée est le plongement profond. L'avantage de cette méthode est que la correction des règles de DC peut être prouvée grâce à la logique de base.

L'inconvénient majeur est qu'on ne peut pas faire la preuve "syntaxiquement", simplement en utilisant les règles d'inférence de DC : il faut passer par une fonction de passage de la sémantique des formules à leur syntaxe, et vice-versa. Voir [Heilmann8a] pour plus de détails.

Isabelle

Ici la démarche est différente de celle de PVS. Isabelle étant un système métalogique permettant de décrire de nombreuses logiques "objet", il faut lui adjoindre un ou des modules, pour obtenir un système d'aide à la preuve dirigé par la syntaxe.

[6]Prototype Verification System

Ainsi, pour pouvoir prouver des formules du calcul des prédicats, il faut utiliser un module décrivant les règles de ce calcul. De la même façon, pour prouver des formules de DC, il a fallu écrire un module basé sur le système de déduction pour DC existant.

L'avantage d'un tel système est évidemment de pouvoir faire des preuves de formules de DC aisément, puisqu'il s'agit d'un plongement léger[7]

L'inconvénient en revanche, n'est pas à l'utilisation, mais à la conception. En effet il faut en plus réécrire tout le système pour les calculs que DC utilise : le calcul des prédicats, le calcul sur les réels, ainsi que les définitions et propriétés des fonctions de calcul et de comparaison que l'on pourrait souhaiter utiliser.

Voir [Heilmann99] pour une vue approfondie de la conception et de l'utilisation de ce système.

Coq

Nous avons également développé des bibliothèques décrivant le calcul des durées pour PhoX, déjà décrites dans [Colin et al.03].

Ces bibliothèques sont encore à l'état de prototype, et illustrent les deux approches citées ci-dessus, en ce sens qu'une bibliothèque est implémentée sous forme de plongement léger, et l'autre sous forme de plongement profond. Cela nous permet à la fois de résoudre des formules du calcul des durées (avec le plongement léger), et de raisonner sur le formalisme lui-même (plongement profond).

[7] Due à la particularité d'Isabelle/HOL, qui est un méta-moteur, on désigne aussi ces plongements par le qualificatif *externes*

Chapitre 3

B

Contents

3.1 B **13**
3.1.1 Présentation rapide 13
3.1.2 Outillage 15

3.1 B

3.1.1 Présentation rapide

Pour des raisons de place, nous supposerons que le lecteur est familier des termes et notions abordées ici (voir [Abrial96] pour une présentation complète de B).

Processus global

La méthode B est une méthode formelle, qui permet, à partir de spécifications exprimées en langage naturel, de générer un code informatique exécutable, prouvé correct et sûr, la partie qui nous intéresse allant des spécifications B jusqu'à la génération des OPs[1].

Une machine abstraite B est composée de clauses de visibilité (inclusion des opérations d'autres machines, simple visibilité, importation d'une machine complète, ...), de clauses définissant la nature des variables utilisées, les invariants devant être respectés par la machine (la clause *INVARIANT* est la plus utilisée lors de la génération des OPs), les valeurs d'initialisation des variables de la machine. Elle est enfin composée d'opérations, qui définissent les possibilités d'évolution dynamiques de la machine.

Une fois la machine abstraite écrite, il est possible de la raffiner. Le *raffinement* d'un composant est l'étape qui permet de le rendre moins abstrait, en remplaçant

[1] Obligations de preuve, formules permettant de déterminer si une machine B est correcte

ce qui était un ensemble par un tableau, par exemple. Il est alors possible de raffiner à nouveau ce raffinement, jusqu'à l'*implémentation*, composant qui doit répondre à des contraintes précises (pas d'opérations non-déterministes,...)

L'étape suivante est la génération des OPs (bien que dans la pratique cette étape et celle de développement des composants soient entremêlés), grâce auxquelles la correction du projet tout entier pourra être prouvée. L'impossibilité à prouver l'une de ces OPs est le signe d'une erreur lors des spécifications ou de la mise en oeuvre dans les opérations.

GSL

Les GSL[2] sont les opérations de base permettant de décrire l'évolution d'une machine B. Toutes les autres substitutions indiquées dans [Abrial96] ne sont que du sucre syntaxique défini à partir des premières. C'est donc aux GSL que nous ajouterons les nouvelles opérations faisant intervenir le temps.

Elles sont utilisées pour calculer les OPs, en étant appliquées à un prédicat (souvent le prédicat de la clause d'invariant de la machine abstraite, raison pour laquelle nous parlerons préférablement d'invariant par la suite, en désignant ce prédicat) selon un procédé de calcul décrit plus loin. Rappelons que la sémantique des programmes est basée sur les triplets de Hoare (précondition, programme, post-condition), et que l'application de la GSL à l'invariant revient à chercher la plus faible précondition qui permettra à l'invariant d'être encore vrai après "exécution" du programme.

En prenant des hypothèses pertinentes (invariant du programme, assertions supplémentaires,etc) et en prenant comme but le prédicat calculé, on obtient une OP qu'il suffira de vérifier pour être sûr que le programme conserve l'invariant. Là encore pour plus de détails, voir [Abrial96].

Obligations de preuve

Avec l'ajout de contraintes temporelles à B, il faut inévitablement songer à l'ajout d'obligations de preuve qui permettront d'indiquer si les machines écrites ont des spécifications temporelles cohérentes.

Ainsi les obligations de preuve concernant l'invariant ou les assertions d'une machine ne suffiront pas pour assurer que certaines conditions temporelles requises sont vérifiées, raison pour laquelle il faut rajouter une clause de spécifications temporelles de la machine. Nous appellerons cette nouvelle clause *TIMING* par la suite.

3.1.2 Outillage

Il existe des outils pour développer en B, permettant d'aller de la spécification à la preuve de correction des machines B écrites. Les outils les plus aboutis sont l'*Atelier B* et le *B-toolkit* (voir respectivement [Atelier b] et [B toolkit]).

[2] substitutions généralisées. GSL est l'acronyme anglais de Generalized Substitutions Language

Il existe également un outil *Open source* encore expérimental mais dont certains aspects sont déjà bien avancés, BCaml ([Bcaml]), projet développé en OCaml. Les expérimentations futures se feront sur cet outil puisque, pouvant accéder au code source, il est possible d'augmenter la syntaxe de B pour y ajouter les commandes ajoutant la gestion du temps aux substitutions généralisées, ainsi que de nouvelles clauses qui devront intervenir lors de la preuve de la machine abstraite.

Chapitre 4

B & temps réel

Contents

4.1 B temps-réel **16**
4.1.1 Expression de spécifications temporelles en B 16
4.1.2 Substitutions généralisées et calcul de formules 17
4.1.3 Obligations de preuves supplémentaires 20
4.2 Perspectives **23**
4.3 Conclusion **23**

4.1 B temps-réel

4.1.1 Expression de spécifications temporelles en B

Notre but ici est d'apporter un moyen puissant pour modéliser des problèmes faisant intervenir le temps, mais bien entendu il existe déjà des travaux traitant de ce problème. Par exemple, [Lano97] et [Treharne et al.99] montrent que l'on peut tenter d'exprimer des spécifications temporelles en AMN[1], avec quelques réserves cependant :

- Il faut définir soi-même les opérations d'une machine permettant de "manipuler" le temps, i.e. qui permettra d'augmenter le temps, d'obtenir sa valeur actuelle, etc. Si avec DC on a besoin d'une horloge, il est possible d'en modéliser une avec une telle machine, mais seule une opération donnant la valeur du temps sera nécessaire, les opérations de manipulation du temps étant rendues caduques par de nouvelles substitutions (voir figure 4.1)
- Devoir justement être obligé de manipuler explicitement le temps, plutôt que de le considérer comme une information à laquelle le problème est soumis (ce qui est le cas dans la réalité)

[1] Notation en machine abstraite. AMN est l'acronyme abglais de Abstract Machine Notation. L'AMN n'existe pas qu'en B (on note usuellement B-AMN), mais puisque nous nous focalisons sur B, nous conserverons la notation AMN

– On est souvent obligé de faire des hypothèses de bon déroulement des opérations, comme dans [Treharne et al.99] : il est possible d'exprimer quand une opération ne peut plus se dérouler, mais pas quand elle doit s'activer pour remplir toutes les conditions temporelles de sûreté
– Aussi explicites les noms de variables puissent-ils être, lors du traitement des obligations de preuve, en cas de problème il peut être ardu de comprendre ce que signifie la formule logique s'il y a beaucoup d'imbrications d'actions et d'intervalles temporels.

4.1.2 Substitutions généralisées et calcul de formules

Sans le temps

Nom	GSL	[GSL]P
skip	skip	P
affectation	$x := E$	$P[E/x]$
affectation multiple	$x, y := E, F$	$P[E/x, F/y]$
délai	delay d	P
attente	await g	$g \Rightarrow P$
précondition	$g \vert S$	$g \wedge [S]P$
garde	$g \Longrightarrow S$	$g \Rightarrow [S]P$
séquence	$S;T$	$[S]([T]P)$
choix borné	$S [] T$	$[S]P \wedge [T]P$
choix non borné	$@x.S$	$\forall x.[S]P$
parallèle	$S \vert\vert T$	transformé en affectation multiple par l'utilisation de règles de réécriture
while	WHILE C DO S VARIANT V INVARIANT I	$I \wedge C \Rightarrow [S]I$ $I \Rightarrow V \in \mathbb{N}$ $I \wedge C \Rightarrow [n := V][S](V < n)$ $I \wedge \neg C \Rightarrow P$

FIG. 4.1 – Tableau décrivant l'application d'une GSL à un prédicat

Le délai et l'attente ne se trouvent pas dans le B standard. Ils ont été rajoutés ici pour permettre de modéliser des problèmes temps-réel (figure 4.1).

Nous avons ajouté des règles d'application pour le délai et l'attente pour que des programmes B pour lesquels le temps est peu important puissent être tout de même analysés.

Donc dans ce cas, pas de surprise : le délai n'est pas pris en compte dans le calcul de la plus faible précondition, mais par contre l'attente devient équivalente à une garde, cela pour permettre de considérer une postcondition contenant éventuellement la garde ou des variables de la garde.

Autre problème, le $\|$: la transformation en affectation multiple indiquée ici suit les conventions indiquées dans [Abrial96], mais également les règles indiquées dans [Siewe et al.01], c'est-à-dire respecter le fait que la concurrence a une sémantique d'entrelacement, excepté autour des *delay* qui se chevauchent. Par exemple, :

$$x, y := a, b; \textbf{delay}\ 1 \| w := c; \textbf{delay}\ 2; z := d$$

se réécrira en

$$x, y, w := a, b, c; \textbf{delay}\ 2; z := d$$

Compatibilités logiques

Si la définition telle qu'écrite dans la figure 4.1 est facile à comprendre pour le *delay*, c'est moins trivial en ce qui concerne le *await*. Rappelons la définition sous forme de triplet de Hoare (étendu avec une formule de DC*), donnée dans [Siewe et al.01] :

$$\{post\}[\text{await } g, \lceil\!\lceil \neg g \wedge post \rceil\!\rceil^*]\{post \wedge g\}$$

Utilisé avec le système d'inférence présenté dans [Siewe et al.01], cette définition est pratique, mais avec un système devant faire des calculs automatiques, considérer la précondition en retirant simplement la garde du *await* de la postcondition n'est pas aussi simple. D'autant que cette définition est adaptée aux problèmes de [Siewe et al.01], mais pas à ceux de B tels qu'ils se présentent, puisqu'en B la postcondition, après quelques étapes de calcul, est déjà «noyée» sous plusieurs implications. Difficile dans ces conditions de valider la garde du *await* dans la précondition.

Ainsi, la définition donnée dans la figure 4.1 a plus d'avantages : l'implication permet de prendre en compte le cas où la garde n'est jamais vérifiée (cas de non-terminaison, hypothèse non considérée en B pour l'instant), et permet de résoudre notre problème, puisque lors de la preuve, la garde passera dans les hypothèses. Enfin, dernier argument qui indique que notre définition est plus expressive, est la règle de conséquence de [Siewe et al.01] :

$$\frac{\{pre\}[P, \phi]\{post\} \quad pre' \Rightarrow pre \quad post \Rightarrow post' \quad \phi \Rightarrow \psi}{\{pre'\}[P, \psi]\{post'\}}$$

Utilisé avec les différentes règles pour le *await*, cela donne :

$$\frac{\begin{array}{cc}\{g \Rightarrow (post \wedge g)\}[P, \phi]\{post \wedge g\} & post \Rightarrow (g \Rightarrow (post \wedge g)) \\ post \wedge g \Rightarrow post \wedge g & \phi \Rightarrow \phi\end{array}}{\{post\}[await g, \phi]\{post \wedge g\}}$$

Ce qui est parfaitement valide, et indique de plus que la définition de [Siewe et al.01] est une conséquence de la nôtre. La raison en est que notre définition *garde* la post-condition, alors que celle de [Siewe et al.01] se contente de «retirer» de la postcondition la garde devenue inutile.

D'un autre côté, il faut vérifier que les structures de contrôle de B sont compatibles avec les règles de construction définies dans [Siewe et al.01]. Par exemple, la conditionnelle est définie par :

$$\frac{\{pre \wedge g\}[P, \phi]\{post\} \quad \{pre \wedge \neg g\}[Q, \psi]\{post'\}}{\{pre\}[\textbf{if } g \textbf{ then } P \textbf{ else } Q \textbf{ fi}\ , \phi \vee \psi]\{post \vee post'\}}$$

Cette règle est vérifiée pour B par :

$$\begin{array}{l} (pre \wedge g) \Rightarrow [P]post \\ \wedge (pre \wedge \neg g) \Rightarrow [Q]post' \\ \wedge pre \\ \Rightarrow \\ [g \Longrightarrow P [\!] \neg g \Longrightarrow Q](post \vee post') \end{array}$$

, ce qui est vrai, la substitution située dans le but étant la forme pour la conditionnelle de B.

On peut faire la même chose pour le *while*, en ayant retiré les prédicats de la clause de variance (V), puisque dans [Siewe et al.01], cette clause n'apparaît pas dans le *while*.

Avec le temps

Ici les formules de durée sont reprises de [Siewe et al.01], sauf bien entendu pour la précondition et le choix non borné. Les règles qui ont donc été fixées ici pour ces deux structures ne sont pas prouvées correctes, seule l'intuition peut le laisser penser. Cependant l'annexe de [Siewe et al.01] indique la façon dont doit être conçue la preuve, et donne, par la même occasion, une idée de la lourdeur des preuves avec certaines extensions de DC.

Remarque

L'affectation de la figure 4.2 a une durée nulle pour respecter l'hypothèse du vrai synchronisme, mais il est tout à fait possible de considérer que le calcul des expressions dans une affectation dure un *certain* temps (comme une addition, une multiplication,...), temps qui pourra ainsi paramétrer les formules de DC obtenues. La raison pour laquelle ce cas n'est pas considéré ici, est qu'il relève plus d'un problème pratique (les opérateurs de B étant nombreux, et les plates-formes d'arrivée des programmes étant fort variées).

Néanmoins, étant donné qu'en B les substitutions «parallélisées» par le $||$ ont des ensembles de variables disjoints, on peut sans scrupules adopter la définition

GSL	$dur([\mathrm{GSL}]P)$
skip	$\lceil\rceil$
$x := E$	$\lceil\rceil$
$x,y := E,F$	$\lceil\rceil$
delay d	$(\ell = d) \wedge \lceil P \rceil$
await g	$\lceil \neg g \wedge P \rceil^*$
$g \mid S$	$dur([S]P)$
$g \Longrightarrow S$	$dur([S]P)$
$S;T$	$dur([S]([T]P))^\frown dur([T]P)$
$S [] T$	$dur([S]P) \vee dur([T]P)$
$@x.S$	$dur([S]P)$
$S \| T$	transformé par l'utilisation de règles de réécriture (voir plus haut)
WHILE C DO S VARIANT V INVARIANT I	$dur([S]P)^*$

FIG. 4.2 – Calcul d'une formule de durées à partir d'une substitution généralisée

utilisée dans [Siewe et al.01]. Lorsque les variables partagées seront autorisées, il faudra vérifier que cette règle reste correcte.

De plus, si le calcul des expressions dans les affectations est considéré durer un certain temps, la sémantique d'entrelacement de la concurrence ne tient plus, et les affectations multiples telles qu'écrites ici ne sont plus valables, puisqu'il y a un ordre d'exécution de fait.

Enfin, toute extension de B basée sur les seules substitutions généralisées, est utilisable avec le calcul des durées ajouté à B, comme par exemple les itérateurs pour B ([Cave et al.00]).

Problèmes

On remarque que la construction $\lceil\rceil$, par la syntaxe, ne devrait contenir que des formules propositionnelles, alors qu'ici on l'autorise à contenir des formules du calcul des prédicats. Mais en fait la restriction aux formules propositionnelles est arbitraire, et l'on pourrait très bien considérer un prédicat non quantifié (du genre de $x > 0$) comme une variable propositionnelle.

De plus, le fait d'utiliser la construction *await* dans un programme lui donne une possibilité de non-terminaison. Mais en B, pour l'instant, on considère que tous les programmes terminent. Nous devrons donc garder cette hypothèse en tête pour le moment. Néanmoins, si B est étendu avec la concurrence et le parallélisme, cette hypothèse devra probablement être reconsidérée.

Ainsi en pratique, le *await* étant utilisé avec des variables partagées entre plusieurs processus dans un langage autorisant la concurrence, il pourra être associé en B «terminant» à des capteurs modélisés par des variables ou dont le comportement sera reflété par le corps d'une opération.

Par exemple, un capteur dont l'activation nécessite un temps τ, pourra se voir modélisé par une opération dont le corps contiendra un *delay* τ

4.1.3 Obligations de preuves supplémentaires

Pour prouver la cohérence temporelle d'un projet, il faut maintenant prouver de nouvelles obligations de preuves temporelles.

Opérations et spécification temporelle

$$(dur([V_1]I) \vee \ldots \vee dur([V_n]I)) \wedge S \Rightarrow T$$

Les V_i sont les opérations de la machine, I son invariant et T la clause TIMING de la machine. S représente les clauses TIMING des machines incluses (si l'inclusion autorise l'appel d'opérations[2]).

Cette OP signifie que chacune des opérations doit pouvoir établir la spécification temporelle.

Ceci a deux conséquences :

- s'il y a beaucoup d'opérations différentes, la spécification se devra probablement d'être simple *OU*
- la machine ne devra être composée que d'une opération pour que l'OP puisse être établie.

En effet, la solution d'utiliser une clause temporelle pour chaque opération était également possible.

Raffinement

Clause de temps La machine abstraite étant l'interface avec les autres machines, c'est là que toutes les clauses se doivent d'être les plus précises, et la clause de temps n'échappe pas à cette règle :

$$T_{n-1} \Rightarrow T_n$$

si la machine n raffine la machine $n-1$. Il arrive en B que certaines clauses puissent disparaître dans un raffinement, puis réapparaître dans le suivant. Dans ce cas-là, le raffinement où la clause disparaît hérite de la clause correspondante de la machine raffinée. Cette particularité fonctionne également avec la clause de temps.

[2]les différentes possibilités d'inclusion de machines B étant relativement complexes, l'objet n'est évidemment pas de les détailler ici

Ainsi, même si la clause n'apparaît plus dans le raffinement, les opérations sont toujours soumises à des contraintes temporelles.

Opérations En B, l'obligation de preuve concernant les opérations est assez particulière, puisqu'elle exploite le fait que ce sont des transformateurs de prédicats. Le prédicat obtenu est un prédicat de sûreté (l'opération ne doit pas établir que l'opération raffinée établit un invariant faux). Cette transformation de prédicats n'est pas possible avec le calcul des durées puisqu'avec DC on calcule plutôt des traces d'exécution.

De plus, puisque l'OP «normale» de raffinement des opérations permet également de vérifier que les valeurs retournées[3] par une opération et son raffinement sont les mêmes, nous pouvons nous intéresser alors uniquement aux traces pures d'exécution (d'où le *true*) :

$$dur([V_n^i]true) \Rightarrow dur([V_{n-1}^i]true)$$

V_n^i étant la i-ème opération du n-ème raffinement de la machine. Cela signifie que :

- Temporellement, les opérations ne peuvent être que de plus en plus précises (entendre par là de plus en plus déterministes). En effet, la source de non-déterminisme qu'est le choix borné, est éliminé au fur et à mesure des raffinements.
- Eu égard à la substitution temporelle *delay*, cela signifie qu'une fois qu'un temps τ a été alloué à une opération, il n'est plus possible de changer cela dans les raffinements.

Remarque Les OPs de raffinements remplissent leur rôle : éviter d'avoir à prouver de nouveau que les opérations remplissent bien les exigences temporelles de la machine, et ce par transitivité.

Il y a certes peu de OPs, mais elles suffisent à saisir toutes les traces d'exécution des machines nécessaires pour prouver la sûreté temporelle.

Aide à la preuve

Assertions temporelles En B, une clause appelée clause d'*ASSERTIONS*, permet de simplifier la validation d'une machine lorsque ses obligations de preuve sont complexes. Elle donne au développeur la possibilité de rajouter des prédicats intermédiaires pour alléger le travail du prouveur, qui se doit d'être capable de prouver le plus de choses possibles automatiquement. Il lui correspond d'ailleurs une OP, puisque pour que les assertions soient utilisables, il faut avoir prouvé leur validité.

Un mécanisme similaire pourrait être imaginé pour la vérification des spécifications temporelles, mais il est d'abord préférable de définir précisément ce qu'est

[3] Un terme plus adapté serait «modifiées», car en B les opérations ne sont pas des fonctions

le niveau de complexité d'une formule de durées, puisqu'en pratique il est plus économique de prouver directement une formule de durées plutôt que de définir de nombreuses de formules intermédiaires, à faire prouver automatiquement, censées faire obtenir un gain de temps.

Support de preuve Quelques essais d'ajout du calcul des durées à Coq ont été faits, d'abord sous forme de plongement profond. Si la plupart des règles et axiomes du système de déduction (section 2.1.1) s'expriment très bien dans Coq, celles qui possèdent des conditions de bord particulières (comme par exemple une condition sur la flexibilité) posent problème. En effet, il est naturel en Coq pour un plongement profond, de définir une variable temporelle comme une variable d'un type donné.

Seulement, lorsque l'on calcule une formule, il n'est pas possible de savoir si on travaille sur une variable ou une formule décomposable, ce qui empêche justement de pouvoir vérifier ces conditions de flexibilité, et *a fortiori* d'automatiser une telle tâche par des tactiques. C'est pourquoi l'ajout d'une librairie pour le calcul des durées à Coq est envisagé actuellement sous la forme d'un plongement léger, certes moins aisé d'utilisation, mais également avec moins de problèmes liés à Coq.

4.2 Perspectives

Ce document se veut avant tout une esquisse de ce à quoi peut ressembler l'apport du temps-réel à B au moyen d'une logique *ad hoc*, et à ce titre il reste beaucoup à faire :

- Définir plus formellement la relation des substitutions à SDC^*. En effet nous nous sommes basés sur les triplets de Hoare de [Siewe et al.01] pour définir les propriétés temporelles des GSL, et il est bon de vérifier tout cela sur la syntaxe particulière de B. Néanmoins, comme nous l'avons vu en section 4.1.2 , il n'y a pas d'incompatibilité logique majeure
- Compléter par des obligations de preuves manquantes, les OPs indiquées ici ne concernant que la preuve temporelle. Il faut en plus vérifier que la clause TIMING n'entre pas en contradiction avec l'invariant. Mais créer un tel pont entre la logique de B et le calcul des durées n'est pas simple
- Ajouter cette extension du formalisme aux outils de gestion de projets B
- Ajouter un support de preuve pour DC à ces mêmes outils. Par exemple, le projet [Bcaml] utilisant [Coq03], il faudra créer une librairie pour la preuve de formules de DC^* dans Coq.
- Réaliser des essais pratiques de ce qui a été dit dans ce document sur des exemples complets, à l'aide desquels des décisions pratiques pourront être prises (voir par exemple section 4.1.3)
- SDC^* donne la possibilité de raisonner sur des programmes qui ne terminent pas, et pour l'instant les programmes en B standard doivent terminer. Avec l'ajout du calcul des durées, c'est une obligation qu'il devient possible de

reconsidérer

– Le langage utilisé dans [Siewe et al.01] autorise la concurrence, et la sémantique (logique et temporelle) des programmes concurrents est bien définie. Il est donc intéressant de reconsidérer la concurrence en B, car même si la substitution $\|$, par son non-déterminisme, peut être assimilée à un opérateur de mise en concurrence, elle laisse tout de même un «flou sémantique» qu'il devient possible de clarifier grâce au calcul des durées.

4.3 Conclusion

Comme nous l'avons vu, le calcul des durées est un formalisme puissant pour la spécification et la preuve de contraintes temporelles, mais pèche par sa lourdeur d'utilisation lors du développement de programmes conformes aux spécifications (voir section 2.1.1). À côté de cela, B est une méthode dont le cycle de développement, modulaire et cohérent, est efficace, mais est quelque peu restreint par son usage du calcul des prédicats lorsqu'il faut décrire des problèmes temps-réel.

La combinaison des deux s'avère être plutôt naturelle, sans qu'il n'y ait de conflit théorique apparent. Mieux encore, cet ajout de DC à B, ouvre des voies nouvelles pour la concurrence en B (voir section 4.2). Ainsi, une fois que les règles d'utilisation des spécifications temporelles dans un programme B sont fixées (voir section 4.1.3), il est envisageable de reconsidérer la concurrence en B sous l'angle temporel, le calcul des durées permettant de gérer habilement les imbrications d'intervalles qu'il peut y avoir dans les traces d'exécution de programmes concurrents.

En conclusion, nous pourrons dire que l'ajout du calcul des durées à B, bien qu'il requiert encore un certain travail de validation, laisse augurer d'intéressantes voies de développement pour la suite.

Index

calcul des durées, 5
    assistants d'aide à la preuve, 11
    itération, 10
    obligations de preuve, 20

domaine d'interprétation, 9

logique temporelle d'intervalle, 8

méthode B, 13
    obligations de preuve, 14
    substitutions généralisées, 14

non-terminaison, 18, 20

plus faible précondition, 17

système de déduction, 8

trace d'exécution, 21

Variabilité finie, 9

Bibliographie

[Abrial96] Abrial (Jean-Raymond). – *The B Book - Assigning Programs to Meanings*. – Cambridge University Press, août 1996.

[Atelier b] http ://www.atelierb.societe.com.

[B toolkit] http ://www.b-core.com.

[Bcaml] http ://www3.inrets.fr/B@INRETS/BCaml.

[Behm et al.99] Behm (Patrick), Benoit (Paul), Faivre (Alain) et Meynadier (Jean-Marc). – Meteor : A successful application of B in a large project. *Lecture Notes in Computer Science*. pp. 369–387. – Springer Verlag, september 1999.

[Cave et al.00] Cave (Francis) et Bert (Didier). – Itérateurs pour le langage B. *Actes de l'Atelier AFADL 2000*. LSR-IMAG, pp. 111–126. – Grenoble, January 2000.

[Colin et al.03] Colin (Samuel), Poirriez (Vincent) et Mariano (Georges). – Thoughts about the implementation of the duration calculus with Coq. *4th International Workshop on the Implementation of Logics*. – University of Liverpool, september 2003. `http://www.csc.liv.ac.uk/research/techreports/`.

[Coq03] Coq, 1989-2003. http ://coq.inria.fr.

[Dca] http ://www.iist.unu.edu/home/Unuiist/newrh/III/1/page.html.

[Dcb] http ://www.iist.unu.edu/dc/.

[Dutertre95] Dutertre (Bruno). – *On first order interval temporal logic*. – Rapport technique nCSD-TR-94-3, Department of computer science, Egham, Surrey TW20 0EX, England, University of London, february 1995.

[Guelev et al.99] Guelev (Dimitar P.) et Hung (Dan Van). – Completeness and decidability of a fragment of duration calculus with iteration. *Asian Computing Science Conference (ASIAN'99)*. pp. 139–150. – Phuket, Thailand, December 1999. Also presented at International Conference on Mathematical Foundation of Informatics, Hanoi, October 25-28, 1999.

[Hansen et al.97] Hansen (M.R.) et Zhou (C.C.). – Duration calculus, logical foundations. *Formal Aspects of Computing*, pp. 283–330. – 1997.

[Heilmann99] Heilmann (Søren T.). – *Proof Support for Duration Calculus*. – Phd-thesis, Department of Information Technology, Technical University of Denmark, Januar 1999.

[Heilmann8a] Heilmann (Søren T.). – *PC/DC Users Guide*, 1998(a).

[Lano97] Lano (K.). – Specifying reactive systems in B AMN. *LNCS*, vol. 1212, 1997, pp. 242–275.

[My96] M.Y. (Vardi). – An automata-theoretic approach to linear temporal logic. *Logics for Concurrency : Structure versus Automata*, pp. 238–265. – Springer-Verlag, 1996.

[Naijun99] Naijun (Zhan). – *Another formal proof for deadline driven scheduler*. – Rapport technique n169, P.O. Box 3058, Macau, UNU/IIST, august 1999.

[Sabatier et al.00] Sabatier (Denis) et Lartigue (Pierre). – The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. *Formal Method in System Design*. pp. 245–272. – Kluwer academic publisher, december 2000.

[Siewe et al.01] Siewe (François) et Hung (Dan Van). – Deriving real-time programs from duration calculus specifications. *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*. pp. 92–97. – Livingston-Edinburgh, Scotland, september 2001. (Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000).

[Treharne et al.99] Treharne (Helen) et Schneider (Steve). – *Capturing timing requirements formally in AMN*. – Rapport technique nCSD-TR-99-06, Department of computer science, Egham, Surrey TW20 0EX, England, Royal Holloway, June 1999.

[Xia98] Xia (Yong). – A raise specification framework and justification assistant for the duration calculus. *ESSLLI-98 Workshop on Duration Calculus*, pp. 51–57. – Germany, august 1998. (Technical report 126, UNU-IIST, P.O.Box 3058, Macau, November 1997).

[Xiaoguang et al.96] Xiaoguang (Mao), Qiwen (Xu) et Ji (Wang). – *Towards a proof assistant for interval logics*. – Rapport technique n 77, P.O. Box 3058, Macau, UNU/IIST, july 1996.

[Zhou et al.91] Zhou (C.C.), Hoare (C.A.R.) et Ravn (A.P.). – A calculus of durations. *Information Processing Letters*, pp. 269–276. – Dezember 1991.

[Zhou et al.95] Zhou (C.C.), Wang (J.) et Ravn (A.P.). – A duration calculus with infinite intervals. *Fundamentals of Computing Theory*, éd. par Reichel (H.), pp. 16–41. – Lübeck, Germany, Springer-Verlag, 1995.

[Zhou et al.99] Zhou (C.C.), Guelev (D.P.) et Naijun (Z.). – A higher-order duration calculus. *Symposium in Celebration of the Work of C.A.R. Hoare*. – Oxford, september 1999. (Technical report 167, UNU-IIST, P.O.Box 3058, Macau, July 1999).