École doctorale de l'université de Valenciennes et du Hainaut-Cambrésis

Contribution à l'intégration de temporalité au formalisme B : Utilisation du calcul des durées en tant que sémantique temporelle pour B

THÈSE

présentée et soutenue publiquement le 3 octobre 2006

pour l'obtention du

Doctorat de l'université de Valenciennes et du Hainaut-Cambrésis

Discipline : informatique
Spécialité : Automatique et Informatique des Systèmes Industriels et Humains

par

Samuel Colin

Composition du jury

Président :	Pascal Yim, professeur à l'École Centrale de Lille
Rapporteurs :	Didier Bert, chargé de recherche CNRS au LSR-IMAG (Grenoble) Hassan Mountassir, professeur à l'université de Franche-Comté (Besançon)
Examinateurs :	Arnaud Fréville, directeur de thèse, professeur à l'UVHC Vincent Poirriez, co-directeur, maître de conférences à l'UVHC Georges Mariano, co-directeur, chargé de recherches à l'INRETS

Mis en page avec la classe thloria.

Remerciements

Il me semble opportun de commencer ce document par une citation de Bernard de Chartres : « Nous sommes des nains assis sur des épaules de géants. Si nous voyons plus de choses et plus lointaines qu'eux, ce n'est pas à cause de la perspicacité de notre vue, ni de notre grandeur, c'est parce que nous sommes élevés par eux ». Je trouve que cette phrase vaut pour tous les gens qui me précèdent : les scientifiques qui ont passé leur vie à comprendre, et à transmettre le savoir qu'ils en ont retiré, et les gens qui m'ont soutenu jusqu'à présent.

Qui sont donc ces récipiendaires des éloges que cette page me permet de dispenser ?

– Mes enseignants, des tout débuts de ma scolarité jusqu'à maintenant, avec une mention spéciale pour mes professeurs de français. Mes enseignants de DEA ont beaucoup contribué à ma compréhension des mathématiques qui fondent l'informatique
– Le conseil régional et l'INRETS, qui ont financé la majeure partie de ma thèse
– L'INRETS-Villeneuve d'Ascq en général, et l'unité de recherche ESTAS en particulier, dont j'ai pu apprécier la diversité scientifique et la richesse humaine. Les pauses café étaient toujours très conviviales
– Le LAMIH-ROI qui m'a accueilli en tant qu'ATER
– Jérôme, que je remercie pour toutes les discussions enjouées que nous avons pu avoir, parfois devant une tasse de café, parfois au dam des autres doctorants qui partageaient le bureau
– Dorian, qui m'a précédé dans le parcours que j'ai suivi jusqu'ici, et dont l'expérience m'a été bénéfique
– Georges, pour toutes les discussions techniques et idéologiques que nous avons eues, et pour m'avoir fait découvrir Debian (que j'utilise majoritairement aujourd'hui)
– Vincent, pour la sagesse dont il a fait preuve, son soutien et ses encouragements malgré son emploi du temps impressionant
– Les rapporteurs, dont les commentaires avisés m'ont beaucoup aidé
– Mes parents, sans qui je ne serais évidemment pas là. Mais ce n'est pas parce qu'une tautologie est intrinsèquement vraie qu'elle n'a pas de valeur
– Oana, care a schimbat vedere mea despre viaţă
– Christophe, Éric, Frédéric, Nicolas, Mickaël, Célia, Fabien D., Fabien M., Fabien C., et les autres doctorants du LAMIH/ROI, ainsi que le gang des barbichettes
– Tous les libristes qui ont contribué à parfaire ma connaissance de Linux et des logiciels qui gravitent autour, puisque je travaille de cette manière depuis longtemps déjà.

Je dédie ce document à Jean, Daniel et Bernadette, qui auraient été sûrement fiers de ne pas tout comprendre à ce qui suit.

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction : Méthodes formelles et expression du temps

Sommaire

Figures

1.1 Méthodes formelles

Lorsqu'il est question de résoudre un problème par l'utilisation d'un système mécanique, électronique ou informatique, il se pose le problème de la manière dont ce système sera conçu. Sa conception doit *in fine* garantir qu'il résout effectivement le problème pour lequel il est utilisé.

Il existe des systèmes dits « critiques » pour lesquels toute défaillance est absolument hors de question. Dans le cas de systèmes logiciels (ou, parfois, électroniques) se pose la question suivante : quelle méthode peut garantir que le système conçu répond bien à ces exigences de criticité ? C'est dans ce contexte que sont invoquées les méthodes formelles.

Nous allons donc, après avoir défini toutes ces notions, détailler quelles sont les particularités des méthodes formelles, et pourquoi ces particularités sont importantes.

1.1.1 Systèmes critiques

La nature d'un problème critique est celle d'un problème qui implique des vies humaines ou, parfois plus prosaïquement, de grandes quantités d'argent. Un système critique est donc un système opérant dans un tel contexte, c'est-à-dire qu'il devra gérer des entités dont la valeur estimée est, sinon incommensurable (vies humaines), particulièrement grande. Des exemples de contextes critiques sont, par exemple et sans souci d'exhaustivité, le domaine du transport (aérien, ferroviaire, routier), le domaine économique (banquier, boursier), le domaine médical ou celui de la production énergétique.

Par conséquent, toute défaillance d'un système opérant dans un contexte critique est à éviter autant que faire se peut. Cela signifie que la conception de ce système doit absolument garantir que celui-ci ne peut pas faillir à sa tâche, et ce, dès sa mise en service. Il est donc nécessaire de disposer de méthodes de conception qui permettent de vérifier cette forte contrainte.

Ces méthodes de conception ont toutes un point commun : elles obéissent strictement à des règles établies (ou découvertes) selon le domaine dans lequel elles sont utilisées. L'architecte en génie civil doit absolument suivre les règles physiques immuables des matériaux qu'il manipule pour garantir que son ouvrage ne s'effondrera pas. Le chimiste sait qu'il doit suivre certaines précautions pour éviter que les produits qu'il fait interagir n'engendrent pas une réaction incontrôlable. De la même manière, l'ingénieur informaticien doit se plier à certaines règles des conception pour éviter que le système qu'il conçoit développe un comportement imprévu et potentiellement dangereux.

Ces règles de conception définissent un cadre formel, en ce sens qu'elles obéissent à des lois (physiques ou mathématiques, par exemple) immuables qui définissent un contexte de conception. C'est pour cette raison que les méthodes de conception logicielle obéissant à des règles de ce type sont appelées *méthodes formelles*.

1.1.2 Méthodes formelles et semi-formelles

À la différence, par exemple, du domaine de l'architecture ou de la chimie cités plus haut, le domaine de l'informatique n'est pas soumis aux lois du monde physique, de par sa nature abstraite. L'informatique est une branche à part entière des mathématiques. Comme l'a souligné Edsger Dijkstra, « l'informatique ne traite pas plus des ordinateurs que l'astronomie traite des téléscopes. »[1]

Il n'est pas rare de comparer la programmation, l'acte de développement logiciel, à de l'art au sens esthétique du terme. Sans traiter ici du bien-fondé ou non de ces points de vue, nous pouvons remarquer qu'ils dénotent le haut niveau d'abstraction souvent associé à la discipline.

Dès lors que le domaine est si abstrait, comment alors garantir qu'un système informatique réalise ce pour quoi il est conçu ? Quelles sont les règles du domaine qui régissent la conception

[1] « Computer Science is no more about computers than astronomy is about telescopes. »

de systèmes informatiques critiques ? La réponse vient également ici d'un point commun avec d'autres domaines d'ingénierie : les mathématiques.

En effet, le besoin ici est de disposer de règles qui permettent de raisonner sur les objets du domaine informatique, les programmes. Ces règles se doivent d'être *formelles*, c'est-à-dire de laisser le moins possible de place à l'intuition et aux erreurs. Donc, idéalement, ces règles doivent être vérifiables automatiquement (mécaniquement, informatiquement) sans devoir recourir au raisonnement d'un être humain.

Le principe est alors d'utiliser les mathématiques pour exprimer ce que signifie (sa *sémantique*) un algorithme, et s'il répond au besoin pour lequel il a été conçu (sa *validation*).

Les propriétés vérifiées par un système s'expriment donc sur la base de la sémantique qui aura été utilisée pour le décrire.

Une méthode sera donc *formelle* si elle dispose des entités suivantes :

- Un langage : un moyen pour exprimer et concevoir un système. La nature de ce langage peut être symbolique, graphique,etc
- Une sémantique : la signification mathématique des éléments de ce langage
- Des règles de validation : il doit être possible d'exprimer les propriétés désirées du système, et d'utiliser la sémantique du système pour le valider.

Une méthode sera *semi-formelle* si la sémantique ou les règles de validation sont incomplètes. Un exemple de méthode semi-formelle est UML [Bur97, UML] : elle dispose d'un langage (graphique, via les différents types de diagrammes), d'une sémantique (associations, quelques propriétés exprimables via OCL) et de règles de validation (celles d'OCL, proche de la logique classique). Un exemple de méthode formelle est la méthode B, présentée ci-après.

1.1.3 Intérêt de la méthode B

Le besoin pour des méthodes formelles étant présent, il a bien entendu fallu le combler : c'est ainsi que la méthode B est apparue au début des années 1980.

En premier lieu, qu'est-ce qui fait de la méthode B (aussi appelée « B ») une méthode formelle ? Reprenons la définition plus haut :

- Elle dispose d'un langage, qui permet de définir des composants et leur comportement dynamique
- Elle dispose également d'une sémantique : les propriétés statiques des modèles se basent sur la logique classique et la théorie des ensembles. Les actions ont une sémantique en transformateurs de prédicats, décrivant comment passer d'un état à un autre
- Les règles de validation se basent sur la démonstration logique que les propriétés statiques d'un modèle B sont maintenues par les propriétés dynamiques de la machine. La validation en elle-même peut être faite par toute méthode de raisonnement sur des formules de la théorie des ensembles.

Tout cela est décrit dans le détail dans le *vade-mecum* de la méthode B écrit par Jean-Raymond Abrial [Abr96].

Par rapport aux autres méthodes cependant, quels sont les points forts de B ? Sa conception fait d'elle un bon compromis entre la formalisation et la facilité d'utilisation :

- La théorie des ensembles est connue depuis plus d'un siècle, et est bien comprise. Elle est de plus enseignée partout dans l'éducation supérieure, ce qui signifie qu'un ingénieur pourra plus facilement s'y adapter

- Son langage est simple à comprendre, sans sacrifier à l'expressivité ni à la facilité de validation. Il dispose de nombreux concepts proches de langages de développements moins formels (la modularité en est un exemple)
- Sa grande force est de disposer d'un mécanisme de développement incrémental parfaitement intégré à la validation : la complexité d'un système développé avec B peut donc être aisément maîtrisée.

Nous détaillons tous ces points au chapitre 3.

1.2 Anciennes méthodes formelles pour nouveaux problèmes

Les méthodes formelles sont utilisées dans l'industrie pour la conception de systèmes critiques, et à chaque nouveau projet logiciel sont associés de nouveaux défis de conception. Chaque nouveau défi est susceptible de remettre en cause l'adoption de la méthode formelle précédemment choisie. C'est pour cela que nous allons détailler ici les raisons de l'intérêt d'être capable d'étendre une méthode formelle.

1.2.1 Motivations industrielles

La discussion de l'adoption des méthodes formelles par le monde industriel est loin d'être close [Sai96], et tout un chacun a une opinion sur la question. Le fait est que cette adoption est lente et souvent difficile, quand bien même les bénéfices soient avérés [HB95].

Selon le même type d'arguments (difficulté apparente, faible visibilité du retour sur investissement), comment alors convaincre un industriel d'abandonner toute l'expertise qu'il a acquise avec une méthode formelle, pour en utiliser une autre plus expressive, plus complète, mais basée sur des fondations différentes ? Sachant que l'adoption fut difficile en premier lieu, le changement risque d'être encore plus ardu.

C'est la raison pour laquelle, en prévision du long-terme, il est important que la méthode formelle adoptée soit suffisamment puissante pour pouvoir exprimer une grande classe de problèmes, ou qu'il soit possible de la faire évoluer.

1.2.2 Méthodes souples

Penchons-nous d'abord sur les méthodes à forte puissance expressive : leur avantage induit leur principal inconvénient, c'est-à-dire que la plupart du temps le formalisme fera appel à des notions abstraites mathématiques plus ou moins complexes. Or du point de vue de l'industriel, trouver un ingénieur capable, ou en former un pour un tel formalisme, est très difficile.

Ensuite, nous avons les méthodes faciles à faire évoluer. Nous pouvons ici distinguer deux écoles :

- La méthode contient déjà une notion d'évolutivité dans la manière dont elle est conçue. Il s'agit là d'une particularité des méthodes dites algébriques, qui sont encore notablement jeunes. C'est par exemple le cas de CASL [BM04]. Remarquons également que les apports théoriques de ces travaux sur l'évolutivité d'un formalisme sont nombreux en ce domaine, par exemple pour CASL : l'ordre supérieur (HasCASL [SM02]), le raffinement entre sémantiques hétérogènes (HetCASL [Mos03])

– La méthode est suffisamment générique pour exprimer tout ou partie d'autres formalismes. Il s'agit souvent du chemin pris pour exprimer des classes de systèmes qui étaient *a priori* difficilement exprimables (nous en donnerons quelques exemples d'utilisation avec B au chapitre 3).

Comme il se trouve que B est en partie fondé par la théorie des ensembles, avec laquelle nombre d'autres formalismes peuvent être exprimés, cette méthode est déjà, dans le contexte de l'évolutivité, un bon candidat.

1.2.3 Choix de la méthode B

Résumons les contraintes et questions qui sont déjà posées : il est difficile de faire changer à un industriel la méthode formelle qu'il utilise, et sur le long terme, l'idéal pour celle-ci est de faire preuve d'évolutivité.

Nous savons que la méthode B est basée sur un formalisme dans lequel une grande classe de problèmes peuvent être modélisés. De plus, B fait partie du cercle relativement fermé des méthodes formelles dont l'adoption par le monde industriel est réussie [BBFM99, SL00].

Toutes ces raisons font de la méthode B le candidat idéal pour étudier son évolutivité vers une classe de problèmes de plus en plus aigus constitués de contraintes temporelles.

1.3 Expression de problèmes à contraintes temporelles

Initialement, les problèmes auxquels les développeurs de systèmes informatiques avaient à faire face pouvaient se réduire à des problèmes de calcul : comment garantir que les états du système correspondent au domaine de valeurs que les contraintes fixaient ?

Puis, avec l'apparition de la notion de composition de systèmes développés indépendamment, est arrivée la question de savoir si cette composition s'avérait harmonieuse. Également, en particulier dans le domaine du transport, des contraintes sur la *manière* dont le système doit fonctionner sont également apparues. Si en effet, par exemple, c'est un logiciel qui pilote une locomotive, une contrainte est de réactualiser l'état du système pour garantir qu'il pourra réagir en un temps minimal à un problème soudain.

Ces notions de fonctionnement composé de systèmes, de contraintes temporelles, font partie de la classe des problèmes de type *temps-réel*. Les formalismes qui devront servir à modéliser les systèmes critiques devront donc refléter de manière plus ou moins fidèle ce type de problèmes. Ces problèmes de type temps-réel sont scindés en deux classes [2] : les approches discrètes et les approches continues.

1.3.1 Approches discrètes

Les approches discrètes se basent sur une modélisation dénombrable des états du système. Des langages tels Lustre [LUS, CPHP87], Esterel [EST, Ber00], des logiques telles LTL [Var96], CTL [EH85], font partie de ce type d'approches.

[2] Bien qu'il existe, pour certains formalismes, des façons de passer d'une classe à l'autre

Le fait qu'elles modélisent des états dénombrables permet de faciliter le contrôle sur le formalisme. Grâce à cela, il est possible d'utiliser des techniques du type de celles utilisées pour les automates pour valider les systèmes modélisés. En effet, les outils de validation pour ces formalismes se basent le plus souvent sur de la vérification de modèle.

Leur avantage est aussi leur défaut : leurs mécanismes sont adaptés à une représentation dénombrable. Cela implique que des opérations telles que l'ajout d'états (pour une modélisation incrémentale) ou la composition de systèmes peuvent poser problèmes. Soit ces opérations sont difficiles à faire, soit c'est la vérification qu'elles sont correctes qui est difficile. Sans entrer dans les détails, ces problèmes correspondent respectivement aux états indiscernables du modèle abstrait, et à l'explosion combinatoire du nombre d'états.

La machine B en figure 1.1, tirée d'une étude de cas de [TS99], introduit un mécanisme de spécification du temps. Deux opérations `ReadTime` et `SetTime` permettent de consulter le temps et de le mettre à jour, respectivement.

```
MACHINE
  Timer
VARIABLES
  time
INVARIANT
  time ∈ ℕ
INITIALISATION
  time :=0
OPERATIONS

tt ⟵ ReadTime =
    BEGIN
       tt :=time
    END ;

    SetTime(tt) =
      PRE
         tt ∈ ℕ
       ∧ tt >time
      THEN
         time :=tt
      END

  END
```

FIG. 1.1 – Une machine B pour modéliser le temps

Cette machine B modélise le temps de manière dénombrable, via une variable entière `time`. Dans l'article de [TS99], cette machine `Timer` est utilisée par un contrôleur dont les opérations ont des contraintes de temps, et l'influencent. Les contraintes sont spécifiées dans les préconditions.

Les prédicats sont par exemple de la forme $inputReceived + 15 \leq time$, pour spécifier que l'opération ne peut être déclenchée qu'au moins 15 secondes après que le contrôleur ait reçu une donnée en entrée. Les substitutions qui modifient le temps ont la forme suivante :

$$
\begin{array}{l}
\text{ANY } endTime \\
\text{WHERE } endTime > time \\
\qquad \wedge endTime \leq inputReceived + 5 \\
\text{THEN } SetTime(endTime) \\
\text{END}
\end{array}
$$

Cette dernière substitution permet d'indiquer que l'opération se termine en moins de 5 secondes après réception d'une donnée.

Cette modélisation du temps est bien adaptée pour le problème auquel elle est appliquée, un contrôleur au comportement séquentiel. Que se passe-t-il alors lorsque plusieurs composants ayant eux aussi des contraintes temporelles doivent être assemblés ? Ont-ils des horloges différentes, ou fonctionnent-ils sur la même ? Dans l'exemple indiqué plus haut, des horloges différentes signifient une synchronisation de ces mêmes horloges : cela se traduirait par une augmentation exponentielle de la taille des préconditions des opérations pour prendre en compte tous les cas. Si l'horloge est partagée, ça n'est tout simplement pas possible en B : plusieurs machines ne peuvent pas modifier (par le biais ici de l'opération `SetTime`, par exemple) les variables d'une même machine qu'elles utilisent.

Comme nous venons de le voir, une modélisation du temps discrète, se servant des mécanismes de B, est limitée à des systèmes simples. Des remarques similaires peuvent être formulées pour d'autres formalismes discrets du temps.

1.3.2 Approches continues

Les approches continues visent à être un reflet plus proche du monde physique où le temps est effectivement continu. Parmi ces approches peuvent se ranger les automates temporisés [AD94](où les transitions sont étiquetées par des contraintes de temps) ou des logiques comme la logique d'intervalle [HMM83] et les logiques se basant sur celle-ci, comme le calcul des durées [HZ04].

L'avantage de tels formalismes réside dans leur puissance d'expression, qui résume en des phrases concises des spécifications temporelles complexes. L'inconvénient est à l'avenant : la puissance expressive est telle que la validation peut être particulièrement difficile pour certaines formes de phrases du formalisme. Certaines validations peuvent être même impossibles, au sens que le problème modélisé correspond à des formules qui font partie de la classe indécidable du formalisme.

1.3.3 Étendre B

La question qui se pose alors est la suivante :

Quelle classe de problèmes de nature temporelle la méthode B peut-elle traiter ? Théoriquement, beaucoup de formalismes peuvent être exprimés via la théorie des ensembles. Il est cependant plus raisonnable d'exprimer des problèmes qui s'adaptent bien à une modélisation discrète. Cette limitation choisie s'explique par les difficultés qu'il est déjà possible de rencontrer lors de la conception et la vérification de formalismes discrets, telles que l'indique l'exemple des horloges partagées de la section 1.3.1.

Dès lors, est-il possible d'adapter B de manière à pouvoir exprimer simplement des spécifications du domaine temporel continu ? C'est la réponse à cette question que nous développons tout au long de ce document.

Nous devrons également tenir compte du fait que l'adoption par la pratique industrielle d'une méthode formelle spécifique, donc pointue, est généralement longue et difficile. En revanche, puisque la méthode B a déjà franchi ce cap, est-il alors possible de s'appuyer sur cette acceptation pour envisager le traitement de problèmes « temps-réels » par une extension ciblée du formalisme.

1.4 Plan

Nous détaillons dans un premier temps les deux formalismes que nous utilisons : les calculs de durées en chapitre 2, et la méthode B (ainsi que sa récente variante, le B événementiel) en chapitre 3. Dans ce même chapitre, nous présentons les différents moyens qui existent pour exprimer des systèmes à contraintes temporelles avec leurs avantages et leurs inconvénients.

Puis, sur la base de ces deux formalismes, nous montrerons qu'il est possible d'étendre B à des spécifications temporelles, dans le contexte d'un domaine temporel continu. Il en résultera B+DC, un B étendu via le calcul des durées. La validation de projets en B+DC se fera donc en deux temps : une validation fonctionnelle, qui utilise les mécanismes usuels de B. L'extension reste donc compatible avec le B d'origine. Le deuxième temps de la validation concerne la validation temporelle, qui passe par l'extraction de prédicats temporels en calcul des durées à partir des opérations. Ces prédicats temporels sont ensuite utilisés pour démontrer les spécifications temporelles associées à ces mêmes opérations. La démarche reste donc très proche de la démarche du B classique, avec l'ajout de l'expressivité apportée par le calcul des durées.

Ensuite, puisque B est une méthode formelle outillée, nous apportons des éléments nouveaux dans la mise en oeuvre du calcul des durées dans un outil de preuve à portée générique, Coq, au chapitre 5. Nous en déduisons une bibliothèque en Coq pour le calcul des durées, ainsi qu'un retour sur expérience ayant trait à l'implémentation de logiques modales dans un prouveur à vocation générique tel que Coq.

Nous terminons au chapitre 6 en rappelant les résultats que nous avons obtenu, d'une part en étendant B par un formalisme temporel continu, d'autre part en mettant en oeuvre les règles de validation de ce formalisme dans un outil de preuve. Nous concluons en traçant quelques prospectives sur l'extensibilité de B et du B événementiel, au niveau de la relation entre temporalité, compositionalité et raffinement.

Bibliographie

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor : A successful application of B in a large project. In *World Congress on Formal Methods 1999*, number 1709 in Lecture Notes in Computer Science, pages 369–387. Springer Verlag, september 1999.

[Ber00] Gérard Berry. The foundations of esterel. *Proof, language, and interaction : essays in honour of Robin Milner*, pages 425–454, 2000.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[Bur97] Rainer Burkhardt. *UML : Unified Modeling Language*. Addison-Wesley, 1997.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[EH85] E. A. Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30(1) :1–24, 1985.

[EST] `http://www.esterel-technologies.com/`.

[HB95] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Series in Computer Science. Prentice Hall International, 1995.

[HMM83] Joseph Y. Halpern, Zohar Manna, and Ben C. Moszkowski. A hardware semantics based on temporal intervals. In *10th International Colloquium on Automata, Languages and Programming*, volume LNCS 154, pages 278–291, London, UK, 1983. Springer-Verlag. ISBN :3-540-12317-2.

[HZ04] Michael R. Hansen and Chaochen Zhou. *Duration Calculus, a formal approach to real-time systems*. Number ISBN : 3-540-40823-1 in Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[LUS] `http://www-verimag.imag.fr/SYNCHRONE`.

[Mos03] Till Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany,* *2002, Revised Selected Papers*, Lecture Notes in Computer Science, pages 359–375. Springer Verlag, London, 2003.

[Sai96] Hossein Saiedian. An invitation to formal methods. *Computer*, 29(4) :16–17, 1996.

[SL00] Denis Sabatier and Pierre Lartigue. The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. In *Formal Method in System Design*, volume 17, pages 245–272. Kluwer academic publisher, december 2000.

[SM02] Lutz Schröder and Till Mossakowski. HasCASL : towards integrated specification and development of functional programs. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST 2002)*, volume 2422 of *Lecture Notes in Computer Science*, pages 99–116. Springer ; Berlin ; http ://www.springer.de, 2002.

[TS99] Helen Treharne and Steve Schneider. Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, Royal Holloway, Department of Computer Science, University of London, Egham, Surrey TW20 0EX, England, June 1999.

[UML] UML. `http://www.uml.org`.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency : Structure versus Automata*, volume 1043 of LNCS, pages 238–265. Springer-Verlag, 1996.

Chapitre 2

Les calculs de durées : syntaxe, sémantique, système de preuve et extensions

Sommaire

Tableaux

Ce chapitre présente le calcul des durées et ses extensions que nous utilisons par la suite. Outre un historique du calcul des durées, nous présentons la sémantique et le système de preuve pour celui-ci, et pour chacune des extensions. Pour une présentation plus approfondie, complète et pragmatique de cette logique, se référer à [HZ04].

2.1 La logique temporelle d'intervalle

Le calcul des durées peut être basé sur différentes logiques temporelles, auxquelles est ajouté un opérateur de durée. Historiquement, il est basé sur la logique d'intervalle temporelle. Cette section est dédiée à un rapide historique de cette logique, et à ses syntaxe, sémantique et système de preuve.

2.1.1 Historique

ITL (Logique Temporelle d'Intervalle) a été introduite par Halpern, Manna et Moszkowski [HMM83, Mos85] : son but est de permettre le raisonnement sur des intervalles continus de temps, leurs sous-intervalles, voire un découpage arbitraire des intervalles de temps. La logique ainsi obtenue possède des propriétés modales : certains connecteurs logiques ne représentent plus la même valeur selon leur place dans la formule. Des travaux ultérieurs de Moszkowski présentent un système de preuve pour pour ITL, et des travaux de Dutertre [Dut95a, Dut95b] montrent différentes axiomatisations pour ITL, avec des résultats de complétude, d'expressivité et d'adaptabilité à certaines extensions de la logique. Nous allons dans un premier temps indiquer la syntaxe et la sémantique d'ITL utilisées par la suite.

2.1.2 Syntaxe et sémantique

Syntaxe

ITL se base sur la logique des prédicats du premier ordre. Nous formalisons en premier lieu les notations pour celle-ci :

$\vee$	la disjonction
$\wedge$	la conjonction
$\neg$	la négation
$\Rightarrow$	l'implication
$\forall x$	la quantification universelle sur une variable x
$\exists x$	la quantification existentielle sur une variable x
$\Leftrightarrow$	l'équivalence

Ces connecteurs logiques ont la signification usuelle. Comme nous nous plaçons dans le cadre de la logique des prédicats du premier ordre, certains connecteurs peuvent être exprimés à l'aide d'autres, comme la conjonction $P \wedge Q \equiv \neg(\neg P \vee \neg Q)$.

ITL étant une logique temporelle continue, elle dispose également de moyens pour exprimer les relations entre des quantités et des opérations sur celles-ci. Ces quantités sont usuellement les nombres réels. Il est possible de définir ITL sur les entiers naturels, mais cela lui fait perdre son caractère continu. ITL dispose donc également de :
- Nombres réels
- Opérations arithmétiques sur les réels : $+, -, *, /$
- Relations sur les réels : $<, \leq, =, \geq, >$

La quantification de la logique du premier ordre s'appliquera donc à des nombres réels.

À la logique du premier ordre et les prédicats sur les nombres réels, ITL ajoute les connecteurs modaux suivants :
- Le découpage d'intervalle, noté « $\frown$ » ou « ; » comme le fait Dutertre[Dut95b]. En effet le découpage d'intervalle représente naturellement la séquence
- La longueur d'intervalle, notée ℓ. Cette quantité change selon sa place dans la formule, puisque des deux côtés d'un $\frown$, elle représentera un (sous-)intervalle différent. Cette particularité est due à sa nature qui, comme nous le verrons plus loin, est celle d'une fonction des intervalles vers les réels. Dans le système de preuve cependant, sa nature fonctionnelle n'est pas visible, d'où ce comportement particulier.
  La variable ℓ se manipule comme un nombre réel. Cependant, nous pouvons déjà nous rendre compte intuitivement que si elle peut ne pas représenter le même intervalle, remplacer une variable quantifiée par ℓ sans discernement pourra causer des problèmes.

La version d'ITL que nous présentons diffère de celle de Moszkowski[Mos85]. Cette différence tient à la réutilisation du système de preuve d'ITL tel que présenté par Dutertre[Dut95b], et cette version-ci d'ITL ne contient pas l'opérateur de répétition * (comparable à l'étoile de Kleene). Cet opérateur de répétition sera néanmoins réintroduit plus tard.

Pour récapituler, la syntaxe d'ITL est représentée par la syntaxe en forme BNF, dans le tableau 2.1. x représente une variable temporelle, v une variable globale, X une variable propositionnelle, R une relation, et f une fonction mathématique.

formule	: :=	atome | $\neg$ formule | formule $\vee$ formule | formule $\frown$ formule | $\exists x$. formule
atome	: :=	**true** | X | R(terme,...,terme)
terme	: :=	x | v | ℓ | f(terme,...,terme)

TAB. 2.1 – Syntaxe d'ITL

La distinction entre la variable temporelle x et la variable globale v est que x verra sa valeur changer en fonction de l'intervalle de temps courant, alors que v a une valeur indépendante du temps (elle sert à représenter les constantes, par exemple).

Les notations suivantes correspondent à des concepts voisins d'autres logiques temporelles :

- $\Diamond(P) \equiv \mathbf{true}^\frown P^\frown \mathbf{true}$: cet opérateur permet de spécifier une propriété P pour un sous-intervalle quelconque. Dans d'autres logiques temporelles, cet opérateur signifie *fatalement*. Dans d'autres logiques modales il peut signifier la notion de « possibilité ».
- $\Box(P) \equiv \neg\,(\Diamond(\neg\,P))$: permet de spécifier une propriété P pour tout sous-intervalle. Dans d'autres logiques temporelles, cet opérateur signifie *toujours*. Dans d'autres logiques modales il peut signifier la notion de « nécessité ».

Sémantique

Comme il s'agit d'une logique temporelle, les valeurs des variables ou des propositions peuvent changer selon le moment où elles sont observées. La précision d'ITL est relative aux intervalles de temps : les états qui ne sont valables qu'à un point du temps (intervalle de temps nul) ne sont donc pas observables. Pour parler des intervalles de temps, nous définissons les domaines suivants :

- $\mathbb{T}ime$, le domaine auquel appartiennent les unités de temps. Usuellement il s'agit du domaine des nombres réels $\mathbb{R}$
- $\mathbb{I}nterval \equiv \{[b,e] | b,e \in \mathbb{T}ime \wedge b \leq e\}$ le domaine des intervalles de temps.
- $\mathbb{B} \equiv \{\mathbf{true}, \mathbf{false}\}$ le domaine des booléens (nous reprenons la notation des atomes)
- $Iterme \equiv (terme \to \mathbb{T}ime) \to \mathbb{I}nterval \to terme \to \mathbb{T}ime$ le domaine des fonctions d'interprétation des termes

Nous définissons les fonctions d'interprétation des termes suivantes :

- $I_T \in Iterme$
- $I_F \in (Iterme) \to (terme \to \mathbb{T}ime) \to \mathbb{I}nterval \to formule \to \mathbb{B}$

Nous introduisons également la fonction de valuation suivante, qui à une variable associe une valeur du domaine temporel :

- $\mathcal{V}_T \in terme \to \mathbb{T}ime$

Deux valuations $\mathcal{V}_T$ et $\mathcal{V}'_T$ sont x-équivalentes pour une variable globale x si pour toute variable y différente de x, $\mathcal{V}_T(y) = \mathcal{V}'_T(y)$.

Les variables temporelles sont interprétées comme des fonctions de $\mathbb{I}nterval$ vers $\mathbb{T}ime$. Les variables globales sont interprétées en obtenant leur valeur grâce à la fonction de valuation mentionnée au-dessus.

La sémantique des termes d'ITL est présentée au tableau 2.2. Nous faisons l'amalgame entre les fonctions et leur interprétation : par exemple la fonction $+$ a son interprétation usuelle d'addition.

$$
\begin{array}{lcl}
I_T(\mathcal{V}_T,[b,e])(x) & = & x([b,e]) \\
I_T(\mathcal{V}_T,[b,e])(v) & = & \mathcal{V}_T(v) \\
I_T(\mathcal{V}_T,[b,e])(\ell) & = & \text{e-b} \\
I_T(\mathcal{V}_T,[b,e])(f(x_1,\ldots,x_n)) & = & f(I_T([b,e])(x_1),\ldots,I_T([b,e])(x_n))
\end{array}
$$

TAB. 2.2 – Sémantique des termes d'ITL

La sémantique des formules d'ITL est présentée au tableau 2.3. Comme avec les fonctions pour les termes, nous amalgamons les relations et leur interprétation. Nous faisons également d'autres abus de notations en amalgamant par exemple la disjonction d'ITL et la disjonction logique usuelle.

$$
\begin{array}{lcl}
I_F(I_T,\mathcal{V}_T,[b,e])(\mathbf{true}) & = & \mathbf{true} \\
I_F(I_T,\mathcal{V}_T,[b,e])(X) & = & X([b,e]), \\
I_F(I_T,\mathcal{V}_T,[b,e])(R(t_1,\ldots,t_n)) & = & R(c_1,\ldots,c_n) \text{ avec } c_i = I_T(\mathcal{V}_T,[b,e])(t_i) \\
I_F(I_T,\mathcal{V}_T,[b,e])(\neg P) & = & \neg\, I_F(I_T,\mathcal{V}_T,[b,e])(P) \\
I_F(I_T,\mathcal{V}_T,[b,e])(P_1 \vee P_2) & = & I_F(I_T,\mathcal{V}_T,[b,e])(P_1) \vee I_F(I_T,\mathcal{V}_T,[b,e])(P_2) \\
I_F(I_T,\mathcal{V}_T,[b,e])(\exists x.P) & = & I_F(I_T,\mathcal{V}'_T,[b,e])(P) \\
 & & \text{pour un } \mathcal{V}'_T \text{ } x\text{-équivalent à } \mathcal{V}_T \\
I_F(I_T,\mathcal{V}_T,[b,e])(P_1 {}^\frown P_2) & = & \exists k.(b \le k \wedge k \le e \\
 & & \wedge I_F(I_T,\mathcal{V}_T,[b,k])(P_1) \\
 & & \wedge I_F(I_T,\mathcal{V}_T,[k,e])(P_2))
\end{array}
$$

TAB. 2.3 – Sémantique des formules d'ITL

Notons que la notion de variable propositionnelle, dans la sémantique d'ITL, est représentée par une fonction d'un intervalle de temps vers les booléens. Cela correspond à la notation $X([b,e])$ du tableau 2.3. La raison en est que la variable propositionnelle peut représenter une valeur de vérité qui peut être dépendante du temps, de la même manière qu'un prédicat comme $\ell = 1$ est dépendant de l'intervalle dans lequel sa valeur de vérité est recherchée ($\ell = 1$ n'est vrai que si l'intervalle $[b,e]$ considéré est de longueur 1, soit $e-b=1$).

La condition de bord sur l'x-équivalence pour une formule de la forme $\exists x.P$ correspond au fait de trouver une valuation acceptable pour x telle que la formule soit vraie. Il s'agit de la signification usuelle du quantificateur existentiel. Nous remarquons que la quantification existentielle ne se fait que sur les variables globales, donc indépendantes du temps.

La sémantique d'ITL représente l'idée que l'on se fait naturellement des intervalles et de leur découpage. Il est pertinent ici de mentionner l'existence d'un formalisme très proche, dont la principale différence réside dans le fait que le point de découpage peut se situer *après* l'intervalle qu'il découpe. Si nous nous référons au tableau 2.3, cela signifie que la contrainte $k \le e$ disparaît. Cette logique est appelée SIL (logique d'intervalle signée) et est présentée en détail dans la thèse de Rasmussen[Ras02]. Ainsi, le $^\frown$ de ITL est dit *contractant* (pour « contracting ») puisqu'il n'agit que sur les sous-intervalles. En revanche $^\frown$ de SIL est *expansif* (pour « expanding ») parce que, le point de découpage pouvant se situer en-dehors (à droite) de l'intervalle, il permet d'exprimer des propriétés sur le futur.

En résumé, la conséquence de la sémantique d'ITL est que celle-ci contraint le raisonnement *à intervalle fermé*. La notion de vivacité reste néanmoins exprimable : en effet les intervalles, bien que fermés, peuvent être de longueur arbitraire. Une propriété qui serait vraie indépendamment de la longueur de l'intervalle courant pourra donc être vraie arbitrairement loin dans le futur, le futur étant pris par rapport au début de cet intervalle de longueur arbitraire.

Nous concluons cette présentation de la sémantique d'ITL par quelques exemples d'interprétation de formule. Pour simplifier la lecture, nous ne noterons que l'ensemble de valuations (sous la forme $\{x_i \mapsto v_i, \ldots\}$) et l'intervalle courant en préfixe du terme ou de la formule, les fonctions d'interprétation pouvant être déduites du contexte.

$$\begin{array}{ll} & \emptyset, [b,e], (\exists x.\ell = x) \\ = & \{x \mapsto ?\}, [b,e], (\ell = x) \\ = & (\{x \mapsto ?\}, [b,e], \ell) = (\{x \mapsto ?\}, [b,e], x) \\ = & e - b = ? \end{array}$$

Nous pouvons donc conclure que la valuation correcte pour x est $e-b$, et que la formule $\exists x.\ell = x$ est vraie.

$$\begin{array}{ll} & \emptyset, [b,e], (P \Rightarrow \mathbf{true}^\frown P^\frown \mathbf{true}) \\ = & \emptyset, [b,e], (\neg P \vee \mathbf{true}^\frown P^\frown \mathbf{true}) \\ = & \neg(P([b,e])) \vee (\exists k.b \leq k \wedge k \leq e \wedge (\exists m.k \leq m \wedge m \leq e.(\mathbf{true} \wedge P([b,e]) \wedge \mathbf{true}))) \end{array}$$

Il suffit de choisir $k = b$ et $m = e$ et la formule se réduit à $\neg(P([b,e])) \vee P([b,e])$, ce qui est vrai quelle que soit la valeur de la variable propositionnelle P sur $[b,e]$.

2.1.3 Système de preuve

Le système de preuve d'ITL est une extension du système de preuve de la logique des prédicats du premier ordre (avec une théorie arithmétique pour les nombres réels) avec des axiomes pour le chop $^\frown$ (découpage d'intervalle) et la variable spéciale ℓ. Cette extension n'est pas conservatrice en ce sens que l'instanciation de variables liées par un quantificateur universel possède des conditions d'utilisation supplémentaires.

Afin de définir quelques conditions de bord particulières sur certains axiomes du système de preuve, nous introduisons les termes suivants :

Flexibilité Une formule est *flexible* si elle contient une variable propositionnelle ou la variable spéciale ℓ

Rigidité Une formule est *rigide* si elle n'est pas flexible

sans coupure Une formule est sans coupure (*chop-free* dans la littérature du domaine) si elle ne contient pas le connecteur $^\frown$

libre pour... Ce terme est utilisé dans la littérature du domaine de la manière suivante : un terme t est *libre pour x* dans la formule P si x n'apparaît pas sous un quantificateur $\forall y$ ou $\exists y$, avec y étant une variable apparaissant dans t. Par exemple, soit la formule $\forall x.P(x) = \forall x.\exists y.y = x$. Le terme $y+1$ n'est pas libre pour x dans P : $P(y+1) = \exists y.y = y+1$. Cette formulation a pour but de pouvoir exprimer la notion de capture de variable.

Autrement dit, t est libre pour x dans P si le remplacement de x par t dans P ne capture aucune variable libre de t. Pour plus de clarté, nous utiliserons cette dernière dénomination. Le terme « libre pour... » n'était introduit ici que pour faire le lien avec la littérature du domaine.

Le tableau 2.4 présente l'axiomatique du système de preuve, où P, Q, R sont des propositions et x, y des variables du domaine $\mathbb{T}ime$.

$A0$	$l \geq 0$
$A1_l$	$((P^\frown Q) \wedge \neg\,(P^\frown R)) \Rightarrow (P^\frown (Q \wedge \neg\, R))$
$A1_r$	$((P^\frown Q) \wedge \neg\,(R^\frown Q)) \Rightarrow ((P \wedge \neg\, R)^\frown Q)$
$A2$	$(P^\frown Q)^\frown R \Leftrightarrow P^\frown (Q^\frown R)$
R_l	$P^\frown Q \Rightarrow P$ si P est rigide
R_r	$P^\frown Q \Rightarrow Q$ si Q est rigide
B_l	$((\exists x)P)^\frown Q \Rightarrow (\exists x)((P)^\frown Q)$ si x n'est pas libre dans Q
B_r	$(P)^\frown (\exists x)Q \Rightarrow (\exists x)((P)^\frown Q)$ si x n'est pas libre dans P
$L1_l$	$(\ell = x)^\frown P \Rightarrow \neg\,((\ell = x)^\frown \neg\, P)$
$L1_r$	$P^\frown (\ell = x) \Rightarrow \neg\,(\neg\, P^\frown (\ell = x))$
$L2$	$x \geq 0 \wedge y \geq 0 \Rightarrow ((\ell = x + y) \Leftrightarrow (\ell = x^\frown \ell = y))$
$L3_l$	$P \Rightarrow P^\frown \ell = 0$
$L3_r$	$P \Rightarrow \ell = 0^\frown P$
Q	$\forall x.P(x) \Rightarrow P(t)$ si P ne capture aucune variable libre de t et t est rigide ou P est sans coupure

TAB. 2.4 – Axiomatique d'ITL

Dans la littérature, le schéma d'axiome Q est parfois regroupé avec les règles d'inférence présentées dans le tableau 2.5. Certaines sont héritées de la logique des prédicats premier ordre, parfois avec quelques modifications, et d'autres sont spécifiques au connecteur modal $^\frown$.

MP	$\dfrac{P \qquad P \Rightarrow Q}{Q}$	Le *Modus Ponens* de la logique du premier ordre
G	$\dfrac{P}{\forall x.P}$	La généralisation
N_l	$\dfrac{P}{\neg\,(\neg\, P^\frown Q)}$	La nécessitation gauche
N_r	$\dfrac{Q}{\neg\,(P^\frown \neg\, Q)}$	La nécessitation droite
M_l	$\dfrac{P \Rightarrow Q}{P^\frown R \Rightarrow Q^\frown R}$	La monotonie gauche
M_r	$\dfrac{P \Rightarrow Q}{R^\frown P \Rightarrow R^\frown Q}$	La monotonie droite

TAB. 2.5 – Règles d'inférence d'ITL

Ce système de preuve est-il correct et complet ? Des preuves de correction (tout théorème du système de preuve est un théorème de la logique) pour divers systèmes de preuve existent, via Dutertre [Dut95b], Zhou et Hansen [ZH98] ou Moszkowski [Mos94]. À chaque fois ce système est prouvé complet pour un domaine de temps ayant les propriétés d'un groupe commutatif totalement ordonné, ce qui bien entendu inclut les nombres réels. Il est cependant possible de construire la logique d'intervalle sur les entiers naturels, par exemple.

Choisir un domaine discret discret pour ITL le rend beaucoup plus proche de logiques temporelles discrètes, et donc moins intéressant si les contraintes temporelles peuvent être directement exprimées dans ces logiques discrètes. C'est pour cette raison que le domaine des réels est choisi à chaque fois.

La limitation principale est qu'il n'existe aucune manière d'axiomatiser au premier ordre les réels, de manière à ce que le système décrive uniquement les nombres réels (Théorème de Löwenheim-Skolem). Donc un système de preuve pour ITL ou, plus loin, le calcul des durées, est au mieux complet par rapport à une axiomatisation particulière des nombres réels.

2.1.4 Exemples

Avant de montrer quelques exemples de formules et leur signification, prenons une convention : puisque la manipulation de formules définies sur un domaine de temps implique l'utilisation d'unités de ce domaine de temps, nous choisissons comme unité de temps la *seconde*. Cela simplifiera l'explication de formules.

Voici quelques exemples de formules ou théorèmes d'ITL :

- $$\ell = 1 ^\frown \ell = 1$$

  Cette formule spécifie un intervalle d'une seconde, suivie d'un autre intervalle d'une seconde. Grâce au système de preuve ou à la sémantique présentée juste avant, il sera possible d'en déduire que $\ell = 2$

- $$P ^\frown \ell = x \wedge Q ^\frown \ell = x \Rightarrow (P \wedge Q) ^\frown \ell = x$$

  Ce théorème d'ITL indique que deux formules P et Q vraie sous des contextes similaires, peuvent être rassemblées sous ce même contexte.

Maintenant que la logique d'intervalle a été introduite, nous pouvons présenter les extensions faites à celle-ci pour obtenir le calcul des durées.

2.2 Un calcul de durées

Nous présentons ici le calcul des durées tel qu'il a été défini, à partir de la logique d'intervalle. Après un historique sur la naissance de DC (pour « Duration Calculus »), nous présentons sa syntaxe, sa sémantique et le système de preuve associé.

2.2.1 Historique

Le calcul des durées est le résultat d'une action de recherche visant à explorer les propriétés alors mal connues des durées d'états en tant que mesures du comportement des systèmes temps-réels. Cette action de recherche a été initiée au sein du projet ProCoS [3] dans les BRA [4] du programme ESPRIT [5] [BHB+89]. Ces actions de recherche ont vite donné lieu à un premier document [ZHR91] décrivant le calcul des durées et ses propriétés. Ce document fut suivi par d'autres décrivant les propriétés intrinsèques de cette logique telles sa complétude et sa décidabilité [HZ97], des extensions (par exemple avec des intervalles de temps infinis [ZWR95]), ou le passage à l'ordre supérieur [ZGN99]. Un livre reprenant la majeure partie des résultats de recherche sur DC a été récemment publié [HZ04]. L'institution la plus active dans ce domaine est à ce jour l'institut international pour la technologie logicielle, affilié à l'université des Nations-Unies[DC] (UNU/IIST[6]).

2.2.2 Syntaxe et sémantique

Syntaxe

Les ajouts de DC à ITL se situent au niveau des *termes*, au tableau 2.6.

formule	: :=	atome | $\neg$ formule | formule $\vee$ formule | formule $\frown$ formule | $\exists x.$ formule
atome	: :=	**true** | X | R(terme,...,terme)
terme	: :=	x | ℓ | f(terme,...,terme) | $\int$ **etat**
etat	: :=	**0 | 1 |** S **| etat** $\vee$ **etat |** $\neg$ **etat**

TAB. 2.6 – Ajouts syntaxiques de DC

La classe des états correspond à la valeur de vérité de certains événements (ou de la relation logique entre eux) en un point précis du temps. L'opérateur de durée $\int$ permet d'exprimer alors la durée d'un événement. Nous faisons un abus de notation en dénotant $\vee$ la disjonction logique, et la disjonction d'états. Heureusement le contexte permet à chaque fois de savoir de laquelle il s'agit. Pour la même raison, il nous arrivera de noter 1 **true** et 0 **false**. Les autres connecteurs logiques usuels ont également leur équivalent pour les états, et se contruisent de la même manière à partir de la disjonction et de la négation.

Nous introduisons également les notations suivantes :

- $\lceil\!\lceil S \rceil\!\rceil \equiv \int S = \ell \wedge \ell > 0$ signifie que nous considérons un événement S ayant lieu pour tout l'intervalle non-nul de temps courant. Cette notation est fréquemment utilisée
- $\lceil\!\lceil \rceil\!\rceil \equiv \ell = 0$ signifie que nous considérons un point du temps

Précisons maintenant plus formellement la sémantique des états et de l'opérateur $\int$.

Sémantique

Nous définissons la fonction d'interprétation suivante pour les états :

- $I_S \in \mathbb{T}ime \rightarrow etat \rightarrow \{0,1\}$

La sémantique des ajouts de DC à ITL est indiquée au tableau 2.7.

[3] Provably Correct Systems, système prouvablement corrects

[4] Basic Research Action, action basique de recherche

[5] European Strategic Program for Research in Information Technology, programme européen stratéguique pour la recherche en informatique

[6] United Nations University/International Institute for Software Technology

$I_T([b,e])(\int S)$	$=$	$\int_b^e I_S(t)(S)dt$
$I_S(t)(0)$	$=$	0
$I_S(t)(1)$	$=$	1
$I_S(t)(S)$	$=$	$\begin{cases} 1 \text{ si S est vrai au temps t} \\ 0 \text{ sinon} \end{cases}$
$I_S(t)(S_1 \vee S_2)$	$=$	$max(I_S(t)(S_1))(I_S(t)(S_2))$
$I_S(t)(\neg S)$	$=$	$1 - I_S(t)(S)$

TAB. 2.7 – Sémantique des ajouts de DC à ITL

Nous pouvons nous rendre compte que par exemple, 0 correspond à l'événement qui n'arrive jamais, et 1 à l'événement qui arrive tout le temps, ou l'état qui est toujours vérifié. La sémantique de l'opérateur $\int$ justifie naturellement sa notation particulière.

Seulement, la notion d'intégration d'une fonction (ici, du temps) est liée à la continuité de cette fonction. DC fait donc une hypothèse très importante sur les états (vus comme des fonctions) : *les fonctions doivent être finiment variables*. Autrement dit, pour un intervalle de temps donné, un état ne doit pas varier indéfiniment, la fonction le représentant ne doit pas avoir un nombre infini de points de discontinuité. Cette hypothèse supplémentaire n'est pas handicapante en pratique, puisque les phénomènes modélisés avec DC ont rarement cette propriété de variabilité infinie.

L'exemple usuel de fonction infiniment variable est le suivant :

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

Cette sémantique permet de voir immédiatement que la durée de l'événement qui arrive toujours ($\int_b^e I_S(1)(S)dt$) est la même que la durée de l'intervalle courant. Cette relation entre $\int 1$ et ℓ, ainsi que d'autres, sont axiomatisées par le système de preuve présenté en section suivante.

2.2.3 Système de preuve

De la même manière qu'en section 2.1.3, nous séparons la présentation du système de preuve entre axiomes, au tableau 2.8 et règles d'inférence, au tableau 2.9.

Notons que pour ce système, la définition de la flexibilité (et par conséquent, de la rigidité) d'une formule évolue :
- Une formule est *flexible* si elle contient une variable propositionnelle, ou un terme flexible
- Un terme est flexible s'il s'agit de ℓ, ou de l'opérateur de durée $\int S$ avec S un état flexible
- Un état est flexible s'il ne se réduit pas à 0, car la seule valeur rigide que peut prendre $\int S$ est 0.

Les axiomes illustrent les concepts suivants :
- La relation entre l'opérateur de durée $\int$ et la longueur d'intervalle
- La relation entre durée d'événements et leurs propriétés logiques.
- Le fait que deux événements indiscernables logiquement ont les mêmes durées

L'axiome DCA4 sert à « factoriser » différemment deux événements pour isoler une possible relation logique entre eux et raisonner dessus par rapport à leurs durées. Cet axiome sert par

$DCA1$	$\int 0 = 0$
$DCA2$	$\int 1 = \ell$
$DCA3$	$\int S \geq 0$
$DCA4$	$\int S_1 + \int S_2 = \int (S_1 \vee S_2) + \int (S_1 \wedge S_2)$
$DCA5$	$\int S = x + y \Leftrightarrow (\int S = x)^\frown(\int S = y)$
$DCA6$	$\int S_1 = \int S_2$ si $S_1 \Leftrightarrow S_2$ en logique propositionnelle

TAB. 2.8 – Axiomes du calcul des durées

exemple à démontrer que $\int S + \int \neg S = \ell$, ce qui ne serait pas possible avec les autres axiomes. La figure 2.1 illustre à quoi correspond l'axiome DCA4.

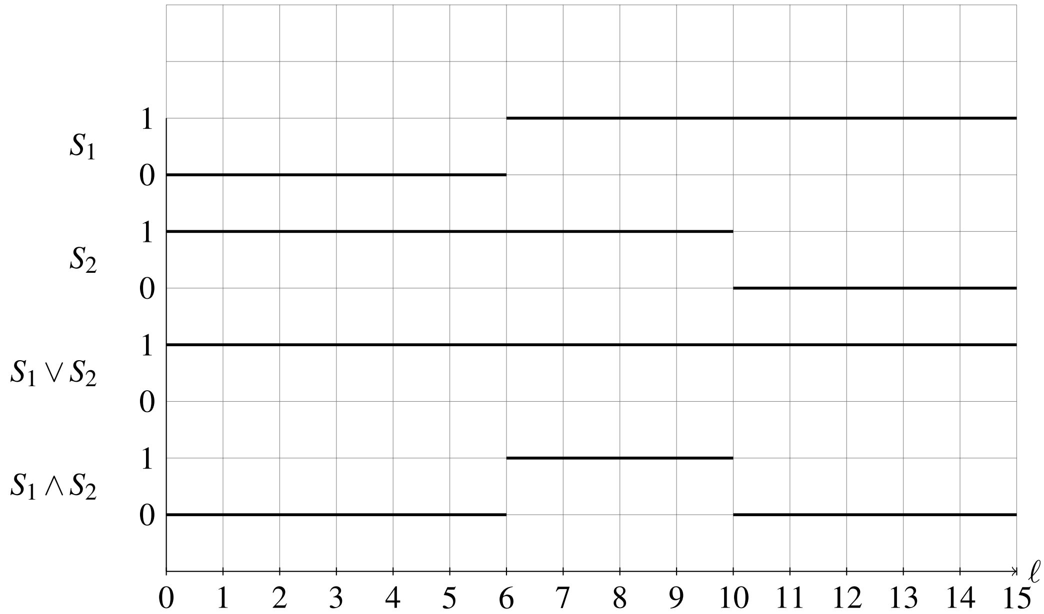

FIG. 2.1 – Illustration de l'axiome DCA4

L'état (ou événement) S_1 est vrai de 6 à 15, donc dure 9 secondes. L'état S_2 est vrai de 0 à 10, donc dure 10 secondes. La disjonction des deux dure 15 secondes (de 0 à 15) alors que la conjonction des deux dure 4 secondes (de 6 à 10). La somme des durées de S_1 et S_2 (9+10) est bien égale à la somme de leur disjonction et de leur conjonction (15+4).

Le tableau 2.9 introduit les règles d'inférence de DC.

Elles expriment une propriété d'induction sur le découpage des intervalles. Elles sont utilisées pour démontrer les propriétés qui se vérifient sur des intervalles quelconques. Un exemple de telle propriété est la formule $\phi \equiv \lceil\rceil \vee \mathbf{true}^\frown \lceil S \rceil \vee \mathbf{true}^\frown \lceil \neg S \rceil$. Cette formule se démontre en prenant pour $H(X)$ la formule $\Box(X \Rightarrow \phi)$.

Le principe d'induction est le suivant : si une propriété dépend d'un sous-intervalle, et que ce sous-intervalle peut être étendu arbitrairement en partant de la longueur nulle, alors cette propriété est vraie pour tout sous-intervalle. Donc ce sous-intervalle peut être remplacé par le sous-intervalle quelconque, **true**.

$$IRDC_l \quad \frac{H(\lceil\rceil) \qquad H(X) \vdash H((X \vee (X^\frown \lceil S_1 \rceil) \vee \ldots \vee (X^\frown \lceil S_n \rceil)))}{H(\mathbf{true})}$$

avec $S_1 \vee \ldots \vee S_n \Leftrightarrow$ **true**

$$IRDC_r \quad \frac{H(\lceil\rceil) \qquad H(X) \vdash H((X \vee (\lceil S_1 \rceil^\frown X) \vee \ldots \vee (\lceil S_n \rceil^\frown X)))}{H(\mathbf{true})}$$

avec $S_1 \vee \ldots \vee S_n \Leftrightarrow$ **true**

TAB. 2.9 – Règles d'inférence du calcul des durées

Quels sont les résultats connus sur ce système de preuve ? Hansen et Zhou [HZ04] démontrent que le système de preuve proposé est correct et complet par rapport à l'axiomatisation des réels utilisée. La décidabilité, en revanche, pose plus de problèmes.

Décidabilité [HZ04] indique quelques résultats sur la décidabilité ou l'indécidabilité de certains classes de formules du calcul des durées.

La grammaire BNF suivante montre un schéma de construction de formules décidables pour DC, que le domaine de temps choisi soit discret ou continu. Soient P une formule et S un état :

$$P \quad ::= \quad \lceil S \rceil | \neg P | P \vee P | P^\frown P$$

La preuve de décidabilité se fait par traduction vers un langage régulier, une formule étant satisfiable si sa traduction dans le langage en question est non-vide. Cette technique est notamment une illustration de la relation entre des sous-classes de DC, et les automates.

Lorsque la longueur d'intervalle intervient explicitement, l'indécidabilité est vite atteinte. Soient P une formule, S un état et k un nombre :

$$P \quad ::= \quad \ell = k | \lceil S \rceil | \neg P | P \vee P | P^\frown P$$

Cette grammaire montre par exemple une sous-classe de formules qui sont décidables pour une longueur spécifique d'intervalle k discret et un domaine temporel discret. En revanche, dans le cas continu, cette sous-classe devient indécidable. La raison en est que dans le cas discret, il est possible de revenir à la sous-classe indiquée plus haut, alors que dans le cas continu cette sous-classe permet de modéliser le problème de l'arrêt de machines à compteurs sous forme d'un problème de satisfaisabilité de formules de cette sous-classe.

Il y a d'autres sous-classes avec peu de règles de construction qui sont également démontrées indécidables, toujours dans [HZ04]. Soient P une formule, S un état et x une variable globale :

$$P \quad ::= \quad \int S = \int S | \neg P | P \vee P | P^\frown P$$

$$P \quad ::= \quad \lceil S \rceil | \ell = x | \neg P | P \vee P | P^\frown P | \exists x.P$$

Ces deux dernières sous-classes sont elles aussi indécidables.

2.2.4 Exemples

Vivacité

Bien que les intervalles soient fermés (du fait de la sémantique d'ITL), il est possible d'exprimer des propriétés de vivacité de type *leadsto*. Soit par exemple la formule suivante :

$$\Box(\llbracket E_1 \rrbracket ^\frown \mathbf{true} \Rightarrow \mathbf{true} ^\frown (\llbracket E_2 \rrbracket \wedge \ell \geq 1))$$

Cette formule signifie que l'activation de l'événement E_1, aussi courte soit-elle (mais non nulle, du fait de la construction $\llbracket \ldots \rrbracket$), déclenche l'activation de l'événement E_2 plus tard (possiblement en même temps, car **true** peut représenter un intervalle nul) pendant au moins une seconde. Comme la longueur de l'intervalle global n'est pas spécifiée, alors cette propriété est vérifiée dans tout l'intervalle de temps pendant lequel le système est surveillé. Ce type de formule est utilisé fréquemment pour exprimer des propriétés de vivacité de ce genre (comme dans l'exemple du brûleur à gaz plus bas).

Le brûleur à gaz

L'exemple classique indiqué dans la plupart des ouvrages décrivant DC est le brûleur à gaz. Un brûleur à gaz chauffe lorsqu'il est enflammé, ou ne fait rien, et alterne entre les deux états (pour maintenir une température dans un récipient, par exemple). Seulement, pour passer d'un état à l'autre, il doit s'écouler un peu de temps entre le début de l'envoi du gaz et la mise à feu du gaz pour assurer que le gaz peut être enflammé. Pendant ce temps où le gaz ne brûle pas, il *fuit*.

Une contrainte de bon fonctionnement peut être donc d'éviter que le gaz s'accumule, et d'autres experts (en mécanique des fluides, par exemple) peuvent déterminer que pour que cette contrainte soit vérifiée, il faut que le gaz ne fuie pas plus d'un vingtième du temps, pour des intervalles de temps plus grands qu'une minute.

Nous définissons donc les événements suivants :

- $Gaz(t)$: si le gaz s'écoule au temps t
- $Flamme(t)$: si une flamme est présente (détectée par des capteurs) au temps t
- $Fuite(t) \equiv Gaz(t) \wedge \neg\, Flamme(t)$: il y a une fuite si le gaz s'écoule mais n'est pas enflammé.

Avec ces définitions, la contrainte s'écrit :

$$\ell \geq 60 \Rightarrow 20 * \int Fuite \leq \ell$$

Maintenant, il faut concevoir le brûleur de telle manière qu'une conséquence de la conception soit le respect de cette contrainte. Par exemple, nous pouvons faire les décisions de conception suivantes :

- Pour toute période de temps où la contrainte est vérifiée, une fuite de gaz peut être détectée et stoppée en moins d'une seconde

  $$\Box(\llbracket Fuite \rrbracket \Rightarrow \ell \leq 1)$$

  Cette formule nous dit que pour tout sous-intervalle où le gaz fuit sans interruption, ce sous-intervalle dure moins d'une seconde.

– Nous pouvons vouloir éviter les fuites, même courtes, trop fréquentes. Donc par exemple, nous décidons qu'après une fuite, l'écoulement du gaz est stoppé pour 30 secondes au moins

$$\Box(\llbracket Fuite \rrbracket ^\frown \llbracket \neg\, Fuite \rrbracket ^\frown \llbracket Fuite \rrbracket \Rightarrow \ell \geq 30)$$

Cette formule nous dit que pour tout sous-intervalle où il y a une alternance entre deux fuites (le temps où il n'y a pas de fuite est non-nul), alors ce sous-intervalle dure plus de 30 secondes.
En effet, chaque côté de l'implication représente le même intervalle. Ce qui signifie que si une alternance existe, elle doit durer au moins 30 secondes. De plus, la formule est préfixée par l'opérateur $\Box$, ce qui signifie qu'elle doit être vraie pour tout sous-intervalle.
Cela induit que l'intervalle où a lieu l'alternance peut être aussi petit que possible, tant que l'alternance est présente, alors il doit durer plus que 30 secondes.

Pour vérifier que la conception du brûleur à gaz respecte ses contraintes de fonctionnement, nous devons donc vérifier que ses propriétés de conception permettent de déduire cette contrainte :

$$\begin{array}{ll} & \Box(\llbracket Fuite \rrbracket \Rightarrow \ell \leq 1) \\ & \wedge\Box(\llbracket Fuite \rrbracket ^\frown \llbracket \neg\, Fuite \rrbracket ^\frown \llbracket Fuite \rrbracket \Rightarrow \ell \geq 30) \\ \Rightarrow & \\ & \ell \geq 60 \Rightarrow 20 * \int Fuite \leq \ell \end{array}$$

Cette formule a été démontrée comme vraie en calcul des durées. Alternativement, observons également le fait que le brûleur à gaz où le gaz ne s'écoule jamais est également une mise en oeuvre correcte (mais peu utile en pratique) : il serait donc possible de rajouter une contrainte de vivacité (le brûleur fonctionne tout de même).

2.3 Calcul des durées avec itération

L'ajout d'un opérateur d'itération à DC a été opéré par Van Hung et Guelev [GH99], avec des résultats sur la complétude et la décidabilité du système obtenu. DC* (DC avec itération) peut être vu de deux manières :

– En tant que la logique d'intervalle originelle (i.e. avec itération), augmentée avec un opérateur de durée
– En tant que DC augmenté avec un connecteur d'itération.

Nous avons choisi de présenter la seconde vision car, comme nous le verrons en section 2.3.2, l'ajout de l'opérateur d'itération se reflète dans le système de preuve par l'ajout d'une règle d'inférence liée à la notion d'état.

Précisons en premier quel est cet opérateur, ainsi que sa sémantique.

2.3.1 Syntaxe et sémantique

Le connecteur étoile * est ajouté aux formules, comme illustré au tableau 2.10.

La signification de cet opérateur correspond intuitivement à l'étoile que l'on rencontre dans d'autres logiques temporelles : P^* est vraie si P est vrai un nombre arbitraire de fois en suivant. Dans le cas de DC, c'est de découpage d'intervalle qu'il s'agit.

formule	: :=	atome | ¬ formule | formule ∨ formule | formule ⌢ formule
		| ∃x. formule | **formule***
atome	: :=	**true** | X | R(terme,...,terme)
terme	: :=	x | ℓ | f(terme,...,terme) | ∫etat
etat	: :=	0 | 1 | S | etat ∨ etat | ¬ etat

TAB. 2.10 – Syntaxe de DC*

$$I_F(I_T,\mathcal{V}_T,[b,e])(P^*) \quad = \quad \begin{cases} b=e \\ \text{ ou } \quad \exists t_1,\ldots,t_n, t_1=b, t_n=e, t_1<t_2<\ldots<t_n, \forall i\in 1..n-1, \\ \qquad I_F((I_T,\mathcal{V}_T,[t_i,t_{i+1}])(P) \end{cases}$$

TAB. 2.11 – Sémantique de l'itération

Usuellement, l'opérateur d'itération n'est pas défini tel quel, mais par le biais d'un opérateur d'itération bornée, défini de la manière suivante :
- $P^0 = \ell = 0$
- $P^k = \underbrace{P^\frown \ldots {}^\frown P}_{k \text{ fois}}$, soit encore $P^k = P^\frown P^{k-1}$

À partir de cette itération bornée, la définition de l'itération est : $P^* = \bigcup_{n\in\mathbb{N}} P^n$. L'intérêt de cette définition est de permettre de spécifier la règle d'inférence pour DC* présentée en section suivante.

2.3.2 Système de preuve

Les axiomes ajoutés à DC ont deux objectifs :
- Reprendre la sémantique de l'opérateur d'itération (DC_1^*, DC_2^*)
- Spécifier la relation entre cet opérateur et les autres connecteurs logiques (DC_3^*).

Ces axiomes sont présentés au tableau 2.12.

$$\begin{array}{ll} DC_1^* & \ell = 0 \Rightarrow P^* \\ DC_2^* & (P^{*\frown}P) \Rightarrow P^* \\ DC_3^* & \begin{cases} ((P^* \wedge Q)^\frown \mathbf{true}) \Rightarrow ((Q\wedge \ell=0)^\frown \mathbf{true}) \\ \vee(((P^*\wedge \neg Q)^\frown P)\wedge Q)^\frown \mathbf{true} \end{cases} \end{array}$$

TAB. 2.12 – Axiomes d'itération

Les règles DC_1^* et DC_2^* sont à rapprocher de la définition de la répétition :
- DC_1^* : si une formule répétée 0 fois est vraie, alors il existe bien une répétition de cette formule qui est vérifiée.
- DC_2^* : s'il existe un découpage entre une formule et sa répétition qui est vérifié, alors il existe bien une répétition de cette formule qui est vérifiée (l'ancienne avec le découpage supplémentaire)

L'intuition de DC_3^* est plus difficile à saisir. Soit Q le préfixe d'un intervalle, préfixe qui peut être découpé en plusieurs sous-intervalles vérifiant chacun P. Alors :

- Soit Q est vérifié dès le début du sous-intervalle (et donc vérifié pour l'intervalle initial de longueur nulle)
- Soit il y a un plus petit découpage tel que Q n'est pas vérifié sur celui-ci.

Notons que ces axiomes proviennent d'une traduction d'axiomes similaires provenant d'une logique proche, la logique dynamique propositionnelle [AGM92]. Plus de détails sur cette traduction se trouvent dans [Gue98].

La règle d'inférence de DC* est à rapprocher de celles de DC : si une propriété est vraie pour un intervalle découpé arbitrairement en des sous-intervalles plausibles (un certain état S est toujours vrai ou toujours faux dans chacun de ces sous-intervalles), alors la propriété est vraie quel que soit l'intervalle de temps. Cette règle d'inférence est présentée au tableau 2.13.

ω	$\dfrac{\forall k < \omega . [(\lceil\!\lceil S \rceil\!\rceil \vee \lceil\!\lceil \neg S \rceil\!\rceil)^k / A]P}{[\mathbf{true}/A]P}$	ω correspond à l'infini des entiers naturels

TAB. 2.13 – Règles d'inférence de DC*

Les règles d'inférence de DC (tableau 2.9) se basaient sur la croissance fournie par un seul découpage. Cette règle-ci permet un raisonnement plus direct sur un découpage arbitraire, grâce à l'itération bornée.

2.3.3 Intérêt

Ce système de preuve est démontré comme étant correct. En revanche, la complétude est uniquement démontrée pour une classe de formules appelée *simples*, i.e. si une formule appartient à cette classe, alors elle est démontrable dans ce système de preuve. Ces formules *simples* correspondent en pratique à des spécifications de systèmes, ce qui les rend si intéressantes. Plus de détails se trouvent dans l'article de Van Hung et Guelev[GH99].

Dans le même ordre d'idées, les formules dites *simples* jouissent d'une autre propriété intéressante : leur satisfaisabilité est décidable, à condition que les inégalités avec ℓ soient des entiers. Ce résultat, indiqué dans [GH99], est une conséquence de l'équivalence des formules simples avec un autre formalisme appelé *expressions rationnelles temporisées*[7], que nous ne détaillerons pas ici.

Toutes ces propriétés font de DC* une logique intéressante pour exprimer la sémantique des langages de programmation et/ou de spécification.

2.3.4 Exemples

L'exemple suivant de formule avec itération spécifie que tant qu'un événement est vrai ($\lceil\!\lceil S \rceil\!\rceil$) alors une certaine variable x est positive ou nulle, et que S est vrai pour un nombre arbitraire d'intervalles ($\lceil\!\lceil S \rceil\!\rceil^*$). Alors il est possible d'en déduire que x est toujours positif ou nul :

$$(\lceil\!\lceil S \rceil\!\rceil^* \wedge \lceil\!\lceil S \rceil\!\rceil \Rightarrow x \geq 0) \Rightarrow \Box(x \geq 0)$$

Cette formule est représentative des obligations de preuve pour des boucles : $\lceil\!\lceil S \rceil\!\rceil^*$ signifie qu'un nombre arbitraire de tours de boucle déclenchant ou maintenant l'événement S a lieu.

[7]Timed Regular Expressions

$\llbracket S \rrbracket \Rightarrow x \geq 0$ spécifie une relation entre cet événement S et une variable x. Ces deux hypothèses nous permettent de spécifier une propriété de sûreté (x n'est jamais négatif) pouvant correspondre à un invariant de boucle.

DC* est efficace pour les répétitions et les intervalles de temps, mais ne permet toujours pas d'exprimer ce qu'il se passe en des points du temps. La prochaine section présente un formalisme qui le permet.

2.4 Calcul des durées avec itération, faiblement monotone

Nous avons constaté dans la section précédente, que DC* avait des propriétés suffisamment intéressantes pour permettre d'exprimer la sémantique des langages de programmation. Cependant, qu'en est-il des programmes concurrents et de leur composition ? Il arrive fréquemment qu'un modèle de programme concurrent définisse des points de synchronisation : comment exprimer alors ce qu'il se passe en ces points ? Nous avons à disposition deux solutions :

- Utiliser effectivement DC* pour exprimer cela : cependant, en pratique, les points de synchronisation d'un programme concurrent sont des endroits ou peu d'actions sont effectuées, la majorité du fonctionnement se passant dans les différents processus
- Utiliser une logique qui sépare deux domaines : un domaine où les contraintes temporelles sont suffisamment importantes pour nécessiter d'être prises en compte, et un domaine ou les actions sont suffisamment rapides en regard des contraintes temporelles pour être assimilées à des actions immédiates.

Cette dernière solution s'appelle l'hypothèse du vrai synchronisme[8] : cette hypothèse spécifie que certaines actions d'un programme temps-réel sont suffisamment rapides en regard des délais ou attentes du programme pour que négliger leur temps d'exécution n'a pas d'impact sur le modèle du programme. Plus concrètement, si le programme est un système de surveillance d'un capteur, par exemple, on peut considérer que l'envoi des informations est immédiat par rapport à la période de surveillance du capteur. L'envoi des informations peut être compris comme la mise à jour d'une variable, i.e. une simple affectation, par exemple. Dans les langages de programmation usuels, l'hypothèse du vrai synchronisme s'applique aux simples affectations.

Partant de l'utilisation de cette hypothèse, il y aura donc des points du temps où plusieurs actions auront lieu (plusieurs affectations en séquence, par exemple). C'est de ce constat qu'est né le calcul des durées faiblement monotone, introduit par Pandya et Van Hung [HP98]. Un opérateur d'itération y a été plus tard ajouté par Siewe et Van Hung [SH01]. Cette logique permet d'exprimer des propriétés selon leurs contraintes temporelles, mais aussi séquentielle : ainsi il est possible de spécifier des propriétés limitées à un point précis du temps.

Contrairement aux sections précédentes, où nous présentions les calculs par ajouts successifs de constructions, nous présentons ici WDC* (que nous appellerons parfois simplement WDC), en commençant par la syntaxe et la sémantique, puis son système de preuve.

2.4.1 Syntaxe et Sémantique

La syntaxe de WDC* change peu par rapport à DC*. Les changements sont plus visibles au niveau de la sémantique.

[8] true synchrony hypothesis

Syntaxe

La syntaxe de WDC* est indiquée au tableau 2.14.

formule	: :=	$\lceil etat \rceil^0$ | atome | $\neg$ formule | formule $\vee$ formule | formule $\frown$ formule | $\exists x$. formule | formule*
atome	: :=	**true** | X | R(réel,...,réel) | R(entier,...,entier)
reel	: :=	x | ℓ | f(reel,...,reel) | $\int$état
entier	: :=	**k | η | f(entier,...,entier)**
etat	: :=	0 | 1 | S | etat $\vee$ etat | $\neg$ etat

TAB. 2.14 – Syntaxe de WDC*

Ici, les relations R sont représentées par la même lettre, mais s'appliquent à deux domaines différents : aux entiers, pour le domaine de η, et aux réels, pour le domaine de ℓ. Le contexte permet de déterminer facilement de quelle relation il s'agit.

Les deux principaux changements par rapport à DC* sont l'apparition d'une nouvelle variable spéciale η (modale, elle aussi), et de la possibilité d'exprimer qu'un événement S est vrai en un point du temps $\lceil S \rceil^0$.

La variable η va servir à spécifier une séparation entre deux actions dans une séquence. De la même manière que ℓ sépare deux points du temps de manière continue (si les nombres réels sont choisis comme domaine), η marque le nombre d'actions séparant deux actions particulières dans une séquence, d'où l'utilisation de nombres entiers (η est une variable discrète). Pandya [HP98] parle de séparation de *phases*. Deux phases successives seront donc séparées par $\eta = 1$, et pourront avoir lieu aussi bien en un point du temps qu'au long d'un intervalle de temps.

La construction $\lceil S \rceil^0$ est le plus petit intervalle de temps et de phase où il est possible d'exprimer la validité de l'événement S : il s'agit de l'intervalle de temps nul et de phase nulle, c'est-à-dire un point unique du temps. La notation du ^{0} en exposant signifie que la phase est nulle. Les crochets d'intervalle permettent de distinguer que S est bien un état.

Sémantique

Puisque désormais, en plus des intervalles, il faut compter les phases, le domaine temporel est changé : il est changé en un domaine qui fait la distinction entre un domaine continu pour les intervalles, et un domaine discret pour les phases. Nous noterons le domaine continu $\mathbb{R}$ et le domaine discret $\mathbb{N}$. Nous appellerons le domaine continu *macro-temps* et le domaine discret *micro-temps*.

Nous introduisons les définitions suivantes (provenant de [SH01]) :

- $\mathbb{T}ime \subset \mathbb{R} \times \mathbb{N}$
- $\mathbb{W}time = (\mathbb{T}ime, \ll)$
- $\ll$ est l'ordre lexicographique sur $\mathbb{W}time$: $(t_1, i_1) \ll (t_2, i_2)$ si $\begin{cases} t_1 < t_2 \\ \text{ou } t_1 = t_2 \wedge i_1 < i_2 \end{cases}$
- Les opérateurs de projection : $\begin{cases} \pi_1 \in \mathbb{T}ime \to \mathbb{R}.\pi_1(T) = \{t | (t,i) \in T\} \\ \pi_2 \in \mathbb{T}ime \to \mathbb{N}.\pi_2(T) = \{i | (t,i) \in T\} \end{cases}$

Des hypothèses supplémentaires sont faites sur l'ordre de $\mathbb{W}time$:

Monotonie de phase $t_1 < t_2 \wedge (t_1, i_1) \in \mathbb{T}ime \wedge (t_2, i_2) \in \mathbb{T}ime \Rightarrow i_1 \leq i_2$

Progrès Pour une formule donnée, seul le macro-temps peut augmenter indéfiniment :
$\pi_1(\mathbb{T}ime) = \mathbb{R}$
Il existe une autre possibilité mentionnée dans [HP98] où le micro-temps peut augmenter indéfiniment : cela autoriserait les formules où l'état du système est bloqué en un point du temps, par exemple.

Fermeture du passé $\pi_2(\mathbb{T}ime)$ est fermé à gauche (fermeture inférieure) pour $\ll$ sur $\mathbb{N}$. Cela signifie que pour arriver à une phase donnée, il faut être passé par toutes les phases précédentes.

Les états peuvent donc désormais changer de valeur plusieurs fois en un même point du temps. Le principe des phases est de pouvoir observer des intervalles de temps où un état est stable, que cet intervalle de temps soit nul ou non. Pour cela, [HP98] introduit une hypothèse de *stabilité de phase*. Soit la fonction $Per \in \mathcal{P}(\mathbb{R} \times \mathbb{N}) \times \mathbb{N} \to \mathcal{P}(\mathbb{R})$ définie par : $Per(\mathbb{T}ime, i) = \{t | (t,i) \in \mathbb{T}ime\}$.

Per représente la *période* d'une phase, c'est-à-dire l'intervalle de macro-temps associée à cette phase. Elle est utilisée pour spécifier l'hypothèse de stabilité de phase qui suit. Soit la fonction d'interprétation des états $I_S \in \mathbb{T}ime \to etat \to \{0,1\}$, et S un état :

$$\forall i \in \pi_2(\mathbb{T}ime), \forall b,e \in Per(\mathbb{T}ime,i), I_S(b,i)(S) = I_S(e,i)(S)$$

Cette hypothèse signifie que si un état change, alors les deux changements d'états se trouveront dans des phases différentes. Cette hypothèse permet d'avoir une granularité très fine dans l'expression des propriétés.

Maintenant que le problème des changements d'états est traité, il faut adapter cette fonction θ pour qu'elle puisse être utilisée par l'opérateur d'intégration $\int$. Nous ajoutons donc la fonction partielle $I_{CS} \in \mathbb{R} \to etat \to \{0,1\}$:

$$I_{CS}(t)(S) = \begin{cases} I_S(t,i)(S) \text{ si } \{i|(t,i) \in \mathbb{T}ime\} \text{ est un singleton} \\ \perp \text{ sinon} \end{cases}$$

Cela signifie que nous pourrons faire une intégration sur un intervalle de temps, à la condition que la phase ne change pas pendant cet intervalle de temps. Armés de ces nouvelles définitions, nous pouvons maintenant définir la sémantique de WDC*. Soient les domaines et les fonctions d'interprétation suivants, modifiés par rapport à la sémantique de DC* :

- $\mathbb{I}nterval \equiv \{[b,e] | b,e \in \mathbb{T}ime \wedge b \ll e\}$ le domaine des intervalles de temps. Ceux-ci sont désormais définis à la fois sur les intervalles de macro-temps et les phases
- $I_S \in \mathbb{T}ime \to etat \to \{0,1\}$ définie plus haut
- $I_{CS} \in \mathbb{R} \to etat \to \{0,1\}$ définie plus haut
- $\mathcal{V}_T \in terme \to \mathbb{R}$ une fonction de valuation
- $Iterme \equiv (terme \to \mathbb{R}) \to \mathbb{I}nterval \to terme \to \mathbb{R}$ le domaine des fonctions d'interprétation des termes
- $I_T \in Iterme$
- $I_F \in (Iterme) \to (terme \to \mathbb{R}) \to \mathbb{I}nterval \to formule \to \mathbb{B}$

La sémantique des formules de WDC* est présentée au tableau 2.15.

Cette sémantique illustre l'existence de deux domaines orthogonaux du temps :

- Un temps dense, ou macro-temps (modélisé par $\mathbb{R}$) : ℓ représente un intervalle de temps, i.e. la distance entre deux points du temps

$I_S(t)(0)$	$=$	0
$I_S(t)(1)$	$=$	1
$I_S(t)(S)$	$=$	$\begin{cases} 1 \text{ si S est vrai au temps t} \\ 0 \text{ sinon} \end{cases}$
$I_S(t)(S_1 \vee S_2)$	$=$	$max(I_S(t)(S_1))(I_S(t)(S_2))$
$I_S(t)(\neg S)$	$=$	$1 - I_S(t)(S)$
$I_T(\mathcal{V}_T,[b,e])(x)$	$=$	$x([b,e])$
$I_T(\mathcal{V}_T,[b,e])(v)$	$=$	$\mathcal{V}_T(v)$
$I_T(\mathcal{V}_T,[b,e])(\ell)$	$=$	$\pi_1(e)-\pi_1(b)$
$I_T(\mathcal{V}_T,[b,e])(\eta)$	$=$	$\pi_2(e)-\pi_2(b)$
$I_T(\mathcal{V}_T,[b,e])(f(x_1,\ldots,x_n))$	$=$	$f(I_T(\mathcal{V}_T,[b,e])(x_1),\ldots,I_T(\mathcal{V}_T,[b,e])(x_n))$
$I_T(\mathcal{V}_T,[b,e])(\int S)$	$=$	$\int_{\pi_1(b)}^{\pi_1(e)} I_{CS}(t)(S)dt$
$I_F(I_T,\mathcal{V}_T,[b,e])(\lceil S \rceil^0)$	$=$	$I_S(b)(S)$ si $b=e$
$I_F(I_T,\mathcal{V}_T,[b,e])(\textbf{true})$	$=$	**true**
$I_F(I_T,\mathcal{V}_T,[b,e])(X)$	$=$	$X([b,e])$,
$I_F(I_T,\mathcal{V}_T,[b,e])(R(t_1,\ldots,t_n))$	$=$	$R(c_1,\ldots,c_n)$ avec $c_i = I_T(\mathcal{V}_T,[b,e])(t_i)$
$I_F(I_T,\mathcal{V}_T,[b,e])(\neg P)$	$=$	$\neg\ I_F(I_T,\mathcal{V}_T,[b,e])(P)$
$I_F(I_T,\mathcal{V}_T,[b,e])(P_1 \vee P_2)$	$=$	$I_F(I_T,\mathcal{V}_T,[b,e])(P_1) \vee I_F(I_T,\mathcal{V}_T,[b,e])(P_2)$
$I_F(I_T,\mathcal{V}_T,[b,e])(\exists x.P)$	$=$	$I_F(I_T,\mathcal{V'}_T,[b,e])(P)$ pour un $\mathcal{V'}_T$ x-équivalent à $\mathcal{V}_T$
$I_F(I_T,\mathcal{V}_T,[b,e])(P_1 \frown P_2)$	$=$	$\exists k.b \leq k \wedge k \leq e$ $\wedge I_F(I_T,\mathcal{V}_T,[b,k])(P_1)$ $\wedge I_F(I_T,\mathcal{V}_T,[k,e])(P_2)$
$I_F(I_T,\mathcal{V}_T,[b,e])(P^*)$	$=$	$\begin{cases} b=e \\ \text{ou} \quad \exists t_1\ldots,t_n, t_1=b, t_n=e, t_1<t_2<\ldots<t_n, \\ \qquad \forall i \in 1..n-1, I_F(I_T,\mathcal{V}_T,[t_i,t_{i+1}])(P) \end{cases}$

TAB. 2.15 – Sémantique de WDC*

– Un temps « logique », ou micro-temps (modélisé par $\mathbb{N}$) : η représente un nombre de phases, i.e. le nombre de phases d'état du système entre deux phases données.

Cela permet donc d'exprimer des évolutions du système imperceptibles temporellement, i.e. qui peuvent avoir lieu en un point donné du macro-temps.

Notations additionnelles

Nous introduisons également les définitions et notations suivantes, qui permettent d'affiner la spécification de propriétés en macro et micro-temps, au tableau 2.16. Ces notations seront beaucoup utilisées lors de la définition de la sémantique temporelle des substitutions B du chapitre 4.

Pt_i	$\lceil 1 \rceil^0$	une phase réduite à un point de temps (l'intervalle et la phase sont nuls)
Ext_i	$\neg\, Pt_i$	une évolution (de macro-temps ou de micro-temps)
Pt_l	$\ell = 0$	un point du temps (pouvant contenir plusieurs phases)
Ext_m	$\neg\, Pt_l$	un intervalle de temps (pouvant contenir plusieurs phases)
$Unit$	$Ext_i \wedge \neg\, (Ext_i \frown Ext_i)$	une et une seule phase, de macro-temps nul
$\lceil P \rceil$	$\neg\, (Ext_i \frown \lceil \neg\, P \rceil^0 \frown Ext_i)$	P est vrai pour toute cette phase strictement
$\lceil P \rceil^1$	$\lceil P \rceil^0 \frown Unit$	P est vrai au début de cette phase
$\lceil\!\lceil P \rceil$	$\lceil P \rceil^0 \frown \lceil P \rceil$	le préfixe de cette phase voit P vrai
$\lceil\!\lceil P \rceil^-$	$\lceil\!\lceil P \rceil \vee \lceil\rceil$	le préfixe de cette phase voit P vrai, ou bien l'intervalle (de phase) est nul, i.e. la phase est inexistante
$\lceil\!\lceil P \rceil^+$	$\lceil\!\lceil P \rceil \vee \lceil P \rceil^0$	le préfixe de cette phase est vraie, ou alors cette phase est réduite à un point du macro-temps
$\lceil\!\lceil P \rceil\!\rceil$	$\lceil P \rceil^0 \frown \lceil P \rceil \frown \lceil P \rceil^0$	toute cette phase voit P vrai, y compris au début et à la fin
$\lceil\!\lceil P \rceil\!\rceil^-$	$\lceil\!\lceil P \rceil\!\rceil \vee \lceil\rceil$	toute cette phase voit P vrai, ou cette phase est inexistante
$\lceil\!\lceil P \rceil\!\rceil^+$	$\lceil\!\lceil P \rceil\!\rceil \vee \lceil P \rceil^0$	toute cette phase voit P vrai, ou bien est réduite à un point du macro-temps

TAB. 2.16 – Notations additionnelles pour WDC*

Quelques commentaires additionnels sur le tableau 2.16 peuvent aider à la compréhension de l'utilité de certaines de ces définitions. Supposons un intervalle $[b,e] \in \mathbb{I}nterval$:

– $Unit$ signifie effectivement une et une seule phase de macro-temps nul car si nous l'expansons selon la sémantique de WDC*, nous obtenons :

$$b \neq e \wedge \neg\, (\exists k \in [b,e], b \neq k \wedge k \neq e)$$
$$\Rightarrow \begin{cases} (\pi_1(b) \neq \pi_1(e) \vee \pi_2(b) \neq \pi_2(e)) \\ \wedge \forall k \in [b,e], \begin{pmatrix} \pi_1(b) = \pi_1(k) \wedge \pi_2(b) = \pi_2(k) \\ \vee \pi_1(k) = \pi_1(e) \wedge \pi_2(k) = \pi_2(e) \end{pmatrix} \end{cases}$$

Ensuite faisons un rapide raisonnement par cas informel :

– Si $\pi_1(b) = \pi_1(e)$: alors il faut que pour tout point du macro-temps dans $[\pi_2(b), \pi_2(e)]$ soit ou bien égal à $\pi_2(b)$, ou bien égal à $\pi_2(e)$. Ce n'est possible que si $\pi_2(b) = \pi_2(e)$, mais cela rendrait la formule fausse

  - Si $\pi_1(b) <= \pi_1(e)+2$: cette fois il est possible de trouver un k ($= \pi_1(b)+1$, par exemple) qui ne soit égal à aucun des bords d'intervalle de phase, donc ici aussi la formule est fausse.
  - Reste $\pi_1(b)+1 = \pi_1(e)$, i.e. il y a exactement une phase :
    - Si $\pi_2(b) = \pi_2(e)$: alors effectivement $\pi_2(k)$ sera égal aux deux extrémités de l'intervalle de macro-temps, donc la formule est vraie
    - Si $\pi_2(b) \neq \pi_2(e)$: d'après la formule il faut absolument que dans tous les cas, $\pi_2(k)$ soit égal à l'une des deux extrémités. Seulement la densité du domaine de macro-temps l'interdit. Donc ce cas est impossible.

  Comme nous le pouvons voir, l'interprétation *Unit* correspond bien à la description que nous en avons faite.
- De la même manière, $\lceil P \rceil$ a pour interprétation : $\forall k, b < k \wedge k < e \Rightarrow \theta(P)(k)$ ce qui signifie que P est vrai *à l'intérieur* de l'intervalle $[b,e]$, c'est-à-dire qu'on ne sait pas s'il est vrai aux extrémités.

2.4.2 Système de preuve

L'introduction d'une nouvelle variable η déterminant les intervalles de micro-temps fait que désormais chaque intervalle est déterminé par au moins ℓ et η. Certains axiomes ne changent pas (ceux qui concernent uniquement $\frown$, par exemple), d'autres sont adaptés ou ajoutés pour refléter l'introduction de η dans la logique. Ils sont illustrés au tableau 2.17.

Les axiomes de DC où ℓ seul était présent sont adaptés pour prendre également en compte l'intervalle de phase η ($WDC_1^*, L1_l$ par exemple). Il y a également duplication des axiomes décrivant ℓ, pour η, puisque celui-ci jouit de propriétés similaires ($L2_\eta$ par exemple). L'axiome supplémentaire AXM permet de donner des informations sur un intervalle de macro-temps nul : en effet celui-ci peut contenir plusieurs phases discrètes de micro-temps (tout sous-intervalle d'un intervalle $\ell = 0$ correspond soit à une absence de phase, soit à un sous-intervalle de macro-temps nul qui commence et termine par une phase).

Les règles d'inférence de ce système sont indiquées au tableau 2.18. Chacune des règles $IRWDC*_r$ de ce tableau possède une règle symétrique $IRWDC*_l$. De la même manière que $IRWDC1_r$ définit l'induction sur le sous-intervalle droit, $IRWDC1_l$ définit l'induction sur le sous-intervalle gauche. Il en va de même pour $IRWDC2_r$ et $IRWDC3_r$.

Sans surprise, les règles d'inférence définissent plusieurs styles d'induction selon les différent cas : intervalle de macro-temps non-nuls, phases, et phases sur un intervalle de macro-temps nul.

Nous allons voir dans la section suivante quelle est l'utilité d'un système logique aussi complexe.

2.4.3 Intérêt

Puisque WDC* permet d'exprimer des propriétés aux points d'intervalle, nous pouvons l'utiliser comme sémantique d'un langage de programmation, comme nous l'avions indiqué en début de section. Des exemples appliqués à un langage proposant la concurrence avec partage de variables, ou d'une version temporisée de CSP, se trouvent dans [HP98].

$A0$	$\ell \geq 0 \wedge \eta \geq 0$
$A1_l$	$((P^\frown Q) \wedge \neg (P^\frown R)) \Rightarrow (P^\frown (Q \wedge \neg R))$
$A1_r$	$((P^\frown Q) \wedge \neg (R^\frown Q)) \Rightarrow ((P \wedge \neg R)^\frown Q)$
$A2$	$(P^\frown Q)^\frown R \Leftrightarrow P^\frown (Q^\frown R)$
R_l	$P^\frown Q \Rightarrow P$ si P est rigide
R_r	$P^\frown Q \Rightarrow Q$ si Q est rigide
B_l	$((\exists x)P)^\frown Q \Rightarrow (\exists x)((P)^\frown Q)$ si x n'est pas libre dans Q
B_r	$(P)^\frown (\exists x)Q \Rightarrow (\exists x)((P)^\frown Q)$ si x n'est pas libre dans P
$Point$	$Pt_i \Leftrightarrow \eta = 0 \wedge \ell = 0$
$L1_l$	$(\ell = x \wedge \eta = k)^\frown P \Rightarrow \neg ((\ell = x \wedge \eta = k)^\frown \neg P)$
$L1_r$	$P^\frown (\ell = x \wedge \eta = k) \Rightarrow \neg (\neg P^\frown (\ell = x \wedge \eta = k))$
$L2_\ell$	$x \geq 0 \wedge y \geq 0 \Rightarrow ((\ell = x + y) \Leftrightarrow (\ell = x^\frown \ell = y))$
$L2_\eta$	$i \geq 0 \wedge j \geq 0 \Rightarrow ((\eta = i + j) \Leftrightarrow (\eta = i^\frown \eta = j))$
$L3_l$	$P \Rightarrow P^\frown Pt_i$
$L3_r$	$P \Rightarrow Pt_i{}^\frown P$
AXM	$\ell = 0 \Rightarrow \Box(Pt_i \vee ((Unit^\frown \mathbf{true}) \wedge (\mathbf{true}^\frown Unit)))$
Q	$\forall x.P(x) \Rightarrow P(t)$ si t est libre pour x dans P, et t est rigide ou P sans coupure
WDC_1^*	$Pt_i \Rightarrow P^*$
WDC_2^*	$(P^{*\frown} P) \Rightarrow P^*$
WDC_3^*	$\left\{ \begin{array}{l} (P^* \wedge Q^\frown \mathbf{true}) \Rightarrow Q \wedge (Pt_i{}^\frown \mathbf{true}) \\ ((P^* \wedge \neg Q^\frown P) \wedge Q)^\frown \mathbf{true} \end{array} \right.$

TAB. 2.17 – Axiomatique de WDC*

Les nombreux axiomes et règles de WDC∗ le rendent difficile à implémenter dans un outil de preuve, en vue de prouver la correction d'un programme par exemple. Posons-nous la question suivante : nous voulons prouver des programmes temps-réels, avons-nous donc besoin de savoir ce qui se passe en tous les points du temps ? Non, pour un programme, ce sont les intervalles de macro-temps (les contraintes temporelles de la spécification initiales) qui nous intéressent.

Donc, sachant que nous avons une sémantique WDC* d'un langage, comment pouvons-nous obtenir, pour une formule donnée, l'équivalent débarrassé des phases ? La réponse, indiquée également dans [SH01], est l'utilisation d'un opérateur de projection. Cet opérateur fera une abstraction sur les intervalles de micro-temps pour permettre d'obtenir une formule de DC*. Soit Π cet opérateur de projection définit au tableau 2.19.

La définition de cet opérateur est intuitive, pour la plupart des cas. Les cas plus subtils sont les suivants :

- $\Pi(R(entier, entier))$: comme nous oublions les contraintes de phase, le fait que deux phases données soient en relation n'a plus d'importance
- $\Pi(\lceil S \rceil^{0\frown} Unit)$: ceci signifie que notre état S est valable juste avant une phase d'intervalle de macro-temps nul. Cela signifie que S est vrai au moins pour $\ell = 0$, donc S est vrai au moins pour une itération nulle de $\llbracket S \rrbracket$. Seulement, comme il est possible que cette formule soit incluse dans un intervalle plus grand, il est possible en fait que S soit vrai pour un

Les règles d'inférence de DC restent valides :

$$IRWDC1_r \quad \frac{H(\lceil\rceil) \qquad H(X) \vdash H((X \vee (X^\frown \lceil S \rceil) \vee (X^\frown \lceil \neg S \rceil)))}{H(\mathbf{true})}$$

L'induction est basée sur les phases des intervalles :

$$IRWDC2_r \quad \frac{H(\eta = 0) \qquad H(X) \vdash H(X \vee (X^\frown \eta = 1))}{H(\mathbf{true})}$$

L'induction est restreinte à un intervalle-point :

$$IRWDC3_r \quad \frac{H(Pt_i) \qquad H(X \wedge \ell = 0) \vdash H(X \wedge \ell = 0 \vee (X \wedge \ell = 0^\frown Unit))}{H(\ell = 0)}$$

$$WDC_\omega \quad \frac{\forall k < \omega.[(\llbracket S \rrbracket \vee \llbracket \neg S \rrbracket)^k / A]P}{[\mathbf{true}/A]P}$$

TAB. 2.18 – Règles d'inférence de WDC*

$$\begin{array}{lcl}
\Pi(\lceil S \rceil^0) & \equiv & \ell = 0 \\
\Pi(\lceil S \rceil^{0\frown} Unit) & \equiv & \llbracket S \rrbracket^* \\
\Pi(R(entier, entier)) & \equiv & \mathbf{true} \\
\Pi(R(reel, reel)) & \equiv & R(reel, reel) \\
\Pi(P \wedge Q) & \equiv & \Pi(P) \wedge \Pi(Q) \\
\Pi(\exists x.P) & \equiv & \exists x.\Pi(P) \\
\Pi(P^\frown Q) & \equiv & \Pi(P)^\frown \Pi(Q) \\
\Pi(P^*) & \equiv & (\Pi(P))^*
\end{array}$$

TAB. 2.19 – Projection de WDC* vers DC*

temps plus grand, d'où l'utilisation de l'itération : en effet dans ce cas, l'intervalle de temps où S est vrai devient morcelable arbitrairement, ce qui correspond à la définition de l'itération.

Cet opérateur définit bien une projection :

$(\mathbb{T}ime, \theta, [b, e]) \models P \Rightarrow (\pi_1(\mathbb{T}ime), \theta_{\pi_1}, [\pi_1(b), \pi_1(e)]) \models \Pi(P)$

Pour tout modèle dans WDC* d'une formule P, son projeté sur le macro-temps est un modèle dans DC* de $\Pi(P)$. Il a également la propriété de monotonicité : Si dans WDC* $P \Rightarrow Q$, alors $\Pi(P) \Rightarrow \Pi(Q)$ (il suffit d'appliquer la définition de Π).

De cette manière nous disposons d'un moyen pour obtenir les propriétés en macro-temps d'un programme, si la sémantique de celui-ci est définie par WDC*.

2.4.4 Exemples

Les quelques formules suivantes aident à donner l'intuition de l'utilisation de WDC* :

- $\ell \geq 1 \Rightarrow \eta \geq 1000$: en une seconde il y a au moins 1000 phases par lesquelles le système peut passer. Bien entendu, comme ici les phases en question ne sont pas spécifiées, ce qu'il se passe peut correspondre à « ne rien faire »

–
$$\ell = 1 \wedge (((\eta = 1 \wedge \lceil P \rceil)^\frown(\eta = 1 \wedge \lceil \neg P \rceil))^*$$
$$\Rightarrow$$
$$\eta \leq 1000 \wedge ((\eta = 1 \wedge \ell \geq 0.001 \wedge \lceil P \rceil)^\frown(\eta = 1 \wedge \ell \geq 0.001 \wedge \lceil \neg P \rceil))^*$$

Si il y a plusieurs alternances d'états P consécutives pendant une seconde, alors il y en a au plus 1000, et chaque alternance dure au moins 2 millièmes de secondes (un millième pour P et un millième pour $\neg P$).

–
$$\begin{cases} Unit^\frown\lceil x = 1 \rceil^0{}^\frown Unit^\frown\lceil y = 0 \rceil^0 \\ \vee\, Unit^\frown\lceil y = 0 \rceil^0{}^\frown Unit^\frown\lceil x = 1 \rceil^0 \end{cases}$$

Il s'agit du genre de formule que l'on est susceptible de retrouver pour la sémantique de programmes concurrents. Soit le programme $x := 1 \| y := 0$ qui met x à 1 et en parallèle y à 0, alors en pratique les instructions seront ordonnancées de telle manière que soit ce sera d'abord x qui est mis à jour et ensuite y, ou bien le contraire. Le *Unit* avant chaque mise à jour de variable sert à indiquer que l'affectation correspond à un changement de phase : cela permet d'ordonnancer les différentes instructions du programme selon les différentes phases.

2.5 Conclusion

Comme nous avons pu le constater, le panel de l'expressivité de DC et de ses extensions est particulièrement large. Rappelons la possibilité d'exprimer l'ordre supérieur [ZGN99], des intervalles de temps infinis[ZWR95], par exemple, à l'aide d'autres extensions de DC. Similairement, il n'est pas nécessaire que DC soit basé sur la logique d'intervalle. Il peut être basé sur la logique de voisinage[ZH98].

Dans cette profusion de logiques, toutes avec des propriétés plus ou moins proches, la question du choix est légitime : avons-nous choisi la bonne logique pour exprimer les propriétés de ma modélisation ? Et la réponse ici est rassurante : il est possible d'unifier beaucoup de ces extensions. En effet, He [HJ04] introduit une logique temporelle dans laquelle peuvent s'exprimer les logiques temporelles suivantes : la logique de voisinage[ZH98], DC*, DC à l'ordre supérieur[ZGN99], DC avec découpage super-dense [ZL94], le calcul des durées récursif [PR95].

En guise de conclusion, nous pouvons donc dire que grâce à ce dernier travail d'intégration, tout travail utilisant un calcul de durées a donc de fortes chances d'être adaptable à une logique temporelle plus expressive, comme l'une de celles mentionnée ci-avant. Une conséquence de cela est que si un outil existe pour cette logique plus expressive, il pourra être utilisé avec des travaux utilisant la logique moins expressive. Cela facilitera d'autant des extensions sémantiques, des preuves plus simples de propriétés,etc.

Maintenant que nous avons présenté en détail les logiques temporelles que nous allons utiliser, le chapitre suivant est consacré au langage de modélisation que nous utilisons.

Bibliographie

[AGM92] Samson Abramsky, Dov M. Gabbay, and T.S.E Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[BHB+89] Dines Bjørner, C.A.R. Hoare, Jonathan Bowen, He Jifeng, Hans Langmaack, Ernst-Rüdiger Olderog, Ursula Martin, Victoria Stavridou, Fleming Nielson, Hanne Riis Nielson, Howard Barringer, Doug Edwards, Hans Henrik Løvengreen, Anders Ravn, and Hans Rischel. A ProCoS project description : ESPRIT BRA 3104. In *Bulletin of the European Association for Theoretical Computer Science*, volume 39, pages 60–73. EATCS, october 1989.

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

[Dut95a] Bruno Dutertre. Complete proof systems for first order interval temporal logic. In *Logic in Computer Science*, pages 36–43, 1995.

[Dut95b] Bruno Dutertre. On first order interval temporal logic. Technical Report CSD-TR-94-3, University of London, Department of computer science, Egham, Surrey TW20 0EX, England, february 1995.

[GH99] Dimitar P. Guelev and Dan Van Hung. Completeness and decidability of a fragment of duration calculus with iteration. In *Asian Computing Science Conference (ASIAN'99)*, volume 1742 of *LNCS*, pages 139–150, Phuket, Thailand, December 1999. Springer-Verlag. Also presented at International Conference on Mathematical Foundation of Informatics, Hanoi, October 25-28, 1999.

[Gue98] Dimitar P. Guelev. Iteration of simple formulas in duration calculus. Technical Report 141, UNU-IIST, P.O. Box 3058, Macau, June 1998.

[HJ04] Jifeng He and Naiyong Jin. Integrating variants of DC. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004, First International Colloquium, Guiyang, China, September 20-24, 2004, Revised Selected Papers*, volume 3407 of *Lecture Notes in Computer Science*, pages 14–34. Springer-Verlag, 2004. ISBN :3-540-25304-1.

[HMM83] Joseph Y. Halpern, Zohar Manna, and Ben C. Moszkowski. A hardware semantics based on temporal intervals. In *10th International Colloquium on Automata, Languages and Programming*, volume LNCS 154, pages 278–291, London, UK, 1983. Springer-Verlag. ISBN :3-540-12317-2.

[HP98] Dang Van Hung and Paritosh K. Pandya. Duration calculus with weakly monotonic time. In Anders P. Ravn and Hans Rischel, editors, *5th symposium of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume LNCS

1486, pages 55–64, Lyngby, Denmark, september 1998. Springer-Verlag. Technical Report 122, UNU-IIST, P.O. Box 3058, Macau, September 1997.

[HZ97] Michael R. Hansen and Chaochen Zhou. Duration calculus : logical foundations. *Formal Aspects of Computing*, 9 :283–330, 1997.

[HZ04] Michael R. Hansen and Chaochen Zhou. *Duration Calculus, a formal approach to real-time systems*. Number ISBN : 3-540-40823-1 in Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[Mos85] Ben C. Moszkowski. Temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2) :10–19, 1985.

[Mos94] Ben C. Moszkowski. Some very compositional temporal properties. In *Programming concepts, methods and calculi*, pages 307–326. Elsevier Science B.V. (North-Holland), 1994.

[PR95] Paritosh K. Pandya and Y. Ramakrishna. A recursive duration calculus. Technical report, TIFR, Mumbai, 1995. Technical Report CS-95/3.

[Ras02] Thomas Marthedal Rasmussen. *Interval Logic - Proof Theory and Theorem Proving*. PhD thesis, Informatics and Mathematical Modeling, Technical University of Denmark, january 2002.

[SH01] François Siewe and Dan Van Hung. Deriving real-time programs from duration calculus specifications. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume LNCS 2144, pages 92–97, Livingston-Edinburgh, Scotland, september 2001. Springer-Verlag. (Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000).

[ZGN99] Chaochen Zhou, Dimitar P. Guelev, and Z. Naijun. A higher-order duration calculus. In *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).

[ZH98] Chaochen Zhou and Michael R. Hansen. An adequate first order interval logic. *Lecture Notes in Computer Science*, 1536 :584–608, 1998.

[ZHR91] Chaochen Zhou, C.A.R. Hoare, and Anders P. Ravn. A calculus of durations. In *Information Processing Letters*, volume 10(5), pages 269–276. Elsevier, December 1991.

[ZL94] Chaochen Zhou and Xiaoshan Li. A mean value duration calculus. In A.W. Roscoe, editor, *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 432–451. Prentice-Hall International, 1994.

[ZWR95] Chaochen Zhou, J. Wang, and Anders P. Ravn. A duration calculus with infinite intervals. In H. Reichel, editor, *Fundamentals of Computing Theory*, volume 965 of LNCS, pages 16–41. Springer-Verlag, Lübeck, Germany, 1995.

Chapitre 3

Méthode B et temporalité

Sommaire

Figures

Quelques-unes des méthodes présentées en section 3.3 ont fait l'objet d'une étude de compatibilité avec un modèle de passage à niveau en B obtenu à partir d'une spécification UML, dans notre article co-écrit avec Marcano [MCM04].

3.1 Présentation de la méthode B

Cette section présente B en mettant l'accent sur ses aspects les plus pertinents : après un rapide historique et un rappel de ses caractéristiques les plus remarquables, nous présentons le coeur du langage B et sa sémantique en transformateurs de prédicats.

3.1.1 Historique

Les idées qui président à la fondation de la méthode B remontent à la première moitié de la décennie 1980-1990 : Jean-Raymond Abrial avait proposé auparavant une notation appelée Z [Abr74], servant à spécifier de manière mathématique, via la théorie des ensembles et la logique du premier ordre, un système informatique. Cherchant à aller plus loin, il a conçu la méthode B [Abr84b, Abr84a, Abr88] pour pouvoir développer un logiciel de manière correcte et sûre. La totalité des idées présentant la méthode B dans ses plus infimes détails est regroupée dans le B-Book[Abr96a]. Notons que le nom de « B » a été choisi pour rendre hommage aux mathématiciens qui ont repensé les mathématiques à partir du début du vingtième siècle et qui signaient sous le pseudonyme commun de *Nicolas Bourbaki*. Les travaux de ce « groupe Bourbaki » continuent encore aujourd'hui.

La méthode B a été utilisée pour démontrer son efficacité sur des études de cas non-triviales, comme un système de contrôle de chaudière [Abr96c] ou un système banquaire [Büc98]. Elle a été également utilisée dans la spécification de systèmes industriels critiques, comme le coeur logiciel d'un métro automatisé [BBFM99]. Elle sera par ailleurs réutilisée pour la modernisation d'une ligne de métro de la même compagnie (Régie Autonome des Transports Parisiens), avec l'expertise de l'entreprise (Siemens Transportation Systems) qui a réalisé la spécification du métro automatisé, METEOR, la première fois. La méthode B est, de manière plus générale, essentiellement utilisée dans l'industrie ferroviaire [BDM97].

Puisque la méthode B a un tel succès, quelles sont les caractéristiques qui la rendent si attrayante ?

3.1.2 Principales caractéristiques

La méthode B permet de spécifier et de vérifier la correction d'un logiciel depuis sa spécification mathématique abstraite, jusqu'au code informatique correspondant, généré automatiquement. Cette correspondance est garantie par des mécanismes de preuve validant le passage d'une spécification abstraite à une spécification plus concrète. Ce passage de l'abstrait au concret est appelé *raffinement*.

L'abstraction utilisée pour spécifier les composants B se base sur la théorie des ensembles et la logique du premier ordre, pour spécifier les propriétés statiques, et un langage de substitutions pour spécifier les propriétés dynamiques des machines. Un composant B est la donnée d'une machine abstraite qui spécifie les propriétés du composant, et de ses raffinements.

Pour rester très schématiques, nous dirons que les machines B sont définies par les classes de clauses suivantes :

- Des clauses `VARIABLES` et `INITIALISATION`, définissant les variables représentant l'état de la machine l'état initial de la machine, respectivement
- Une clause `INVARIANT`, un prédicat qui définit les conditions de sûreté de la machine
- Une clause `OPERATIONS` définissant les changements d'état possibles pour cette machine.

Une machine B peut également utiliser d'autres machines B, via des clauses de modularité `INCLUDES` ou `IMPORTS` qui rendent accessibles les entités des machines. Les variables sont accessibles en lecture, et ne peuvent être modifiées que via l'appel des opérations *ad hoc*. Des clauses de modularité `SEES` permettent de référencer les variables d'une autre machine, mais en lecture seulement, i.e. les opérations qui modifient les variables ne peuvent pas être appelées. Il existe d'autres clauses de modularité, mais il est préférable pour notre discours de s'en tenir là : des descriptions détaillées de la modularité de B et de ses extensions possibles se trouvent dans [BPR96, PR98, Pet03].

Enfin, une machine B peut être le raffinement (`REFINEMENT`) d'une autre machine B : le but est alors de réduire l'indéterminisme de la machine plus abstraite, ainsi que de changer la représentation de l'état. L'idée dans un projet B est de constituer une chaîne de raffinements, lesquels « déterminisent » à chaque fois un peu plus le raffinement précédent et rendent l'état plus facile à manipuler pour un programme informatique (par exemple la transformation de certaines fonctions en tableaux), pour arriver à l'implémentation (`IMPLEMENTATION`) qui définira un pseudo-code suffisamment déterminisé pour être traduit sous forme de code informatique.

Après cette étape d'implémentation vient l'étape de génération de code (C ou Ada, par exemple), que nous ne détaillerons pas ici, et qui est elle aussi décrite dans la thèse de Dorian Petit[Pet03]. Nous décrivons dans les sections suivantes les substitutions du langage B, la notion de raffinement en B, et les contraintes sur les machines B pour démontrer qu'elles sont correctes.

Nous illustrons l'organisation d'une machine B représentant une horloge (reprise d'un exemple de Dominique Cansell) précise aux heures près, en figure 3.1.

```
MACHINE
  Clock
VARIABLES
  hour
INVARIANT
  hour ∈ 0..23
INITIALISATION
  hour :∈ 10..16
OPERATIONS
  tick =
    IF
      hour=23
    THEN
      hour :=0
    ELSE
      hour :=hour+1
    END

END
```

FIG. 3.1 – Un exemple de machine B

Cette machine spécifie un état conservant les heures (`hour`), initialisé à la mise en route de l'horloge à un nombre entre 10 et 16. Cela spécifie par exemple que l'horloge sera forcément mise en route pendant ces heures. La condition de sûreté pour que l'horloge fonctionne correctement, est que les heures affichées restent bien entre 0 et 23. Enfin une opération `tick` permet de faire avancer l'heure : cette opération pourra par exemple être appelée par un composant lorsqu'un cristal de quartz aura oscillé un certain nombre de fois, comme cela est fait dans les montres à quartz.

3.1.3 Substitutions

Le langage des actions de B est défini à partir d'un noyau d'instructions, les GSL (Langage des Substitutions Généralisées). il s'agit d'instructions dont la sémantique est définie en plus faible précondition : si une substitution S doit établir une postcondition P, quelle est l'état *sine qua non* dans lequel doit se trouver notre système pour que cette postcondition soit vraie après application de la substitution ? Cette plus faible précondition est notée $[S]P$. Les règles d'application d'une substitution à un prédicat pour calculer la plus faible précondition sont indiquées en figure 3.2.

Faisons quelques remarques supplémentaires :

Nom	S	$[S]P$
skip	skip	P
affectation	$x := E$	$P[E/x]$
affectation multiple	$x, y := E, F$	$P[E/x, F/y]$
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	$S \Vert T$	transformé en affectation multiple par l'utilisation de règles de réécriture
boucle	WHILE C DO S VARIANT V INVARIANT I END	$I \wedge C \Rightarrow [S]I$ $I \Rightarrow V \in \mathbb{N}$ $I \wedge C \Rightarrow [n := V][S](V < n)$ $I \wedge \neg C \Rightarrow P$

FIG. 3.2 – Substitutions généralisées et sémantique en transformateur de prédicat

– Nous utilisons le terme *substitution* pour désigner à la fois une substitution généralisée, et une instruction du langage B. Nous pourrons utiliser aussi le terme *GSL* plutôt que substitutions.
– Pour toutes les substitutions indiquées en figure 3.2, il s'agit de la plus faible précondition, sauf pour la boucle. En ce qui concerne celle-ci, les formules indiquées servent à vérifier que l'invariant est suffisamment expressif pour vérifier P. Il s'agit donc plutôt d'une précondition suffisamment expressive, plutôt qu'une plus faible précondition.

La sémantique en transformateurs de prédicats provient d'une extension des triplets de Hoare [Hoa69] par Dijkstra [Dij75]. Des structures de contrôles complexes peuvent être construites à partir de ces transformateurs de prédicats, tels le `if-then-else` à partir d'un choix borné et de gardes. Cette apparente austérité confère aux GSL deux avantages :
– La sémantique en transformateurs est plus concise et plus simple à formuler pour le jeu réduit des GSL, qu'elle le serait pour toute la syntaxe de B
– Le comportement d'un composant B est définissable de manière encore plus abstraite. Par exemple, le choix borné spécifie deux chemins que peut prendre le programme, sans discrimination : il pourra être déterminisé (via raffinement) en ajoutant des gardes. Cet ajout de gardes peut correspondre par exemple à des états de la machine qui ne sont pas censés être connus à l'extérieur de cette machine.

Nous introduisons également les prédicats permettant de vérifier certaines propriétés des substitutions :

Terminaison : $trm(S) \equiv [S](x = x)$: ce prédicat représente l'espace des états pour que S puisse établir une postcondition.

Prédicat *avant-après* : $prd_x(S) \equiv \neg [S](x' \neq x)$: ce prédicat caractérise la modification par S de la variable x. Le x' représente la valeur de la variable x après évaluation de S, par rapport à sa valeur de départ.

Toute substitution peut être décrite en termes de trm et prd. Pour pouvoir faire le lien entre les GSL et la syntaxe B usuelle, nous indiquons en figure 3.3 la manière dont s'écrivent les substitutions de B avec les GSL.

Nous disposons jusqu'ici de mécanismes qui permettent de démontrer la validité de n'importe quel programme. Est-ce seulement faisable en pratique ? Il peut devenir rapidement très difficile de spécifier un système suffisamment complexe à l'aide d'une simple machine B. Donc pour simplifier la spécification et la rendre plus progressive, la méthode B introduit un mécanisme permettant de spécifier incrémentalement un système, présenté en section suivante.

3.1.4 Raffinement

Le raffinement d'une machine par une autre permet de déterminiser, de préciser deux aspects de la machine : son état, et son comportement dynamique.

Lors du raffinement, l'état de la machine peut être détaillé en précisant avec une granularité plus fine les variables de la machine, ou en ajoutant de nouvelles variables. De manière duale, le comportement dynamique de la machine est précisé en modifiant les opérations conformément aux anciennes et nouvelles variables. La figure 3.4 illustre un raffinement de notre machine `Clock` de la figure 3.1.

Ce raffinement appelle à plusieurs commentaires :

- Le raffinement définit une nouvelle variable `h` : comme les heures ont un comportement plus précis dans le raffinement (à cause de l'initialisation), alors il est nécessaire de définir cette nouvelle variable. Ensuite il est nécessaire de spécifier quelle est la relation entre h et $hour$: le prédicat $hour = h$ est dit *de collage*, car il fait la relation entre les états de la machine abstraite et ceux du raffinement.
- L'initialisation dans le raffinement est plus précis : l'heure qui auparavant pouvait être initialisée entre 10 et 16, est désormais initialisée à 11
- Le raffinement ajoute une variable `minute` qui permettra de préciser à quel moment la variable des heures changera
- L'opération `tick` est modifiée pour exprimer le changement d'état de la machine lorsque cette opération est appelée. Or nous nous rendons compte que cette fois-ci, lorsque l'opération `tick` est appelée, la variable des heures ne change pas forcément. Donc l'opération de la machine abstraite n'était pas correcte : il nous faut désormais spécifier que les heures peuvent ne pas changer. La machine abstraite est corrigée dans ce sens.

De par l'obligation pour le raffinement d'être plus précis (plus exactement, « moins indéterministe ») que la machine qu'il raffine, il se constitue donc en B une chaîne de raffinements pour une machine donnée. Cette chaîne de raffinements est complétée par son composant le plus déterministe, l'implémentation. L'implémentation est la machine B dont le code sera traduit vers un autre langage, comme le C ou Ada. Ce déterminisme de la machine, est assuré par l'interdiction de certaines constructions non-déterministes (telles que le choix borné, la substitution parallèle) dans le code de l'implémentation.

La machine B et son raffinement, présentés en figure 3.4, sont corrects : la méthode B définit en effet des formules à démontrer, appelées *obligations de preuve*, pour les machines abstraites, leurs raffinements, et leurs implémentations, et ce pour assurer la correction du projet B. Ces

PRE P THEN S END	$P\|S$

CHOICE
  S
OR
  T
END
	$S [] T$

IF P THEN S ELSE T END	$P \Longrightarrow S [] \neg P \Longrightarrow T$
SELECT WHEN P THEN S ... WHEN Q THEN T ELSE R	$P \Longrightarrow S [] \dots [] Q \Longrightarrow T [] R$
VAR x IN S END	$@x.S$
ANY x WHERE P THEN S END	$@x.P \Longrightarrow S$
x:(P)	$@x'.[x, x_0 := x', x]P \Longrightarrow x := x'$
x::E	$@z.z \in E \Longrightarrow x := z$

FIG. 3.3 – Correspondance des substitutions B avec les substitutions généralisées

<pre>**MACHINE**
  Clock
VARIABLES
  hour
INVARIANT
  hour ∈ 0..23
INITIALISATION
  hour :∈ 10..16
OPERATIONS
  tick =
    CHOICE
      IF
        hour=23
      THEN
        hour :=0
      ELSE
        hour :=hour+1
      END
    OR
      skip
    END
 	
END</pre>
	<pre>**REFINEMENT**
	  Clockhm
	REFINES
	  Clock
	VARIABLES
	  h,minute
	INVARIANT
	  minute ∈ 0..59 ∧h=hour
	INITIALISATION
	  h:=11
	|| minute :∈ 0..45
	OPERATIONS
	  tick =
	    IF
	      minute=59
	    THEN
	      IF
	        h=23
	      THEN
	        h:=0
	      || minute:=0
	      ELSE
	        h:=h+1
	      || minute:=0
	      END
	    ELSE
	      minute:=minute+1
	    END
	
	END</pre>

FIG. 3.4 – Une machine B et un raffinement possible

formules sont présentées dans la section suivante.

3.1.5 Obligations de preuve

Le mécanisme des obligations de preuve peut se résumer ainsi :

- L'invariant de machine doit être vrai quel que soit l'état de la machine.
  - Il doit donc être vérifié à l'initialisation de la machine
  - Il faut aussi vérifier que si la machine est dans un état correct (l'invariant est vrai), alors toute opération de cette machine maintient celle-ci dans un état correct.
- Le raffinement de machine doit vérifier lui aussi les mêmes contraintes pour l'invariant, avec la contrainte que la nouvelle machine soit au pire, aussi non-déterministe, au mieux, moins non-déterministe (donc, plus déterministe), que la machine qu'elle raffine.
  - Cela signifie que l'initialisation du raffinement ne doit pas se situer *en dehors* des états d'initialisation de la machine raffinée : l'initialisation n'est pas moins déterministe que celle de la machine raffinée.
  - Pour chaque opération raffinée, il faut également vérifier que le code de l'opération dans le raffinement ne permet pas d'arriver dans un état au-dehors de l'état correct de la machine abstraite.

  Ces dernières obligations de preuve n'assurent pas que le raffinement sera plus détermiste : simplement que ses états sont inclus dans les états de la machine abstraite. Cela correspond à la contrainte, indiquée plus haut, que la machine doit au mieux amoindrir le non-déterminisme de la machine raffinée.

Donc un raffinement ne peut pas accepter « plus » d'états que la machine abstraite correspondante : c'est ce qui garantit qu'un raffinement peut être utilisé en lieu et place de la machine qu'il raffine. *In fine*, ce sera donc l'implémentation qui vérifiera les spécifications de la machine abstraite : nous sommes donc certains que le code traduit de l'implémentation fera ce pour quoi il a été spécifié (en supposant que cette étape de traduction est sûre elle aussi).

La figure 3.5 présente des schémas de machines et de raffinements.

Les obligations de preuve des machines de la figure 3.5 sont indiquées en figure 3.6. Nous les explicitons en nous focalisons sur le rôle joué par chacun des buts à prouver :

- $[U_1]I_1$: la plus faible précondition U_1 pour établir I_1 est-elle toujours vraie ? En d'autres termes, est-ce que les valeurs initiales établissent l'invariant dans tous les cas ?
- $I_1 \wedge Q_1 \Rightarrow [V_1]I_1$: Si nous plaçons la machine dans un état où l'invariant est vérifié, et l'opération op_1 appelée dans ses conditions de définitions, est-ce suffisant pour garantir que V_1 établit l'invariant I_1 ?
- $[U_n]\neg\,[U_{n-1}]\neg\,I_n$:
  - $\neg I_n$ est l'ensemble des états interdits à la machine
  - $[U_{n-1}]\neg\,I_n$ représente les états initiaux incohérents de la machine abstraite M_{n-1}
  - $\neg\,[U_{n-1}]\neg\,I_n$ est l'ensemble des initialisations de la machine raffinée qui garantissent que le raffinement est dans un état cohérent
  - Donc $[U_n]\neg\,[U_{n-1}]\neg\,I_n$ est l'ensemble des initialisations du raffinement qui sont incluses dans les initialisations de la machine raffinée garantissant un état cohérent pour le raffinement

  En d'autres termes, l'initialisation du raffinement est moins imprécise que celle de la machine raffinée, et est correcte car vérifie l'invariant du raffinement.

MACHINE $M_1(X_1, x_1)$	**REFINEMENT** $M_n(X_1, x_1)$
CONSTRAINTS C_1	**REFINES** M_{n-1}
SETS $S_1; T_1 = \{a_1, b_1\}$	**SETS** $S_n; T_n = \{a_n, b_n\}$
CONSTANTS c_1	**CONSTANTS** c_n
PROPERTIES P_1	**PROPERTIES** P_n
VARIABLES v_1	**VARIABLES** v_n
INVARIANT I_1	**INVARIANT** I_n
INITIALISATION U_1	**INITIALISATION** U_n
OPERATIONS	**OPERATIONS**
$u_1 \longleftarrow op_1(w_1) \equiv$	$u_1 \longleftarrow op_1(w_1) \equiv$
PRE Q_1	**PRE** Q_n
THEN V_1	**THEN** V_n
END ;	**END ;**
...	...
END	**END**

FIG. 3.5 – Schéma de machines B

<table>
<tr><td colspan="2">Soient les formules additionnelles suivantes :
$A_i \equiv X_i \in \mathbb{P}_1(INT)$
$B_i \equiv S_i \in \mathbb{P}_1(INT) \wedge T_i \in \mathbb{P}_1(INT) \wedge T = \{a, b\} \wedge a \neq b$</td></tr>
<tr><td>$A_1 \wedge B_1 \wedge C_1 \wedge P_1 \wedge$
$\Rightarrow$
$[U_1]I_1$

Initialisation de M_1</td><td>$A_1 \wedge B_1 \wedge C_1 \wedge P_1 \wedge I_1 \wedge Q_1$
$\Rightarrow$
$[V_1]I_1$

Opération op_1 de M_1</td></tr>
<tr><td>$A_1 \wedge B_1 \wedge \ldots \wedge B_n \wedge$
$C_1 \wedge P_1 \wedge \ldots \wedge P_n \wedge$
$\Rightarrow$
$[U_n]\neg\, [U_{n-1}]\neg\, I_n$

Initialisation du raffinement M_n</td><td>$A_1 \wedge B_1 \wedge \ldots \wedge B_n \wedge$
$C_1 \wedge P_1 \wedge \ldots \wedge P_n \wedge$
$I_1 \wedge \ldots \wedge I_n \wedge Q_1 \wedge$
$\Rightarrow$
$Q_n \wedge [[u_1 := u_1']V_n]\neg\, [V_{n-1}]\neg\, (I_n \wedge u_1 = u_1')$

Opération op_1 du raffinement M_n</td></tr>
</table>

FIG. 3.6 – Obligations de preuve pour la figure 3.5

Pour clarifier les explications, l'obligation de preuve pour le raffinement d'une opération est scindée en deux parties :

- $I_1 \wedge \ldots \wedge I_n \wedge Q_1 \Rightarrow Q_n$: cette formule garantit que les états de départ autorisés pour l'opération raffinante contient au moins les états de départ autorisés de la machine abstraite. Elle permet de garantir qu'il est possible d'appeler l'opération raffinante à la place de l'opération abstraite
- $I_1 \wedge \ldots \wedge I_n \wedge Q_1 \Rightarrow [[u_1 := u'_1]V_n]\neg\, [V_{n-1}]\neg\, (I_n \wedge u_1 = u'_1)$:
  - $\neg\, (I_n \wedge u_1 = u'_1)$ correspond aux états incohérents du raffinement, en termes d'invariant et de valeur calculée
  - $[V_{n-1}]\neg\, (I_n \wedge u_1 = u'_1)$ sont les états permettant d'arriver dans un état incohérent ou de calculer une mauvaise valeur en activant l'opération raffinée
  - $\neg\, [V_{n-1}]\neg\, (I_n \wedge u_1 = u'_1)$ sont donc les états garantissant que l'opération raffinée laisse le raffinement dans un état cohérent et fait un calcul correct
  - $[[u_1 := u'_1]V_n]\neg\, [V_{n-1}]\neg\, (I_n \wedge u_1 = u'_1)$ correspond donc aux états de la machine raffinée qui garantissent d'arriver après exécution de l'opération raffinante dans un état où l'opération raffinée établira un état cohérent pour le raffinement, et produira un calcul correct.

En d'autres termes, l'opération raffinante est aussi (voire plus) précise que l'opération qu'elle raffine, et est correcte puisqu'elle conserve l'invariant de machine et produit un résultat similaire à l'opération qu'elle raffine.

La théorie des ensembles et le calcul des prédicats, utilisés par B pour la modélisation des propriétés, semble suffisamment puissante pour exprimer la plupart des problèmes. Cependant, la mise en pratique pose quelques problèmes, que nous illustrons par deux exemples concrets :

Premier exemple Lorsque nous reprenons l'exemple de la figure 3.4, nous voyons que nous avons dû modifier une opération la machine abstraite pour « préparer » l'introduction d'un comportement qui n'est pas visible dans l'abstraction mais l'est dans le raffinement.

Ce phénomène est dû à un changement de granularité : le but du raffinement est d'observer des actions qui n'étaient pas visibles au niveau de l'abstraction. Ces actions invisibles correspondent à ce qui dans d'autres formalismes est appelé $\varepsilon - transitions$ (dans les systèmes de transition [Gil62, HU79]), ou *bégaiement* (stuttering) comme dans TLA+[Lam02]. Il est en règle générale désirable de pouvoir spécifier celles-ci.

Pour ce faire, B impose au développeur de prévoir des cas où l'état n'évolue apparemment pas, au niveau abstrait. Ces cas correspondent aux états qui apparaîtront lors du raffinement, dans les niveaux plus concrets.

De plus, le raffinement a pour but de simplifier la validation d'un modèle en permettant l'ajout progressif de complexité à la vision abstraite du modèle, qui se traduira par des formules plus simples à prouver.

La limitation citée plus haut sur la granularité et les états invisibles peut donc obliger à garder l'absence apparente d'évolution de l'état pendant plusieurs étapes de raffinement. Le développeur devra prendre bien garde à l'étape de raffinement où un état reste invisible ou non. L'apparition d'un état visible est liée le plus souvent à l'apparition d'une nouvelle variable (ou du changement de représentation de l'une d'entre elles). Si ces nouvelles variables sont fortement corrélées entre elles et avec les anciennes, il peut être difficile de séparer le développement en étapes de raffinement suffisamment nombreuses. Ce dernier point dépend cependant de la nature du problème modélisé.

Enfin, le fait de ne pas pouvoir morceler les opérations à mesure du raffinement peut conduire à des formules qui, bien que simples à prouver, contiennent beaucoup de grand termes qui correspondent au code final du modèle. Cela est particulièrement visible dans les raffinements. Ce point peut être déterminant si l'outil utilisé pour la preuve ne sait pas faire de l'unification de terme efficacement. Là encore, le problème peut être contourné en donnant une définition à ces grands termes, un « alias », qui raccourcira les formules logiques engendrées. Il s'agit cependant d'un contournement qui incombe au développeur, et amoindrit les possibilités d'automatisation.

Deuxième exemple Très souvent la finalité des opérations B est de calculer un résultat déjà pensé comme *final*. La figure 3.7 montre un calcul de factorielle :

- La machine abstraite spécifie la signification de la factorielle à l'aide de prédicats, et spécifie l'opération « calculer la factorielle de b » en faisant appel à cette définition
- Son implémentation spécifie comment calculer la factorielle de n, sans donner la possibilité d'accéder aux états intermédiaires. Il pourrait cependant être intéressant de pouvoir utiliser certaines des factorielles intermédiaires calculées par la boucle, par exemple.

Parallèlement, l'exemple de l'horloge en figure 3.4 spécifie comment évolue l'horloge, mais pas ce qu'elle fait, i.e. son but final. De plus, le fonctionnement de la machine est dévolu à une machine tiers qui appellerait le `tick` d'horloge lorsque nécessaire. Cette manière de faire est contre-intuitive, puisque ce qui est attendu ici est plutôt que l'horloge avance par elle-même, et d'autres composants se mettent à jour en fonction de celle-ci. Cette manière de faire n'est pas possible en B, puisque cette notion de fonctionnement perpétuel est interdite par le fait que les boucles doivent s'arrêter, alors que ce sont les seules structures de contrôle exprimant une notion de répétition.

Tous ces exemples nous indiquent que B possède des limitations (pratiques ou théoriques) dans les domaines suivants :

- L'apparition d'étapes de calcul auparavant indiscernables
- L'impossibilité d'accéder aux états intermédiaires d'une machine lors d'une opération (la granularité observable se limite aux états *entre* les opérations)
- L'impossibilité de spécifier qu'un système fonctionne *tout le temps*

Les réponses à ces problèmes sont apportées par le B événementiel.

3.2 Présentation du B événementiel

3.2.1 Historique

Le déplacement de paradigme de modèles invoqués vers des modèles autonomes provient d'un article sur la spécification de contraintes dynamiques en B [AM98]. Il illustre que B peut être utilisé pour spécifier des systèmes distribués, en partant d'une spécification tellement abstraite que cette notion de distribution n'y est pas apparente, seulement les propriétés voulues du système distribué.

L'idée initiale est de spécifier un projet B normalement, et d'ajouter des contraintes dynamiques pour spécifier que la machine atteindra un certain état futur. Il faudra s'assurer que ces

<table>
<tr>
<td>

```
MACHINE
  Factorielle
CONCRETE_CONSTANTS
  n
ABSTRACT_CONSTANTS
  factorielle
PROPERTIES
  n ∈ ℕ
∧ factorielle ∈ ℕ→ℕ
∧ factorielle(0)=1
∧ ∀n.(n ∈ ℕ1
      ⇒ factorielle(n)=n*factorielle(n-1))
OPERATIONS

r ←— calcul =
    BEGIN
      r :=factorielle(n)
    END

END
```

</td>
<td>

```
IMPLEMENTATION
  Factorielle_1
REFINES
  Factorielle
VALUES
  n=5
OPERATIONS

r ←— calcul =
    VAR
      i
    IN
      i :=0
     ; r :=1
     ; WHILE
         i<n
       DO
         i :=i+1
        ; r :=r*i
       INVARIANT
        i ∈ 0..n
       ∧ r =factorielle(i)
       ∧ r ≤factorielle(n)
       VARIANT
         n-i
       END
    END

END
```

</td>
</tr>
</table>

FIG. 3.7 – Exemple de code B en « one-shot » pour la factorielle

contraintes dynamiques sont compatibles avec les contraintes statiques déjà spécifiées.

Cet ajout de contraintes dynamiques, et les problèmes de B évoqués plus haut ont amené à repenser la notion de fonctionnement et de raffinement d'un projet B. Les systèmes ne sont plus vus comme des composants contrôlés, mais comme des systèmes autonomes où différents événements activent différents comportements au sein du système. Le raffinement est pensé comme une vue de plus en plus précise de ces événements et de ces comportements.

Une première description d'un système conçu dans cet esprit a vu le jour au sein du projet Matisse [Abr00] et jette les bases d'un B *événementiel*, que nous appellerons eventB. Les exemples se succèdent depuis, notamment une modélisation complète d'algorithmes [ACM03a, ACM03b] pour la recherche d'arbres minimaux couvrants de graphes, nécessaires dans le cadre du protocole IEEE1394.

Les spécifications finales du B événementiel ont vu le jour vers le milieu de l'année 2005 au sein du projet RODIN [ROD05], avec une mise à disposition d'outils libres vers le mois de mai 2006.

3.2.2 Différences avec B

Les avantages d'eventB par rapport au B classique sont déjà résumés à la fin de la section 3.1 :

- Un composant n'est plus « appelable » : il est désormais un système autonome dénommé *modèle*
- Chaque modèle est composé de plusieurs actions activables lorsque les gardes de celles-ci sont déclenchables. Ces actions gardées sont appelées *événements*
- Chaque modèle définit un invariant qui établit une propriété qui restera vraie tout le temps de l'exécution du système
- Un modèle fonctionne tant qu'un de ses événements est activable. Lorsque le contraire arrive, selon le point de vue, soit le calcul fait par le système est terminé, soit le système est bloqué. Cette dernière propriété n'est pas désirable lorsque le système est conçu pour justement ne pas s'arrêter.
- Un modèle peut être raffiné en un ou plusieurs sous-modèles reprenant les anciens événements et en définissant de nouveaux. Ce raffinement de modèles est très similaire à celui de B : le nouveau système raffinant rend plus déterministe le système abstrait.

Les substitutions autorisées en eventB sont moins nombreuses, elles sont indiquées en figure 3.8. La raison de cette réduction du nombre de substitutions est liée au fait qu'eventB oblige à raisonner en termes d'activation d'événéments plutôt qu'en termes d'appel de sous-routines. Ceci a pour conséquence que toutes les substitutions décrivant le flot de contrôle en B classique ne se retrouvent pas en eventB.

La substitution parallèle ici n'est pas à proprement parler une substitution de base, puisqu'avec des règles de réécriture il est possible de la réduire à une subsitution déterministe ou non-déterministe, selon la forme de S et T. Notons au passage que les règles de réécriture pour cette substitution *en B classique* souffrent de quelques exceptions : certaines substitutions ne peuvent pas être réduites. En effet deux substitutions contenant chacune une garde ne peuvent être réduite que si leur terminaison est vérifiée (cf [Abr96a, section 7.1])

En eventB, pas d'exception, la sémantique des substitutions permet d'arriver à une forme normale. La substitution classique « devient tel que » $x : (P)$ et d'appartenance $x :\in S$ peuvent

Vide	skip
Déterministe	x :=E(v)

Non-déterministe	**ANY** t
	WHERE
	  P(t,v)
	THEN
	  x :=t
	END

Parallèle	$S\|\|T$

FIG. 3.8 – Substitutions en B événementiel

être tous deux définis en termes de substitutions non-déterministes.

Les événements, construits sur les substitutions, ont la forme décrite en figure 3.9. Ils sont définis par un nom, une garde optionnelle et une substitution. Si la garde n'est pas présente, alors seule la substitution reste, délimitée par les mots-clefs `BEGIN` et `END` s'il s'agit d'une substitution déterministe ou parallèle.

event_name ≡
WHEN *Guard*
THEN *Substitution*
END

FIG. 3.9 – Événements en eventB

Les systèmes décrits en eventB ont la forme indiquée en figure 3.10. La présentation diffère de celle de [ROD05], puisqu'ici nous nous efforçons de faire le lien avec les concepts duaux du B classique. Les différences principales tiennent à ceci :
- Le nommage des variables qui représentent les différentes clauses
- Les expressions et les prédicats dans [ROD05] sont montrés explicitement comme dépendants des variables et des constantes du modèle dans lequel elles sont définies. Par exemple, une expression est écrite $E(c,v)$ pour indiquer qu'elle dépend des entités c et v. Ici nous supposons que cette dépendance est implicite pour alléger la notation.

Nous ajoutons deux remarques sur ces schémas de modèles :
- Il n'y a en fait pas de clause `PROPERTIES` : les modèles eventB ont à la place une clause `SEES` qui « voit » un contexte qui définit des ensembles, des constantes et des propriétés sur ceux-ci. Son rôle est similaire à la clause du même nom en B classique
- Il n'y a pas non plus de clause `INITIALISATION` : l'initialisation des variables se fait dans un événement spécial, sans garde, appelé `initialization`. Son rôle est similaire à la clause d'initialisation du B classique.

En eventB, un événement est donc composé d'un prédicat appelé la *garde* et de l'action correspondant à cette garde. La garde est un prédicat qui définit la condition d'activation d'activation, et l'action est définie par une substitution.

La modularité en eventB est en quelque sorte l'inverse de celle du B classique. Alors que des machines B peuvent être développées séparément puis composées plus tard, le développe-

MODEL M_1
VARIABLES x
INVARIANT I_1
INITIALISATION U_1
OPERATIONS

event$_1 \equiv$
WHEN Q_1
THEN V_1
END ;

...
event$_n \equiv$
WHEN Q_n
THEN V_n
END

END

REFINEMENT M_2
REFINES M_1
VARIABLES y
VARIANT A_2
INVARIANT I_2
INITIALISATION U_2
OPERATIONS

event$_1 \equiv$
WHEN R_1
THEN W_1
END ;

...
event$_n \equiv$
WHEN R_n
THEN W_n
END ;

event$_{n+1} \equiv$
WHEN R_{n+1}
THEN W_{n+1}
END ;

...
event$_{n+m} \equiv$
WHEN R_{n+m}
THEN W_{n+m}
END

END

FIG. 3.10 – Schéma de modèles en B événementiel

ment en eventB part de l'expression des propriétés les plus abstraites du modèle, puis au fur et à mesure des raffinements les modèles peuvent être séparés en différents sous-modèles. Ils peuvent également être rassemblés plus tard en un autre point du raffinement, bien que l'utilité de ce rassemblement soit discutable, puisqu'il fait perdre la modularité. La modularité d'eventB ne dispose pas des particularités de celle du B classique : ni les différentes sortes d'accès au machines (lecture seule, appel d'opérations), ni la notions de paramètres de machine.

Un modèle eventB est correct s'il ne diverge pas. La *divergence*, mentionnée à plusieurs reprises dans le document de référence de eventB, peut désigner différentes notions :
- L'invariant n'est plus établi (le système n'est plus sûr)
- Les calculs sont incohérents (le système atteint un état incohérent)
- Aucune garde ne peut plus être activée (le système est bloqué)

Valider un modèle eventB revient donc à démontrer qu'il ne diverge pas (les obligations de preuve correspondantes se trouvent plus loin en section 3.2.4).

3.2.3 Raffinement de données et d'opérations

En B classique, le raffinement était à la fois de données et d'opérations, mais sans aucune gestion de complexité au niveau du raffinement d'opérations. Le modèle abstrait initial devait définir suffisamment d'opérations pour que l'implémentation finale ne soit pas trop complexe. En effet il n'était pas possible de morceler une opération pendant un raffinement si elle devenait trop « grande ». La seule possibilité résidait dans l'extension du code de l'opération.

En B événementiel, le raffinement gère la complexité pour les deux aspects :

– Des données correspondant à une vue de plus en plus précise peuvent être introduites au fur et à mesure des raffinements, à l'instar du B classique.
– De nouveaux événements peuvent être définis pour prendre en compte le comportement de ces nouvelles données.

Si nous reprenons l'exemple de l'horloge en figure 3.4, le modèle en B événementiel correspondant peut être décrit comme en figure 3.11.

MODEL Clock **VARIABLES** hour **INVARIANT** hour $\in 0..23$ **INITIALISATION** hour $:\in 10..16$ **EVENTS** update_h = **WHEN** hour <23 **THEN** hour :=hour+1 **END** ; reset_h = **WHEN** hour =23 **THEN** hour :=0 **END** **END**	**REFINEMENT** Clockhm **REFINES** Clock **VARIABLES** h,minute **INVARIANT** minute $\in 0..59$ $\wedge$h=hour **INITIALISATION** h:=11 || minute $:\in 0..45$ **EVENTS** update_h = **WHEN** hour <23 $\wedge$m=59 **THEN** hour :=hour+1 || m:=0 **END** ; reset_h = **WHEN** hour =23 $\wedge$m=59 **THEN** hour :=0 || m:=0 **END** ; update_m = **WHEN** m <59 **THEN** m :=m+1 **END** **END**

FIG. 3.11 – L'horloge en eventB, et son raffinement

Les contraintes pour un raffinement sont d'être plus déterministe que le modèle qu'il raffine, et de ne pas introduire de comportements que le modèle abstrait n'avait pas. Cela signifie qu'il y aura des vérifications à faire pour les raffinements d'anciens événements, et pour les nouveaux événements introduits :

– Pour les anciens événements :

    - Ils calculent encore des valeurs valides
    - Si l'événement raffinant est activé, alors cela correspond nécessairement à un cas où l'événement raffiné était activé
    - L'événement raffinant établit l'invariant, et calcule des états *au plus* du même « domaine » que celui de l'événement raffiné (il ne fait pas « plus qu'il ne faudrait »).
- Pour les nouveaux événements :
    - Ils calculent des valeurs valides
    - Ils établissent l'invariant du modèle
    - Ils ne bloquent pas indéfiniment les événements déjà définis (ceux du modèle raffiné). En effet, dans le cas contraire, les anciens événements ne seraient plus activables, ce qui constitue un changement de comportement par rapport au modèle raffiné.

Ces contraintes sur les événements, nouveaux et anciens, contribuent elles aussi à éviter les phénomènes de divergence dont il était question plus haut. De plus, pour tous les événements du modèle raffiné, pour garantir l'absence de blocage, il faut assurer que la possibilité d'activation d'un événement correspond toujours à une possibilité d'activation dans le raffinement. Autrement dit, le modèle raffinant doit être activable lorsque le modèle raffiné l'était. Cela se traduit de deux manières :

- Contrainte faible : si un événement était activable, alors au moins un événement (peut importe lequel) est activable dans le raffinement.
- Contrainte forte : si un événement était activable, alors il l'est aussi dans le raffinement, ou bien un des nouveaux événements est activable.

La deuxième contrainte est plus forte dans le sens qu'elle interdit un changement dans l'ordre des événements activables. La première était plus souple puisqu'elle permettait de « remplacer » un ancien événement par un autre événement.

Toutes ces contraintes, sur les modèles et les raffinements, donnent lieu à des obligations de preuve en section suivante.

3.2.4 Obligations de preuves

Les OPs pour les modèles abstraits sont indiquées en figure 3.10. Chaque OP est accompagnée par sa signification plus informelle, et correspond au discours que nous avons eu dans les sections précédentes.

Notons qu'une autre lecture, purement logique, des OPs d'existence des valeurs calculées peut être faite : en effet le résultat (prédicat avant-après) de l'action est ajouté dans l'OP d'invariant parmi les hypothèses. S'il est incohérent avec les autres hypothèses, alors la preuve est triviale (la proposition fausse permet de déduire n'importe quelle formule). Cela signifie qu'il n'y a plus aucune garantie sur le système s'il atteint un état incohérent. Ces OPs d'existence assurent également que les événements sont capables de s'exécuter, i.e. que le calcul fait par l'action est possible, ce qui se rattache à la vivacité du système. D'où le besoin d'OPs d'existence pour assurer qu'à chaque fois le système ne devient pas incohérent ou ne se bloque pas.

En appliquant les OPs de la figure 3.12 au modèle de la figure 3.11, la validité du modèle pour l'horloge. La remarque est la même pour son raffinement, avec les OPs indiquées en figure 3.13.

La figure 3.14 montre les deux formes de contraintes (faible et forte) possibles pour le raffinement des états bloquants.

L'événement *i* est faisable si la garde est activable, sous l'hypothèse que l'invariant et les éventuelles propriétés sont vérifiées	$P_1 \wedge I_1 \wedge Q_i$ $\Rightarrow \exists x'.prd_x(V_i)$
L'événement *i* conserve l'invariant	$P_1 \wedge I_1 \wedge Q_i \wedge prd_x(V_i)$ $\Rightarrow I_1[x'/x]$
Les valeurs d'initialisation sont faisables	P_1 $\Rightarrow \exists x'.prd_x(U_1)$
L'initialisation établit l'invariant	$P_1 \wedge prd_x(U_1)$ $\Rightarrow I[x'/x]$
À tout moment au moins un événement est activable	$P_1 \wedge I_1$ $\Rightarrow Q_1 \vee \ldots \vee Q_n$

FIG. 3.12 – Obligations de preuve pour un modèle eventB

L'événement raffinant calcule une valeur qui existe	$P_1 \wedge I_1 \wedge I_2 \wedge R_i$ $\Rightarrow \exists y'.prd_y(W_i)$
Lorsque l'événement raffinant est activé, le raffiné l'était également	$P_1 \wedge I_1 \wedge I_2 \wedge R_i$ $\Rightarrow Q_i$
L'événement raffinant établit l'invariant du modèle, et calcule des valeurs du même domaine que l'événement raffiné	$P_1 \wedge I_1 \wedge I_2 \wedge R_i \wedge prd_y(W_i)$ $\Rightarrow \exists x'.(prd_x(V_i) \wedge I_2[x'/x, y'/y])$
Le nouvel événement calcule une valeur qui existe	$P_1 \wedge I_1 \wedge I_2 \wedge R_{n+i}$ $\Rightarrow \exists y'.prd_y(W_{n+i})$
Le nouvel événement établit l'invariant	$P_1 \wedge I_1 \wedge I_2 \wedge R_{n+i} \wedge prd_y(W_{n+i})$ $\Rightarrow I_2[y/y']$
Le nouvel événement n'empêche pas indéfiniment les anciens événements d'être activés	$P_1 \wedge I_1 \wedge I_2 \wedge R_{n+i} \wedge prd_y(W_{n+i})$ $\Rightarrow A_2 \in \mathbb{N} \wedge A_2[y'/y] < A_2$

FIG. 3.13 – Obligations de preuve pour le raffinement d'événements (anciens et nouveaux)

Si la garde d'un ancien événement était activable, alors cela signifie qu'au moins un événement du raffinement est activable	$P_1 \wedge I_1 \wedge I_2 \wedge Q_i$ $\Rightarrow R_1 \vee \ldots \vee R_n \vee R_{n+1} \vee \ldots \vee R_{n+m}$
Si la garde d'un ancien événement était activable, alors son extension par les nouveaux événements dans le raffinement est aussi activable	$P_1 \wedge I_1 \wedge I_2 \wedge Q_i$ $\Rightarrow R_i \vee R_{n+1} \vee \ldots \vee R_{n+m}$

FIG. 3.14 – Obligations de preuve pour le raffinement de l'absence de blocage

Les formules présentées ici sont restreintes à des formes de machines simples. En pratique elles s'appliquent à des chaînes de raffinements qui peuvent définir de nouvelles propriétés, de nouveaux ensembles,... Elles s'appliquent aussi à la modularité d'eventB, que nous ne traiterons pas ici.

Cette introduction courte des deux formalismes (par ailleurs décrits complètement dans la littérature [Abr96a, ROD05]) va nous permettre de décrire les solutions qu'ils apportent à l'expression et à la validation de problèmes à contraintes temporelles.

3.3 B et l'expression de problèmes temps-réel

Nous présentons dans cette section des solutions utilisant B pour modéliser et valider des problèmes temps-réels. Ces solutions se basent sur B, ou l'augmentent pour faire ce traitement.

Nous introduisons les notions ayant trait aux problèmes de temporalité (causalité entre états du système, liens avec une notion de temps ou de durée) et de simultanéité (concurrence, vivacité,...), puis nous faisons un large tour d'horizon sur les problèmes de l'expression de la concurrence en B, et enfin abordons la question de la non-terminaison.

3.3.1 Terminologie

Concurrence

En premier lieu, qu'entendons-nous par concurrence ? Nous prendrons pour définition de la concurrence la possibilité pour deux processus (pris dans un sens très général) de s'exécuter en même temps. Sur les machines modernes, lorsqu' il n'y a qu'un seul processeur, les différents processus du système ne s'exécute bien entendu pas de manière concurrente, ils se partagent le temps de calcul du processeur selon leur priorité. Néanmoins, de la même manière que pour la *vraie* concurrence, cette exécution en temps partagé ne renseigne pas sur la manière dont s'exécuteront les processus, dans quel ordre, à quels moments ils peuvent être interrompus pour faire exécuter un autre processus,etc. Ainsi, quelle que soit la manière dont la concurrence est implantée d'un point de vue concret, les problèmes usuels restent valables, i.e. :

- La mise à jour de variables partagées : on ne sait pas à l'avance à quel moment une variable sera modifiée par rapport à un autre processus
- Les problèmes d'équité : selon que l'on fait l'hypothèse que tous les processus avancent ou que certains peuvent rester bloqués car les autres processus sont plus avantagés (plus rapides, ou ont plus de temps processeur), alors le raisonnement sur le problème diffèrera

Dans le cadre d'une méthode formelle, ces problèmes peuvent devenir des contraintes à faire vérifier par le modèle.

Sûreté, vivacité

Les différents concepts présentés ici sont étroitement liés à la notion de concurrence. Ils permettent de caractériser les comportements de processus agissant les uns sur les autres en concurrence. Afin d'illustrer chacun d'entre eux, nous rappelons le fameux exemple du dîner des philosophes [Dij71]. Cinq philosophes sont attablés pour manger un plat de spaghettis particulièrement retors, et qui de ce fait nécessitent l'utilisation de deux fourchettes pour les manger. Le système est présenté comme suit :

– Un philosophe alterne indéfiniment entre la réflexion, et la consommation de spaghettis
– Chaque philosophe dispose d'une assiette contenant les spaghettis et d'une unique fourchette pour manger
– Les spaghettis, cuisinés de manière particulière, nécessitent deux fourchettes pour manger.

Tout le problème est donc d'assurer que les philosophes pourront arriver à la fin de leur repas. Un philosophe affamé peut donc prendre la fourchette de son voisin de droite pendant que celui-ci réfléchit par exemple. La conséquence en est que deux philosophes voisins ne peuvent pas manger simultanément. L'exemple du dîner des philosophes permet d'illustrer de manière intuitive beaucoup de concepts liés aux systèmes concurrents partageant des ressources :

Sûreté La notion de sûreté exprimée la plus simplement est « quelque chose de mauvais n'arrive jamais ». En d'autres termes, le modèle que l'on propose ne doit pas exhiber un comportement décrit dans le cahier des charges comme non souhaitable.

Dans le cadre de notre exemple, une notion de sûreté peut être qu'aucun philosophe ne prend jamais la fourchette de son voisin de *gauche*.

Vivacité La vivacité est le dual de la sûreté, et s'exprime par "quelque chose de bon est toujours possible". Les propriétés de vivacité d'un modèle permettent de spécifier la progression du programme.

Cela correspond par exemple au fait qu'à tout moment nous pouvons assurer qu'il y a au moins un philosophe qui peut penser ou manger (i.e. faire quelque chose).

Équité Un modèle équitable est un modèle où les processus qui veulent progresser ne sont pas bloqués indéfiniment par d'autres qui seraient favorisés par l'environnement (plus de temps processeur, par exemple).

Dans l'exemple du dîner des philosophes, une absence d'équité signifierait qu'un ou plusieurs philosophes n'ont plus le droit de faire quoi que ce soit (réfléchir, manger, prendre la fourchette du voisin,...).

Interblocage Il s'agit du cas où chaque processus concurrent d'un modèle attend qu'un autre fasse une certaine action (généralement, libérer une ressource critique, modifier une variable,...). Donc puisque tous les processus attendent que les autres progressent, ils sont bloqués entre eux.

Si l'on se réfère à notre exemple, cela correspondrait au cas où chaque philosophe a une seule fourchette, attend d'en prendre une autre à un voisin mais n'a pas l'intention de laisser la sienne.

Famine Les problèmes de famine se divisent en deux catégories : le cas où un processus ne peut pas progresser, ou le cas plus général où le système dans son ensemble ne progresse plus, sans nécessairement être bloqué.

Le premier cas correspond au philosophe auquel sont perpétuellement refusées les fourchettes par les autres philosophes.

Le second cas, plus difficile à détecter, correspond au cas où tous les philosophes ont une fourchette mais jamais deux, bien qu'ils en changent (par exemple ils se départissent de leur première fourchette et prennent l'autre, mais comme chaque voisin fait de même, tous les philosophes se retrouvent à nouveau avec une seule fourchette).

Les problèmes de famine peuvent éventuellement être évité s'ils sont pris en compte dans les propriétés de vivacité.

Selon le formalisme utilisé, les définitions ci-dessus peuvent être plus ou moins formelles. Par exemple, en TLA+ [Lam02], ces propriétés s'expriment en utilisant les connecteurs *always* $\Box$ et *eventually* $\Diamond$:

- L'équité est représentée par les notions de *Weak Fairness* et de *Strong Fairness*, i.e. d'équité faible et d'équité forte. Par exemple, l'équité faible est exprimée par la formule $\Diamond\Box\text{ENABLED}\langle A\rangle_v \Rightarrow \Box\Diamond\langle A\rangle_v)$. Elle signifie que si l'action A est continûment activable alors elle sera infiniment souvent activée. Plus de détails sur ces notions se trouvent dans le livre de Lamport [Lam02]
- La vivacité s'exprime par le fait qu'une propriété F finit toujours par être vérifiée : $\Box\Diamond F$. Notons qu'en TLA+ les propriétés de vivacité peuvent être exprimées comme une conséquence des propriétés d'équité
- La sûreté est exprimée en général par un théorème disant que si un modèle TLA+ vérifie certaines propriétés, alors il vérifie une propriété particulière associée au cahier des charges du modèle. Par exemple, le premier modèle que Lamport [Lam02] introduit est celui d'une horloge. Le modèle exprime un état initial, la manière dont les heures progressent, et une propriété disant que la variable représentant les heures appartient à un certain intervalle d'entier (de 1 à 12). La propriété de sûreté est représentée par le fait que l'état initial de l'horloge et ses progressions possibles permettent de déduire que la variable des heures reste toujours dans l'intervalle 1..12, vérifiant ainsi la propriété de sûreté.

3.3.2 B et la spécification de contraintes temps-réel

Des exemples d'extension de la méthode B pour exprimer la concurrence et/ou le temps-réel ont déjà été proposés par le passé. Il y a globalement deux manières d'étendre B pour ce faire : étendre la sémantique de B, ou bien associer B à un autre formalisme pour bénéficier des avantages de chacun. Ces mécanismes d'extension sont détaillés dans des sections ultérieures, l'objectif étant ici de tracer simplement un point de vue global.

Extensions sémantiques Les extensions sémantiques faites à B peuvent se faire à plusieurs niveaux : les substitutions, la logique utilisée ou au niveau des machines B. Les extensions suivantes se placent dans la catégorie des extensions sémantiques :

- Étendre le champ d'expression de la substitution parallèle $\|$ comme Bodeveix [BFM99], ou Bert [BPR96]. Ces approches sont détaillées en section 3.3.4
- Étendre les machines B en les considérant comme des activateurs d'événements : il suffit pour cela de contraindre ces machines à avoir des traces d'exécution précises, comme Lano et Dick [LD96]. Plus de détails en section 3.3.5
- Nous avons déjà mentionné le B événementiel en section 3.2, en exhibant le fait qu'une des obligations de preuve assurant la correction d'un modèle était une obligation de preuve de vivacité

Extensions par d'autres formalismes Les exemples d'extension suivants se placent plutôt dans la catégorie des adjonctions à B :

- Treharne et Schneider [TS99, TS00, ST02] montrent qu'il est possible d'utiliser CSP pour décrire le contrôle d'exécution des opérations de machines B. Les opérations des machines B deviennent des événements pour les termes de CSP, et l'enchaînement des différents événements est décrit en utilisant la syntaxe de CSP. La correction d'un tel système est assurée par la vérification de l'établissement d'un *invariant de boucle de contrôle*, i.e. une formule permettant d'assurer que la boucle d'exécution ne peut jamais se bloquer. Ainsi on peut modéliser et vérifier que certains schémas d'exécution de machines B fonctionnent sans nécessairement terminer.
- Bodeveix [BFR04] utilise un fragment TLTL [ZM95] (*Timed linear temporal logic*) permettant d'exprimer des propriétés temporelles simples. De ces formules temporelles il déduit une machine B les implémentant. Il suffit alors de vérifier le modèle B pour démontrer la correction fonctionnelle et temporelle du modèle complet.
- Hammad [HJMO03] montre qu'il est possible d'associer un automate temporisé à un modèle event B pour exprimer des contraintes temporelles (progression, changement d'état, durée)

3.3.3 Extensions de la méthode B — gestion du temps

Les diverses extensions présentées ci-après présentent plusieurs manières d'associer un flot de contrôle à un projet B ou event B. Elles ne se veulent pas exhaustives, mais donnent un aperçu assez complet des techniques utilisées pour y parvenir.

Étendre la logique

Lano et Dick [LD96] proposent d'étendre les clauses de B par les suivantes :

- Une clause `TRACES` permettant d'exprimer la trace d'exécution des opérations de la machine. Ces traces sont par exemple de la forme $(a;b;c)^*$, qui signifient que la machine n'a le droit que d'exécuter les opérations a, b et c, consécutivement, et ce un nombre arbitraire de fois.
- Une clause `GUARDS` permettant d'empêcher l'exécution de certaines opérations si elles ne vérifient pas une certaine contrainte. Cette clause semble *a priori* superflue, puisqu'il est déjà possible de spécifier des gardes en B. En fait, la différence se situe dans le fait que cette garde peut aussi contenir un connecteur **count** énumérant le nombre d'exécutions de l'opération. Cela permet ainsi d'exprimer des contraintes simples à propos de l'historique de l'opération ainsi gardée
- Une clause `THREADS` associant à des opérations des gardes et des commandes supplémentaires : lorsqu'une opération est appelée, la garde doit être vérifiée à ce moment, sinon l'appelant est bloqué tant qu'elle n'est pas vérifiée. Puis l'opération est exécutée, et lorsqu'elle est finie elle rend la main à la machine appelante (qui était bloquée tant que l'opération s'exécutait). Ensuite la commande correspondante est exécutée. De cette manière, la commande fonctionne en même temps que la suite des opérations dans la machine appelante. Ce mécanisme permet de définir un opérateur pouvant faire fonctionner deux substitutions en parallèle

– Une clause `HISTORY` contenant des formules de logique temporelle discrète, pour exprimer des contraintes d'équité et de vivacité sur les opérations de la machine.

Ils montrent ensuite que cette extension préserve le raffinement de B. Ils présentent une étude de cas (la cellule de production) reprenant toutes les nouvelles notions présentées avant. Pour résumer, cette extension définit une forme de concurrence en B, avec la possibilité d'exprimer des contraintes sur les traces d'exécution, tout en conservant les mécanismes de raffinement de B. Certaines des notions introduites dans cette extension (déclenchement d'opérations, équité) préfigurent d'ailleurs le B événementiel introduit quelques années plus tard.

Associer une autre logique

Il est également possible d'associer un fragment d'une logique temporelle à un projet B. Butler et Falampin [BF02] montrent comment utiliser une certaine classe des formules de TLTL (*timed linear temporal logic*) pour spécifier des propriétés temporelles avec les machines B. Cette classe correspond aux formules de type *leadsto* et permet de spécifier des propriétés de vivacité. Plus tard, Bodeveix [BFR04] reprend ce travail en ajoutant la possibilité d'exprimer la conjonction de plusieurs événements, et la possible répétition d'évènements. Soit la formule TLTL en figure 3.15, avec P et Q des prédicats et k une constante.

$P \leadsto_{\leq k} Q$	Le fait que la propriété P soit vraie à un temps t implique que la propriété Q sera vraie à un temps au plus $t+k$.

FIG. 3.15 – Formule de type *leadsto*

Butler et Falampin [BF02] spécifient une façon de dériver une formule de logique classique pour l'utiliser dans un projet B. Cette formule a la forme : $P_{time} + k \leq t \Rightarrow Q$ signifiant que si le compteur de temps t a dépassé la limite $P_{time} + k$ (temps où P était vraie plus la limite de temps), alors la propriété Q est vraie.

Les ajouts de Bodeveix [BFR04] se font à deux niveaux :

– la formule B correspondante ne reflétait pas correctement de multiples activations possibles du même événement,
– La conjonction de plusieurs contraintes de temps (de la forme de la figure 3.15) n'était pas prise en compte

La méthode utilisée revient à dériver une machine B représentant la formule temporelle dont il est question. Cette machine contient des variables représentant le moment de l'activation d'un événement, le fait que l'événement est activé ou non (pour garder trace des activations multiples) et une variable représentant le temps. Elle contient aussi la formule B correspondant à la formule temporelle, en utilisant les variables de la machine.

Le modèle prenant en compte la conjonction de formules est obtenu quant à lui en générant une machine pour chaque interaction entre deux événements. En effet, Bodeveix [BFR04] suppose qu'il y a une formule temporelle pour chaque interaction entre deux événements, possiblement le même. Le modèle de machine B résulte de la conjonction parallèle des différentes opérations correspondant à un même événement d'origine, et ce pour chaque événement, i.e. si $P_1, P_2, \ldots, P_n$ sont des événements, alors il y aura une opération OP_i qui sera la conjonction parallèle $P_i \leadsto P_1, \ldots, P_i \leadsto P_n$, pour chaque événement P_i. Les opérations d'avancement du temps sont réunies de la même manière. Enfin le modèle final est obtenu en composant parallèlement l'opération d'origine et l'opération contenant la contrainte temporelle.

Les avantages d'une telle extension sont la disponibilité des contraintes temporelles de type vivacité sans devoir changer la méthode B elle-même, puisque ces mêmes contraintes temporelles sont exprimables par des machines B. Les deux principaux inconvénients sont en revanche la multiplication des machines dans le cas de la conjonction de formules temporelles, comme indiqué plus tôt, et la limitation justement à cette seule classe de contraintes temporelles.

Associer un autre formalisme

Schneider et Treharne [TS99, TS00, ST02] utilisent CSP pour décrire le contrôle d'exécution des opérations de machines B. Les opérations des machines B deviennent des événements pour les termes de CSP, et l'enchaînement des différents événements est décrit en utilisant la syntaxe de CSP. La sûreté d'un tel système est obtenue en vérifiant un *invariant de boucle de contrôle* : cette formule spécifie les conditions selon lesquelles le système ne peut pas se bloquer. Cet invariant modélise les schémas d'exécution selon lesquels les machines B appelées devront fonctionner, indéfiniment (d'où la dénomination *boucle de contrôle*). Plus précisément, la modélisation est séparée en trois étapes :

– Le comportement du modèle, et la façon dont il évolue au cours du temps. À cet effet, une machine introduisant une variable de temps ainsi que des opérations de consultation/avancement du temps est modélisée. Une machine peut donc « avancer le temps » pour indiquer ses propriétés temporelles. C'est nécessaire du point de vue de la modélisation du fonctionnement du modèle, mais il s'agit d'un paradoxe pour le sens commun. En effet les opérations du modèle devraient être « soumises » au temps plutôt qu'être capables de le faire avancer.
  Les machines B qui décrivent le comportement du modèles font appel à cette « machine-horloge » pour spécifier comment elles évoluent, temporellement parlant.
– Les contraintes du système, spécifiées via des machines non-temporisées. Celles-ci sont ensuite incluses dans des machines au dessein similaire, mais rajoutant les contraintes temporelles, comme mentionné ci-dessus.
– Un modèle CSP décrivant les interactions entre les différentes machines, où les passages de messages correspondent aux opérations des machines.

De cette manière, les différentes classes de spécification du système sont bien séparées :

– Ce que font les éléments du système (machines B à la base du fonctionnement du système)
– Les contraintes temporelles sur l'évolution du système (les machines B « temporisées »)
– Le comportement global du système (ordonnancement de l'activation des différents éléments du système).

Valider un tel projet revient à valider d'abord les machines B, puis à générer les traces d'exécution depuis le modèle CSP et de vérifier la correction de ces traces d'exécution par rapport à des contraintes temporelles sur le système dans son ensemble. Les traces d'exécution correspondent aux séquences d'appels d'opérations B autorisées par le modèle CSP.

3.3.4 Extensions de la méthode B — parallélisme

La substitution généralisée de B se rapprochant le plus de la notion de simultanéité est la substitution parallèle. Cette substitution a comme particularité d'avoir été définie après les autres dans le B-Book[Abr96a] en termes de *terminaison*[9] et de *prédicat avant-après*. Soient S et T deux substitutions, avec $\mathcal{VAR}(S) = x$ et $\mathcal{VAR}(T) = y$ (nous représentons chaque ensemble de variables par une seule variable par souci de lisibilité), tels que $x \cap y = \emptyset$:

$$\begin{array}{rcl} S \parallel T & \equiv & trm(S) \wedge trm(T) | \\ & & @\{x,y\}'.(prd_x(S) \wedge prd_y(T) \\ & & \Longrightarrow \{x,y\} := \{x,y\}') \end{array}$$

La substitution parallèle permet de composer deux substitutions pour modéliser le fait que l'ordre d'exécution des deux substitutions n'a pas d'importance, d'où sa qualification de substitution non-déterministe. Deux remarques importantes concernant l'utilisation de cette substitution :

- Les ensembles des variables modifiées (donc à gauche d'une affectation) doivent être disjoints. En effet $[x := 1 \parallel x := 2]$ n'est pas défini, par exemple
- Lors de l'application à un prédicat, ce n'est pas la définition plus haut de cette substitution qui est utilisée, mais une réécriture du parallèle (qui donne le même résultat final).

La première remarque empêche donc d'envisager des variables partagées avec cette définition de $\parallel$. La deuxième remarque indique que lorsque la réécriture aura lieu, les conditions de bord devront être vérifiées : certaines, comme la vérification qu'il n'y a pas de conflit de noms dans les variables, sont normales, alors que la réécriture pour la garde impose de vérifier la terminaison – au sens B – de la substitution que l'on réduit dans la substitution de la garde. Par exemple, $S \parallel (P \Longrightarrow T)$ se réduit en $P \Longrightarrow (S \parallel T)$ à condition de vérifier $trm(S)$.

Disjonction parallèle

Cependant, les travaux de Bodeveix, Filali et Munoz [BFM99] montrent qu'il est possible de donner une définition différente à la substitution $\parallel$. Soient S et T deux substitutions, avec $\mathcal{VAR}(S) = x$ et $\mathcal{VAR}(T) = y$:

$$\begin{array}{rcl} S \parallel T & \equiv & trm(S) \wedge trm(T) | \\ & & @\{x,y\}'.(prd_{x \setminus (x \cap y)}(S) \wedge prd_{y \setminus (x \cap y)}(T) \wedge \\ & & (prd_{x \cap y}(S) \vee prd_{x \cap y}(T)) \\ & & \Longrightarrow \{x,y\} := \{x,y\}') \end{array}$$

Cette nouvelle définition dispose de plusieurs avantages :

- Elle reste compatible avec l'ancienne : lorsque les ensembles de variables modifiées sont disjoints ($x \cap y = \emptyset$), on obtient l'ancienne formule
- Elle conserve de bonnes propriétés : commutativité, associativité, et monotonie par rapport au raffinement. Il est à noter que cette dernière propriété n'est vérifiée que si les variables modifiées sont préservées par le raffinement.

[9] au sens B, i.e. si la substitution est capable d'établir quelque chose

– Les composants de la substitution parallèle sont considérés comme *atomiques*, c'est-à-dire qu'on ne considère pas l'entrelacement des deux. Ainsi il n'est plus nécessaire d'appliquer la réécriture.

Cependant, elle ne suffit pas toujours pour exprimer les propriétés d'une exécution concurrente de deux substitutions. Prenons comme exemple :

$$x := x+1 \parallel x := x+1$$

Avec la sémantique de la substitution parallèle indiquée plus haut, il ne serait pas possible de prouver la formule suivante :

$$\begin{array}{rl} & x = 0 \Rightarrow [x := x+1 \parallel x := x+1](x = 2) \\ \Leftrightarrow & x = 0 \Rightarrow [@x'.(x' = x+1 \vee x' = x+1) \Longrightarrow x := x'](x = 2) \\ \Leftrightarrow & x = 0 \Rightarrow \forall x'.((x' = x+1 \vee x' = x+1) \Rightarrow x' = 2) \end{array}$$

En effet, la sémantique du parallèle introduite par Bodeveix[BFM99] reste une sémantique de *modélisation*, i.e. servant à décrire une transition globale sur les variables, et non à décrire de manière pertinente une sémantique pour l'exécution en parallèle de deux substitutions (avec les problèmes d'ordre d'évaluation que cela comporte).

Conjonction parallèle

Une autre extension sémantique du parallèle, alors noté $\otimes$, est introduite par Bert, Potet et Rouzaud [BPR96].

Soient$\mathcal{VAR}(S) = x \cup z$ et $\mathcal{VAR}(T) = y \cup z$:

$$\begin{array}{rcl} S \otimes T & \equiv & trm(S) \wedge trm(T) | \\ & & @x', y', z'.(prd_{z,x}(S) \wedge prd_{z,y}(T) \\ & & \Longrightarrow \{z, x, y\} := \{z, x, y\}') \end{array}$$

Encore une fois, nous remarquons que lorsque les deux substitutions S et T n'ont pas de variable en commun ($z = \emptyset$), la sémantique de l'opérateur $\otimes$ se réduit à celle du parallèle classique $\parallel$. Lorsqu'en revanche z n'est pas vide (i.e. les deux substitutions spécifient des changements d'état différents en z), alors l'opérateur $\otimes$ permet de spécifier que le résultat du changement de z est l'intersection des changements d'état de z. Ainsi, pour reprendre quelques exemples de l'article de Bert [BPR96] :

– $(v :\in \{1,2,3\}) \otimes (v :\in \{2,3,4\}) \equiv (v :\in \{2,3\})$
– $(v := 1) \otimes (v := 2) \equiv (v :\in \emptyset)$

Ce dernier exemple est important, puisqu'il permet de s'apercevoir que cette sémantique ne convient pas non plus pour décrire l'exécution parallèle de deux substitutions. En effet dans d'autres langages permettant d'exprimer la concurrence nous nous attendrions plutôt à ce que l'exécution de $(v := 1) \parallel (v := 2)$ résulte en le prédicat $v = 1 \vee v = 2$, puisqu'un entrelacement de ces deux affectations conduit effectivement à une valeur de 1 ou de 2 pour la variable v.

Les diverses extensions sémantiques de la substitution parallèle de B présentées ci-avant ne conviennent pas en tant qu'expression de l'exécution concurrente de substitutions, pour les raisons invoquées pour chacune d'entre elles. Il nous reste donc à trouver un opérateur de composition concurrente permettant d'exprimer la notion d'entrelacement des substitutions.

3.3.5 Composition concurrente et variables partagées

Lano et Dick [LD96] introduisent une première notion de composition concurrente par le biais d'un opérateur ||| permettant l'appel asynchrone d'opérations, avec synchronisation lorsque les deux opérations ont terminé leurs traitements. Soient OP_1 et OP_2 des opérations se trouvant dans une machine $N1$ et $N2$, respectivement. Il est possible de définir la composition concurrente $OP\_1 \,|||\, OP\_2$ en exploitant la clause `THREAD` (citée en section 3.3.2), et créant les implémentation $Async_{N_i-imp}$ comme indiqué figure 3.16 (tirée de [LBS96]). Les machines abstraites correspondantes sont similaires, sans la clause `THREAD`.

```
IMPLEMENTATION
   Async_Ni_imp
 REFINES
   Async_Ni
 IMPORTS
   Ni
 OPERATIONS

async_OP_i = skip ;
complete_OP_i = skip ;

   THREAD
     WHILE
        true
      DO
        SELECT
           ANSWER async_OP_i
         THEN
           OP_i
          ; ANSWER complete_OP_i
         END
      INVARIANT
        true
      END
   END
```

FIG. 3.16 – Introduire la concurrence en B

Les clauses `ANSWER` sont des points de la boucle infinie de la machine sur lesquels les machines appelantes doivent se synchroniser. Ainsi dans l'exemple en figure 3.16, une machine appelante peut *déclencher* l'exécution de OP_i en appelant l'opération $async\_OPi$. Puis $Async\_Ni\_imp$ rendra la main à la machine appelante tout en exécutant l'opération OP_i. Pendant ce temps la machine appelante peut faire un autre traitement. La synchronisation avec la fin de l'exécution de OP_i se fait en appelant $complete\_OP\_i$. De toute manière il n'est pas possible pour la machine $Async\_Ni\_imp$ d'exécuter un autre traitement tant que $complete\_OP\_i$

n'est pas appelée : cette contrainte est vérifiée lorsque les obligations de preuve de la machine sont prouvées. Cette mise en oeuvre du parallélisme permet la synchronisation de tous les OP_i appelés en même temps par le connecteur $|||$.

Cette manière de faire est avantageuse pour les raisons suivantes, indiquées de manière informelle :

- Le mécanisme d'inclusion/importation de Lano et Dick [LD96] reste celui du B standard : les substitutions mises en parallèle ont des ensembles de variables disjoints. Il n'y a donc pas d'interférence possible entre deux substitutions mises en parallèle
- Le blocage induit par cette substitution (introduit par $complete\_OP\_i$ dans la version expansée en figure 3.16) garantit qu'aucune autre opération de la machine Ni ne peut être appelée dans le même temps. De cette manière l'invariant de la machine ne peut pas être violé.

Il est donc possible de spécifier une exécution simultanée de plusieurs tâches en B, sous condition (par les contraintes d'inclusion de B) que les ensembles de variables sur lesquels agissent les tâches respectives soient disjoints. Que se passe-t-il en revanche dans le cas de variables partagées ?

Le problème se pose en fait lorsque deux processus consultent et/ou mettent à jour des variables qu'ils ont en commun. En B, cela est interdit par le fait que dans un projet B, il ne soit pas possible d'accéder à une même machine incluse par deux chemins d'inclusion différents. À ce sujet, le B-Book [Abr96a, section 7.2.3] donne un exemple de machine, repris en figure 3.17, pour laquelle une composition parallèle des opérations brise l'invariant, par exemple :

$$\text{PRE } v < w \text{ THEN increment} || \text{decrement END}$$

Les préconditions de chacun des appels d'opération sont respectées, mais la composition des deux peut briser l'invariant, par exemple dans le cas où $v+1 = w$, l'effet des deux opérations aura pour résultat $v = w+1$, ce qui invalide l'invariant de la machine. Notons également que les autres substitutions de composition ne posent pas ce problème. Soit une machine abstraite possédant deux substitutions S et T :

- La composition séquentielle de deux substitutions ne brise pas l'invariant.

  PROOF:
  1. ASSUME: Prédicats P et R
  2. ASSUME: $I \Rightarrow [S]I$
  3. ASSUME: $I \Rightarrow [T]I$
  4. ASSUME: $[S]P \wedge (P \Rightarrow [T]R) \Rightarrow [S;T]R$ ([Abr96a, propriété 9.1.1])
  5. $[S]I \wedge (I \Rightarrow [T]I) \Rightarrow [S;T]I$ (1,$R \equiv I$,$P \equiv I$,4)
  6. $I \wedge (I \Rightarrow [T]I) \Rightarrow [S;T]I$ (2,5)
  7. $I \Rightarrow [S;T]I$ (3,6)

  $\Box$

- La composition en choix borné de deux substitutions ne brise pas l'invariant

  PROOF:
  1. ASSUME: $I \Rightarrow [S]I$
  2. ASSUME: $I \Rightarrow [T]I$
  3. $I \Rightarrow [S]I \wedge [T]I$ (1,2)

```
MACHINE
    M1
 VARIABLES
    v,w
 INVARIANT
    v ∈ ℕ
∧ w ∈ ℕ
∧ v ≤w
 OPERATIONS

increment =
     PRE
        v <w
     THEN
        v :=v +1
     END
     ;

decrement =
     PRE
        v <w
     THEN
        w :=w −1
     END
```

FIG. 3.17 – Exemple de machine pour laquelle on ne doit pas trouver deux chemins d'inclusion différents

4. $I \Rightarrow [S[\!]T]I$ (1,[Abr96a, 4.14])
□

– En revanche, la composition $\|$ peut ne pas préserver l'invariant. En effet, soient S une substitution modifiant une variable x et T une substitution modifiant une variable y (x et y sont distinctes), on a la propriété suivante [Abr96a, théorème 7.1.1] :

$$[S]P \wedge [T]Q \Rightarrow [S\|T](P \wedge Q) \text{ si } x \backslash Q \wedge y \backslash P$$

Si nous nous référons à la figure 3.17, que nous instancions $P \equiv I$ et $Q \equiv I$ et sachant que $I \Leftrightarrow I \wedge I$, alors les conditions de bord ne sont pas respectées, puisque les variables v et w sont libres dans l'invariant I de la machine.

Pour résoudre ce problème, une approche basée sur l'*absence d'interférence* a été proposée à au moins deux moments : par Lano [LBS96, section 4.5], où l'auteur décrit informellement à quelles conditions une machine peut être partagée, et de manière similaire et plus détaillée par Büchi [BB99] où l'auteur introduit la notion de *rôle* permettant de spécifier les différentes classes de changement d'état autorisés par la machine.

Dans les deux cas cependant, la méthode sous-jacente décrite n'est autre que la vérification d'absence d'interférence[10] décrite originellement par Owicki et Gries [OG76]. On en trouve par exemple une description détaillée dans le livre d'Apt et Olderog [AO97].

$$\begin{array}{llcll} \{0 \bmod 2 = 0\} & x := 0 & \| & \{x+2 \bmod 2 = 0\} & x := x+1; \\ & & & \{x+1 \bmod 2 = 0\} & x := x+1 \\ & & \{x \bmod 2 = 0\} & & \end{array}$$

FIG. 3.18 – Exemple simple montrant la présence d'interférence

La figure 3.18 montre par exemple un programme composé de deux sous-programmes concurrents et qui contient une interférence entre les deux sous-programmes. En effet, supposons que nous voulions que ce programme établisse que x est pair ($x \bmod 2 = 0$). Nous avons préfixé chacune des instructions de base de notre programme par la plus faible précondition permettant d'établir notre postcondition. Nous voyons qu'elle est trivialement établie pour le sous-programme de gauche, et qu'à droite elle est établie également si x est pair à l'origine. Pour montrer que la composition des deux sous-programmes continue d'établir la postcondition, il suffit de montrer (comme indiqué par Apt et Olderog [AO97]) que chaque instruction d'un sous-programme ne brise aucune assertion (précondition ou postcondition) des autres sous-programmes. Plus formellement, soient S et T deux sous-programmes, soit s une instruction de base de S (dans le cas de B il s'agirait d'une substitution), $Pre(s)$ la précondition associée et P une assertion de T :

– s n'interfère pas avec P si et seulement si :

$$\{P \wedge Pre(s)\}s\{P\}$$

c'est-à-dire que l'exécution de s (d'où la présence de $Pre(s)$) continue d'établir l'assertion P.

[10]interference freedom

– Si aucun s de S n'interfère avec aucune assertion de T, et si aucun t de T n'interfère avec aucune assertion de S, alors S et T sont sans interférence

Si nous reprenons l'exemple de la figure 3.18, cela revient à vérifier les triplets de Hoare suivants :

1. $\{0 \bmod 2 = 0 \wedge x+1 \bmod 2 = 0\}\ \ x := x+1\ \ \{0 \bmod 2 = 0\}$
2. $\{0 \bmod 2 = 0 \wedge x+2 \bmod 2 = 0\}\ \ x := x+1\ \ \{0 \bmod 2 = 0\}$
3. $\{x+1 \bmod 2 = 0 \wedge 0 \bmod 2 = 0\}\ \ x := 0\ \ \{x+1 \bmod 2 = 0\}$
4. $\{x+2 \bmod 2 = 0 \wedge 0 \bmod 2 = 0\}\ \ x := 0\ \ \{x+2 \bmod 2 = 0\}$

Il est assez facile de voir que le troisième triplet n'est pas vérifié, puisqu'il impliquerait de prouver que $1 \bmod 2 = 0$.

De la même manière, les aménagements proposés par Lano [LBS96] et Büchi [BB99] proposent d'ajouter des obligations de preuve supplémentaires pour vérifier cette absence d'interférence au niveau de l'invariant. Büchi [BB99] propose de rajouter une clause `CONTRACTS` qui permet de spécifier quels différents rôles (ensembles d'appels d'opérations différents) la machine partagée peut jouer.

Soit une machine M_S pouvant être partagée par plusieurs autres machines dont M, la vérification d'absence d'interférence se fait en vérifiant que les opérations de M_S autres que celles appelées par M ne brisent pas l'invariant de M : il s'agit donc de vérifier que les opérations pouvant provoquer une interférence, n'en provoquent effectivement pas :

– Lano [LBS96] propose pour cela de vérifier que toutes les opérations de M_S pouvant modifier les variables de M_S référencées dans l'invariant de M ne brisent pas l'invariant de M
– Büchi [BB99] propose de vérifier que ce sont les *rôles* de M_S que M ne déclare pas (et donc n'utilise pas) qui ne brisent pas l'invariant de M. Les rôles sont des substitutions spécifiant la manière dont les opérations de M_S sont appelées (selon un choix borné, selon une séquence particulière,...)

Dans les deux cas, le résultat est que nous disposons d'un moyen de composer et partager des machines en B.

Cependant, cette manière de faire pose problème lorsque l'on souhaite rajouter une substitution qui compose d'autres substitutions en prenant en compte l'entrelacement. En effet, en se basant sur l'exemple de la figure 3.18, les différents entrelacements possibles des deux sous-programmes permettent également de vérifier la postcondition : en effet l'entrelacement $x := x+1; x := 0; x := x+1$ ne vérifie pas la postcondition donc la composition concurrente des deux sous-programmes ne vérifie pas la post-condition. Nous voyons donc qu'ici la granularité des programmes (ou, duallement, l'atomicité des instructions qui les composent) joue un rôle dans l'expressivité des programmes.

Ce problème apparaît également lors du raffinement : en effet, comme en B le raffinement est un raffinement de données, la manière dont elles sont traitées peut varier d'une machine à son raffinement, comme illustré en figure 3.19.

Il existe bien entendu plusieurs solutions permettant de régler ce problème :

– Par exemple, il serait possible de définir une substitution généralisée de la forme :

$$\begin{array}{rcl} [S;T \,|||\, U;V] & \equiv & S;[T \,|||\, U;V] \\ & & [\!]U;[S;T \,|||\, V] \end{array}$$

```
MACHINE
   M_S
 VARIABLES
   x
 INVARIANT
   x mod2 = 0
 OPERATIONS

Set0 =
     x :=0
     ;

Incr2 =
     x:=x +2
```

```
REFINEMENT
   M_S'
 REFINES
   M_S
 OPERATIONS

Set0 =
     x :=0
     ;

Incr2 =
     BEGIN
       x:=x +1
       ; x:=x +1
     END
```

FIG. 3.19 – Raffinement et granularité

avec bien entendu une expansion similaire dans d'autres substitutions. Cette solution a le désagrément de générer beaucoup de traces d'exécution à vérifier (que l'on peut réduire lorsque les espaces de variables sont disjoints) mais s'intégrerait assez facilement aux outils pour B déjà existants.

- L'approche compositionnelle présentée dans [Qiw96] pourrait également être utilisée dans ce sens, en raison de certaines de ses caractéristiques :
  - Elle est basée sur les méthodes bien connues et comprises de *rely-guarantee* (aussi dites d'*assumption-commitment*)
  - Le raffinement traité est un raffinement en *forward simulation*, qui est aussi le type de raffinement de B (construction d'une spécification concrète à partir d'une spécification abstraite)
  - La conclusion de [Qiw96] laisse entendre qu'il est possible de dériver les conditions de garantie (sur lesquelles les preuves de non-interférence se basent) à partir du système de transition lui-même. Par exemple il serait possible d'utiliser l'invariant de machine et les différentes opérations pour obtenir ces conditions de garantie.
- Une dernière approche, plus en phase avec les développements B tels qu'ils sont faits et compris en pratique, est de garantir qu'à tout moment dans le modèle, deux opérations de la même machine partagée ne sont jamais appelées en même temps. En effet, le renommage de machines (conséquence de la modularité particulière de B) permet de représenter fidèlement des processus du monde physique (comme par exemple les différentes pompes du modèle du *Boiler* d'Abrial [Abr96c])

L'approche que nous choisissons est la dernière, puisque l'adjonction d'une sémantique temporelle en calcul des durées à B permet d'exprimer ce genre de contraintes temporelles. Nous autoriserons l'utilisation de variables partagées, avec comme restriction que ces variables ne peuvent apparaître que dans la même machine. Cette restriction correspond à l'interdiction pour deux machines B différentes d'inclure une même machine, si ces deux machines font partie du même développement.

Après avoir indiqué les tenants de l'introduction de la concurrence en B, nous traitons dans la section suivante des problèmes de non-terminaison.

3.3.6 Non-terminaison

Originellement, la non-terminaison d'un programme a été vue comme un problème plutôt qu'une propriété voulue d'un programme. Cette dualité dans la manière de considérer la terminaison d'un programme se retrouve encore dans la manière dont sont considérés les systèmes permettant de prouver la correction d'un programme. Par exemple, Apt et Olderog [AO97] font la distinction entre correction *partielle* et *totale* d'un programme : cette distinction se retrouve dans les règles de correction pour l'instruction *while*.

Dans le cas de la correction partielle, les formules à démontrer permettent d'établir que l'invariant est établi pour les trois étapes suivantes : à l'entrée de la boucle, à l'exécution du corps de la boucle, sous l'hypothèse que la condition d'arrêt n'est pas vérifiée, et la postcondition sous l'hypothèse que l'invariant et la condition d'arrêt sont vérifiés. Dans le cas de la correction totale, la démonstration ajoute une formule qui permet de garantir la décroissance d'une quantité (liée au test de la condition d'arrêt).

En B également se retrouve cette notion de correction totale. Le *while* est défini à partir d'une substitution de répétition, comme indiqué figure 3.20.

$S\hat{}$	$(S;S\hat{})\ [\!]\ \text{skip}$
WHILE C DO S END	$(C \Longrightarrow S)\hat{};\neg\, C \Longrightarrow \text{skip}$

FIG. 3.20 – Définition du *while* en B

Ainsi que nous l'avons vu en section 3.1.3, la terminaison d'une substitution est vue comme l'ensemble des états dans lequel doit se trouver le programme avant exécution pour que le programme puisse être exécuté jusqu'à son terme.

Cependant, il arrive de plus en plus souvent que les programmes pour lesquels il faille une preuve de sûreté soient des programmes embarqués qui devront fonctionner indéfiniment, et donc qui ne termineront jamais. De ce fait, ce genre de programmes utilisent un mécanisme permettant de définir une répétition infinie d'étapes, soit de manière implicite (Lustre, OCCAM, CSP) ou explicite (ESTEREL, et d'une manière générale tout langage permettant les boucles *while (true) do…*).

De ce fait pour pouvoir traiter de modèles ou le système ne doit jamais terminer, nous devrons affaiblir les contraintes pour la boucle en ne requérant plus la décroissance d'une quantité liée à la condition de boucle.

3.4 B événementiel et temps-réel

3.4.1 Mécanismes du B événementiel

Les mécanismes de B événementiel lui fournissent cependant une expressivité proche des logiques temporelles discrètes, comme LTL par exemple, si l'on considère les événements comme des points d'observation. La continuité peut être modélisée implicitement par des variables qui représentent des horloges et par des gardes qui bornent ces variables à certains intervalles. Des propriétés de vivacité peut aussi être exprimées, sans changement fondamental [AM98] ou en modifiant les mécanismes de raffinement du B événementiel [ACM05].

Abrial et Mussat ont démontré que B pouvait être utilisé pour spécifier facilement des systèmes distribués [Abr96b], avec l'exemple d'un protocole de transmission [AM97]. Cette utilisation de B est déjà un premier pas vers le B événementiel. Ce pas est continué plus tard en montrant que ce que les auteurs appellent «contraintes dynamiques» peuvent être exprimées avec le B événementiel. Dans d'autres méthodes ou domaines, ces contraintes portent le nom de contraintes temporelles, ou de contraintes de vivacité. Ces contraintes dynamiques [AM98] sont traitées en quatre temps :

– Autoriser le raffinement à complexifier le système en y rajoutant de nouveaux états (et donc de nouveaux événements qui les traitent)
– Spécifier les évolutions possibles des états du programme
– Spécifier les états que le système peut atteindre, i.e. l'*atteignabilité* de certains états
– Empêcher les blocages.

Ces quatre points correspondent à la modélisation d'un système, ses contraintes de sûreté ainsi que sa vivacité. Cette dernière est obtenue en spécifiant l'atteignabilité de certains états désirables du système et l'impossibilité pour le système de se bloquer. Les trois derniers points ramènent aux définitions de la section 3.3.1, respectivement la sûreté, la vivacité (voire l'équité) et l'absence d'interblocage. Nous allons maintenant voir brièvement comment ces mécanismes sont obtenus par Abrial et Mussat [AM98].

Raffinement

Le problème du raffinement dans un système dynamique est l'introduction de nouveaux événements pour le système. Pour ceux-ci, la question suivante se pose : à quel événement du système abstrait correspondent-ils ? En fait, il est dit que ces "nouveaux" événements raffinent un événement qui ne fait rien (*skip* en B). En effet, au niveau abstrait, ces événements ne sont pas visibles, et donc sont "contenus" dans un événement invisible, qui semble ne rien faire. D'ailleurs, Abrial [AM98] compare ces événements invisibles aux «stuttering steps» de TLA, connus depuis longtemps, dont la fonction est similaire.

Pour garantir que le raffinement est correct, il faut donc s'assurer que les nouveaux événements ne s'exécutent pas indéfiniment. Si c'était le cas, cela signifierait que dans le système abstrait, *skip* s'exécuterait indéfiniment, ce qui empêcherait les événements déjà présents d'être déclenchés. Le mécanisme proposé par Abrial et Mussat [AM98] pour empêcher cela est d'introduire une clause `VARIANT` contenant une quantité qui doit forcément décroître à chaque déclenchement des *nouveaux* événements. De cette manière le déclenchement des événements déjà présents dans le système abstrait est garanti.

Notons toutefois que si le fonctionnement de cette clause `VARIANT` est similaire à la clause homonyme de la boucle `WHILE` de B, son rôle est différent :

- Dans le premier cas, il s'agit d'empêcher un fonctionnement indéfini des nouveaux événements. Rien n'interdit aux anciens événements de faire croître la quantité diminuée par les nouveaux événements. Dans ce cas le système dans son ensemble peut fonctionner indéfiniment
- Dans le second cas, il s'agit de garantir que la boucle s'arrête forcément.

Dynamique

Une clause appelée `DYNAMICS` contenant un prédicat est introduite pour pouvoir spécifier l'évolution des variables. De l'aveu d'Abrial [AM98], cette clause est redondante avec les événements eux-mêmes (qui sont au fond des prédicats exprimés différemment). Elle permet cependant d'éclaircir un modèle en B événementiel, et éventuellement de faciliter la preuve par l'expression des propriétés importantes.

Modalités

Les auteurs retiennent pour modalités LEADSTO $\leadsto$ et UNTIL pour exprimer des propriétés de vivacité, invoquant la constatation qu'elles suffisent pour exprimer la plupart des contraintes temporelles usuelles. La modalité $P \leadsto Q$, P et Q étant des prédicats, est exprimée par le biais d'une substitution gardée par P, et contenant une boucle dont la garde est la négation de Q. Le corps de la boucle est composé d'un choix borné entre les événements du système. Des contraintes supplémentaires (que nous ne détaillerons pas ici) et les obligations de preuve de cette substitution, une fois prouvées, garantissent que la formule $\leadsto$ dont est issue la substitution est vérifiée.

Le UNTIL est spécifié de la même manière : en fait il est même exprimé à partir d'une construction particulière de LEADSTO.

Absence de blocage

Le raffinement de modèles B événementiel pose un problème supplémentaire : comment garantir qu'un événement n'est pas *trop* raffiné, c'est-à-dire que sa garde est tellement renforcée qu'elle en devient impossible à déclencher ? En effet dans ce cas, si un événement (voire tous) ne peut plus être déclenché, le système peut se bloquer. La solution proposée par Abrial [AM98] est de s'assurer que la garde de l'opération se retrouve dans la disjonction de la nouvelle garde et des gardes de nouveaux événements. En d'autres termes, on s'assure que le déclenchement est toujours possible quelque part, dans l'événement raffiné ou dans les nouveaux événements introduits.

Travaux ultérieurs

Des travaux ultérieurs de Abrial, Cansell et Méry [ACM05] sur le B événementiel résolvent des problèmes posés par les mécanismes introduits plus haut :

- Les introductions successives de nouveaux VARIANTS au cours des raffinements rend la preuve de décroissance de plus en plus difficile : en effet certains nouveaux événements peuvent avoir besoin de modifier les variables déjà présentes. Ils interfèrent avec les anciens variants.
- La multiplication des nouveaux événements peut rendre le modèle plus difficile à analyser et à traiter
- Les nouvelles constructions traitant de contraintes temporelles (LEADSTO et UNTIL) peuvent peut-être être évitées.

Ces problèmes et interrogations sont résolus des manières suivantes :

- Le problème vient en fait de la manière dont sont introduits les nouveaux événements : sont-ils plutôt liés aux nouvelles variables introduites, ou modifient-ils l'évolution des variables déjà présentes (et donc sont liés aux variants) ? Abrial [ACM05] propose de lier plutôt les nouveaux événements aux variants en introduisant des *événements anticipatifs* dans la version abstraite du modèle. Autrement dit, il s'agit d'introduire dans le modèle des événements dont on ne sait presque rien, sauf qu'ils modifieront les variables du modèle. Abrial [ACM05] définit pour ces événements anticipatifs les contraintes suivantes :
  - Ils raffinent *skip* (comme attendu)
  - Ils modifient de manière non-déterministe la variable qu'ils manipuleront plus tard de manière déterministe dans le raffinement
  - Ils ne modifient pas le VARIANT pour cette même variable dans le modèle courant.

  Ces trois contraintes sur les événements anticipatifs permet donc de simplifier le raffinement dans un modèle complexe
- Un mécanisme de *fusion* des événements est défini. Il revient simplement à faire la disjonction de gardes d'événements dont le corps est identique.
- La méthode pour obtenir la vérification d'une contrainte d'atteignabilité (qui correspond en fait à une contrainte de vivacité) est elle illustré par une technique étayée d'un exemple. En quelques mots, l'atteignabilité est déjà spécifiée dans le modèle abstrait par un événement dit « one-shot » (en un coup). Il suffit donc de partir du principe que la propriété recherchée est obtenue en une seule fois, et de raffiner pour compléter le système avec les événements manquants.

  L'exemple étudié par Abrial, Cansell et Méry [ACM05] est celui d'un client dans un restaurant. Pour garantir qu'il sera forcément servi (absence de famine, il s'agit donc bien d'une contrainte de vivacité), le modèle abstrait montre un événement où il est servi immédiatement. Puis les étapes successives de raffinement ajoute des événements supplémentaires où les autres clients sont servis, jusqu'au raffinement final. Les techniques utilisées pour parvenir à ce résultat sont celles décrites dans le même article, pour la simplification des raffinements et la fusion des événements.

Ce dernier article montre donc que le B événementiel peut traiter de problèmes temporels simples sans faire appel à des mécanismes extérieurs. Cependant ces problèmes temporels simples peuvent aussi vus comme de simples problèmes de causalité, puisque la notion de temps n'apparaît jamais. Il est évidemment possible de déclarer des variables qui représenteront le temps, un compte d'horloge,etc mais on en revient aux problèmes de complexité que l'on retrouve pour les approches similaires en B classique (cf section 3.3.3.0).

3.4.2 Automates temporisés

Hammad [HJMO03] présente une approche créée spécifiquement pour exprimer en B les systèmes réactifs temps-réel. Cette démarche est à rapprocher des démarches en section 3.3.3. L'exemple retenu pour l'illustrer est celui du passage à niveau.

Le principe est de coupler l'utilisation d'automates temporisés à un modèle en B événementiel. Un *automate temporisé à invariants* est un automate dont les états sont étiquetés par un invariant. Celui-ci doit être vérifié pour un état donné tant qu'une transition ne le fait pas passer dans une autre transition. Cet automate est temporisé parce qu'il définit des horloges, sur lesquelles des contraintes peuvent être définies dans les invariants d'états, et que les transitions peuvent remettre à zéro ces horloges.

À partir des ces automates temporisés à invariants, Hammad [HJMO03] définit une traduction en automates des régions. Ces automates sont à états finis et permettent de rendre discret un automate temporisé : il suffit de regrouper dans de mêmes états les configurations dont les évolutions ne modifient pas l'état du système. L'avantage immédiat d'une telle traduction est de pouvoir utiliser les outils sur les automates « classiques », puisque les automates des régions sont à états finis. Un de ces outils fait notamment la traduction en un modèle B événementiel.

La traduction se passe de la manière suivante :

- Les différentes régions de l'automate forment un domaine auquel appartiendra une variable décrivant le temps. Il peut y avoir plusieurs variables si plusieurs horloges avaient été définies dans l'automate
- Les événements décrivent les différentes transitions de l'automate : leurs gardes sont des prédicats indiquant l'état courant dans l'automate temporisé ainsi que la région du temps dans laquelle ils se trouvent. Il y a deux sortes d'événements : les événements qui modélisent les états de l'automate, et un événement qui fait avancer l'horloge. Donc à tout moment, deux événéments sont activables, un événement de transition, et un événement d'horloge. Les événements de transition remettent l'horloge à 0. L'événément d'horloge est activable dans les limites imposées par les contraintes sur les transitions. Donc à un moment donné, seul un événément de transition sera activable
- Un événement supplémentaire, *tic*, fait avancer le temps, i.e. fait passer le système d'une région du temps à l'autre, toujours en utilisant les régions temporelles déclarées dans le modèle. La garde de cet événement assure qu'il n'est activé que lorsque nécessaire. Ainsi dans la pratique, les transitions de l'automate sont entrecoupées d'autant d'événements *tick* qu'il y a de régions temporelles à traverser pour un état donné.

Il s'avère que le modèle B obtenu correspond à l'automate des régions minimisé de l'automate initial.

Hammad [HJMO03] suggère ensuite d'étendre le B événementiel pour simplifier la notation de ces machines temporisées :

- une clause `CLOCK` contient les horloges du système. Ces horloges remplacent l'ensemble énuméré décrivant les régions de l'automate temporisé
- Les contraintes portant anciennement sur ces régions deviennent des prédicats bornant sur les valeurs de l'horloge
- L'évènement *tick* devient implicite

En bref, la notation obtenue est une traduction directe de l'automate temporisé à invariants originel. Sachant que le passage de cette notation “continue” à la notation discrète précédente est automatique, la vérification par model-checking devient possible.

Le deuxième intérêt de cet ajout de notation est la possibilité du raffinement. Hammad [HJMO03] définit pour les systèmes de transition une relation de raffinement très proche de celle du B événementiel (cf section 3.4.1.0). Les auteurs définissent à partir de cette relation de raffinement une autre relation, celle-ci pour les systèmes de transition temporisés (STT). Ils illustrent sur un exemple que le raffinement d'un STT par l'autre implique le raffinement entre leurs automates des régions respectifs. La démonstration de ce résultat impliquerait qu'il est possible de relater le raffinement de B et le raffinement de B étendu avec des horloges, ce qui simplifierait la modélisation et surtout la validation de ce genre de système.

3.5 Vue d'ensemble des extensions

Que pouvons-nous retenir des différentes extensions de B et du B événementiel pour traiter du temps et de la concurrence ? Nous pouvons tout d'abord retenir que ces extensions peuvent diviser en deux grandes catégories :

– Les extensions utilisant les mécanismes de B
– Les extensions lui adjoignant une nouvelle sémantique, de manière interne ou externe, devant permettre d'exprimer de nouvelles propriétés.

Les extensions utilisant les mécanismes de B restent assez simples : Bodeveix [BFR04] qui exprime certaines classes de contraintes temporelles, et Büchi [BB99] qui propose un mécanisme de type assumption/commitment. L'intérêt d'utiliser les mécanismes de B est de pouvoir utiliser les outils existants. Le défaut de ce genre d'approche est le risque que les modèles obtenus aient une complexité résultante inadaptée à ces mêmes outils.

Les extensions sémantiques internes permettent d'augmenter l'expressivité de B lui-même. Les diverses extensions du $\|$ illustrent à petite échelle comment changer la sémantique des instructions de B. Ce changement reste cependant dans le cadre théorique de la théorie des ensembles. D'autres extensions touchant à la logique utilisée, ou aux hypothèses implicites faites, démontrent la possibilité d'exprimer plus directement des contraintes de type temporel : atteignabilité, vivacité, simultanéité. Le B événementiel fait par exemple l'hypothèse implicite de non-terminaison du système : des problèmes de vivacité, de blocage apparaissent et deviennent explicite dans la validation du modèle. L'extension de Lano et Dick [LFD96] propose d'utiliser une logique temporelle simple pour la validation des machines, et étend la notion de machine à la notion de *machine threadée*. Il devient donc possible d'exprimer la simultanéité, et de rajouter des contraintes temporelles de vivacité sur les machines.

Les extensions sémantiques externes utilisent B pour sa facilité à exprimer des propriétés mathématiques et lui adjoignent un autre formalisme pour spécifier des séquences d'actions non-terminantes. CSP$\|$B utilise B pour valider les composants d'un modèle, et la partie CSP les mets en séquence. La validation se fait en vérifiant certaines contraintes additionnelles sur l'interconnexion entre les machines B et les processus CSP. Hammad [HJMO03] utilise les automates temporisés pour donner à B la notion de continuité des actions. B y est utilisé pour accueillir les contraintes mises sur les états et transitions, et spécifier des contraintes temporelles additionnelles portant sur ceux-ci. La difficulté ici est de pouvoir faire correspondre le raffinement d'automates temporisés et le raffinement du B événementiel.

Globalement, les extensions utilisant B comme format final restent d'une expressivité assez simple. Le problème réside donc surtout dans la complexité des spécifications B obtenues, qui peuvent être alors plus difficile à valider. Les extensions augmentant B de manière interne permettent d'élargir considérablement le champ d'action de B, en gardant une certaine simplicité de modélisation. Le problème de la validation dépend, lui, de la sémantique utilisée pour l'extension. Les extensions externes ont pour avantage d'utiliser le “meilleur de deux formalismes”. La difficulté vient ici du point de rencontre entre les deux formalismes, où il faut vérifier que certaines bonnes propriétés (modularité, raffinement) sont conservées.

Nous présentons au chapitre suivant une extension interne de B, qui se base sur une logique temporelle continue pour en exprimer la sémantique.

Bibliographie

[Abr74] Jean-Raymond Abrial. Data semantics. In *IFIP Working Conference Data Base Management*, pages 1–60, 1974.

[Abr84a] Jean-Raymond Abrial. The mathematical construction of a program. *Science of Computer Programming*, 4(1), April 1984.

[Abr84b] Jean-Raymond Abrial. Specification or how to give reality to abstraction. *Technique et Science Informatiques*, 1984.

[Abr88] J. R. Abrial. The B tool. In R. Bloomfield ; L. Marshall ; R. Jones, editor, *Proceedings of the 2nd VDM-Europe Symposium*, volume 328 of *LNCS*, pages 86–87, Berlin, September 1988. Springer.

[Abr96a] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[Abr96b] Jean-Raymond Abrial. Extending B without changing it (for developing distributed systems). In Habrias [Hab96], pages 169–191.

[Abr96c] Jean-Raymond Abrial. A steam-boiler control specification problem. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications : Specifying and Programming the Steam Boiler*, volume 1165 of *Lecture Notes in Computer Science*, pages 500–509. Springer-Verlag, Berlin, Germany, 1996, 1996.

[Abr00] Jean-Raymond Abrial. Event driven sequential program construction. MATISSE project, October 2000.

[ACM03a] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal aspects of computing*, 14(3), April 2003.

[ACM03b] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal derivation of spanning trees algorithms. In Bert et al. [BBKW03], pages 457 – 476.

[ACM05] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in event_b. In Helen Treharne, Steve King, and Martin Henson, editors, *ZB 2005 : Formal Specification and Development in Z and B : 4th International Conference of B and Z Users, Guilford, UK*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer Verlag, Apr 2005.

[AFA03] IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.

[AM97] Jean-Raymond Abrial and Louis Mussat. Specification and design of a transmission protocol by successive refinements using B. In Manfred Broy and Birgit Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F : Computer ans Systems Sciences*, pages 129–200. Springer, 1997.

[AM98] Jean-Raymond Abrial and L. Mussat. Introducing dynamic constraints in B. In Bert [Ber98], pages 83–128.

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs (2nd ed.).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[BB99] Martin Büchi and Ralph Back. Compositional symmetric sharing in B. In Wing et al. [WWD99], pages 431–451.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor : A successful application of B in a large project. In *World Congress on Formal Methods 1999*, number 1709 in Lecture Notes in Computer Science, pages 369–387. Springer Verlag, september 1999.

[BBKW03] Didier Bert, Jonathan P. Bowen, S. King, and M. Waldén, editors. *ZB'2003 – Formal Specification and Development in Z and B,International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, Turku, Finland, June 2003. Springer.

[BDM97] Patrick Behm, Pierre Desforges, and Fernando Mejia. *Application de la méthode B dans l'industrie ferroviaire*, chapter III, pages 59–88. Volume 20 of OFTA [OFT97], 1997.

[Ber98] Didier Bert, editor. *B'98 : The 2nd International B Conference, Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science (Springer-Verlag)*, Montpellier, April 1998. LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag.

[BF02] Michael Butler and Jerome Falampin. An approach to modelling and refining timing properties in B. In *RCS02 (Refinement of Critical Systems)*, Grenoble, January 2002. CERT-ONERA.

[BFM99] Jean-Paul Bodeveix, Mamoun Filali, and César Munoz. A formalization of the B method in Coq and PVS. In BUGM99 [BUG99], pages 32–48.

[BFR04] Jean-Paul Bodeveix, Mamoun Filali, and Miloud Rached. Méthodes de spécification de systèmes temps réel en B. In *FAC2004 (Formalisation des Activités Concurrentes)*, Toulouse, Mars 2004. CERT-ONERA. `http://www.cert.fr/feria/svf/FAC/2004/actes.html`.

[BPR96] D. Bert, M.-L. Potet, and Y. Rouzaud. A study on components and assembly primitives in B. In Habrias [Hab96], pages 47–62.

[Büc98] Martin Büchi. The B bank : A complete case study. In *Proceedings of ICFEM98, the Second International Conference on Formal Engineering Methods*, pages 190–199. IEEE Press, December 1998.

[BUG99] *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*. Springer-Verlag, 1999.

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1 :115–138, 1971. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF`.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[DY00] DCS-York, editor. *ZB'2000 – International Conference of B and Z Users*, volume 1878 of *Lecture Notes in Computer Science (Springer-Verlag)*, Helsington, York, UK YO10 5DD, August 2000.

[Gil62] Arthur Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[Hab96] Henri Habrias, editor. *Proceedings of the 1st Conference on the B method*, Putting into Practice methods and tools for information system design, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de recherche en informatique de Nantes.

[HJMO03] Ahmed Hammad, Jacques Julliand, H. Mountassir, and Dieudonné Okalas Ossami. Expression en B et raffinement des systèmes réactifs temps réel. In AFADL2003 [AFA03], pages 211–226.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[Lam02] Leslie Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, July 2002.

[LBS96] K. Lano, J. Bicarregui, and A. Sanchez. Using B to design and verify controllers for chemical processing. In Habrias [Hab96], pages 237–270.

[LD96] Kevin Lano and Jeremy Dick. Development of concurrent systems in B AMN. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, july 1996.

[LFD96] Kevin Lano, J. Fiadeiro, and Jeremy Dick. Extending B AMN with concurrency. Technical report, Dept. of Computing, Imperial College, 1996.

[MCM04] Rafaël Marcano, Samuel Colin, and Georges Mariano. A formal framework for uml modelling with timed constraints : Application to railway control systems. In *SVERTS : Specification and Validation of UML models for Real Time and Embedded Systems*, Lisbon, Portugal, October 2004. (in conjunction with 7th International Conference on the Unified Modeling Language, UML 2004).

[OFT97] OFTA, editor. *Application des techniques formelles au logiciel*, volume 20. Observatoire Français des Techniques Avancéees & Lavoisier TEC & DOC, 1997.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6 :319–340, 1976.

[Pet03] Dorian Petit. *Génération automatique de composants logiciels sûrs à partir de spécifications formelles B*. Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambrésis, December 2003.

[PR98] Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the B method. In Bert [Ber98], pages 46–65.

[Qiw96] Xu Qiwen. On compositionality in refining concurrent systems. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, Berlin, Germany, 1996.

[ROD05] 2005. `http://rodin.cs.ncl.ac.uk/`.

[ST02] Steve Schneider and Helen Treharne. Communicating B machines. In ZB02 [ZB002], pages 416–435.

[TS99] Helen Treharne and Steve Schneider. Using a process algebra to control B OPERATIONS. Technical Report CSD-TR-99-01, Royal Holloway, Department of Computer Science, University of London, Egham, Surrey TW20 0EX, England, June 1999.

[TS00] Helen Treharne and Steve Schneider. How to drive a B machine. In DCS-York [DY00], pages 188–208.

[WWD99] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *Proceedings of FM'99 : World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag). Springer Verlag, September 1999.

[ZB002] LSR-IMAG. *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, Grenoble, France, January 2002.

[ZM95] Ying Zhang and Alan K. Mackworth. Synthesis of hybrid constraint-based controllers. In *Hybrid Systems II*, volume 999 of *LNCS*, pages 552–567, London, UK, 1995. Springer-Verlag.

Chapitre 4

B + Calcul des durées : une sémantique temporelle pour B

Sommaire

Tableaux

Figures

Nous allons présenter dans ce chapitre les aménagements apportés par l'adjonction de DC* à la méthode B et à son récent développement en B événementiel. En réponse aux méthodes présentées pour étendre B temporellement au chapitre 3, nous présentons dans une première section les ajouts que nous faisons à la méthode B. Nous précisons la sémantique en DC* des substitutions, puis nous montrons comment construite une machine B temporisée. Nous présentons ensuite les particularités de la concurrence telle que nous l'avons introduite, et présentons la notion de raffinement temporel. Nous finissons sur quelques remarques concernant le lien entre B événementiel et B+DC, revenons sur ce dont nous aurons traité dans ce chapitre. Nous terminons sur quelques perspectives visant à simplifier les points plus difficiles du formalisme. Des parties de ce chapitre ont fait l'objet d'une publication [CMP04a].

4.1 Extension temporelle de B

Les extensions à B présentées en section 3.3 indiquaient deux approches : soit utiliser les définitions préexistantes de B (différentes sémantiques du $\|$, par exemple), soit utiliser un autre formalisme apportant à B ce qui manque pour l'expression de problèmes temps-réel et/ou concurrents. Notre approche ici est plutôt de définir une sémantique en logique temporelle des substitutions B et ainsi d'obtenir un moyen de prouver la correction temporelle des machines B. Cette définition en logique temporelle, présentée en détail en annexe A est mise en place selon le principe suivant :

- Définir la signification en WDC^* des substitutions B
- S'assurer que les propriétés *fonctionnelles* des substitutions sont conservées (cf annexe A)
- Déduire de la sémantique WDC^* une sémantique DC^* plus adaptée à la preuve de correction de machine : en effet la première utilise des constructions complexes peu utiles pour la démonstration de propriétés temporelles basées sur les intervalles

Dans cette section, nous partons du principe que cette sémantique en WDC^* est déjà à notre disposition, et que nous devons appareiller B pour permettre d'utiliser toute l'expressivité par cette nouvelle sémantique. Ainsi, nous introduisons les substitutions de délai, d'attente réactive (ou synchronisation), de mise en concurrence et de postcondition.

4.1.1 Délai d'attente et attente réactive

Nous pouvons vouloir spécifier que certaines opérations prennent un certain temps avant de pouvoir s'effectuer, comme par exemple l'interrogation de l'état d'un capteur. Nous définissons à cet effet une substitution *delay d*, où *d* est une quantité réelle (mais n'est pas une variable de la machine).

Nous la définissons de la manière informelle suivante (voir annexe A pour la définition formelle) : *delay d* laisse inchangé l'état de la machine dans laquelle elle se trouve, et fait attendre au programme *d* unités de temps.

Comme nous faisons de plus l'hypothèse du vrai synchronisme (les affectations ne prennent pas de temps), il est laissé à la discrétion du développeur l'ajout ou non de délais pertinents pour les opérations prenant un temps non négligeable. Notons cependant que ce dernier choix pourrait être supporté de manière automatique, en associant pour chaque opérateur arithmétique un délai arbitraire.

Dans le cadre de systèmes temps-réels, une autre propriété désirable est de pouvoir réagir en un temps (prouvablement) minimal lorsqu'une certaine condition est vérifiée. Cette condition peut correspondre à une requête, un changement d'environnement,etc. Pour modéliser une telle condition, il est possible d'avoir recours à une structure de contrôle qui surveille régulièrement si la condition est vérifiée. La granularité temporelle du système correspondra donc à la fréquence de cette vérification.

Le problème de cette approche est lié au fait que c'est la boucle qui sert de support à cette construction, et les boucles sont notablement difficile à manipuler formellement. De par le fait, vérifier une simple attente réactive modélisée par une boucle devient difficile. C'est la raison pour laquelle une autre construction, plus flexible, est utilisée pour spécifier cette notion d'attente réactive, et est usuellement notée **await** suivi de la condition sur laquelle réagir.

Nous ajoutons donc à B une substitution *await c* qui permet d'exprimer l'attente réactive. Notons que d'autres langages définissent une version plus longue de cette instruction, de la forme *await c then S end*. Cette construction permet d'exécuter de manière *atomique* l'instruction *S* lorsque la condition *c* est vérifiée. En B, la granularité des opérations n'est pas connue *a priori*, elle n'est visible qu'à l'étape d'implémentation. Pour cette raison, donner la capacité à une substitution possiblement complexe d'être atomique n'est pas avisé. En effet, selon le langage de destination, celles-ci pourront être plus ou moins fiables ou plus ou moins faciles à définir. Nous considérerons donc plutôt la forme simple (*await c*).

L'attente réactive joue également un autre rôle : elle permet la synchronisation dans le cadre d'un système où de la concurrence est présente.

4.1.2 Concurrence

L'usage d'une sémantique temporelle continue a comme avantage d'exprimer facilement la simultanéité (il s'agit d'une conjonction). Il devient donc intéressant d'équiper B d'une construction permettant d'exprimer la simultanéité de plusieurs spécifications, donc une construction de composition concurrente.

Les substitutions précédentes (délai et attente réactive) sont des substitutions de base : elles ne sont pas construites sur d'autres substitutions. En revanche, la concurrence, qui permet d'exprimer la notion de simultanéité, compose des morceaux de programme ensemble, ces programmes pouvant eux-même être définis par le biais de substitutions structurantes (choix borné, séquencement,...). Se pose donc à nous la question de ce qui pourra être composé : sont-ce les machines, les opérations, ou les substitutions ?

La composition concurrente de machines fait peu de sens dans un cadre où celles-ci sont des composants appelables, et non des entités indépendantes les unes des autres, comme dans [LD96]. Dans le cadre du B classique, il est alors plus naturel de choisir de composer entre elles des substitutions (les opérations étant de toute façon des substitutions).

Ensuite il nous faut savoir quelles sont les contraintes qui opèrent sur cette composition concurrence des substitutions. En effet la section 3.3 du chapitre précédent nous indique que la composition peut briser l'invariant de machines, si la sémantique de la composition n'est pas suffisamment liée à la modularité du langage (l'exemple cité étant la composition via $\|$ de deux opérations d'une même machine). Nous préciserons donc les contraintes d'utilisation de la concurrence lorsque nous présenterons plus formellement celle-ci.

Nous rappelons que des travaux introduisant la concurrence en B existent, nous en avons présenté un large panel au chapitre 3. Cependant la concurrence introduite est la plupart du temps triviale : amenée par un formalisme extérieur [ST02], implicite (eventB) ou sans interférence des variables entre elles [LD96]. Les travaux mentionnant une notion de partage de composants ou de synchronisation sont plus rares [BB99, LBS96].

4.1.3 Postcondition

Pour obtenir un prédicat temporel, nous devrons partir d'un état connu atteignable après la substitution. Plus cet état est précis, plus le prédicat temporel sera précis. Comme la méthode que nous utilisons se base sur le calcul de plus faible précondition, nous nous tournons donc vers un moyen d'exprimer une postcondition la plus précise possible. Si l'invariant d'une machine peut effectivement servir de postcondition, il possède néanmoins deux défauts en regard de notre méthode :

- Il est par principe peu précis, puisque qu'il doit rester vrai peu importe l'opération qui a été exécutée. Il informe sur une condition de sûreté des données en général, et non sur l'état précis des données après exécution d'une opération donnée
- Toutes les opérations d'une machine ne manipulent pas forcément toutes les variables de cette même machine : calculer une formule temporelle pour une opération en prenant en compte toutes les variables de la machine peut s'avérer inutilement encombrant.

Pour ces deux raisons nous [CMP04b] avons défini une substitution de postcondition en utilisant les mécanismes de B. Si le principe apporte peu au niveau théorique (le même effet peut être obtenu par un raffinement), il est utile d'un point de vue pragmatique :

- Les postconditions permettent de spécifier une information précise sur le résultat du calcul d'une opération
- Les opérations des machines B peuvent être désormais caractérisées uniquement par rapport aux variables qu'elles modifient.

Ces deux points résolvent les problèmes évoqués plus haut.

De plus, la postcondition, alliée à la précondition, peuvent faire des prédicats idéaux dans le cas où les machines B ne sont qu'une étape intermédiaire avant implémentation : en effet il est possible de générer à partir de machines B un code source dans un autre langage (historiquement, ce furent d'abord C et Ada). Or ce langage peut être équipé de contrats, ou tout du moins en avoir certains traits, comme le décrit le travail de thèse de Dorian Petit [Pet03].

Par exemple, des opérateurs d'assertions peuvent se retrouver en C, en Java ou en Eiffel (même si ils ne recouvrent pas exactement les mêmes notions dans ces différents langages). Ainsi les postconditions peuvent aider à spécifier des propriétés à retrouver dans le code final d'implémentation en vue de simplifier l'étape de test.

Du point de vue théorique, les postconditions sont définies selon la forme générale des substitutions généralisées de B :

$$S \equiv trm(S) | @x'.(prd_x(S) \Longrightarrow x := x')$$

Les variables contenues dans x n'apparaissent pas forcément dans la substitution S, et donc leur absence de changement est reflétée par l'expression $prd_x(S)$.

Nous notons $S \rhd P$ la substitution S caractérisée par la postcondition P. Elle a la sémantique présentée en figure 4.1.

$$\begin{array}{rcl} trm(S \rhd P) & \equiv & [x_0 := x]([S]P) \\ prd_x(S \rhd P) & \equiv & [x_0, x := x, x']P \end{array}$$

FIG. 4.1 – Définition de la postcondition

Nous [CMP04b] montrons que cette définition de la postcondition est cohérente avec les formules résultantes observées : la postcondition caractérise la substitution à laquelle elle est attachée. De plus, la forme des obligations de preuve contenant une postcondition rappelle la forme des obligations de preuve de raffinement. Ces deux dernières observations confirment que les postconditions peuvent effectivement être émulées *via raffinement* : la postcondition serait alors représentée par une substitution généralisée (« devient tel que »), et le code correspondant serait représenté par son raffinement.

Néanmoins c'est pour leur intérêt pratique que nous les utilisons : elles sont un point d'ancrage pour le calcul des formules temporelles. Elles décrivent de manière précise l'état des variables modifiées par la substitution juste après exécution de celle-ci. Ceci nous ramène d'ailleurs aux arguments invoqués au début de cette section (précision, faible encombrement des variables). Nous pouvons de surcroît invoquer l'argument de la clarté comme pour la clause DYNAMICS d'eventB en section 3.4.1.

La machine en figure 4.2 gère un ensemble en diminuant ou augmentant son contenu via les opérations *diminuer* et *augmenter*. L'ensemble contient des entiers naturels et ne doit jamais être vide. L'opération *augmenter* ajoute un élément, passé en paramètre de l'opération, à l'ensemble. Il est obligatoire que l'opération *augmenter* incrémente effectivement la taille de l'ensemble, ce qui explique la précondition qui précise que l'élément fourni n'est pas déjà présent dans l'ensemble. L'opération *diminuer* enlève un élément non déterminé de l'ensemble, mais, pour des raisons pratiques par exemple, il est plus simple d'enlever le plus petit élément.

MACHINE
  Ensemble
VARIABLES
  ensemble
INVARIANT
  ensemble $\neq \varnothing \wedge$ ensemble $\subset \mathbb{N}$

INITIALISATION
  ensemble :={ 0 }

OPERATIONS

diminuer =
  PRE
    card(ensemble) ≥ 2
  THEN
    ensemble :=ensemble $\setminus\{$ min(ensemble)$\}$
  POST
    $\exists$x.(x $\in \mathbb{N} \wedge$ensemble\$0 =ensemble $\cup\{$x$\}$)
  END
augmenter(n) =
  PRE
    n $\in \mathbb{N} \wedge$ n $\notin$ ensemble
  THEN
    ensemble :=ensemble $\cup\{$n$\}$
  POST
    ensemble\$0 =ensemble $\cup\{$n$\}$
  END

  END

FIG. 4.2 – Un exemple de postcondition

En B classique, cette dernière spécification se ferait via un raffinement, nous le faisons ici avec une postcondition.

L'obligation de preuve (PO) de l'opération *diminuer* aura la forme suivante :

PROOF:
1. ASSUME: $ensemble \neq \varnothing \wedge ensemble \subset \mathbb{N} \wedge card(ensemble) \geq 2$

(Hypothèses de la PO)

2. $[ensemble := ensemble \backslash \{min(ensemble)\}) \triangleright \exists x.(ensemble_0 = ensemble \cup \{x\})]$
$(ensemble \neq \varnothing \wedge ensemble \subset \mathbb{N})$

(But de la PO)

3. $[ensemble_0 := ensemble][ensemble := ensemble \backslash \{min(ensemble)\})]$
$(\exists x.(ensemble_0 = ensemble \cup \{x\}))$

(2,Expansion du terme *trm* de la postcondition)

PROOF:
   3.1. $\exists x.(ensemble = (ensemble \backslash \{min(ensemble)\}) \cup \{x\})$

(3, application des substitutions)

   3.2. $x = min(ensemble)$

(3.1, $ensemble \neq \varnothing$)

   3.3. Q.E.D.

4. $\forall ensemble'.$
$([ensemble_0, ensemble := ensemble, ensemble'](\exists x.(ensemble_0 = ensemble \cup \{x\}))$
$\Rightarrow [ensemble := ensemble'](ensemble \neq \varnothing \wedge ensemble \subset \mathbb{N}))$

(2,Expansion du terme *prd* de la postcondition)

PROOF:
   4.1. $\forall ensemble'.(\exists x.(ensemble = ensemble' \cup \{x\}))$
$\Rightarrow ensemble' \neq \varnothing \wedge ensemble' \subset \mathbb{N})$

(4, application des substitutions)

   4.2. $Vrai$

($card(ensemble) \geq 2$)

   4.3. Q.E.D.

□

L'opération *diminuer* vérifie donc bien l'invariant de la machine, et sa postcondition permet de renforcer l'indéterminisme sur l'élément qui est enlevé de l'ensemble.

Après avoir présenté les nouveaux concepts que nous introduisons dans B, la prochaine section traite de leur définition formelle.

4.2 Sémantique temporelle

La sémantique que nous introduisons ici doit servir deux buts :
- Conserver le lien avec la sémantique « théorie des ensembles » et calcul des prédicats de B, et conserver ses propriétés
- Introduire les nouvelles propriétés et contraintes liées au temps, i.e. la concurrence, la non-terminaison possible et bien entendu l'expression de contraintes temporelles

Nous laissons la description de la sémantique en WDC* et les preuves afférentes en annexe A. Nous nous limiterons ici à la description des propriétés en DC* des substitutions, et de la sémantique en transformateur de prédicats des nouvelles substitutions introduites.

4.2.1 Principe de la sémantique

Le principe sous-jacent à cette sémantique temporelle, présenté en détail en annexe A, se base sur une représentation en WDC* d'un ordonnanceur. Plusieurs substitutions peuvent être composées par l'opérateur de concurrence : chacune de ces substitutions correspond alors à un processus abstrait. Chaque processus est représenté par des variables spécifiant s'il est en fonction ou en attente d'être ordonnancé. Des hypothèses sur ces variables sont faites pour simuler le comportement « physique » d'un ordonnanceur (deux processus ne peuvent pas mobiliser le processeur en même temps, ...).

Ensuite, la sémantique de chacune des substitutions est donnée. L'affectation correspond par exemple à un changement atomique, et qui fait avancer d'une phase (au sens de WDC*) le processus dans lequel elle se trouve. Grâce à la formulation en WDC* des substitutions et des hypothèses additionnelles, il est alors possible d'exprimer le comportement de substitutions (construites à partir de substitutions plus simples) mises en concurrence.

La difficulté réside dans la complexité des formules obtenues : cette complexité découle de la possibilité d'exprimer des propriétés sur les intervalles de temps nuls (qui peuvent contenir plusieurs phases). Cependant, les obligations de preuve de B ne s'intéressent en fait qu'à un seul point précis du temps : celui qui se trouve entre les appels d'opérations d'une machine, c'est-à-dire l'invariant. Nous sommes plutôt intéressés par ce qu'il se passe *pendant* le déroulement des appels d'opérations, donc sur des intervalles de temps non nuls. Pour circonvenir à cette complexité des formules de WDC*, un opérateur de projection permet de passer des formules WDC* à des formules DC*, i.e. où les informations de phase ont disparu. Ce sont ces dernières formules qui seront utilisées pour la démonstration de propriétés temporelles au sein des machines B.

La démonstration temporelle de machines B sera donc en fait concentrée au niveau des opérations, puisque ce sont de celles-ci que peuvent être obtenues les formules temporelles. Une machine B validable temporellement sera donc une machine B normale, où les opérations seront équipées de formules DC* qui spécifient les contraintes temporelles à vérifier pour l'opération qui y est associée. Le processus de validation temporelle consistera en l'obtention de formules temporelles à partir du corps des opérations, et à la vérification des spécifications temporelles par rapport à ces formules temporelles extraites.

Une contrainte additionnelle est que cette étape de validation soit compatible avec la validation « classique » de machines B, c'est-à-dire qu'une validation d'une machine B temporelle sera divisée en deux étapes :
- L'étape de validation « classique » qui utilise la sémantique usuelle en transformateurs de prédicats pour vérifier l'invariant
- Une étape de validation temporelle basée sur le corps de opérations pour vérifier les contraintes temporelles spécifiées pour chacune des opérations.

Nous allons donc introduire en premier lieu quelques définitions, puis nous séparerons la présentation en deux étapes : l'une focalisée sur les substitutions purement temporelles, l'autre sur la notion de concurrence.

4.2.2 Notations et définitions

Boucle while

Nous utiliserons une notations condensée pour représenter la boucle while, de deux façons :
- $\mathcal{W}(g,S,I,V)$: il s'agit de la boucle définie par la condition d'arrêt g, le corps S, l'invariant I et le variant V
- $\mathcal{W}(g,S,I)$: il s'agit de la même chose, mais sans le variant (ce qui en outre signifie que la démonstration de l'arrêt de la boucle via décroissance du variant n'est pas requise).

(Non)-terminaison

La notion de terminaison en B correspond à la cohérence logique d'une substitution. Ici nous l'utiliserons au sens plus usuel du terme, c'est-à-dire la possibilité pour un programme de s'achever (terminaison) ou non (non-terminaison). En effet, l'utilisation de substitutions possiblement non-terminantes (boucle, attente réactive) impose de traiter le cas où un programme peut ne pas s'arrêter. Lorsqu'il s'agira de terminaison au sens de B, nous le préciserons comme nous venons de le faire.

La terminaison et la non-terminaison sont définies formellement au tableau 4.1. Elles seront notées $Term(S)$ pour la terminaison de S et $NTerm(S)$ pour la non-terminaison de S. Si $Term(S)$ est vrai, alors S termine *toujours*. Si $NTerm(S)$ est vrai, S ne termine *jamais*.

Substitution S	Terminaison de S
skip	**true**
$x := E$	**true**
delay d	**true**
$g \vert S$	$Term(S)$
$g \Longrightarrow S$	$g \Rightarrow Term(S)$
$S;T$	$Term(S) \wedge Term(T)$
$S [] T$	$Term(S) \wedge Term(T)$
$@x.S$	$\forall x.Term(S)$
$S \Vert T$	$Term(S) \wedge Term(T)$
$S \vert\vert\vert T$	$Term(S) \wedge Term(T)$
$await\, g$	$g \Leftrightarrow$ **true**
$\mathcal{W}(g,S,I,V)$	**true**
$\mathcal{W}(g,S,I)$	$g \Leftrightarrow$ **false**

Substitution S	Non-terminaison de S
skip	**false**
$x := E$	**false**
delay d	**false**
$g \vert S$	$NTerm(S)$
$g \Longrightarrow S$	$g \Rightarrow NTerm(S)$
$S;T$	$NTerm(S) \vee NTerm(T)$
$S [] T$	$NTerm(S) \wedge NTerm(T)$
$@x.S$	$\forall x.NTerm(S)$
$S \Vert T$	$NTerm(S) \vee NTerm(T)$
$S \vert\vert\vert T$	$NTerm(S) \vee NTerm(T)$
$await\, g$	$g \Leftrightarrow$ **false**
$\mathcal{W}(g,S,I,V)$	**false**
$\mathcal{W}(g,S,I)$	$g \Leftrightarrow$ **true**

TAB. 4.1 – Terminaison et non-terminaison

Opérateur de durées

Cet opérateur servira à obtenir à partir des substitutions les formules temporelles en DC* qui y correspondent. Il sera noté de deux manières différentes :
- Dans le cas terminant : $dur(S,P)$ prend pour paramètres une substitution S et un prédicat P en logique classique
- Dans le cas non-terminant : $dur_\infty(S)$ prendra pour seul paramètre une substitution S

Traces temporelles

La notion de « trace » se trouve principalement dans le domaine du model-checking, où une trace est une représentation des états successifs que peut prendre un programme. Nous utiliserons ce terme pour désigner les formules temporelles en DC* extraites des substitutions. La raison en est que, bien qu'abstraites, ces formules temporelles ont une expressivité proche de la notion de traces. Plus précisément, $dur(S,P)$ sera une (représentation abstraite de la) trace temporelle de la substitution S (par rapport à un état connu immédiatement postérieur représenté par la postcondition P).

Préfixes

Pour pouvoir exprimer des propriétés temporelles dans le cas où un processus ne termine jamais, nous avons besoin de pouvoir exprimer des propriétés sur l'ensemble de ses préfixes. À cette fin nous utilisons l'opérateur d'extraction de préfixe $PREF$:

$$\begin{array}{rcl}
PREF(\llbracket S \rrbracket) & \equiv & \llbracket S \rrbracket^* \\
PREF(a \leq \ell) & \equiv & \ell \geq 0 \\
PREF(\ell \leq a) & \equiv & \ell \leq a \\
PREF(\phi \vee \phi') & \equiv & PREF(\phi) \vee PREF(\phi') \\
PREF(\phi \wedge \phi') & \equiv & PREF(\phi) \wedge PREF(\phi') \\
PREF(\phi \frown \phi') & \equiv & PREF(\phi) \vee \phi \frown PREF(\phi') \\
PREF(\phi^*) & \equiv & \phi^* \frown PREF(\phi)
\end{array}$$

Cette opération d'extraction est par exemple bien illustrée par l'extraction de tous les préfixes d'un découpage : il s'agit de tous les préfixes de la première partie de l'intervalle, unis avec tous les préfixes qui ont la première partie de l'intervalle en commun. L'extraction de préfixe pour $\llbracket S \rrbracket$ est peut-être plus difficile à saisir : $\llbracket S \rrbracket$ implique que $\ell > 0$. Il pourrait être suffisant de spécifier pour le préfixe de cette formule $\int S = \ell \wedge \ell \geq 0$. Il est en fait plus efficace d'utiliser la forme itérée de la formule, pour simplifier l'étape de raisonnement en utilisant directement les axiomes et théorèmes liés à l'itération pour faire la correspondance avec d'autres sous-intervalles en disjonction. Par exemple, soient les deux formules suivantes :

– $(\llbracket S \rrbracket)^* \vee (\llbracket S \rrbracket \frown \llbracket \neg S \rrbracket)^*$
– $(\int S = \ell \wedge \ell \geq 0) \vee (\llbracket S \rrbracket \frown \llbracket \neg S \rrbracket)^*$

Les deux parties de ces formules comprenant S sont équivalentes, cependant la première est plus facile à utiliser pour obtenir $((\llbracket S \rrbracket \vee \llbracket S \rrbracket) \frown (\llbracket S \rrbracket \vee \llbracket \neg S \rrbracket))^*$, ce qui se réduit en $(\llbracket S \rrbracket \frown \mathbf{true})^*$. C'est donc l'utilisation de l'itération dans les formules qui motive cette forme pour l'extraction des préfixes d'une formule de la forme $\llbracket S \rrbracket$. Remarquons enfin que la notion de préfixe s'accorde naturellement avec la notion de trace temporelle telle que nous l'avons décrite plus haut.

4.2.3 Sémantique en DC^* des substitutions

À l'aide des définitions introduites plus tôt, nous pouvons maintenant définir les traces temporelles des substitutions. Comme nous introduisons de nouvelles substitutions, nous devons également indiquer leur sémantique en transformateur de prédicat.

GSL	$[\mathrm{GSL}]P$	$dur(\mathrm{GSL},P)$
skip	P	$\lceil\rceil$
$x := E$	$P[x := E]$	$\lceil\rceil$
delay d	P	$(\ell = d) \wedge \lceil P \rceil$
$C\|S$	$C \wedge [S]P$	$dur(S,P)$
$C \Longrightarrow S$	$C \Rightarrow [S]P$	$dur(S,P)$
$S;T$	$[S]([T]P)$	$dur(S,([T]P))^\frown dur(T,P)$
$S[]T$	$[S]P \wedge [T]P$	$\vee\ dur(S,P) \vee dur(T,P)$ $\vee\ NTerm(S) \Rightarrow dur(T,P)$ $\vee\ NTerm(T) \Rightarrow dur(S,P)$
$@x.S$	$\forall x.[S]P$	$\exists x.dur(S,P)$
$S\|\|T$	Transformé selon des règles de réécriture	Transformé selon des règles de réécriture
$S \|\|\| T$	$[S \|\|\| T]P$	$dur(S,P) \wedge dur(T,P)$
await C	$C \Longrightarrow P$	$\lceil \neg C \wedge P \rceil^*$
WHILE C DO S VARIANT V INVARIANT I	$I \wedge C \Rightarrow [S]I$ $I \Rightarrow V \in \mathbb{N}$ $I \wedge C \Rightarrow [n := V][S](V < n)$ $I \wedge \neg C \Rightarrow P$ I	$dur(S,I) \Rightarrow \ell > 0$ $\wedge dur(S,I)^*$

TAB. 4.2 – Calcul d'une formule de durées à partir d'une postcondition P, dans le cas terminant

Le tableau 4.2 présente cette sémantique, ainsi que les traces temporelles des substitutions dans un cadre terminant, i.e. la substitution considérée est supposée terminer.

Les définitions fonctionnelles des nouvelles substitutions introduites au tableau 4.2 ont les significations suivantes :

– Le délai, fonctionnellement parlant, ne fait rien : il ne change pas l'état de la machine
– L'attente ressemble beaucoup à la garde, bien qu'elles aient des rôles différents. Cette ressemblance correspond à la notion d'obligation de vérité pour la condition gardée ou attendue. En effet le code gardé n'est exécuté que si la garde est vérifiée, donc, temporellement parlant, si ce bout de code est en train d'être considéré alors il est possible de supposer qu'à un moment donné la garde a été vérifiée. De la même manière, si le code qui suit l'attente réactive est en train d'être considéré, alors cela signifie qu'il est possible de supposer qu'elle a été vérifiée. La vérification elle-même de cette garde est liée à la propagation de la garde lors du calcul de plus faible précondition. De ceci nous pouvons dire que l'attente peut être vue comme une garde, fonctionnellement parlant, qui dispose également d'une dimension temporelle
– La substitution concurrente : il s'agit en fait d'une transformation en une substitution sans concurrence. Cette transformation aboutit à une substitution définissant au moins la même dynamique que la substitution originale. Par conséquent, la validité de la substitution transformée implique la validité de la susbtitution concurrente. Le fonctionnement de cette transformation est indiqué plus loin en section 4.4.

Explicitons ensuite les formules temporelles qui peuvent être calculées à partir des différents substitutions :

- L'affectation (et le skip) ne prennent aucun temps, ce qui correspond à l'hypothèse du vrai synchronisme
- Le délai maintient la postcondition connue pendant le nombre de secondes spécifié
- La précondition et la garde ne sont vrais qu'en un point du temps. Notons que leur sémantique en WDC* illustre cela. Ils n'apparaissent donc pas dans la forme calculée.
- La séquence est représentée par un découpage d'intervalle, comme l'on doit s'y attendre
- Le choix borné présente une alternative dans les intervalles, donc les deux chemins devront être utilisés pour la vérification des propriétés temporelles d'une opération. De plus, la possibilité que l'un des deux chemins ne termine pas impose que l'autre termine, pour qu'effectivement le contexte reste celui d'une substitution terminante
- Pour le choix non-borné, nous conservons simplement la quantification universelle, en considérant que toutes les alternatives doivent être terminantes.
- La substitution parallèle peut être réduite. Elle est de nature très abstraite et indépendante de toute notion de durée, donc nous laissons son traitement au mécanisme usuels de B
- La concurrence est définie en termes fonctionnel et temporel. Temporellement parlant, les deux traces temporelles des substitutions concurrentes doivent être vraies « en même temps ». Leurs traces sont donc mises en conjonction. La définition fonctionnelle de la substitution concurrente est définie plus loin, par une transformation en un entrelacement de substitutions usuelles
- L'attente réactive définit une attente de longueur inconnue a priori, pendant laquelle la condition d'attente n'est pas vérifiée ($\neg C$) et la postcondition P n'évolue pas
- La boucle est représentée par une répétition arbitraire de son corps. Notons que cette répétition abstrait la connaissance indirecte du moment de sa terminaison : en effet dans certains cas où le corps est simple, le nombre de boucles peut être prédit (comme dans le cas du calcul de complexité d'un algorithme simple, par exemple). Puisque nous restons généraux (pas d'hypothèses sur la simplicité du corps de la boucle), nous en resterons à cette abstraction sur le nombre d'itérations. De plus, nous rajoutons une obligation de preuve servant à s'assurer que le corps de la boucle dure un temps non-nul, pour éviter les phénomènes de Zénon.

Les traces temporelles des substitutions dans un cadre non-terminant sont présentées au tableau 4.3.

GSL	dur_∞(GSL)
skip	**false**
$x := E$	**false**
delay d	**false**
$C \mid S$	$dur_\infty(S)$
$C \Longrightarrow S$	$dur_\infty(S)$
$S;T$	$\vee \begin{cases} \vee\, dur_\infty(S) \\ \vee\, PREF(dur(S,(\mathbf{true})) \\ \vee\, dur(S,(\mathbf{true}))^\frown dur_\infty(T) \end{cases}$ $\vee\, Term(S) \Rightarrow \begin{cases} \vee\, dur(S,(\mathbf{true}))^\frown dur_\infty(T) \\ \vee\, PREF(dur(S,(\mathbf{true})) \end{cases}$ $\vee\, NTerm(S) \Rightarrow dur_\infty(S)$
$S[\!]T$	$\vee\, dur_\infty([S]) \vee dur_\infty(T)$ $\vee\, Term(S) \Rightarrow dur_\infty(T)$ $\vee\, Term(T) \Rightarrow dur_\infty(S)$
$@x.S$	$\exists x.dur_\infty(S)$
$S \parallel T$	Transformé selon des règles de réécriture
$S \mid\mid\mid T$	$dur_\infty(S) \vee dur_\infty(T)$
await C	$[\![\neg\, C \wedge P]\!]^*$
WHILE C DO S INVARIANT I	$\vee \begin{cases} Term(S) \wedge (dur(S,I) \Rightarrow \ell > 0) \Rightarrow \\ (dur(S,I))^{*\frown} PREF(dur(S,I)) \end{cases}$ $\vee\, NTerm(S) \Rightarrow dur_\infty(S)$ $\vee \begin{cases} dur_\infty(S) \wedge (dur(S,I)) \Rightarrow \ell > 0) \wedge dur(S,I) \Rightarrow \\ \begin{cases} \vee\, (dur(S,I))^{*\frown} dur_\infty(S) \\ \vee\, (dur(S,I))^{*\frown} PREF(dur(S,I)) \end{cases} \end{cases}$

TAB. 4.3 – Calcul d'une formule de durées dans le cas non-terminant

Les formules pour le cas non-terminant sont plus complexes car certaines distinctions sur la terminaison connue ou non de parties du programme doivent être prises en compte :

- L'affectation, le skip et le délai terminent forcément. Il est donc normal que leur trace dans le cas non-terminant soit fausse
- La précondition et la garde ont la même justification que dans le cas terminant : elles ne « durent » que pendant un point du temps
- Pour la séquence, plusieurs cas doivent être considérés :
  - Il n'est pas possible de savoir laquelle des parties de la séquence est non-terminante. Nous devons donc prendre en compte le cas où la première ne termine pas, le cas où un préfixe de la première est nécessaire s'il y a une autre substitution en concurrence, et le cas où c'est la deuxième qui ne termine pas
  - Si la première substitution termine forcément, alors nous devons prendre en compte un préfixe d'exécution, ainsi que le cas où la deuxième substitution de la séquence ne termine pas
  - Si seule la première substitution ne termine pas, alors le calcul dans le cas non-terminant se limite à celle-ci.

Les **true** qui apparaissent ici sont utilisés pour fournir une postcondition la plus générique possible à S, à partir de laquelle calculer les traces. En effet il n'est pas possible d'extraire une précondition de T, puisque les parties de la formule où elle n'est pas considérée sont aussi les cas où elle est supposée ne pas terminer

- Le choix borné dans ce cas-ci doit prendre en compte le cas où l'un des deux chemins termine
- Le choix non-borné est réellement non-terminant si toutes les alternatives possibles sont non-terminantes
- La justification ici pour la substitution parallèle est la même que pour le cas terminant
- Pour la substitution concurrente, la justification est la même que pour le cas terminant
- Pour la boucle, son corps doit durer un temps non nul. De plus, plusieurs cas de figures peuvent se produire :
  - Le corps termine : dans ce cas nous devons prendre en compte le fait qu'en réalité c'est une substitution concurrente qui ne termine pas, et donc nous devons calculer un préfixe pour une exécution de la boucle
  - Le corps ne termine pas : donc nous obtenons sa trace dans le cas non-terminant
  - Ou enfin, il y a des cas où le corps termine, et d'autres où il ne termine pas. Nous devons alors prendre en compte les cas où l'exécution parvient dans un état non-terminant, et les cas où nous devons avoir un préfixe d'exécution de la boucle, dans le cas où il s'agit d'une substitution concurrente qui ne terminerait pas.

Toujours dans le cas non-terminant, certaines substitutions ont des formules de durées contenant plusieurs disjonctions, certaines gardées par une analyse de la terminaison (ou non) de parties de ces substitutions. Ces disjonctions gardées sont superflues dans la sémantique en WDC* des substitutions correspondantes. Mais comme nous utilisons celles-ci dans un cadre DC*, elles permettent de compléter et simplifier la validation (*NTerm* et *Term* sont basés sur la syntaxe de la substitution).

Avec la possibilité d'extraire les propriétés temporelles des substitutions, vient la capacité à vérifier des contraintes temporelles dans le cadre de DC*. Nous présentons dans la section suivante comment ajouter ces contraintes temporelles à une machine B.

4.3 Construction des machines B temporisées

Le principe derrière la sémantique que nous avons présentée plus tôt est d'exprimer les substitutions sous une forme validable, DC* en l'occurrence. Cette sémantique est de nature opérationnelle, et est de ce point de vue orthogonale à la sémantique en transformateurs de prédicats des substitutions. Nous devons donc introduire de nouvelles clauses pour exprimer les propriétés temporelles que nous souhaitons voir vérifiées par le système en conception.

4.3.1 Schéma de machine temporisée

La figure 4.3 indique les clauses supplémentaires, du point de vue de la machine et des opérations.

La machine est équipée d'une clause `TIME_INVARIANT` qui désigne la propriété temporelle qu'elle vérifie à tout moment. Chacune des opérations est équipée, en plus de la précondition et

```
MACHINE
  M
TIME_INVARIANT
  XM
INVARIANT
  IM
OPERATIONS

opM_1 =
    PRE PM_1
    THEN SM_1
    POST QM_1
    TIMING XM_1
    END

  END
```

FIG. 4.3 – Machine B temporisée

de la substitution, des clauses supplémentaires suivantes :

- `POST` : elle représente la postcondition vérifiée par l'opération. Elle doit permettre de caractériser plus finement le comportement de l'opération, et servira au calcul des propriétés temporelles de l'opération. Elle est exprimée sous forme d'un prédicat de B.
- `TIMING` : elle représente la formule temporelle que l'opération doit vérifier. Elle est exprimée sous forme d'un prédicat du calcul des durées. Nous interposons une clause par opération pour deux raisons : pour simplifier la preuve de la clause temporelle de la machine, et pour prendre en compte le cas où l'opération peut ne pas terminer.

La machine de la figure 4.3 sera *temporellement* correcte si les formules suivantes sont démontrées :

- l'opération opM_1 vérifie sa spécification temporelle, dans le cas où elle termine et dans le cas où elle ne termine pas.

$$\begin{array}{l} Term(SM_i) \Rightarrow dur(SM_i, QM_i) \Rightarrow XM_i \\ \wedge NTerm(SM_i) \Rightarrow dur_\infty(SM_i) \Rightarrow XM_i \end{array}$$

- $XM_1 \vee \ldots \vee XM_n \Rightarrow XM$: les opérations garantissent que la condition de sûreté temporelle de la machine est vérifiée. La disjonction en hypothèse est la conséquence du fait que certaines opérations puissent ne pas être appelées. Donc chacune d'entre elles, séparément, doit permettre de s'assurer que la propriété temporelle de la machine entière est vérifié.

Ces contraintes sont bien entendues soumises à l'existence des clauses qui y correspondent :

- Une clause `TIME_INVARIANT` absente est remplacée par **true**, donc l'implication indiquée ci-avant est trivialement vérifiée
- Une clause `TIMING` absente (de op_i) peut être remplacée par le calcul direct des propriétés temporelles de l'opération ($dur(SM_i, QM_i)$)

– Une clause `POST` absente (de op_i) pourra être remplacée par l'invariant (puisque l'invariant de machine est une postcondition commune à toutes les opérations) pour calculer la propriété temporelle de l'opération ($dur(SM_i, IM)$).

Reste le cas où le corps de l'opération contient un appel d'opération d'une machine incluse. La validation se passe alors comme suit :

– La validation fonctionnelle reste similaire à celle de B, c'est-à-dire que l'appel d'opération est expansé
– La validation temporelle en revanche bénéficie du fait que la sémantique soit opérationnelle. Dans le calcul de $dur(OP, P)$, plutôt que de remplacer l'appel d'opération par son corps, nous remplaçons l'expression complète par la contrainte temporelle vérifiée par l'opération. Si par exemple la trace temporelle de $dur(opM_1, Q)$ doit être calculée, il suffit de remplacer l'expression entière par XM_1

4.3.2 Exemple

La figure 4.4 présente une machine capable d'ouvrir et de fermer une valve d'admission de gaz, pour un brûleur par exemple. Les opérations ont des temps minimaux d'activation, à la fin desquels la valve atteint l'état demandé.

Les corrections fonctionnelle et temporelle sont facilement vérifiées :

– La correction fonctionnelle est obtenue par la vérification de l'OP d'initialisation, et des OPs d'opérations :
  – $valve \in Valve\_state$
  $\Rightarrow [valve := closed](valve \in Valve\_state)$
  – $valve \in Valve\_state \wedge valve = closed$
  $\Rightarrow [(\text{delay } 0.5; valve := open) \rhd valve = open] valve \in Valve\_state$
  – $valve \in Valve\_state \wedge valve = open$
  $\Rightarrow [(\text{delay } 0.1; valve := closed) \rhd valve = closed] valve \in Valve\_state$

  Ces trois prédicats sont trivialement vérifiés.
– La correction temporelle est obtenue en calculant la formule de durées pour chacune des opérations
  – Soit $S \equiv \text{delay } 0.5; valve := open$.
  $((Term(S) \Rightarrow dur(S, valve = open)) \wedge (NTerm(S) \Rightarrow dur_\infty(S))) \Rightarrow \ell \leq 0.5$
  $((\mathbf{true} \Rightarrow (\ell = 0.5 \wedge \llbracket \mathbf{true} \rrbracket) \wedge (\mathbf{false} \Rightarrow dur_\infty(S))) \Rightarrow \ell \leq 0.5$
  $\ell = 0.5 \Rightarrow \ell \leq 0.5$
  L'expression complète $dur_\infty(S)$ se réduit en $\ell = 0.5$, par le biais des sous-disjonctions qui utilisent $PREF$ pour le séquencement. Mais comme dans les hypothèses $NTerm(S)$ est faux, le cas non-terminant se réduit finalement en **true**, donc la validation se limite naturellement ici au cas terminant.
  – Soit $S \equiv \text{delay } 0.1; valve := closed$.
  $((Term(S) \Rightarrow dur(S, valve = closed)) \Rightarrow \ell \leq 0.1$
  $((\mathbf{true} \Rightarrow (\ell = 0.1 \wedge \llbracket \mathbf{true} \rrbracket) \Rightarrow \ell \leq 0.1$
  $\ell = 0.1 \Rightarrow \ell \leq 0.1$

  Les deux OPs temporelles sont similaires pour les deux opérations. L'OP pour la version non-terminante est trivialement vérifiée, puisque $NTerm(S)$ se réduit à faux.
  Le calcul de la trace temporelle se fait comme suit, pour l'opération d'ouverture de valve :

```
MACHINE
  Gas_valve
VARIABLES
  valve
SETS
  Valve_state ={ open, closed }
INVARIANT
  valve ∈ Valve_state
TIMEINVARIANT
  true
INITIALISATION
  valve :=closed
OPERATIONS

open_valve =
     PRE
        valve =closed
     THEN
        delay 0.5
       ; valve :=open
     POST
        valve =open
     TIMING
        ℓ≤0.5
     END
     ;

close_valve =
     PRE
        valve =open
     THEN
        delay 0.1
       ; valve :=closed
     POST
        valve =closed
     TIMING
        ℓ≤0.1
     END

  END
```

FIG. 4.4 – Une machine B temporisée pour une valve de gaz

$dur(S, valve = open) \rightsquigarrow dur(\text{delay } 0.5, open = open) \rightsquigarrow \ell = 0.5$
Nous pouvons observer que le changement d'état (la valve passe de l'état fermé à l'état ouvert) se place à la fin de la substitution. La trace temporelle illustre que pendant 0.5 secondes, l'état de la valve n'est pas connu. Cette manière de spécifier les changements d'états est plus courte que celle de[LD96], où les changements d'états sont divisés en le démarrage du changement d'état, sa continuation puis sa fin, avec les opérations correspondantes. Cette distinction est nécessaire pour capturer à la fois les points du temps où surviennent les changements d'état, et les intervalles du temps où les états sont stables.
Dans notre exemple, avec l'utilisation de DC* pour la spécification et la vérification d'intervalles du temps, nous pouvons écrire des spécifications de ce genre.

Nous utiliserons la machine que nous venons de présenter pour proposer un modèle de brûleur à gaz, validé fonctionnellement et temporellement. Le prochain exemple illustrera l'utilisation de la concurrence, présentée dans la section suivante.

4.4 Concurrence

Nous précisons dans cette section comment calculer l'expression $[S \,|||\, T]P$ du tableau 4.2.

4.4.1 Concurrence et plus faible précondition

Si nous introduisons la concurrence en B pour pouvoir exprimer le caractère simultané de l'exécution d'au moins deux parties d'un système, nous devons disposer d'un moyen pour vérifier la correction fonctionnelle de ce système. Il est donc nécessaire de pouvoir donner une assise formelle à la composition concurrente de plusieurs processus. Nos critères pour choisir ce moyen sont précis : il doit pouvoir être automatisable, et compatible avec la sémantique en plus faible précondition de B. Ce moyen devra pouvoir prendre en compte l'entrelacement des substitutions et l'introduction de délais. En effet, si nous tenons compte de l'hypothèse du vrai synchronisme, une affectation prendra toujours moins de temps qu'un délai aussi petit soit-il.

Le chapitre 3 a été l'occasion de traiter des problèmes de composition concurrente (notions de parallélisme, problème des variables partagées). Nous présentons ici quelques solutions possibles ainsi que les raisons pour lesquelles elles conviennent ou pas.

La méthode d'Owicki et Gries [OG76] : il s'agit de vérifier que toutes les assertions locales d'un processus sont conservées par chacune des instructions des autres processus, en supposant que ces instructions sont exécutées dans leur précondition. En B cela reviendrait à annoter chaque étape d'une séquence d'un processus par sa plus faible précondition (par rapport à l'invariant de machine) et à vérifier que les instructions des autres processus conservent cette plus faible précondition. L'avantage de cette méthode est de permettre de prouver l'interaction harmonieuse de plusieurs processus, mais son principal inconvénient est de ne pas permettre de calculer effectivement la plus faible précondition de l'ensemble des processus.

La méthode de Lamport de vérification du plus faible invariant [Lam90] : cette méthode, si on la rapporte à B, généralise l'invariant entre chaque appel d'opération d'une machine, en un invariant qui doit être maintenu aussi *pendant* chaque appel d'opération. De cette manière l'absence d'interférence est ici une conséquence naturelle de la méthode. De plus, il n'y a aucune hypothèse sur l'atomicité des instructions, ce qui permet de découvrir plus facilement des hypothèses cachées ayant été incluses dans la conception du programme (l'auteur indique l'exemple de l'algorithme de la boulangerie). Cependant, le principal inconvénient de cette méthode est de nécessiter des informations sur le flot de contrôle pour pouvoir exprimer des propriétés suffisamment précises : par le biais d'étiquettes pour chaque étape du programme, ou bien en utilisant des variables locales pour simuler le flot de contrôle. B ne dispose pas de tels mécanismes dans le premier cas, et la mise en place des secondes en B alourdirait pour un faible gain les obligations de preuves générées. Notons cependant que les transitions *one-shot* [11] d'eventB se prêteraient mieux à un formalisme de ce genre.

La génération des traces du programme : C'est la méthode utilisée en vérification de modèle. Cela revient à exécuter tous les cas possibles d'entrelacements, et vérifier que chacun d'entre eux vérifie les propriétés spécifiées pour l'ensemble des processus. Le bénéfice de cette approche est son automatisation aisée, mais son défaut est d'être limitée à de petits programmes. Vouloir faire la même chose en B reviendrait à tester les entrelacements pour toutes les valeurs qui peuvent être prises par une variable introduite par un choix non-borné. Il est possible de faire des approximations, mais dès que beaucoup d'entre elles sont nécessaires, la méthode revient à faire de l'interprétation abstraite. Cela a pour effet, à cause des multiples approximations, de risquer de rejeter plus de programmes qui pourtant étaient valides. Or le choix de la vérification de modèle était peut-être lié à ce besoin de précision initialement.

La prochaine section présente la manière dont se fait le calcul d'entrelacement. Ce calcul est une conséquence de la sémantique en WDC* donnée à B.

4.4.2 Calcul d'entrelacement

L'idée est ici de remplacer un ensemble de substitutions composées concurrentiellement par une expression équivalente *fonctionnellement*. Il suffit pour cela d'identifier la manière dont s'entrelacent les substitutions qui sont considérées comme atomiques, puis de faire passer cet entrelacement aux substitutions de structure. Pour cela nous définissons des règles pour réécrire la substitution $S \mathbin{|||} T$ en une substitution plus simple, en itérant jusqu'à ce que le symbole $|||$ n'apparaisse plus.

La réécriture se fait par discrimination sur l'opérateur de séquencement, puisque c'est celui-ci qui dirige la réécriture. En effet, l'annexe A montre que la séquencialisation est dépendante de la conservation d'un état d'un intervalle à l'autre, ainsi que de l'état du processus courant. Nous fermons ici cette parenthèse (voir l'annexe A pour plus de détails).

Nous découpons la présentation de cette réécriture selon les substitutions qui se trouvent en tête d'un séquencement : atomiques, ou non. Si la substitution considérée n'est pas un séquencement, non considérons qu'elle est suivie par *skip*. Cela signifie que si les substitutions en concurrence sont, par exemple, $x := E \mathbin{|||} y := F$, nous les réécrivons en $(x := E; skip) \mathbin{|||} (y := F; skip)$. Le cas de base pour la réécriture d'entrelacement est, pour toute substitution S :

$$(skip \mathbin{|||} S) \equiv (S \mathbin{|||} skip) \equiv S$$

[11]en une seule instruction

De plus, la substitution de concurrence est commutative et associative :

$$(S \,|||\, T) \equiv (T \,|||\, S)$$
$$(S \,|||\, T) \,|||\, U \equiv S \,|||\, (T \,|||\, U)$$

Ceci est une conséquence de sa sémantique en WDC*, puisqu'il s'agit d'une conjonction. Cela nous permet de ne présenter que la moitié des cas : la discrimination sur la substitution de séquencement pourra aussi bien se faire à droite qu'à gauche de l'opérateur de concurrence. De plus, dans le cas de plus de deux substitutions mises en concurrence, l'expansion pourra se faire en choisissant les opérateurs de concurrence à expanser dans un ordre arbitraire.

Substitutions atomiques

Les substitutions purement atomiques sont l'affectation, le skip, l'attente réactive et le délai. Leur entrelacement est présenté au tableau 4.4.

$(S_a;S) \,|||\, (T_a;T)$	$S_a;(S \,|||\, (T_a;T))$ $T_a;((S_a;S) \,|||\, T)$	si $S_?, T_? \equiv \begin{cases} x := E \\ skip \\ \text{await } c \end{cases}$
$(\text{delay } a;S) \,|||\, (T_a;T)$	$T_a;((\text{delay } a;S) \,|||\, T)$	si $T_a \equiv \begin{cases} x := E \\ skip \end{cases}$
$(S_a;S) \,|||\, (\text{delay } b;T)$	$S_a;(S \,|||\, (\text{delay } b;T))$	si $S_a \equiv \begin{cases} x := E \\ skip \end{cases}$
$(\text{delay } a;S) \,|||\, (\text{await } b;T)$	$(\text{delay } a;(S \,|||\, (\text{await } b;T)))$ $[\!] (\text{await } b;((\text{delay } a;S) \,|||\, T))$	
$(\text{await } a;S) \,|||\, (\text{delay } b;T)$	$(\text{await } a;(S \,|||\, (\text{delay } b;T)))$ $[\!] (\text{delay } b;((\text{await } a;S) \,|||\, T))$	
$(\text{delay } a;S) \,|||\, (\text{delay } b;T)$	$\text{delay } a;(S \,|||\, (\text{delay } (b-a);T))$	si $a < b$

TAB. 4.4 – Séquentialisation : substitutions atomiques

Les entrelacements de l'affectation, du skip et de l'attente réactive sont très similaires, raison pour laquelle nous les avons regroupés. Comme dans d'autres formalismes, l'entrelacement de deux affectations résulte en un ordonnancement arbitraire de l'une par rapport à l'autre.

L'attente réactive fonctionne différemment, comme cela sera illustré par la suite : le fait qu'elle puisse être activée à tout moment dans l'exécution fait que son entrelacement la place à tout endroit du code, *y compris si une attente est présente dans l'autre processus*.

Le délai se comporte de la manière attendue : il « laisse passer » les autres substitutions qui ne prennent aucun temps. Lorsque deux délais sont mis en concurrence, il faut d'abord laisser passer la plus petite durée, puis continuer l'entrelacement en prenant en compte la diminution du délai dans le processus concurrent.

Cette présentation de l'expansion montre que la réduction de l'entrelacement se fait des deux côtés. Cela est rendu nécessaire à cause de la possibilité de l'existence de délais des deux côtés de l'opérateur de concurrence. Il s'agit de la manière la plus simple de faire, si nous considérons la sémantique en WDC* qui sous-tend toutes ces substitutions. La réduction des substitutions a lieu des deux côtés (à droite autant qu'à gauche) pour les mêmes raisons.

Substitutions de structure

Nous allons exploiter quelques-unes des propriétés du séquencement de B. Le BBook [Abr96, section 9.1.3] nous présente les propriétés d'absorption du séquencement par rapport aux autres substitutions. Nous les rappelons au tableau 4.5.

$skip;S$	$=$	S	
$(P\|S);T$	$=$	$P\|(S;T)$	
$(P \Longrightarrow S);T$	$=$	$P \Longrightarrow (S;T)$	
$(S [] T);U$	$=$	$(S;U) [] (T;U)$	
$(@x.S);T$	$=$	$@x.(S;T)$	si x n'est pas libre dans T
$(S;T);U$	$=$	$S;(T;U)$	
$S;skip$	$=$	S	
$S;(P\|T)$	$=$	$[S]P\|(S;T)$	
$(x := E);(P \Longrightarrow S)$	$=$	$[x := E]P \Longrightarrow (x := E;S)$	
$S;(T [] U)$	$=$	$(S;T) [] (S;U)$	
$S;@x.T$	$=$	$@x.(S;T)$	si x n'est pas libre dans S

TAB. 4.5 – Relations entre séquencement et autres substitutions

Les règles qui nous intéressent sont celles où la séquence *suit* la substitution de structure, i.e. de la forme $S;T$ où S est une précondition, une garde, un choix borné ou un choix non borné.

Nous définissons donc l'expansion de l'entrelacement comme suit. Soit S une précondition, une garde, un choix borné ou non borné :

$$((S;T) \mid\mid\mid U) \equiv (R \mid\mid\mid U)$$

Avec R une réécriture comme décrite au tableau 4.5. Ensuite nous définissons au tableau 4.6 l'entrelacement entre une substitution de structure, et une substitution atomique.

$(C \Longrightarrow S) \mid\mid\mid (T_b;T)$	$(C \Longrightarrow (S \mid\mid\mid (T_b;T)))$ $[](T_b;(C \Longrightarrow S) \mid\mid\mid T)$ si $T_b \equiv \begin{cases} x := E \\ skip \\ \text{await } c \end{cases}$
$(C \Longrightarrow S) \mid\mid\mid (\text{delay } b;T)$	$C \Longrightarrow (S \mid\mid\mid (\text{delay } b;T))$
$(C\|S) \mid\mid\mid (T_b;T)$	$(C\|(S \mid\mid\mid (T_b;T)))$ $[](T_b;(C\|S) \mid\mid\mid T)$ si $T_b \equiv \begin{cases} x := E \\ skip \\ \text{await } c \end{cases}$
$(C\|S) \mid\mid\mid (\text{delay } b;T)$	$C\|(S \mid\mid\mid (\text{delay } b;T))$
$(S [] T) \mid\mid\mid U$	$(S \mid\mid\mid U)$ $[](T \mid\mid\mid U)$
$(@x.S) \mid\mid\mid T$	$@x.(S \mid\mid\mid T)$

TAB. 4.6 – Entrelacement de substitutions atomiques et de structure

Dans la sémantique en WDC*, la garde et la précondition ont la même sémantique, et sont des prédicats vrais uniquement en un point du temps. C'est ce qui explique le fait qu'ils puissent être entrelacés à de nombreux points du code, à moins qu'un délai ne force ces prédicats à être évalués immédiatement. Cela se voit dans la différenciation entre les règles lorsque la substitution atomique en tête du processus concurrent est une affectation ou une attente réactive, et lorsqu'il s'agit d'un délai. Notons également que dans le cas de la substitution non bornée, le problème se réduit plutôt à un placement de quantification universelle : la forme correspond au délai aurait tout autant suffi pour toutes les substitutions atomiques.

Nous définissons l'entrelacement de deux substitutions de structure au tableau 4.7. Nous supposons que les réécritures par rapport au séquencement ont déjà été opérées. Les choix bornés et non bornés ne sont pas présents, puisqu'il ne font aucune discrimination sur la forme du processus à droite. Comme la substitution de concurrence est commutative, le choix borné (ou non-borné) se traite de manière similaire au tableau 4.6.

$(C_a \hookrightarrow_a S_a) \mid\mid\mid (C_b \hookrightarrow_b S_b)$	$(C_a \hookrightarrow_a (S_a \mid\mid\mid (C_b \hookrightarrow_b S_b)))$ $[\!]\,(C_b \hookrightarrow_b ((C_a \hookrightarrow_a S_a) \mid\mid\mid S_b))$ avec $\hookrightarrow_? \equiv \begin{cases} \mid \\ \Longrightarrow \end{cases}$

TAB. 4.7 – Entrelacement de substitutions de structure

La sémantique en WDC* de la précondition et de la garde sont similaires. De plus, comme nous l'avons indiqué plus haut, les conditions sont vraies en un point du temps, et donc peuvent être vraies en tout point du processus mis en concurrence, aux délais stoppant l'expansion près. Il est donc normal que leur expansion soit similaire.

Boucle

Il nous reste à traiter d'une substitution qui n'est pas atomique, et n'obéit pas aux mêmes règles d'expansion que les autres substitutions : la boucle. Le lien entre la boucle est problématique : elle contient forcément un délai (pour éviter les phénomènes de Zénon) répété un nombre arbitraire de fois. De plus, l'entrelacement entre le corps de la boucle et une substitution concurrente devient arbitraire, justement à cause la répétition.

Il devient donc plus difficile de proposer un entrelacement qui soit capable de prendre en compte le délai de la boucle, sans pour autant sacrifier la correction du programme. La conséquence est que la solution que nous proposons pour la boucle a pour défaut de causer de « faux positifs », i.e. des programmes vus comme incorrects à la validation, alors qu'en réalité les états du programme détectés comme invalides ne sont jamais atteints.

Les règles d'expansion pour la boucle sont présentées au tableau 4.8.

Nous n'avons pas traité le cas où le processus à droite est un choix (borné ou non-borné) : les règles présentées au tableau 4.6 sont valables ici aussi (la règle s'applique alors au processus de droite plutôt qu'à celui de gauche).

La fonction *undelay* introduite dans le tableau 4.8 est une fonction qui débarrasse une substitution de tous ses délais, en les remplaçant par *skip*.

Les règles pour l'entrelacement d'une boucle sont plus complexes que les précédentes, parce qu'il faut s'efforcer de prendre en compte les délais dans la progression de part et d'autre de la substitution de concurrence.

- Nous observons que pour l'affectation, le skip, la précondition et la garde, l'entrelacement induit deux cas. Dans le premier cas, la boucle n'est pas exécutée, et dans le deuxième, il y a au minimum une exécution du corps. À cause du fait que le corps prend forcément un temps non nul (non-Zénon), il contient un délai. C'est donc ce délai qui absorbera tous les

$(\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} (T_b;T)$	$S \mathbin{\vert\vert\vert} (T_b;T))$ $[\!]((C \Longrightarrow S_a);\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} (T_b;T)$ si $T_b \equiv \begin{cases} x := E \\ skip \end{cases}$
$(\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} (\text{await } b;T)$	$\text{await } b;((\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} T)$ $[\!]((I \Rightarrow [S_a \mathbin{\vert\vert\vert} \text{await } b]I) \vert (\mathcal{W}(C,S_a,I);S)) \mathbin{\vert\vert\vert} T$ $[\!]\mathcal{W}(C,S_a,I);(S \mathbin{\vert\vert\vert} (\text{await } b;T))$
$(\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} (C_b \hookrightarrow_b T)$	$S \mathbin{\vert\vert\vert} (C_b \hookrightarrow_b T)$ $[\!]((C \Longrightarrow S_a);\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} (C_b \hookrightarrow_b T)$ avec $\hookrightarrow_b \equiv \begin{cases} \vert \\ \Longrightarrow \end{cases}$
$(\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} (\text{delay } b;T)$	$S \mathbin{\vert\vert\vert} (\text{delay } B;T)$ $[\!]((C \Longrightarrow S_a);\mathcal{W}(C,S_a,I);S) \mathbin{\vert\vert\vert} (\text{delay } b;T)$
$(\mathcal{W}_a;S) \mathbin{\vert\vert\vert} (\mathcal{W}_b;T)$	$S \mathbin{\vert\vert\vert} T$ $[\!] \begin{cases} [(C_a \Longrightarrow (S'_a)) \mathbin{\vert\vert\vert} (C_b \Longrightarrow (S'_b))]((I_a \wedge I_b))\vert \\ \left(\begin{array}{l} \mathcal{W}_a;(S \mathbin{\vert\vert\vert} (\mathcal{W}_b;T)) \\ [\!]\mathcal{W}_b;((\mathcal{W}_a;S) \mathbin{\vert\vert\vert} T) \end{array} \right. \end{cases}$ avec $\mathcal{W}_X \equiv \mathcal{W}(C_X,S_X,I_X)$ et $S'_X \equiv undelay(S_X)$

TAB. 4.8 – Entrelacement d'une boucle

éléments du processus concurrent qui prennent un temps nul, c'est-à-dire l'affectation, le skip, la précondition et la garde. Il fallait ne pas oublier de prendre en compte le cas où la boucle n'est pas exécutée (i.e. le cas où la condition n'est pas vérifiée du départ).
De plus, cette forme de réécriture de l'entrelacement est en fait une conséquence implicite de la définition de la boucle :
$\mathcal{W}(C,S,I) \equiv ((C \Longrightarrow S)\hat{}; \neg\, C \Longrightarrow skip)$
Cette définition se retrouve dans l'expansion que nous proposons, où le choix est fait entre une répétition nulle, et au moins une répétition.

– Comme pour toutes les autres substitutions, l'attente réactive se propage partout. Nous devons cependant nous assurer que l'attente réactive n'apparaîtra pas *plusieurs fois* dans le corps de la boucle. Pour ce faire, nous ajoutons (deuxième cas du choix borné) une précondition qui vérifie que la condition d'attente réactive n'interfère pas avec le corps de la boucle.
– La forme de l'expansion pour le délai s'explique de la même manière que pour les substitutions atomiques : la différence est que le corps de la boucle pourra potentiellement être absorbé plusieurs fois par le délai en concurrence. Au final, l'expression obtenue correspond à tous les points du temps, par rapport au délai concurrent, où la boucle risque de s'arrêter.
– La mise en concurrence de deux boucles implique que les corps des boucles peuvent s'entrelacer de toutes les manières possibles. La seule exception à ce cas est lorsque les corps de la boucle ont exactement les mêmes délais, avec des chemins identiques. Pour simplifier l'expansion, nous conservons donc le cas général. L'expression de l'expansion

se lit comme suit :
- Dans le premier cas, il est possible qu'aucune des deux boucles ne soit exécutée
- Sinon, nous nous assurons que l'entrelacement arbitraire (donc sans l'absorption provoquée par les délais) des corps des boucles est valide par une précondition. Puis deux cas peuvent se produire, selon que la boucle de gauche ou de droite s'achève en premier.

4.4.3 Utilisation dans les machines temporisées

La concurrence, telle que nous l'avons présentée ici, mais également dans la sémantique en WDC*, autorise les variables partagées. Seulement, comme nous l'avons vu en section 3.3.5, prendre en compte les variables partagées requiert des changements profonds en B.

D'un côté, nous pourrions proposer un aménagement de ce type qui permettrait de prendre en compte les variables partagées dans un cadre concurrent. Ce serait cependant risquer un faible retour sur investissement, eu égard aux différentes limitations des solutions proposées, et augmenterait la complexité de B+DC d'un cran supplémentaire.

D'un autre côté, il semble possible d'exprimer des spécifications précises sans l'aide de variables partagées (voir section 3.3.5). Nous pouvons aussi remarquer qu'en règle générale, l'utilisation de variables partagées sous-tend un besoin de synchronisation entre processus. Il y a donc en pratique une séparation entre les processus mis en concurrence, et les moyens mis en oeuvre pour les synchroniser.

C'est pour cette dernière raison que nous proposons de conserver la notion de variable partagée (en regard de la concurrence), sans pour autant amender les contraintes sur les espaces de variables dans les machines B. Cela signifie que ceux-ci restent disjoints. Le seul endroit où pourront apparaître des variables partagées sera la machine où elles sont déclarées. Implicitement, si elles sont partagées, cela indique qu'elles se retrouvent, dans le corps des opérations, de part et d'autre des substitutions concurrentes utilisées.

Par exemple, soit l'opération de la forme suivante :

operation =
  BEGIN
    S |||T
  END

Soient $\mathcal{VAR}(S)$ et $\mathcal{VAR}(T)$ les ensembles des variables apparaissant dans S et T respectivement. $\mathcal{VAR}(S) \cap \mathcal{VAR}(T)$ doit être inclus dans l'ensemble des variables de la machine dans laquelle se trouve l'opération `operation`.

4.4.4 Exemples

Les machines en figure 4.5, 4.6 et 4.7 complètent un projet B+DC décrivant le fonctionnement d'un brûleur à gaz. Celui-ci est donc composé d'un contrôleur de valve de gaz, d'un déclencheur d'étincelles, lesquels sont utilisés pour décrire les opérations possibles du brûleur (déclencher une flamme ou l'arrêter). Le brûleur est dirigé par un contrôleur qui implémente les décisions de conception devant assurer la sûreté du système dans son ensemble.

```
MACHINE
  Sparker
SETS
  Sparks ={ presence, absence}
VARIABLES
  sparks
INVARIANT
  sparks ∈ Sparks
INITIALISATION
  sparks :=absence
OPERATIONS

do_sparks =
    BEGIN
        sparks :=presence ; (await sparks ≠presence |||delay 0.1)
    POST
        sparks = absence
    TIMING
        ⟦sparks = presence⟧∧ℓ=0.1
    END

  END
```

FIG. 4.5 – Une machine qui envoie des étincelles

```
MACHINE
  Burner
INCLUDES
  Gas_valve, Sparker
PROMOTES
  close_valve
INVARIANT
  true

OPERATIONS

start_burner =
    PRE
        valve =closed
    THEN
        open_valve |||(do_sparks ; await valve =open)
    POST
        valve =open
    TIMING

    END

  END
```

FIG. 4.6 – Fonctionnement du brûleur

```
MACHINE
   Burner_controller
INCLUDES
   Gas_valve
INVARIANT
   true

OPERATIONS

burner =
      BEGIN
         ( WHILE
             true
          DO
             await (¬flame)
            ; close_valve
            ; delay 30
            ; start_burner
          INVARIANT
            true
          END) |||( WHILE
             true
          DO
             detect_flame
            ; delay 0.5
          INVARIANT
            true
          END)
      TIMING

      END

   END
```

FIG. 4.7 – Contrôleur du brûleur à gaz

4.5 Raffinement temporel

La définition du raffinement habituel de B a toujours cours : ici c'est de raffinement temporel dont il est question.

Nous pouvons définir le raffinement temporel comme l'inclusion de traces temporelles. C'est-à-dire qu'une machine en raffine temporellement une autre s'il est possible de déduire les propriétés temporelles de la machine raffinée à partir des propriétés temporelles de la machine raffinante.

Prenons comme raffinement de la machine M de la figure 4.3 la machine N de la figure 4.8.

```
REFINEMENT
  N
REFINES
  M
TIME_INVARIANT
  XN
INVARIANT
  IN
OPERATIONS

opM_1 =
    PRE PN_1
    THEN SN_1
    POST QN_1
    TIMING XN_1
    END

  END
```

FIG. 4.8 – Raffinement temporisé

La machine N raffine *temporellement*la machine M si les formules suivantes sont vérifiées :
- $XN \wedge IN \Rightarrow XM$: la sûreté temporelle de N peut être déduite de la sûreté temporelle de son raffinement, en précisant la relation entre les anciennes et les nouvelles variables via l'invariant de collage
- $(Term(SN_i) \Rightarrow dur(SN_i, QN_i)) \vee (NTerm(SN_i) \Rightarrow dur_\infty(SN_i)) \Rightarrow XN_i$: les opérations vérifient leurs spécifications temporelles plus précises
- $XN_i \Rightarrow XN$:
- $XN_1 \vee \ldots \vee XN_n \Rightarrow XN$:
- $XN_i \Rightarrow XM_i$: La spécification temporelle de l'opération raffinante doit inclure celle de l'opération raffinée

Les conséquences pratiques de cette définition souffrent de défauts que nous n'avons eu que peu l'occasion d'étudier. Le raffinement temporel est-il pertinent ? En effet, il est déjà en pratique difficile de raffiner un modèle de manière à en faire la preuve simplement. Rajouter un raffinement temporel au-dessus de ce processus de concrétisation multiplie les contraintes :

- Le raffinement permet de rajouter des états du programme via des nouvelles variables ou des variables qui miment le comportement des variables raffinées. Du point de vue des traces d'exécution du système, il est fréquent que si, fonctionnellement, le changement est transparent, temporellement la nouvelle variable se comporte différemment. Donc, temporellement parlant, l'invariant de collage n'est que de peu d'aide pour la preuve temporelle.
- Le fait d'avoir une spécification du comportement du programme très tôt contrecarre l'objectif initial de B, qui est de permettre de spécifier les propriétés très abstraites du système (*ce qu'il calcule*) et de procéder par étapes pour spécifier la manière dont il vérifie ces propriétés (*comment il le calcule*). En ce sens, devoir spécifier trop tôt le flot de contrôle du système interfère avec la manière dont un développement B est conduit.

Deux solutions peuvent être apportées à ce problème, cependant :

- Le système abstrait peut spécifier les propriétés temporelles, mais celles-ci pourront n'être à vérifier que dans une étape de raffinement où le flot de contrôle du programme est suffisamment spécifié, au plus tard dans l'implémentation
- Le système abstrait et ses raffinements doivent inclure un flot de contrôle implicite. De par le fait, c'est le cas d'eventB. Nous décrivons donc l'utilisation de DC* avec eventB dans la prochaine section 4.6. plus loin.

Bien entendu, il existe des méthodes permettant de raffiner des systèmes concurrents selon des logiques temporelles simples, comme le système de Mokkedem et Méry [MM95]. D'autres travaux construisant des systèmes concurrents au raffinement modulaire comme ceux de Qiwen [Qiw96] suggèrent que les méthodes dites d'« assumption-commitment » semblent plus adaptées. C'est cette même méthode qui est utilisée pour spécifier un système utilisant DC* (une logique temporelle continue, donc) dans les travaux de Siewe [SHZC04]. Ces deux derniers résultats semblent donc indiquer que les méthodes d'assumption-commitment équipées d'une logique temporelle semblent plus adaptées pour la validation de la spécification et du raffinement de systèmes concurrents à contraintes temporelles.

Nous présentons dans la prochaine section quelques résultats de l'utilisation du calcul des durées avec eventB.

4.6 B événementiel et sémantique temporelle

Comme nous l'avons pu voir, B se prête bien à un passage à une sémantique temporelle. Il en va différemment d'eventB. Le « glissement » de B vers eventB fait perdre certaines des propriétés qui nous intéressaient auparavant et qui s'avèrent cruciales pour la définition d'une sémantique temporelle.

Notons que cette section constitue plutôt un travail de réflexion sur eventB et le temps, puisque la majeure partie de notre travail portait sur le B classique. Cependant, puisque les spécifications finales d'eventB sont disponibles depuis la moitié de l'année 2005 [ROD05], il nous a semblé intéressant de donner quelques arguments en faveur ou en défaveur de l'utilisation d'une sémantique en calcul de durées pour eventB.

Les différences d'eventB par rapport à B sont de deux ordres :

– EventB est focalisé sur le changement d'état. Ainsi les substitutions qui expriment des comportements complexes d'évolution des états n'apparaissent plus. Ne restent que les substitutions les plus simples, ou abstraites mathématiquement parlant, comme l'affectation et la substitution non-déterministe
– Le cadre d'eventB, focalisé sur le changement d'état, s'intéresse moins à la manière dont les états évoluent. C'est là la différence entre événéments et opérations. Cela signifie que si certaines propriétés d'évolution doivent être respectées par un modèle, il faut les préciser dans les contraintes à vérifier par le modèle. Ce point est notamment illustré par la manière dont est exprimée la vivacité dans [ACM05].

Par conséquent, eventB n'a plus de notion de flot de contrôle explicite, comme il était apporté par le choix borné ou la séquence en B classique. C'est désormais le déclenchement de gardes d'événements qui conditionne l'évolution de l'état du système.

Dans ce contexte, une idée pour ajouter des contraintes temporelles, comme nous l'avons fait pour le B classique, est d'ajouter une formule temporelle à chaque événement. Cette formule décrirait les propriétés de l'action associée, comme sa durée par exemple. Ces formules resteraient dans le cadre d'eventB, i.e. dans un cadre où le flot de contrôle est implicite.

Le problème de cette approche survient au raffinement : si nous nous plaçons dans le cadre d'eventB, les événements abstraits peuvent ne pas être la conséquence des événements raffinés respectifs. L'exemple le plus simple est celui de l'horloge : un événement qui avance d'heure en heure a une granularité de 3600 secondes. Le même événement raffiné pour introduire les minutes a une granularité de 60 secondes. La formule qui démontrerait le raffinement est de la forme $(\ell = 60)^* \Rightarrow (\ell = 3600)^*$, ce qui est faux. Plusieurs moyens pour surmonter ce problème peuvent être mis en oeuvre :
– Limiter la validation aux préfixes des formules temporelles. Dans ce cas, la validation sur le suffixe est perdue, sachant que le suffixe en eventB représente l'exécution indéfinie du modèle
– S'assurer d'un nombre minimal de répétitions qui corresponde au modèle abstrait. Comme l'opérateur d'itération de DC* correspond à un nombre arbitraire de répétitions, cette solution ne peut pas non plus être retenue.

Dans cette optique, naïve mais révélatrice, le problème peut venir de deux sources :
– DC* n'est pas adapté pour ce problème. Dans ce cas, il est préférable de choisir une logique temporelle plus adaptée aux hypothèses de fonctionnement des modèles eventB
– Les hypothèses de fonctionnement d'eventB, parce qu'implicites, manquent d'une définition du point de vue opérationnel. Dans ce cas, il est possible d'utiliser (W)DC* pour expliciter ces hypothèses, voire en définir d'autres.

Nous explicitons ces points en perspective, en section 4.7.2.

4.7 Conclusions

Nous voulions au départ définir un moyen d'exprimer et valider des propriétés temporelles en B. Pour ce faire, nous avons décidé d'utiliser DC*, une logique temporelle continue permettant de raisonner sur des durées d'événements. Nous avons utilisé une extension de DC* appelée WDC* : elle possède des propriétés simlaires à DC*, avec en plus la possibilité de raisonner sur des intervalles de temps nuls. Cela en fait une logique idéale pour représenter les entrelacements d'exécution qui peuvent survenir à divers points d'un programme composé de plusieurs processus.

4.7.1 Apports d'une sémantique en logique temporelle continue

Nous avons exprimé la sémantique des substitutions de B avec WDC* en suivant une technique déjà présentée par Siewe dans [SH01]. La difficulté supplémentaire ici était de préciser la sémantique de l'entrelacement de processus concurrents, et de s'assurer que la sémantique en calcul de plus faible précondition reste compatible avec la nouvelle sémantique temporelle.

Notre proposition étend donc B selon deux axes :

- B est étendu *fonctionnellement* par l'expression de la concurrence (|||) et la notion de non-terminaison (await , while). Il est désormais envisageable de développer des programmes B qui ne terminent pas, alors qu'auparavant la notion de non-terminaison d'un programme n'était pas permise : les boucles devaient forcément terminer. Cela correspond à une correction totale d'un programme dans la logique de Hoare. Autoriser la non-terminaison nous conduit donc à une correction partielle des programmes : introduire des formules de durées pour les programmes permet de spécifier et vérifier leur comportement lorsqu'ils ne terminent pas
- B est étendu *temporellement* grâce à la génération de formules temporelles représentant les traces d'exécution d'un composant B. Cela étend d'autant les possibilités d'expression de sûreté et de vivacité en B

Résultats

C'est principalement le fait que B soit basé sur des paradigmes bien connus (théorie des ensembles, logique de Hoare) qui a favorisé l'adjonction d'une logique temporelle, elle-même définie selon des paradigmes bien connus (calcul des prédicats)

Le résultat obtenu est conforme aux attentes : il apporte la concurrence, la non-terminaison, ce qui permet d'étendre le champ d'utilisation de la méthode B. Ce résultat est important, puisque la méthode B, pour de nombreuses entreprises, constitue un pari vers les méthodes formelles. La possibilité de conserver une méthodologie similaire de développement pour des systèmes plus complexes est donc un atout non négligeable. Un composant B validé selon le B classique restera alors correct avec BDC. Cette extension B+DC a donc les conséquences suivantes pour la méthode B :

- Elle permet d'exprimer les systèmes infinis, explicitement
- Elle permet un suivi du flot de contrôle des opérations
- Elle permet la composition concurrente (assez modestement, au vu de la technique utilisée, cependant)

L'extension de B se rapprochant le plus de la nôtre est celle de Lano et Dick [LD96], aussi nous raffiner un peu la comparaison des deux extensions :

- Elles permettent toutes deux les systèmes infinis
- Seule la nôtre ajoute la composition concurrente avec variables partagées
- L'extension de Lano et Dick permet le suivi du flot de contrôle via une logique temporelle discrète. Nous utilisons une logique temporelle continue.

Ces deux extensions montrent par ailleurs le potentiel d'extension du B classique.

De plus, l'utilisation de paradigmes bien connus possède un effet de bord désirable : il existe déjà des outils implémentant chacun de ces paradigmes. La disponibilité d'outils pour BDC peut donc se résumer à la mise en commun d'outils pour B et d'outils pour DC^*, aux modifications près. Pour peu que les outils suivent la théorie au plus proche, adapter les extensions à l'un ou l'autre outil devient alors aisé. Au vu de la maturité de B et de DC, nous pouvons considérer que c'est le cas.

Enfin, l'expressivité théorique de DC^* fait que l'extension BDC peut englober toutes les autres extensions temporelles pour B présentées en section 3.3. Donc, théoriquement, pour chaque extension, il est possible de trouver une traduction de composants B de cette extension vers BDC. Cette possibilité théorique offre donc à disposition un moyen supplémentaire de vérifier la correction de composants B de ces extensions.

Limitations

La première conséquence de l'utilisation de DC^* est que le développeur devra se familiariser avec cette logique, dont certains aspects ne sont pas triviaux (voir chapitre 5). Cette remarque est cependant valable pour tout autre système utilisant des cadres logiques moins habituels.

Une autre difficulté est de garder trace des changements de valeur des variables du programme pour pouvoir en exprimer des propriétés plus précises. En effet, le fait de continuer à utiliser le calcul de plus faible précondition concentre le maximum d'expressivité pour un programme à sa fin. Or si le programme est correct, il n'en reste pas moins vrai que certains morceaux de « code mort » (du point de vue de la preuve) peuvent être retrouvés dans l'implémentation finale. Cela signifie que le programme peut se retrouver dans un état non souhaitable lorsqu'il traverse ces portions de code non détectées à la preuve. Les mécanismes nécessaires pour résoudre ce problème peuvent s'avérer néanmoins très lourdes et font perdre de l'expressivité au système. Des solutions existent cependant, comme indiqué en section 4.7.2.

Il reste également une imprécision concernant le rapport entre une machine abstraite et ses raffinements, et les questions suivantes restent posées :

– Le code de la machine abstraite doit-il être validé temporellement, ou ne doit-il avoir pour but que la spécification ? Dans ce dernier cas, est-il préférable de valider temporellement un raffinement par rapport à la machine qu'il raffine, ou par rapport à la spécification temporelle abstraite ?
– À quel moment est-il possible d'utiliser les substitutions intéressantes du point de vue du flot de contrôle (séquencement, boucle) ? Théoriquement parlant, il est possible de les utiliser dans les abstractions, mais qu'en est-il en pratique ?

Enfin, quelques tentatives triviales d'utilisation de DC^* pour exprimer et valider temporellement des modèles d'eventB ont illustré en quoi cette logique n'était pas adaptée à eventB. Il sera donc nécessaire de chercher d'autres voies (ou, plus simplement, un dérivé de DC proche) pour valider temporellement des machines B.

4.7.2 Perspectives

Des travaux récents [HJ04] d'unification des différentes extensions de DC montrent comment, à partir d'une seule logique, déduire plusieurs variantes de DC. Plusieurs de ces variantes ont des propriétés intéressantes pour la représentation de paradigmes particuliers des langages de programmation : l'itération (pour la boucle), découpage *super-dense* et voisinage (vrai synchronisme, séquencement), ordre supérieur et récursivité (fonctions récursives, par exemple). Toutes ces propriétés en font un candidat intéressant à considérer pour exprimer une sémantique plus précise de B. Mais pourquoi, alors que nous avons déjà une sémantique temporelle pour B ? Elle pourrait en fait unifier la sémantique sous-jacente des substitutions avec la logique utilisée pour les preuves de programmes. De plus, cette logique unifiante semble également être un candidat idéal pour l'exploration des propriétés temporelles des modèles eventB.

Nous avons d'ailleurs indiqué plus haut que DC* ne semble pas être la sémantique idéale pour eventB. Néanmoins, les travaux de Hammad [HJMO03] sur eventB et les automates de région, et la possibilité d'exprimer les automates de région en calcul des durées, laissent à imaginer d'intéressantes possibilités pour la validation et le raffinement temporels de modèles B. Au vu de l'engouement suscité par eventB, et la proximité d'eventB avec les systèmes de transition, qui sont un formalisme bien compris, il nous semble que la piste de modèles B exprimés en corrélation avec une logique temporelle continue suffisamment expressive est un terrain de recherche particulièrement prometteur.

Bibliographie

[Abr96] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[ACM05] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in event_b. In Helen Treharne, Steve King, and Martin Henson, editors, *ZB 2005 : Formal Specification and Development in Z and B : 4th International Conference of B and Z Users, Guilford, UK*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer Verlag, Apr 2005.

[AFA03] IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.

[BB99] Martin Büchi and Ralph Back. Compositional symmetric sharing in B. In Wing et al. [WWD99], pages 431–451.

[CMP04a] Samuel Colin, Georges Mariano, and Vincent Poirriez. Duration calculus : A real-time semantic for B. In *First International Colloquium on Theoretical Aspects of Computing*, Guiyang, China, september 2004. UNU-IIST.

[CMP04b] Samuel Colin, Georges Mariano, and Vincent Poirriez. A natural extension of B substitutions : postconditions. Technical report, LAMIH/ROI, 2004. `http://www.univ-valenciennes.fr/ROI/WP/`.

[Hab96] Henri Habrias, editor. *Proceedings of the 1st Conference on the B method*, Putting into Practice methods and tools for information system design, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de recherche en informatique de Nantes.

[HJ04] Jifeng He and Naiyong Jin. Integrating variants of DC. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004, First International Colloquium, Guiyang, China, September 20-24, 2004, Revised Selected Papers*, volume 3407 of *Lecture Notes in Computer Science*, pages 14–34. Springer-Verlag, 2004. ISBN :3-540-25304-1.

[HJMO03] Ahmed Hammad, Jacques Julliand, H. Mountassir, and Dieudonné Okalas Ossami. Expression en B et raffinement des systèmes réactifs temps réel. In AFADL2003 [AFA03], pages 211–226.

[Lam90] Leslie Lamport. *win* and *sin* : Predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.*, 12(3) :396–428, 1990.

[LBS96] K. Lano, J. Bicarregui, and A. Sanchez. Using B to design and verify controllers for chemical processing. In Habrias [Hab96], pages 237–270.

[LD96] Kevin Lano and Jeremy Dick. Development of concurrent systems in B AMN. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, july 1996.

[MM95] Abdelillah Mokkedem and Dominique Méry. On using temporal logic for refinement and compositional verification of concurrent systems". *Theoretical Computer Science*, 140(1) :95–138, 1995.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6 :319–340, 1976.

[Pet03] Dorian Petit. *Génération automatique de composants logiciels sûrs à partir de spécifications formelles B*. Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambrésis, December 2003.

[Qiw96] Xu Qiwen. On compositionality in refining concurrent systems. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, Berlin, Germany, 1996.

[ROD05] 2005. `http://rodin.cs.ncl.ac.uk/`.

[SH01] François Siewe and Dan Van Hung. Deriving real-time programs from duration calculus specifications. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume LNCS 2144, pages 92–97, Livingston-Edinburgh, Scotland, september 2001. Springer-Verlag. (Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000).

[SHZC04] Francois Siewe, Dang Van Hung, Hussein Zedan, and Antonio Cau. A formal design technique for real-time embedded systems development using duration calculus. In A. Weitzenfeld and A. Barrera, editors, *1st IEEE Latin American Robotics Symposium*, pages 60–65, Mexico City, Mexico, october 2004.

[ST02] Steve Schneider and Helen Treharne. Communicating B machines. In ZB02 [ZB002], pages 416–435.

[WWD99] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *Proceedings of FM'99 : World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag). Springer Verlag, September 1999.

[ZB002] LSR-IMAG. *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, Grenoble, France, January 2002.

Chapitre 5

Implémentation du Calcul des Durées en Coq

Sommaire

Tableaux

Figures

Ce chapitre traite des problèmes liés à l'implémentation du calcul des durées dans un assistant de preuve à vocation générique (ici Coq). Nous commençons ce chapitre par quelques définitions préliminaires. Ensuite nous décrirons les raisons qui nous ont poussés à faire une telle implémentation. Puis, après une rapide présentation de Coq, nous évoquerons les problèmes soulevés par cette implémentation, les solutions déjà proposées par ailleurs pour des implémentations similaires, et les solutions retenues ainsi que leur description. Nous conclurons sur quelques exemples d'utilisation et réflexions conséquentes à cette implémentation. Les réflexions présentées ici ont fait l'objet d'une publication [CPM03].

5.1 Définitions

5.1.1 Plongement léger, plongement profond

Le terme de plongement[12] est utilisé pour qualifier la mise en oeuvre d'un formalisme à l'aide d'un autre. Dans le cadre des logiques, il aide usuellement à décrire la manière dont est implémentée une logique donnée à l'aide d'un outil de preuve. Inversement, il permet de caractériser un outil par la manière dont il permet d'implémenter d'autres logiques. Définissons plus précisément de quoi il s'agit.

Soit $\mathcal{L}$ une logique, et $\mathcal{L}'$ une logique basée sur $\mathcal{L}$. Si nous supposons que nous avons à disposition un assistant de preuve qui définit $\mathcal{L}$, alors :

– Un *plongement léger*(ou superficiel) de $\mathcal{L}'$ dans cet assistant de preuves est l'extension des règles de $\mathcal{L}$ à $\mathcal{L}'$. En d'autres termes, seules les nouvelles règles de $\mathcal{L}'$ par rapport à $\mathcal{L}$ doivent être ajoutées
– Un *plongement profond* de $\mathcal{L}'$ dans cet assistant de preuves est l'implémentation complète de $\mathcal{L}'$, y compris l'implémentation de $\mathcal{L}$. Un tel plongement est *conservatif*, au sens qu'il ne nécessite aucune adaptation de $\mathcal{L}'$ à l'outil de preuve.

Notons également que la notion de plongement se définit avec un rapport de point de vue : si $\mathcal{L}$ est elle-même un plongement profond, alors $\mathcal{L}'$ sera un plongement profond par rapport à l'outil de preuve, mais sera un plongement léger par rapport à $\mathcal{L}$.

Le choix entre plongement léger et profond a également des conséquences pratiques :

– Un plongement léger nécessite d'adapter les nouvelles règles de la logique aux mécanismes fondamentaux de l'outil de preuve. La mise en oeuvre peut donc être plus difficile. Cependant, comme ce sont les mécanismes fondamentaux qui sont utilisés, l'outil résultant bénéficie de toutes les optimisations et les particularités de l'assistant de preuve. Un exemple d'optimisation sont les règles de déduction automatique fournies par l'outil. Un exemple de particularité est de pouvoir déduire un code informatique d'une preuve de théorème (conformément à l'isomorphisme de Curry-Howard).
– Un plongement profond a les avantages et inconvénients inverses : comme il s'agit d'une réimplémentation depuis la base de la logique, il n'y a pas lieu de manipuler celle-ci pour

[12]Les termes anglophones correspondants sont *shallow embedding* pour le plongement léger/superficiel, et *deep-embedding* pour le plongement profond

l'adapter à l'outil (hormis la syntaxe). En revanche, le surplus de syntaxe et de définitions engendré rend peu exploitables les optimisations et particularités de l'assistant de preuve.

Enfin, certains outils de preuve, en particulier Isabelle-HOL [Isa93], sont des méta-moteurs logiques, i.e. ils n'ont à disposition que les règles de déduction à l'ordre supérieur, et des règles d'unification pour les termes-preuves. Dans de tels outils, les logiques implémentées sont donc forcément des plongements profonds. On parle d'ailleurs le plus souvent du duo Isabelle/HOL : il s'agit de Isabelle avec une librairie de règles et axiomes pour la logique d'ordre supérieure. Dans de tels cas, on parle alors de *logiques objets* ou de *plongements objets*.

5.1.2 Déductions

Pour éviter des erreurs de lecture par la suite, nous définissons ici les notations que nous utiliserons pour représenter des déductions et des raisonnements. Dans un système logique, la notation suivante signifie que par la règle d'inférence nommée $\mathcal{R}$ il est possible de déduire Q si la proposition P est vraie :

$$\frac{\mathrm{P}}{\mathrm{Q}}\,\mathcal{R}$$

Une suite d'applications de règles d'inférences, appelée déduction, peut être notée selon ce qui dans la déduction porte l'emphase : la description précise de la preuve, ou bien les théorèmes démontrés. Soient T_i des théorèmes ou des axiomes et R_i des règles d'inférence d'un système, la table 5.1 décrit une même déduction écrite de deux manières différentes.

$\dfrac{T_1 \qquad \dfrac{P \quad T_2}{L_1}\,\mathcal{R}_1}{\mathrm{Q}}\,\mathcal{R}_2$	$P \vdash Q$

TAB. 5.1 – Plusieurs notations pour une déduction

La notation de gauche de la table 5.1 présente une preuve (une déduction) de Q à partir de P en utilisant les règles $\mathcal{R}_1$ et $\mathcal{R}_2$ en conjonction avec les axiomes (ou théorèmes) T_1 et T_2. Cette notation met en évidence une manière de faire une déduction. La notation de droite de la table 5.1 met plutôt en évidence le fait que Q peut se déduire de P, sans plus de détails. Le symbole $\vdash$ signifie «permet de déduire». Il peut être utilisé pour indiquer les différentes étapes d'un raisonnement, ou en tant que simple connecteur.

Selon le contexte, donc, nous utiliserons $\vdash$ pour représenter soit une déduction complète, soit comme un connecteur. Par rapport à notre discours, cela signifiera que si le contexte correspond au système de preuve de DC, $\vdash$ indiquera une déduction d'après application d'une ou plusieurs règles d'inférences de DC. Si le contexte correspond plutôt à Coq, alors $\vdash$ représentera l'habituel connecteur des séquents, séparant les hypothèses et la conclusion.

5.2 Motivations

Le point de départ de cette implémentation peut se résumer à la question suivante : supposons que nous soyons capables de générer des obligations de preuves temporelles. Comment pouvons-nous les vérifier ? Cette section présente dans un premier temps les solutions déjà à disposition. Partant de remarques sur ces solutions, nous proposons notre solution en section 5.2.2.

5.2.1 Différents choix

Notre problème est donc ici de pouvoir valider des formules temporelles, si possible automatiquement, donc d'implémenter la logique utilisée d'une manière ou d'une autre. À quoi servira l'implémentation : à vérifier seulement la validité de formules, ou plus ?

- Dans le cas de la vérification de la validité d'une formule par algorithme de décision, une approche de type vérification de modèle (ou « model-checking ») n'est-elle pas préférable ? Ou une traduction des formules vers un autre formalisme est-elle possible ?
- Si les algorithmes de décision s'avèrent trop complexes (ou si l'idée est de traiter des fragments de la logique qui ne sont pas complètement décidables), l'utilisation d'un outil de preuve est plus profitable. L'automatisation est ici perdue, au profit de théorèmes qui passent difficilement les procédures de décision mentionnées plus haut. Mais il reste des choix d'implémentation à faire :
  - Devons-nous utiliser la sémantique de la logique ?
  - Devons-nous (et pouvons-nous) adapter la logique à l'outil, ou l'inverse ?

Détaillons maintenant ces différentes approches par rapport aux solutions existantes.

Approche par procédures de décision

Model-checking Notre but initial est d'être capable de valider toute formule de DC : a priori les obligations de preuve provenant d'un modèle $B+DC$ n'ont pas de forme particulière. En effet, les approches en vérification de modèle pour le calcul des durées, comme par exemple DCVALID [Pan01], sont restreintes à une sous-classe particulière du calcul des durées. Cela suppose donc de pouvoir modéliser un problème à résoudre en termes de cette sous-classe. Cette contrainte est vérifiée en pratique dans de nombreux cas, ce qui justifie de l'utilité de cette approche. Cependant, puisque ne faisons aucune hypothèse sur la classe des formules, nous devons écarter l'approche par vérification de modèles.

Traduction de formules Une autre approche existante est la traduction des formules du calcul des durées en un autre formalisme : Enslev et Nielsen [EN05] proposent un outil, basé sur les travaux de Fränzle [Frä02], traduisant les formules de DC en systèmes de contraintes linéaires. Ces systèmes linéaires seront ensuite résolus par un solveur dédié. L'attrait de cette approche est l'efficacité obtenue dans la validation des formules, bien que cette efficacité soit dépendante du solveur utilisé. L'inconvénient est que la résolution ne peut se faire que pour des intervalles de temps bornés (dans le cas non borné, la résolution est indécidable).

Comme nous venons de le voir, les méthodes par vérification de modèle sont limitées par la classe des formules à résoudre. Qu'en est-il des approches par outil de preuve ?

Suivre la sémantique

L'approche la plus immédiate, si un outil de preuve est utilisé, est d'implémenter directement la sémantique des formules de DC. C'est ce que fait Skakkebæk [Ska94] en implémentant DC avec PVS : les formules sont interprétées dans leur sémantique naturelle, c'est-à-dire qu'elles sont paramétrées par des intervalles de temps.

Une approche assez similaire, DC/RJ^- [Xia98], a été utilisée pour l'extension de RAISE [RAI85]. Le langage de spécifications a tout d'abord été étendu pour pouvoir exprimer des formules de DC, en utilisant la sémantique en intervalles de DC (comme pour PVS). Des termes permettant de spécifier des propriétés sur les formules ont également été implémentés (comme la variabilité finie d'une formule, par exemple). Ensuite, des modules de théorie utilisant cette extension implémentent l'axiomatique du calcul comme une conséquence de la définition des termes de DC. Yong [Xia98] montre par exemple des preuves d'un de ces axiomes, d'un théorème classique pour les logiques temporelles, et une preuve de correction du modèle du brûleur à gaz spécifié avec DC/RJ^-. L'intérêt de cette approche tient dans son intégration à un outil de spécification pratique d'usage (RAISE), et la simplicité de la mise en oeuvre de l'extension (via la sémantique naturelle de DC). Les inconvénients sont duaux des avantages, cependant : l'efficacité de l'outil est dépendante de l'ensemble de théorèmes utilisé et du problème à traiter. De plus, l'outil souffre de ce que RAISE est moins un assistant de preuve qu'un outil de conception : de ce fait, tout changement aux théories de base sont plus difficilement rejouables au niveau des théories qui les utilisent. Ceci rend la conception de l'outil logique plus ardue.

Suivre le système de preuve

L'autre approche existante est l'utilisation d'un méta-moteur logique, Isabelle-HOL [Isa93], pour mettre en oeuvre les règles du système d'axiomes et de règles d'inférence de DC. Cette mise en oeuvre est présentée exhaustivement par Heilmann [Hei99]. Rappelons ici rapidement comment une logique est mise en oeuvre dans Isabelle/HOL :

- Isabelle/HOL est, fondamentalement, un assistant de preuve pour une logique d'ordre supérieur intuitionniste, appelée méta-logique
- Pour implémenter une logique en Isabelle/HOL, la méta-logique doit être complétée avec les axiomes et règles d'inférence de cette logique, appelée alors logique *objet*. Ces axiomes et règles d'inférence représenteront la syntaxe et le système logique de la logique objet
- Isabelle/HOL dispose également de mécanismes de réécriture et d'unification pour la résolution de formules et l'application de règles d'inférence
- Enfin Isabelle/HOL fournit également des possibilités de redéfinition syntaxique pour simplifier la manipulation de la logique objet.

Dans le cas de Heilmann [Hei99], la logique objet est la logique d'intervalle, augmentée ensuite avec le calcul des durées, bien que ce ne soient pas les seules : en effet, la logique d'intervalle s'appuie sur la logique classique et une théorie de l'arithmétique sur les réels. Le résultat final est un assistant de preuve (baptisé *Isabelle/DC*) utilisant directement le système de preuve pour DC introduit par Hansen et Zhou [HZ04], et qui met à disposition une bibliothèque particulièrement riche de théorèmes sur DC. Cette approche a néanmoins deux défauts :

- Il faut réimplémenter les logiques sur lesquelles s'appuient IL et DC, plutôt que réutiliser des bibliothèques existantes. La raison en est que certains axiomes de base, triviaux, ne sont plus valides tels quels dans IL. L'axiome de l'égalité en est un exemple.
- Un fait plus gênant, est que l'implémentation choisie représente le système de preuve comme un système de séquents. Or le système présenté par Hansen et Zhou [HZ04] n'est

pas représenté sous cette forme. De ce fait, certaines règles d'inférence de DC doivent être modifiées.

Notons cependant que ce dernier problème tient plus au système de preuve de DC qu'à l'approche choisie. En effet, un système où les règles d'inférence sont sous forme de séquents est usuellement beaucoup plus facile à manipuler pour la preuve de théorèmes. Néanmoins, cette nécessité de devoir réimplémenter les logiques de base amoindrit la modularité du système résultant, et c'est ce qui motive notre travail comme présenté en section 5.4.

Il existe également une extension de IL appelée SIL (la logique d'intervalle signée) qui affaiblit la définition du découpage d'intervalle pour permettre d'exprimer des propriétés sur le futur. C'est-à-dire qu'avec SIL, il est possible de spécifier des propriétés qui portent au-delà de l'intervalle de temps que l'on considère. SIL dispose d'une mise en oeuvre avec Isabelle/HOL (par Rasmussen [Ras02]) basée sur un système en déduction naturelle, rappelant le système présenté au paragraphe précédent. De la même manière que la logique de voisinage et DC sont basés sur IL, il est possible de les baser également sur SIL, les faisant bénéficier de l'expression étendue de celle-ci. La clarté de la mise en oeuvre de SIL dans Isabelle/HOL contribue également à faciliter la mise en oeuvre de DC basé sur SIL, toujours dans Isabelle/HOL.

Récapitulons donc les choix qui s'offrent à nous, selon les diverses approches :

Vérification de modèle : Le problème à traiter doit pouvoir être exprimable selon une forme particulière. Comme nous choisissons de rester généraux, nous écartons cette solution.

Traduction de formules : Il s'agit d'une approche récente. Elle se cantonne aux intervalles de temps bornés, et son efficacité dépend fortement du solveur utilisé en bout de chaîne.

Suivre la sémantique : L'approche est simple puisque la sémantique de DC est facile à saisir. Cependant les quantificateurs existentiels imposés par le découpage d'intervalle peuvent rendre la résolution de formules très difficile. De plus, l'extension par changement de certains théorèmes de base est difficile.

Suivre le système de preuve : cette solution semble la plus prometteuse, au vu du nombre d'assistants de preuve existants actuellement. Dans le cas d'un plongement léger, l'inconvénient est l'obligation d'adapter le système de preuve à un nouveau fonctionnement (avec des séquents). Dans le cas d'un plongement léger, la mise en oeuvre pose moins de difficultés, mais les bibliothèques résultantes ne bénéficient plus des tactiques et optimisations dédiées conçues pour l'assistant de preuve.

5.2.2 Choix retenus

D'après les exemples que nous avons décrit ci-avant, les outils basés ou à destination de la vérification de modèle ne nous sont que de peu d'utilité, puisqu'ils se limitent à certaines classes de formules de DC, alors que nous n'avons aucun *a priori* à ce niveau.

Il reste donc à choisir parmi les possibilités offertes par les assistants de preuve : une mise en oeuvre qui suit la sémantique permettrait d'obtenir plus rapidement un outil, au prix de son efficacité (voir section 5.2.1.0). En effet, la base des formules de DC étant le découpage d'intervalles, celui-ci est défini sémantiquement par l'utilisation d'une quantification existentielle : un découpage d'intervalle suppose l'existence d'un nombre (compris dans l'intervalle de temps considéré) pour lequel la formule est valide. Or les quantificateurs existentiels sont souvent source de difficultés pour les assistants de preuve, puisqu'il leur est difficile de trouver avec quelle valeur instancier ladite variable existentielle. En revanche, la disponibilité d'une mise en oeuvre de DC qui suit la sémantique permet d'en déduire le système de preuve, ou tout au moins quelques axiomes et théorèmes usuels.

Outre suivre la sémantique, l'autre approche possible, qui suit le système de preuve, serait plus efficace, mais nécessiterait d'effectuer quelques adaptations au système de preuve originel. En effet, le système de preuve de DC ne prend pas en compte les séquents, alors que c'est le cas de la plupart des assistants de preuve d'ordre supérieur. L'inconvénient, pour l'implémentation existante avec Isabelle/HOL, est de devoir redéfinir une bibliothèque pour des théories répandues (premier ordre, arithmétique des réels) sans en avoir réutilisé d'existantes. Ceci fait d'ailleurs qu'il n'est pas possible de profiter de théorèmes ou de tactiques puissantes déjà existantes sur ces mêmes théories.

Des critères pratiques entrent également en ligne de compte : l'outil le plus abouti, basé sur Isabelle/HOL, n'est plus maintenu, et n'est pas utilisable avec des versions récentes d'Isabelle. Un outil similaire, qui base DC sur SIL [Ras02], a vu le jour il y a peu, et nous n'avons pu considérer à l'époque son utilisation pour la preuve de théorèmes de DC. Néanmoins, comme SIL est une version de ITL qui supprime certaines contraintes, son utilisation dans un cadre ITL/DC est envisageable, en rajoutant certaines contraintes manquantes (le point de découpage d'intervalle dans SIL n'est plus forcément au milieu, mais peut se trouver au-delà, en particulier).

De plus, nous avons déjà utilisé un assistant de preuve dans le cadre d'un outil pour le développement de projets B. Cet outil s'appelle B/PhoX et fait partie du projet BRILLANT [CPR+05]. L'assistant de preuve PhoX est initialement basé sur un lambda-calcul (AF2), ce qui le dote de caractéristiques proches de celles de Coq. La plus intéressante est la possibilité de raisonnement à l'ordre supérieur. Ainsi les travaux qui restent dans un contexte suffisamment générique dans Coq peuvent être adaptés à PhoX, et vice-versa.

Récapitulons :

- En suivant la sémantique de DC, nous obtiendrions un outil peu efficace
- Suivre le système de preuve est plus prometteur, mais les adaptations à opérer nous obligeraient à redéfinir également des bibliothèques pour des définitions triviales car déjà existantes (logique du premier ordre, arithmétique des réels)
- L'outil le plus abouti, en Isabelle/HOL, n'est (n'était) plus maintenu
- PhoX, un assistant de preuve au formalisme proche de Coq, est déjà utilisé par ailleurs en tant que prouveur d'obligations de machines B.

Pour toutes ces raisons, nous proposons d'utiliser l'assistant de preuve Coq, qui dispose de bibliothèques très fournies sur le premier ordre et sur l'arithmétique des réels. L'implémentation du calcul des durées que nous proposons sera un plongement léger, ce qui signifie que le travail est divisé en deux temps : adapter le système de preuve de DC pour qu'il reste correct dans le moteur d'inférence de Coq (qui, comme Isabelle/HOL, fonctionne avec des séquents), et la mise en oeuvre proprement dite.

5.3 Coq, une mise en oeuvre du calcul des constructions inductives

Cette section présente, sans trop la détailler, la théorie sur laquelle est basée Coq, puis les propriétés de l'outil pertinentes pour nos besoins d'implémentation.

5.3.1 Le calcul des constructions inductives

Le CCI (calcul des constructions inductives) est un système logique constitué du calcul des constructions, étendu par la possibilité d'exprimer des types inductifs. Intéressons-nous dans un premier temps au simple calcul des constructions. Il s'agit d'un lambda-calcul qui permet d'exprimer :

- Les types dépendants. Ils permettent par exemple de modéliser des types qui dépendent d'un terme. Par exemple, le type des tableaux de taille n peut être modélisé de cette manière
- Les constructions de types. Elles permettent de construire des types plus complexes à partir des types de base. Elles rendent possibles d'exprimer des types plus complexes (des listes de listes d'entiers, par exemple)
- Les types polymorphes. Ils permettent de paramétrer les types par des sortes, ce qui rend ces mêmes types indépendants d'autres types sous-jacents. Ils permettent d'exprimer des propriétés générales sur des prédicats ou des fonctions (comme une fonction qui associe à un type sa fonction identité, ou une propriété sur les ensembles d'entiers, par exemple)

En se restreignant à deux univers (les types et les sortes), différentes combinaisons de ces trois constructions abstraites forment différents systèmes logiques : le $\lambda\Pi$-calcul, le système F,... Ces systèmes forment le système du cube [Bar92], en figure 5.1. Le lambda-calcul simplement typé $\lambda_{\rightarrow}$) est le formalisme de départ. Le calcul des constructions λC est obtenu en rajoutant à ce dernier les trois extensions du typage citées plus haut. Il est décrit plus en détail dans [Coq85, CH85, CH87, CH88].

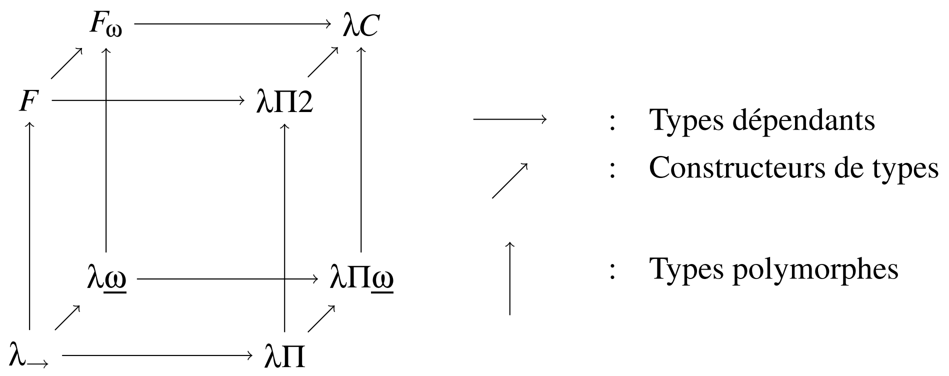

FIG. 5.1 – Le système du cube de Barendregt

Le calcul des constructions peut également exprimer nombre de fonctions mathématiques, puisque l'ordre supérieur permet d'exprimer des notions de récurrence. Cependant, cette manière de représenter est peu efficace pour deux raisons :

- Les calculs ne peuvent pas exploiter une forme d'inductivité génériques sur les termes pour être plus efficaces
- Utiliser l'ordre supérieur plutôt que des schémas de récurrence rend difficile la manipulation du système logique obtenu

Donc le calcul des constructions a été étendu en CCI [CPM90]. En effet il existait déjà une extension de la théorie des types simples : la théorie des types de Martin-Löf. Elle étend celle-ci avec les axiomes suivants :

- Un schéma d'axiomes sur les propositions : il permet de remplacer un terme par un autre dans une proposition s'ils sont égaux
- Un schéma d'axiomes pour la récurrence sur les entiers naturels
- Les axiomes de Peano

Cette extension dispose de règles de réduction pour calculer efficacement les termes définis inductivement (ici, les entiers naturels). Le calcul des constructions a été étendu en CCI de manière presque similaire. Seulement, les règles sur les types et les sortes du calcul des constructions permettaient d'aller plus loin que les seuls entiers naturels. Par conséquent, le CCI est une extension du calcul des constructions, avec un mécanisme de définitions inductives des types (sur les termes, les propositions, ainsi qu'à l'ordre supérieur). Ce mécanisme définit implicitement les schémas d'axiomes de récurrence pour les types définis inductivement, que le système de réduction peut utiliser pour calculer efficacement sur les termes.

Toutes ces propriétés font du CCI, et par extension, de Coq, un outil idéal pour pouvoir implanter et raisonner sur tout système défini par règles d'inférences, en particulier si des calculs sont impliqués. Nous illustrons dans la section suivante les propriétés qui nous intéressent pour la mise en oeuvre de DC dans Coq.

5.3.2 Propriétés de l'outil

Exprimons tout d'abord quelques-unes des caractéristiques de DC, pour ensuite déterminer en quoi Coq est adapté par rapport à ces caractéristiques.

Premier ordre : DC est basé sur la logique classique, donc est du premier ordre. Coq est basé sur une logique constructiviste, mais permet de raisonner en logique classique en rajoutant simplement l'axiome du tiers-exclu. Ainsi dans le cadre d'un plongement superficiel, il n'est pas besoin de recréer tout ce pan de la logique.

Définitions inductives : Les formules, termes et états de DC sont définis inductivement : il est alors possible d'implanter exactement ces définitions en Coq dans le cadre d'un plongement profond.

De plus les conditions de bord sur les axiomes (par exemple la *rigidité* de certaines formules) ou sur les règles d'inférence (l'absence d'opérateur $\frown$) rendent nécessaire la possibilité de parcourir la structure. Selon le type de plongement, cela peut être fait à l'aide de prédicats définis inductivement, ou à l'aide de fonctions qui analysent la structure des formules.

Nombres réels : Bien qu'il soit possible d'utiliser comme domaine numérique pour DC celui des entiers, le domaine des nombres réels est plus intéressant. Par les nombres réels il est possible d'exploiter toute la notion de continuité autorisée par DC. Il se trouve que Coq possède une librairie particulièrement fournie et complète sur les nombres réels : il s'agit donc encore d'un des pans de la définition du calcul des durées qu'il devient inutile de recréer.

Grammaire et syntaxe : D'un point de vue strictement syntaxique, DC est assez particulier, puisqu'il dispose de connecteurs logiques en double : les uns pour les formules logiques, les autres pour les formules d'état. Il y a d'ailleurs une relation entre ces dernières et la logique propositionnelle, cette relation étant illustrée par un des axiomes de DC (noté DCA6). De plus, comme toute logique temporelle, il est d'usage d'avoir une notation particulière pour certains connecteurs complexes définis à partir de la logique de base : $\Box$ ("toujours"), $\Diamond$ ("fatalement"), $\leadsto$ ("jusqu'à ce que").

Coq permet, grâce à des possibilités de (re)définitions syntaxiques, d'associer des notations particulières à certains termes. Les premières versions des bibliothèques de support de DC étaient implémentées en utilisant la version 7.3 de Coq, mais les dernières utilisent la version 8.0. Comme dans cette dernière version la manière d'introduire de nouvelles notations a été revue, les redéfinitions syntaxiques sont encore plus claires et plus simples à spécifier.

5.4 Implémentation : problèmes et solutions

Nous avons donc choisi d'implémenter DC sur le mode d'un plongement léger dans Coq. Cela signifie que, dans l'idéal, seuls auront à être ajoutés les connecteurs spécifiques à DC, i.e. ℓ, $\frown$, $\int$ et les états. Cette implémentation se fera dans le cadre des bibliothèques du premier ordre de Coq.

5.4.1 Intérêt

Les problèmes décrits en section 5.2.1 concernant l'implémentation de DC en Isabelle laissent à penser qu'une adaptation de DC sera nécessaire pour un système d'inférence utilisant la notion de séquents. Pour ce plongement léger, nous partons du principe qu'il faut modifier Coq le moins possible[13], et plutôt adapter DC à celui-ci.

Pourquoi vouloir apporter le moins de changements à Coq ? Les raisons en sont théoriques et pratiques :

Théoriques

- Une des règles d'inférence du système de preuve de DC concerne la généralisation d'une variable : or la généralisation en Coq est l'un des mécanismes fondamentaux. Vouloir le modifier directement reviendrait à modifier profondément le calcul des constructions.
- L'axiome d'égalité devrait aussi être modifié, or celui-ci, dans Coq, est une définition inductive de l'égalité de Leibniz. La technique (présentée par Heilmann [Hei99]) donne une hypothèse supplémentaire pour l'application d'un remplacement dans le cadre d'une égalité. Or les définitions inductives, en Coq, requièrent la décroissance de leur argument.

[13]Même si la disponibilité du code source de Coq nous laisse beaucoup de choix et de libertés à ce niveau

La version de l'égalité de Coq ne pourrait donc pas accueillir cette hypothèse supplémentaire.

Pratiques

- Parallèlement à la raison théorique sur la généralisation, en pratique cela revient à intervenir directement sur le programme source de Coq. Une telle intervention est bien trop complexe à réaliser dans le cadre de la simple implémentation d'une logique.
- Si réellement nous reprenions l'axiome de l'égalité pour qu'il fonctionne dans le cadre de DC, il nous faudrait réimplémenter un très grand pan des bibliothèques de Coq. En effet, la bibliothèque de théorèmes sur les réels ne pourrait pas faire usage d'une telle définition de l'égalité. Donc cela reviendrait à suivre l'approche de Heilmann [Hei99], et nous n'y gagnerions rien.
- Enfin, un plongement léger permet de bénéficier des optimisations déjà apportées au moteur de Coq pour la résolution de formules du premier ordre (puisque que c'est dans cet univers que la plupart des bibliothèques sont définies).

Avec toutes ces raisons à l'esprit, nous résumons dans un premier temps les difficultés d'implémentation auxquelles nous avons dû faire face, puis explicitons celles-ci en introduisant les solutions que nous avons apportées.

5.4.2 Description

Les difficultés d'implémentation tiennent principalement à ce qu'une mise en oeuvre directe du système de preuve peut rendre le système complet incohérent. L'implémentation de DC en Isabelle, présentée en section 5.2.1, illustre bien ce problème. Pour ce qui nous concerne ici, changer les règles de fonctionnement de Coq est hors de question : toute intervention à ce niveau nécessite de revérifier la cohérence du système dans son ensemble. Cela n'apporterait rien de plus que la mise en oeuvre de DC en Isabelle/HOL. Le problème principal est donc ici pour nous d'adapter le système de preuve de DC à (la bibliothèque du premier ordre de) Coq.

Notre argumentation se divisera selon les points qui nous ont posés problème. Ces points correspondent globalement à chacune des règles d'inférence du système de preuve de DC, avec en plus les hypothèses implicites posées par la variable spéciale ℓ. Beaucoup des contre-exemples montrant la fausseté de certaines implémentations naïves proviennent de Heilmann [Hei99]. Après avoir présenté les problèmes cas par cas et les solutions que nous avons choisies, nous présenterons le système de preuve résultant, ainsi que la preuve de correction de celui-ci.

La variable spéciale ℓ

D'après la syntaxe de DC [HZ04], ℓ est une variable spéciale (puisqu'elle est intégrée à la syntaxe) représentant la longueur de l'intervalle courant. La première idée qui vient est donc de la représenter telle quelle en Coq :

```
Parameter l : R.
```

Cette définition signifie que l est une variable réelle. Le problème est que cette définition ne fonctionne pas. Dans un système avec séquents, l'égalité de ℓ avec une valeur fait que l'on est tenté de remplacer ℓ par la valeur avec laquelle elle est identifiée. Pour reprendre l'exemple de Heilmann [Hei99] sur ce sujet :

$$\frac{\ell = 3 \;\vdash\; \ell = 1 \frown \ell = 2}{\ell = 3 \;\vdash\; 3 = 1 \frown 3 = 2}$$

La prémisse montre qu'on l'on peut découper un intervalle de 3 unités de temps en deux intervalles, l'un de une seconde et le suivant de deux secondes. Le résultat de cette déduction montre une conclusion fausse, puisque le découpage de deux intervalles faux est un intervalle faux. Donc cette règle est incohérente avec DC, puisqu'on n'aurait pas dû pouvoir remplacer ℓ par sa valeur.

La proposition de Heilmann [Hei99] à ce niveau est de rajouter une hypothèse à la règle de substitution (qui découle de l'égalité de deux termes), comme suit :

$$\frac{}{\Box(a = b), P(a) \vdash P(b)}$$

avec P un prédicat. Intuitivement, l'idée fonctionne, puisque l'hypothèse supplémentaire $\Box(a = b)$ garantit que les deux termes seront égaux sur tout l'intervalle de temps considéré. Ainsi la déduction erronée faite précédemment n'est plus possible. Cependant, dans notre cas, cette modification de l'égalité est autrement plus difficile. La définition de l'égalité de Coq est la suivante :

```
Inductive eq (A:Type) (x:A) : A -> Prop :=
refl_equal : x = x :>A
where "x = y :> A" := (@eq A x y) : type_scope.
```

C'est-à-dire que l'égalité est définie comme étant le plus petit prédicat réflexif. On retrouve la définition de l'égalité de Leibniz dans le schéma d'axiome pour les preuves inductives généré par Coq pour cette définition. En d'autres termes, l'égalité est définie minimalement, et l'égalité propositionnelle en est une conséquence. Or cela nous pose problème pour une implémentation en Coq, puisque l'ajout d'une hypothèse de la forme $\Box(a = b)$ induit deux difficultés :

- Cette définition n'est plus minimale
- Parallèlement, Coq interdit les définitions inductives dont les arguments croissent : ajouter une hypothèse à cette définition serait donc rejeté

Enfin, comme nous l'avons déjà indiqué, redéfinir l'égalité rendrait caducs la plupart des théorèmes sur les réels, ce qui obligerait à redéfinir ceux-ci. Donc la solution de Heilmann [Hei99] n'est clairement pas viable dans notre cas. Nous proposons donc une solution plus adaptée à l'intuition de la longueur d'un intervalle.

Celle-ci est en fait dépendante du contexte de la formule (et donc du découpage d'intervalle $\frown$), ce qui, justement, fait de DC une logique modale. Donc il s'agit en fait d'un prédicat dépendant de $\frown$. Notre idée est de rendre sa prédicativité à ℓ, en en faisant un prédicat comme les autres, et en spécifiant ses propriétés une à une. Comme il s'agit de représenter la longueur d'intervalle, ses propriétés se bornent en fait à être comparé avec d'autres valeurs. Donc nous proposons de représenter ℓ par deux prédicats :

```
Parameter length_eq : R -> Prop.
Parameter length_lt : R -> Prop.
```

où `length_eq` (resp. `length_lt`) est le prédicat représentant l'égalité (resp. l'infériorité stricte) de la longueur de l'intervalle avec une valeur. Les autres prédicats de différence, et d'inégalité, peuvent être définis à partir de ceux-là. Grâce à cette définition, nous sommes certains d'éviter les effets indésirables d'une définition plus triviale de l'égalité. En effet, comme désormais nous devons expliciter les propriétés de ces prédicats, nous redevenons maîtres de celles-ci.

À partir de ces deux définitions, nous pouvons définir les autres inéquations mettant en jeu la longueur d'intervalle :

```
Definition length_le (x:R):=(length_eq x) \/ (length_lt x).
Definition length_gt (x:R):=~(length_le x).
Definition length_ge (x:R):=(length_eq x) \/ (length_gt x).
```

Cette définition des autres prédicats de relation permet d'obtenir immédiatement un théorème de totalité pour la durée d'intervalle :

```
Theorem total_length:forall x:R, l<x \/ l=x \/ l>x.
```

Il suffit de remarquer que la définition de $\ell > x$ est la négation de $\ell < x \vee \ell = x$. Notons également ici que les notations prédicatives sont cachées grâce aux redéfinitions syntaxiques permises par Coq. Ainsi, par exemple, $\ell < x$ correspond en fait à `(length_lt x)`.

Il nous faut également une axiomatique permettant de mettre en relation la durée d'intervalle, et le domaine des réels. Suivent quelques exemples d'axiomes :

```
Axiom length_eq_Rgt:forall x y:R,l=x -> x>y -> l>y.
Axiom length_gt_length_lt:forall x y:R,l>x -> l<y -> x<y.
```

La bibliothèque Coq contient tous les axiomes nécessaires, ainsi que des théorèmes utiles pour que la véritable nature prédicative de ℓ soit transparente à l'utilisateur. Néanmoins, ces axiomes ne sont pas suffisants. En effet, nous avons désormais à disposition une axiomatique de la longueur d'intervalle. Cependant cette axiomatique fait apparaître une possibilité que la longueur d'intervalle ne puisse pas être représentée comme une constante exprimée en Coq.

La raison pour laquelle ce problème n'arrivait pas dans l'implémentation de DC en Isabelle, est que ℓ était une variable spéciale définie comme appartenant à $\mathbb{R}$. Et le fait d'être membre d'une classe en théorie des types, garantit automatiquement l'existence. Ainsi, toute implémentation qui fait de ℓ un membre des réels permet de prouver facilement le théorème $\exists x, \ell = x$. Il suffit pour cela de choisir pour x ℓ lui-même, et d'utiliser l'axiome de l'égalité.

Pour notre implémentation prédicative de la longueur d'intervalle, ce n'est plus possible. L'axiome $\ell \geq 0$ permet effectivement de prouver que ℓ n'est pas l'infini négatif, mais il n'est pas possible de prouver que ℓ n'est pas infini positif. Nous sommes donc obligés de rajouter un axiome nous garantissant que la longueur d'un intervalle est forcément une valeur réelle atteignable. Donc le théorème $\exists x, \ell = x$ devient un axiome dans notre implémentation. Ce nouvel axiome nous permet alors de profiter des propriétés du nombre réel correspondant à ℓ lorsque sa valeur n'est pas spécifiée dans un théorème.

Notation : Bien que dans la suite nous ne considérerons que cette implémentation prédicative pour la longueur d'intervalle, nous noterons $\ell_{\mathcal{R}}$ (i.e. la variable indicée par le prédicat de relation représenté) le prédicat correspondant à $\ell\mathcal{R}x$.

Règle d'inférence : monotonie

Soient P, Q, R des propositions, la règle de monotonie gauche est décrite de la manière suivante :

$$\frac{P \Rightarrow Q}{P^\frown R \Rightarrow Q^\frown R}$$

avec la règle symétrique pour la monotonie droite. Comme pour la nécessitation, notre discours se limitera à une seule des deux règles de monotonie, puisque les conclusions sont identiques à la symétrie près.

Remarquons tout d'abord que l'ajout naïf d'hypothèses permet d'avoir une déduction fausse :

$$\frac{\ell = 2 \vdash \ell = 1 \Rightarrow \ell = 33}{\ell = 2 \vdash \ell = 1^\frown \ell = 1 \Rightarrow \ell = 33^\frown \ell = 1}$$

La prémisse est valide (implication entre deux propositions fausses), mais la conclusion est fausse. Comment pouvons-nous alors éviter ce problème ? La règle que nous voulons définir aura la forme :

$$\frac{\Gamma \vdash X(P, Q)}{\Gamma \vdash P^\frown R \Rightarrow Q^\frown R}$$

avec $X(P, Q)$ une certaine formule mettant en jeu P et Q. Il se trouve que l'implication de Coq implémente le *modus ponens* à plusieurs ordres (termes de type Set, Prop, Type). Il est donc envisageable de profiter de cela, pour définir un axiome qui aura le même effet :

$$\frac{\Gamma \vdash X(P, Q) \qquad \vdash X(P, Q) \Rightarrow (P^\frown R \Rightarrow Q^\frown R)}{\Gamma \vdash P^\frown R \Rightarrow Q^\frown R}$$

Il nous reste donc à définir à quoi correspond $X(P, Q)$. La réponse nous est en fait déjà fournie par Heilmann [Hei99] : $X(P, Q) \equiv \Box(P \Rightarrow Q)$. Donc l'axiome que nous devons implémenter dans Coq sera :

```
Axiom monotony_left :
forall p q : Prop, [](p -> q) ->
  forall r : Prop, chop p r -> chop q r.
```

$$\begin{array}{lll} P & \vdash & \neg(\neg P^\frown Q) \\ & \vdash & \neg(R^\frown \neg(\neg(\neg P^\frown Q))) \\ & \vdash & \neg(R^\frown(\neg P^\frown Q)) \\ & \vdash & \neg(\mathbf{true}^\frown(\neg P^\frown \mathbf{true})) \\ & \vdash & \Box P \end{array}$$

FIG. 5.2 – Un théorème est vrai pour tous ses sous-intervalles

Règle d'inférence : nécessitation

Soient P, Q deux propositions, la règle de nécessitation gauche est décrite par Hansen et Zhou [HZ04] de la manière suivante :

$$\frac{\mathrm{P}}{\neg(\neg P^\frown Q)}$$

avec la règle symétrique pour la nécessitation droite. Nous nous limiterons à la nécessitation gauche puisque les arguments que nous développons ici sont applicables à la symétrie près.

Cette règle nous indique que tout théorème P du système de preuve ne peut devenir faux sur aucun sous-intervalle gauche de l'intervalle complet courant. Cette règle est importante, puisqu'elle permet d'obtenir la *règle de déduction* de la logique d'intervalle (et de DC par extension). Soit une proposition P, appliquons-lui la nécessitation gauche et droite, en figure 5.2.

Donc la figure 5.2 nous indique que tout théorème de la logique d'intervalle devient automatiquement vrai pour tous ses sous-intervalles (nous avons omis l'étape où nous éliminons la double négation, il suffit pour cela de se référer à la logique classique).

Remarquons cependant que cette déduction n'est valide que parce nous n'avons pas d'hypothèses supplémentaires. En effet, supposons que nous définissions comme règle de nécessitation la règle d'inférence suivante :

$$\frac{\Gamma \vdash P}{\Gamma \vdash \neg(\neg P^\frown Q)}$$

Alors il devient facile de dériver une preuve fausse de la forme :

$$\cfrac{\cfrac{\ell_{=}2 \vdash \ell_{>}1}{\ell_{=}2 \vdash \neg(\neg(\ell_{>}1)^\frown Q)}}{\vdash \neg(\ell_{\leq}1^\frown Q)}$$

Cette dérivation signifierait que sur un intervalle de 2 secondes on ne peut pas trouver de sous-intervalle gauche de moins d'une seconde. Donc si la règle initiale est vraie (un intervalle de 2 secondes est plus long qu'une seconde), la conclusion est fausse. Ce contre-exemple montre que la nécessitation ne doit pas être modélisée naïvement lorsque d'autres hypothèses entrent en jeu.

La solution proposée par Heilmann [Hei99] est une règle de la forme :

$$\frac{\Gamma^{\Box} \vdash P}{\Gamma \vdash \Box P}$$

où $\Gamma^{\Box}$ sont les formules de Γ préfixées par $\Box$. Cette solution n'est pas envisageable pour nous, puisque nous n'avons pas le contrôle de *toutes* les hypothèses locales d'une preuve. Par exemple, supposons que nous mettions cette règle sous forme d'axiomes utilisant la simple implication [14] : $(\Box P \Rightarrow Q) \Rightarrow (P \Rightarrow \Box Q)$. Si nous choisissons $P \equiv \mathbf{true}$, nous pouvons dériver une déduction de $\ell_{=}1 \vdash \Box(\ell_{=}1)$, ce qui est faux. Sachant que nous ne pouvons pas intervenir directement sur les tactiques de base de Coq, comment pouvons-nous alors obtenir le comportement désiré de la nécessitation ? Observons un instant la règle de nécessitation telle qu'elle existe dans d'autres logiques. Elle a souvent la forme :

$$\frac{P}{\Box P}$$

C'est-à-dire que tout théorème de la logique est forcément vrai *tout le temps*. C'est également ce qu'il se passe dans le cas de IL : la règle de nécessitation de IL n'est qu'une forme compatible avec $\frown$ de la règle indiquée ci-dessus. De plus, nous avons également les théorèmes suivants : $\Box P \Rightarrow P$ et $P \Rightarrow \Diamond P$. Nous pouvons donc observer une sorte de « gradation » de la puissance des formules selon leur définition d'intervalle : respectivement pour tout sous-intervalle, sans information d'intervalle, pour un sous-intervalle donné.

Cette notion de gradation est par ailleurs utilisée dans d'autres logiques modales (par exemple pour l'encodage de la logique S5[LL32] par de Wind[dW01]). Elle est présente également dans IL. La règle de nécessitation présente dans le système de preuves de IL sert donc deux buts :

- créer un pont entre les théorèmes hors nécessitation, et les théorèmes soumis à la nécessitation
- donner la relation entre la notion de nécessitation, le découpage d'intervalles et les théorèmes de la logique

Le deuxième point est implicite tant dans le système de preuve que dans ses encodages, puisque $\Box$ est à chaque fois défini en termes de découpage d'intervalle $\frown$. En revanche, le premier point est plus délicat puisque notre contrainte d'encodage ne nous permet pas de redéfinir les tactiques de Coq. Dès lors, la solution est de rétablir nous-mêmes ce pont entre les théorèmes « normaux » et les théorèmes de formules nécessitées.

Notre encodage de la règle de nécessitation passe par les axiomes sur lesquels celle-ci a une influence. Il s'agit de faire passer les axiomes à la nécessitation (ainsi nous « court-circuitons » la règle d'inférence de nécessitation) et de fournir un moyen pour raisonner sur les formules nécessitées (i.e. valables pour tout sous-intervalle de temps).

En pratique, tous les axiomes de la logique d'intervalle sont passés à la nécessité. De la même manière, pour les connecteurs logiques déjà définis (ceux de Coq), il faut également rajouter les théorèmes mais cette fois valables pour tout sous-intervalle. Enfin, nous devons toujours être capable de :

- Faire le pont entre les formules nécessitées et les formules normales. Pour cela nous rajoutons un axiome $\Box P \Rightarrow P$.

[14] Précisons ici que le passage de la déduction à l'implication est en général faux

– Raisonner sur des formules nécessitées. Pour cela nous rajoutons un axiome qui porte le Modus Ponens au niveau des formules nécessitées : $\Box(P \Rightarrow Q) \Rightarrow \Box P \Rightarrow \Box Q$.

Ces deux axiomes sont des théorèmes dans d'autres mises en oeuvre d'IL. Pour résumer, la mise en oeuvre de la règle de nécessitation dans notre plongement léger consiste en :

– L'ajout de deux axiomes faisant le lien entre les formules normales et les formules tout le temps vraies : $\vdash \Box P \Rightarrow P$ et $\vdash \Box(P \Rightarrow Q) \Rightarrow \Box P \Rightarrow \Box Q$
– Une implémentation particulière des axiomes de la logique :
  – Les axiomes de IL (et, plus loin, de DC) sont implémentés sous la forme *nécessitée*, i.e. ils sont pris comme vrais pour tout sous-intervalle. Les axiomes classiques de IL peuvent être déduits de ceux-là par l'un des deux axiomes cités au-dessus
  – Les axiomes déjà présents sont augmentés par leur version nécessitée. Si P est un axiome de Coq, alors nous ajoutons l'axiome $\Box P$.

Un dernier problème subsiste : lorsque nous parlons des axiomes de Coq, de quels axiomes s'agit-il en fait ? Les connecteurs logiques de Coq ne sont pas définis par une liste de propositions bien précises reprenant par exemple les axiomes de Hilbert pour la logique classique. Ils sont en fait définis *inductivement*, et des schémas d'axiomes sont générés par Coq à partir de cette définition inductive. Pour chaque définition inductive faite en Coq, trois schémas d'axiome sont générés : chacun d'entre eux définit le schéma d'axiome d'induction sur une sorte de Coq (`Set`, `Prop` ou `Type`).

Ce qui nous intéresse donc ici, c'est de faire passer pour chaque connecteur logique son schéma d'axiome sur `Prop` à la nécessité. Par exemple, le connecteur logique de conjonction est défini de la façon suivante :

```
Inductive and (A B:Prop) : Prop :=
    conj : A -> B -> A /\ B
 where "A /\ B" := (and A B) : type_scope.
```

De cette définition, Coq génère les trois schémas d'axiomes :

```
and_ind : forall A B P : Prop,
  (A -> B -> P) -> A /\ B -> P
and_rec : forall (A B : Prop) (P : Set),
  (A -> B -> P) -> A /\ B -> P
and_rect : forall (A B : Prop) (P : Type),
  (A -> B -> P) -> A /\ B -> P
```

Ici c'est le schéma d'axiome `and_ind` qui nous intéresse, puisque c'est lui qui définit la conjonction sur le type `Prop`. Nous devons donc définir pour ce schéma d'axiome (dit *d'élimination*) et l'axiome $A- > B- > A/\ B$ (dit *d'introduction*), définir les axiomes correspondants, passés à la nécessité :

```
Axiom always_conj :
  forall p q:Prop, [](p -> q -> p /\ q).

Axiom always_and_ind :
  forall p q r:Prop, []((p -> q -> r) -> p /\ q -> r).
```

Des axiomes similaires doivent être introduits pour chaque axiome utilisé dans le système de preuve de DC : connecteurs de la logique classique, et bibliothèque sur les réels.

Au vu de cette contrainte, la question se pose : dans ce cas, plutôt que compléter chaque axiome avec la version nécessitée manquante, créer une tactique Coq dédiée n'aurait-il pas été plus intéressant ? En supposant que nous soyons capables effectivement d'écrire la tactique en question (qui requerrait l'intervention d'un programme en OCaml, comme pour les autres tactiques), cette tactique risquerait d'avoir des effets de bord néfastes :

– Elle fonctionnerait pour tout axiome ou théorème utilisé, y compris ceux de nouvelles théories introduites par l'utilisateur. Considérons par exemples les axiomes sur les listes : avec une telle tactique, les axiomes nécessités deviendraient disponibles. Or, qu'est-ce qui nous indique que ces axiomes nécessités sont voulus dans l'extension théorique de l'utilisateur ? Le danger ici est donc de potentiellement rendre une extension théorique incohérente en pratique.
– Comment cette tactique se comporterait-elle avec les autres, notamment les tactiques à visée automatique ? Bien qu'il n'y ait pas de réponse claire à cette question, les scénarios de mauvais comportement existent : par exemple la tactique de nécessitation pourrait retirer trop tôt la nécessité, et rendrait le théorème impossible à prouver.

Ces deux dernières exemples indiquent que notre solution, qui est de rajouter les versions nécessitées des axiomes, et l'approche la plus prudente et contrôlable. La prochaine section traite des règles d'instanciation et de généralisation.

Règles d'inférence : instanciation et généralisation

Les règles concernant les quantificateurs, dans le système de preuve de DC, agissent sur des valeurs du domaine numérique sur lequel est basé le système. Il s'agit donc du domaine des nombres réels, usuellement.

Généralisation : La règle de généralisation de DC est :

$$\frac{P(x)}{\forall x.P(x)}$$

Cette règle correspond à la règle que l'on a l'habitude de retrouver en logique classique. La particularité qui nous la fait citer ici, c'est son nom : généralisation. En effet si nous lisons dans le sens prémisse-conclusion, nous voyons que la variable x passe à la généralisation. En Coq, soit un type R, la règle similaire est la suivante :

$$\frac{\Gamma, x : R \vdash P(x)}{\Gamma \vdash \forall x : R, P(x)}\ \text{intro}$$

est plutôt appelée une règle d'introduction, puisque la variable x est *introduite* dans les hypothèses pendant le déroulement de la preuve. En effet, en Coq les preuves s'écrivent plutôt de la conclusion (à prouver) vers les prémisses (qui à leur tour doivent être prouvés). C'est la différence entre l'approche *bottom-up* du système de preuve de DC, qui va des axiomes vers les théorèmes, et l'approche *top-down* de Coq, qui va des théorèmes à prouver vers les axiomes à partir desquels la preuve est dérivée.

Cette remarque était nécessaire pour éviter le contresens dangereux qui peut arriver lorsque l'instanciation de DC sera traitée. Comme nous le voyons juste après, celle-ci aura le nom opposé en Coq.

Instanciation La règle d'instanciation (abrégée en Q pour "quantification") de DC est en fait un schéma d'axiome :

$$\frac{}{\forall x.P(x) \Rightarrow P(t)} \quad \begin{array}{l} \text{si } t \text{ est libre pour } x \text{ dans } P(x) \text{ et} \\ \left\{ \begin{array}{l} \text{ou } t \text{ est rigide} \\ \text{ou } P(x) \text{ ne contient pas de } \frown \end{array} \right. \end{array}$$

La règle de Coq correspondante, qui est lue dans l'autre sens, a pour nom `generalize`. Si nous adaptons l'exemple du manuel de référence de Coq, la règle a la forme :

$$\frac{x, y \in \mathbb{N} \vdash \forall n \in \mathbb{N}, n \geq 0}{x, y \in \mathbb{N} \vdash x + y \geq 0}\,\text{generalize}$$

La question que nous nous posons donc est la suivante : cette tactique, sans aucune modification, est-elle compatible avec les autres adaptations que nous avons faites ? Penchons-nous un instant sur la raison d'être de la condition d'utilisation de la règle d'instanciation. Elle existe pour éviter des déductions de la forme :

$$\frac{\forall x. \ell = x \frown \ell = x \Rightarrow \ell = 2x}{\ell = \ell \frown \ell = \ell \Rightarrow \ell = 2\ell}$$

La prémisse est correcte (la concaténation de deux intervalles de même longueur est un intervalle de longueur double) et la conclusion est fausse (ℓ est égal à deux fois lui-même, ce qui n'est valide que si ℓ vaut 0). La condition de bord permet donc d'éviter de remplacer ce qui était pensé être *rigide* par un terme flexible. Dans l'exemple, x était pensé comme étant une valeur référence rigide à laquelle ℓ était mesuré. Remplacer x par ℓ était donc un contresens.

Or nous avons codé ℓ prédicativement : la fausse déduction de la forme indiquée plus haut n'est plus possible, puisque x ne peut plus être remplacé directement par ℓ. Mais cela suffit-il pour nous garantir que d'autres déductions fallacieuses sont impossibles ? Observons plus attentivement la condition de bord du schéma d'axiome d'instanciation de IL : la formule doit être ou bien exempte de découpage d'intervalle, ou bien le terme instancié doit être rigide (nous faisons abstraction des problèmes de capture de visibilité, puisqu'ils sont traités automatiquement par Coq). Cela signifie qu'une instanciation « dangereuse » se fait sur une formule contenant plusieurs intervalles, avec un terme flexible. Cela a pour but d'éviter la « capture » par les contextes locaux de la portée réelle du terme flexible, initialement défini dans un contexte plus global. Autrement dit, cette condition de bord particulière est le dual temporel de la condition de bord habituelle de l'instanciation classique : de la même manière que la condition de bord sur la liberté d'une variable permet son instanciation en évitant sa capture par un quantificateur, les conditions de bord pour l'instanciation des termes permettent aussi l'instanciation, en évitant la capture par un contexte local.

La solution de Heilmann [Hei99] est de remplacer l'axiome d'égalité par une version qui passe au contexte local, en obligeant les valeurs égales à l'être *pour tout sous-intervalle* (cf section 5.4.2.0). Donc cette solution, plutôt que d'éviter la capture d'une variable flexible par un contexte local, s'assure plutôt que la variable conserve ses propriétés pour tout sous-intervalle, donc y compris pour un sous-intervalle particulier.

Notre solution est plutôt de ne rien spécifier : en effet, pour qu'une variable soit « dangereuse », il faut qu'elle soit flexible, c'est-à-dire qu'elle doit être au final la représentation d'une fraction de ℓ, même si cette fraction change selon les sous-intervalles locaux. Or ℓ, dans notre implémentation, n'existe plus en tant que quantité : elle est devenue un prédicat. Il n'est donc plus possible d'instancier une variable x par ℓ ou une fraction de celle-ci.

En conclusion, les règles d'instanciation et de généralisation de Coq ne sont aucunement modifiées dans notre implémentation. Les prochaines sections traitent des ajouts pour passer de la logique d'intervalle au calcul des durées.

Axiomes de DC

La partie difficile de la mise en oeuvre reste la partie qui concerne la logique d'intervalle. En effet, le calcul des durées ne fait pas d'hypothèses sur la logique d'intervalle sur laquelle il se base : il existe des versions du calcul des durées basées sur la logique de voisinage (Neighbourhood logic, voir l'article de Pilegaard [PHS03]) et sur la logique d'intervalle signée (voir la thèse de Rasmussen [Ras02]). La seule réelle difficulté pour la mise en oeuvre de DC sera donc l'adaptation aux changements apportés à la logique d'intervalle. Nous nous bornerons donc ici à la description de ces changements particuliers.

Plusieurs axiomes de DC^* mettent en relation d'égalité la longueur d'intervalle ℓ et la durée d'un état, par exemple : $\vdash \ell = \int 1$. La durée d'état est donc un terme intrinsèquement flexible. L'erreur serait donc de l'implanter comme une fonction à valeurs réelles, de la même façon que l'erreur naïve était d'implanter ℓ comme une variable réelle. Les modifications apportées à l'opérateur de durée d'état $\int$ seront donc du même type que celle apportées à ℓ : nous rendons $\int$ prédicatif, et définissons un ordre sur celui-ci comme nous l'avons fait pour ℓ :

```
Inductive DCState : Set :=
  | SZero : DCState
  | SOne : DCState
  | SOr : DCState -> DCState -> DCState
  | SNot : DCState -> DCState

Parameter duration_eq:DCState -> R -> Prop.
Parameter duration_lt:DCState -> R -> Prop.

Notation "# s #= x":=(duration_eq s x)
  (at level 0, no associativity).
Notation "# s #< x":=(duration_lt s x)
  (at level 0, no associativity).
```

Supposons que nous ayions choisi une représentation naïve pour $\int$: dans ce cas une fausse dérivation comme il pouvait être fait avec le schéma d'instanciation aurait été possible. Donc, avec cette définition, l'axiome $\vdash \int 1 = \ell$ devient dans notre implémentation :

$$\forall x \in \mathbb{R}, \int_{=}(1,x) \Leftrightarrow \ell_{=}x$$

Cette formule signifie qu'il est équivalent de dire que l'intervalle dure un certain temps, et que l'événement qui est tout le temps vrai dure tout ce temps. La version Coq de cet axiome est la suivante :

```
Axiom DCA2 : forall (x:R), [](#1#= x <-> l=x).
```

Comme les codages de ℓ et $\int$ ne peuvent plus être mis en relation directe, nous devons passer par une valeur intermédiaire. Les adaptations aux autres axiomes sont similaires. Également, nous devons passer l'axiome à la nécessité $\Box$, pour les raisons invoquées au sujet de la nécessitation.

Règles d'inférence de DC

Les changements des règles d'inférence pour la durée d'états sont encore plus minimes. Nous pouvons nous servir directement du *modus ponens* de Coq pour définir les règles d'induction sur le découpage d'intervalles d'événements. L'une de ces règles (il y a une règle pour le sous-intervalle gauche et une pour le sous-intervalle droit) est la suivante :

$$\frac{\vdash H(\lceil\rceil) \qquad H(X) \vdash H(X \vee X^\frown \lceil S_1 \rceil \vee \ldots \vee X^\frown \lceil S_n \rceil)}{\vdash H(\mathbf{true})}$$

avec $S_1 \vee ... \vee S_n \Leftrightarrow$ **true**. Cet axiome est mis en oeuvre tel quel sous la forme d'une implication, et nous utilisons une structure de liste ainsi que quelques fonctions *ad hoc*. Nous en déduisons également le théorème similaire à cette règle d'inférence, où $S_1 \vee \ldots \vee S_n \equiv S \vee \neg S$, plus pratique d'utilisation.

Règles d'inférence de DC*

La règle d'inférence de DC* fait appel à une définition bornée de l'itération, et à la durée d'états. Elle est très similaire dans son principe aux règles d'induction sur le découpage d'intervalle :

$$\frac{\forall k < \omega, [(\lceil S \rceil \vee \lceil \neg S \rceil)^k / A]\varphi}{\vdash [\mathbf{true}/A]\varphi}$$

Le ω est l'infini positif des entiers naturels. Étant donné que la prémisse de cette règle est quantifiée par un entier naturel, elle sera démontrée la plupart du temps via un raisonnement par induction sur celui-ci. Nous disposons donc de deux choix dans la définition de l'itération :
- Une définition purement axiomatique, qui sera plus tard mise en relation avec la notion d'itération bornée φ^k
- Une définition qui est une conséquence de la définition de l'itération bornée.

Puisque la règle d'inférence utilise naturellement cette dernière, c'est de cette manière que nous choisissons de la mettre en oeuvre. Nous définissons une fonction Coq qui réalise la concaténation en *n* intervalles d'une formule *P*, puis une fonction construisant une disjonction entre toutes les concaténations composées de 0 à *n* formules concaténées :

```
Fixpoint repetition (p:Prop) (n:nat) {struct n} :Prop :=
match n with
 | 0 => (length_eq 0%R)
 | (S k) => chop (repetition p k) p
end.

Fixpoint repetition_closure (p:Prop) (n:nat) {struct n} :Prop :=
match n with
 | 0 => repetition p 0
 | (S k ) => (repetition p n) \/ (repetition_closure p k)
end.
```

Grâce à cette dernière fonction, nous pouvons donc donner la définition d'une formule itérée :

```
Parameter iteration : Prop -> Prop.

Axiom star : forall p :Prop,
  []((iteration p) <-> (forall n:nat, (repetition_closure p n))).
```

Cet axiome est passé à la nécessitation pour les raisons déjà invoquées précédemment. Nous donnons également la définition de la règle d'inférence exploitant la définition de la répétition plus haut :

```
Axiom IRDCstar : forall P : Prop -> Prop,
  (forall n : nat, forall S:DCState,
    (P (repetition ([[ S ]] \/ [[ ~S ]]) n))) -> (P True).
```

Axiomes de DC*

De la même manière que pour les axiomes de DC, nous ajoutons la version nécessitée des axiomes de DC*. À vrai dire, l'axiomatique pour le connecteur d'itération s'applique plus à la logique d'intervalle qu'au calcul des durées, puisqu'aucun opérateur de durée n'apparaît dans ces axiomes. Un exemple d'axiome est : $\vdash \ell = 0 \Rightarrow \phi^*$, qui signifie qu'un intervalle de longueur nulle est aussi un intervalle itéré (itéré 0 fois, précisément).

La version Coq de cet axiome, à la nécessitation près, ne change pas :

```
Theorem DCstar1 : forall P:Prop,
  []((length_eq 0) -> (iteration P)).
```

La différence notable est que nous ne l'avons pas défini comme un axiome, mais comme un théorème : en effet l'`itération` telle que nous l'avons définie nous permet de déduire cet axiome. Il en va de même pour les autres axiomes de DC*.

Avec l'ajout de ces derniers axiomes, le système résultant permet alors de raisonner sur des formules du calcul des durées avec itération. La section suivante résume de quoi ce système est constitué.

Système de preuve résultant

Pour faciliter la lecture, nous avons noté $\vdash_\Box$ les axiomes implémentés sous forme nécessitée, et dont la forme non nécessitée est immédiatement déduite par la règle $N1_{refl}$. Les implémentations complètes se trouvent en annexe B.

Le tableau 5.2 présente sous formes de formules mathématiques les axiomes ajoutés à Coq pour obtenir un système de preuve utilisable pour démontrer des théorèmes de la logique d'intervalle.

$\ell_< \neq$	$\vdash_\Box \ell_< x \Rightarrow \ell_{\neq} x$	$\ell_> >$	$\vdash_\Box \ell_> x \Rightarrow x > y \Rightarrow \ell_> y$
$\ell_= >$	$\vdash_\Box \ell_= x \Rightarrow x > y \Rightarrow \ell_> y$	$\ell_> <$	$\vdash_\Box \ell_> x \Rightarrow \ell_< y \Rightarrow x < y$
$\ell_= <$	$\vdash_\Box \ell_= x \Rightarrow x < y \Rightarrow \ell_< y$	$\ell_> =$	$\vdash_\Box \ell_> x \Rightarrow \ell_= y \Rightarrow x < y$
$\ell_= \ell_<$	$\vdash_\Box \ell_= x \Rightarrow \ell_< y \Rightarrow x < y$	$\ell_< <$	$\vdash_\Box \ell_< x \Rightarrow x < y \Rightarrow \ell_< y$
$\ell_= \ell_=$	$\vdash_\Box \ell_= x \Rightarrow \ell_= y \Rightarrow x = y$	$\ell_< =$	$\vdash_\Box \ell_< x \Rightarrow \ell_= y \Rightarrow y < x$
$\ell_= \ell_>$	$\vdash_\Box \ell_= x \Rightarrow \ell_> y \Rightarrow x > y$	$\ell_< >$	$\vdash_\Box \ell_< x \Rightarrow \ell_> y \Rightarrow y < x$
		$\ell_\exists$	$\vdash_\Box \exists x. \ell_= x$
$A0$	$\vdash_\Box \ell_\geq 0$	B_l	$\vdash_\Box (\exists x.\phi)^\frown \psi \Rightarrow \exists x.\phi^\frown \psi$
$A1_r$	$\vdash_\Box \begin{cases} (\phi^\frown \psi) \wedge \neg (\phi^\frown \varphi) \\ \Rightarrow (\phi^\frown (\psi \wedge \neg \varphi)) \end{cases}$	B_r	$\vdash_\Box \phi^\frown (\exists x.\psi) \Rightarrow \exists x.\phi^\frown \psi$
		$L1_r$	$\vdash_\Box \ell_= x^\frown p \Rightarrow \neg (\ell_= x^\frown \neg p)$
$A1_l$	$\vdash_\Box \begin{cases} (\phi^\frown \psi) \wedge \neg (\varphi^\frown \psi) \\ \Rightarrow ((\phi \wedge \neg \varphi)^\frown \psi) \end{cases}$	$L1_l$	$\vdash_\Box p^\frown \ell_= x \Rightarrow \neg (\neg p^\frown \ell_= x)$
$A2$	$\vdash_\Box (p^\frown q)^\frown r \Leftrightarrow p^\frown q^\frown r$	$L2$	$\vdash_\Box \begin{cases} x \geq 0 \wedge y \geq 0 \\ \Rightarrow ((\ell_= x + y) \Leftrightarrow \ell_= x^\frown \ell_= y) \end{cases}$
R_l	$\vdash_\Box \phi^\frown \psi \Rightarrow \phi$ si ϕ est rigide	$L3_l$	$\vdash_\Box \phi \Rightarrow \phi^\frown \ell_= 0$
R_r	$\vdash_\Box \phi^\frown \psi \Rightarrow \psi$ si ψ est rigide	$L3_r$	$\vdash_\Box \phi \Rightarrow \ell_= 0^\frown \phi$
M_l	$\vdash \Box(p \Rightarrow q) \Rightarrow (p^\frown r \Rightarrow q^\frown r)$		
M_r	$\vdash \Box(p \Rightarrow q) \Rightarrow (r^\frown p \Rightarrow r^\frown q)$		
$N1_{MP}$	$\vdash \Box(p \Rightarrow q) \Rightarrow \Box p \Rightarrow \Box q$		
$N2_{refl}$	$\vdash \Box p \Rightarrow p$		
$True_\Box$	$\vdash \Box True$		
$True_{ind\Box}$	$\vdash \Box(p \Rightarrow True \Rightarrow p)$		
$and_\Box$	$\vdash \Box(p \Rightarrow q \Rightarrow p \wedge q)$		
$and_{ind\Box}$	$\vdash \Box((p \Rightarrow q \Rightarrow r) \Rightarrow p \wedge q \Rightarrow r)$		
$or_{left\Box}$	$\vdash \Box(p \Rightarrow p \vee q)$		
$or_{right\Box}$	$\vdash \Box(q \Rightarrow p \vee q)$		
$or_{ind\Box}$	$\vdash \Box((p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow p \vee q \Rightarrow r)$		
$always\_S$	$\vdash \Box(p \Rightarrow q \Rightarrow p)$		
$always\_K$	$\vdash \Box((p \Rightarrow q) \Rightarrow (p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow r))$		

TAB. 5.2 – Axiomes et règles d'inférence pour IL

Le tableau 5.3 montre les axiomes à ajouter au tableau précédent pour obtenir un système pour le calcul des durées.

Le tableau 5.4 montre les axiomes à ajouter aux tableaux précédents pour obtenir un système de preuve pour le calcul des durées avec itération. Il est à noter que les axiomes présentés dans ce tableau sont en fait des théorèmes conséquence de la définition de la répétition de formules sur un intervalle de temps, comme indiqué à la fin de la section 5.4.2

$DCA1$	$\vdash_{\Box} \int_{=}(0,0)$
$DCA2$	$\vdash_{\Box} \int_{=}(1,x) \Leftrightarrow \ell_{=}x$
$DCA3$	$\vdash_{\Box} \int_{\geq}(s,0)$
$DCA4$	$\vdash_{\Box} \begin{cases} \int_{=}(s_1,x_1) \\ \Rightarrow \int_{=}(s_2,x_2) \\ \Rightarrow \int_{=}(s_1 \wedge s_2, and_{12}) \\ \Rightarrow \int_{=}(s_1 \vee s_2, or_{12}) \\ \Rightarrow x_1 + x_2 = and_{12} + or_{12} \end{cases}$
$DCA5$	$\vdash_{\Box} \int_{=}(s,x)^\frown \int_{=}(s,y) \Rightarrow \int_{=}(s,x+y)$
$DCA6$	$\vdash_{\Box} (PL(s1 \Rightarrow s2)) \Rightarrow (\forall x \in \mathbb{R}. \int_{=}(s1,x) \Rightarrow \int_{=}(s2,x))$
$IR1$	$\vdash \begin{cases} (PL(s_1 \vee \ldots \vee s_n)) \\ \Rightarrow (H \llbracket\rrbracket) \\ \Rightarrow ((H\,X) \Rightarrow (H\,(X \vee (X^\frown \llbracket s_1 \rrbracket) \vee \ldots \vee (X^\frown \llbracket s_n \rrbracket))) \\ \Rightarrow (H\,\mathbf{true}) \end{cases}$
$IR2$	$\vdash \begin{cases} (PL(s_1 \vee \ldots \vee s_n)) \\ \Rightarrow (H \llbracket\rrbracket) \\ \Rightarrow ((H\,X) \Rightarrow (H\,(X \vee (\llbracket s_1 \rrbracket^\frown X) \vee \ldots \vee (\llbracket s_n \rrbracket^\frown X))) \\ \Rightarrow (H\,\mathbf{true}) \end{cases}$

TAB. 5.3 – Axiomes et règles d'inférence pour DC

DC^*1	$\vdash_{\Box} \ell_{=}0 \Rightarrow \phi^*$
DC^*2	$\vdash_{\Box} \phi^{*\frown}\phi \Rightarrow \phi^*$
DC^*3	$\vdash_{\Box} (\phi^* \wedge \psi^\frown \mathbf{true} \Rightarrow (\psi \wedge \ell_{=}0^\frown \mathbf{true})) \vee (((\phi^* \wedge \neg\,\psi^\frown \phi) \wedge \psi)^\frown \mathbf{true})$
ω	$\vdash (\phi(\llbracket S \rrbracket \vee \llbracket \neg\, S \rrbracket)^k) \Rightarrow (\phi(\mathbf{true}))$

TAB. 5.4 – DC*-Coq

La façon dont ce système est construit nous permet d'énoncer le théorème suivant :

Théorème 5.1 *Le système de preuves, appelé DC-Coq, mis en oeuvre par les tableaux 5.2, 5.3 et 5.4 est correct. C'est-à-dire que tout théorème de DC-Coq est également un théorème du système de preuves de DC.*

PROOF: La preuve est simple et se fait par induction sur la longueur de l'arbre de preuve :

- Les preuves de taille 0 sont les axiomes de notre système : ce sont tous des axiomes ($A0$ par exemple), des théorèmes ($True_{\Box}$ par exemple) ou des définitions (l'itération par exemple). Donc ces preuves correspondent à des preuves existantes du système de preuve de DC
- Qu'en est-il des preuves de taille n+1 ? Nous pouvons déjà dire qu'elles sont le résultat de l'application d'une tactique de Coq, sur des preuves dont au moins l'une d'entre elles a la longueur n et les autres des longueurs d'au plus n. Donc ces sous-preuves ont pour conclusion des théorèmes de DC, par application de l'hypothèse d'induction. Ces théorèmes permettent-ils de déduire la conclusion de notre preuve de longueur n+1 ? Nous devons vérifier au cas par cas, selon la tactique employée. Puisque les tactiques de Coq sont des cas particuliers des règles d'inférence du calcul des constructions inductives, les cas à examiner se ramènent à ces mêmes règles d'inférence :

  Let La règle `Let` un mécanisme de définition de variable locale, surtout utile à Coq pour inférer le terme correspondant à une preuve, et aux règles de conversion des lambdas-expressions, et n'a aucune influence sur la proposition elle-même. Donc la preuve de longueur $n+1$ est valide.

  App Il s'agit du Modus Ponens de Coq, que l'on fait correspondre au Modus Ponens de DC ou à l'instanciation de DC. Le Modus Ponens se retrouve également dans DC. L'instanciation telle que nous l'avons mise en oeuvre empêche d'utiliser des termes flexibles, puisque nous avons rendu ℓ prédicatif. Donc la preuve de longueur $n+1$ est valide.

  Lambda Cette règle permet de construire les formules de type produit, i.e. une conclusion de la forme $\forall x : T, U : Prop$ (de même pour les types Set et Type, mais nous nous limitons à Prop). Selon que x apparaît ou non dans U, il s'agit d'un produit dépendant ou non-dépendant (et dans ce cas Coq l'affichera $T \rightarrow U$). Cette règle correspond soit à la généralisation, soit à l'affaiblissement de formule (si Q alors $P \rightarrow Q$, si P est un théorème) dans DC. Donc notre preuve de longueur $n+1$ est valide.

  Les autres règles concernent le typage et la bonne formation des termes du CCI.

Donc nous pouvons en conclure qu'à toute preuve d'un théorème dans DC-Coq correspond un théorème dans le système de preuve de DC.

Notre mise en oeuvre de DC* est assez proche de celle de Heilmann [Hei99], qui laisse de nombreux indices arguant en faveur de la complétude de sa mise en oeuvre (i.e. tout théorème de DC a une preuve en Isabelle/DC). La proximité de notre encodage laisse à penser que celui-ci est également complet. Le seul argument en défaveur correspondrait au fait que les variables du systèmes de preuve de DC n'apparaissent plus explicitement dans notre bibliothèque, et sont plutôt gérées par Coq lui-même. Nous savons donc pas pour le moment quel est l'impact de cette limitation par rapport à la complétude du système.

5.5 Conclusion

Quels apports pouvons-nous déduire de cette bibliothèque de manipulations de formules de DC^* en Coq ? Les apports se situent à deux niveaux, théorique et pratique.

5.5.1 Apport théorique

Les questions de départ, avant l'implémentation, étaient les suivantes :

1. Est-il possible de fournir une bibliothèque pour un outil de preuve pour une logique temporelle en temps continu telle que DC, sans se limiter à une certaine sous-classe de la logique ?
2. Est-il possible de le faire en plongement léger ?
3. Est-il possible de le faire sans devoir implémenter de nouveau les théories sur lesquelles elle est fondée ?
4. Pouvons-nous le faire sans apporter de modifications dangereuses à l'outil de preuve (i.e. qui interviennent directement sur son fonctionnement interne) ?

Des implémentations déjà existantes répondent affirmativement à la première question, en plus de la nôtre. En revanche, pour les questions suivantes notre mise en oeuvre se distingue :

– La mise en oeuvre de Skakkebæk[Ska94] correspond plutôt à un plongement profond, de par l'encodage de la logique selon sa sémantique. La mise en oeuvre de Heilmann[Hei99] correspond également à un plongement profond, cependant le fonctionnement d'Isabelle fait que toute implémentation sera nécessairement un tel plongement. Sachant que les théories sur lesquelles DC est basé sont bien comprises (arithmétique des réels, logique classique), et que les tactiques sont optimisées pour cela, il est raisonnable de l'assimiler à un plongement léger.
– Dans le cadre des assistants de preuve, notre implémentation est la seule en plongement léger (à notre connaissance) qui ne réimplémente pas en grande partie les théories qui fondent DC.
– Toute intervention malvenue sur un assistant de preuve peut le rendre incohérent : notre mise en oeuvre ne manipule en rien les mécanismes internes de Coq. Cet avantage se fait cependant au prix de l'ajout de lourdeur au niveau de l'axiomatique du système (mise en nécessitation de théorèmes). Ce coût est rapidement amorti par la mise à disponibilité de théorèmes permettant de manipuler plus aisément ces formules nécessitées.

Ces réponses nous confirment donc que le contrat que nous nous étions donné à l'origine est rempli. De plus, cette mise en oeuvre indique une façon possible pour dépasser la difficulté d'implémentation liée aux règles d'inférence particulières (nécessitation) des logiques modales, dans un plongement léger.

5.5.2 Apport pratique

L'apport pratique est, de la même manière que l'implémentation en Isabelle, un outil de preuve pour des formules du calcul des durées. Nous allons même un peu plus loin puisque notre implémentation s'applique au calcul des durées *avec itération*. L'apport de Heilmann

[Hei99] en ce domaine est un outil disposant de tactiques dédiées, appuyées par un outil de raisonnement sur l'arithmétique des réels, pour la preuve dans le calcul des durées.

Plus de sûreté est également apportée, puisqu'à aucun moment le recours à un autre outil n'est nécessaire. L'exemple illustrant le mieux ceci est la pré-existence d'une bibliothèque pour Coq et de tactiques dédiées à la résolution d'équations et d'inéquations sur les réels. Une éventuelle introduction d'erreurs est ainsi évitée.

Impact sur les preuves en B

Il existe des travaux implémentant B ou sa partie preuve en Coq par Bodeveix [BFM99], ou dans un formalisme proche de Coq par notre équipe[RCMP04]. Quel impact a donc notre implémentation sur ces travaux ?

Le système de preuve de DC reste compatible avec un calcul des prédicats de premier ordre : il en va de même pour notre mise en oeuvre. Du reste, elle est elle-même déjà construite pardessus un système permettant le raisonnement en logique classique. Donc fondamentalement, pour les preuves « classiques » de machines B, aucune modification n'est à apporter, puisque le système logique de B est du premier ordre. Dans le cas de formalismes proches, si l'implémentation est elle aussi du premier ordre, alors la problématique du passage à Coq se situe principalement au niveau syntaxique.

Donc en définitive, notre bibliothèque est transparente par rapport aux preuves classiques de machines B : elle pourra donc servir à la fois pour celles-ci, et pour des machines augmentées de contraintes temporelles.

5.5.3 Perspectives

Ce chapitre présentait une mise en oeuvre en plongement léger. Il est également possible de définir un type inductif représentant les formules, et définir plusieurs fonctions ou prédicats qui représenteront l'interprétation de ces formules dans leur sémantique naturelle, ou bien dans le système de preuve lui-même. Il s'agirait là d'un plongement profond plus abstrait que celui fait en Isabelle/HOL, puisqu'il laisserait la place à plusieurs sémantiques possibles. Cette manière de faire est par exemple, pour un autre type de logique temporelle, très proche de la mise en oeuvre de de Wind [dW01]. Un début d'implémentation est en cours à l'heure où ces lignes sont écrites.

Notre travail peut également être amélioré, au niveau du plongement profond, par l'écriture de tactiques dédiées aux formules particulières du calcul des durées. Nous nous sommes focalisés sur l'aspect interactif puisque dans le cas de formules difficiles à prouver, c'est toujours cet aspect qui est perçu par le développeur. Maintenant le côté automatique peut être développé par l'apport de tactiques, autant pour prouver automatiquement des formules du calcul des durées que pour faciliter la preuve interactive.

Nous avons également rapidement traité en section 5.4.2 du cas spécifique de la nécessitation, qui est « éliminée » par l'utilisation d'une axiomatique judicieusement choisie. Nous avions justifié de cette élimination par le fait que l'autre choix possible était de devoir intervenir dans les mécanismes de Coq. À des fins d'expérimentation, il peut cependant être intéressant d'observer l'impact de l'introduction d'une nouvelle tactique du point de vue de la facilité d'utilisation. Les changements à apporter au système pour ce faire sont minimes : les axiomes nécessités devraient être changés en leur version non nécessitée. La difficulté provient simplement de la programmation dans la mécanique interne de Coq de cette tactique. Cette expérimentation peut donc constituer un objectif à court terme.

Bibliographie

[Bar92] Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types, pages 117–309. Clarendon Press, 1992. (Technical Report 91-19, Catholic University Nijmegen, 1991).

[BFM99] Jean-Paul Bodeveix, Mamoun Filali, and César Munoz. A formalization of the B method in Coq and PVS. In BUGM99 [BUG99], pages 32–48.

[BUG99] *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*. Springer-Verlag, 1999.

[CH85] Thierry Coquand and Gérard Huet. Constructions : A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, volume 203 of *Lecture Notes in Computer Science*, Linz, 1985. Springer-Verlag.

[CH87] Thierry Coquand and Gérard Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In The Paris Logic Group, editor, *Logic Colloquium'85*, pages 123–146. Elsevier, 1987. ISBN :0-444-70211-3.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3) :95–120, february 1988.

[Coq85] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, january 1985.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[CPM03] Samuel Colin, Vincent Poirriez, and Georges Mariano. Thoughts about the implementation of the duration calculus with coq. In *4th International Workshop on the Implementation of Logics*, volume Technical report ULCS-03-018. University of Liverpool, september 2003. `http://www.csc.liv.ac.uk/research/techreports/`.

[CPR$^+$05] Samuel Colin, Dorian Petit, Jérôme Rocheteau, Rafaël Marcano, Georges Mariano, and Vincent Poirriez. BRILLANT : An open source and XML-based platform for rigourous software development. In *SEFM (Software Engineering and Formal Methods)*, Koblenz, Germany, september 2005. AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press.

[dW01] Paulien de Wind. Modal logic in Coq. Master's thesis, Vrije Universiteit, Amsterdam, 2001. Supervised by dr. F. van Raamsdonk.

[EN05] Jacob Enslev and Anne-Sofie Nielsen. Bounded model construction for duration calculus. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005. Supervised by Assoc. prof. Martin Franzle and Assoc. prof. Michael R. Hansen.

[Frä02] Martin Fränzle. Take it NP-easy : Bounded model construction for duration calculus. In Springer Verlag, editor, *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant systems (FTRTFT 2002)*, volume 2469, pages 245–264, 2002.

[Hei99] Søren T. Heilmann. *Proof Support for Duration Calculus*. Phd-thesis, Department of Information Technology, Technical University of Denmark, Januar 1999.

[HZ04] Michael R. Hansen and Chaochen Zhou. *Duration Calculus, a formal approach to real-time systems*. Number ISBN : 3-540-40823-1 in Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[Isa93] Isabelle/HOL, 1993. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[LL32] C. I. Lewis and C. H. Langford. *Symbolic logic*. Dover Publications, 1932. 1959 reprint.

[Pan01] Paritosh K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. In *RT-TOOLS'2001*, Aalborg, August 2001. (affiliated with CONCUR 2001). Technical report TCS-00-PKP-1, Tata Institute of Fundamental Research, Mumbai, 2000.

[PHS03] Henrik Pilegaard, Michael R. Hansen, and Robin Sharp. An approach to analyzing availability properties of security protocols. *Nord. J. Comput.*, 10(4) :337–, 2003.

[RAI85] The RAISE typechecker, 1985. `http://www.iist.unu.edu/home/Unuiist/newrh/III/3/1/page.html`.

[Ras02] Thomas Marthedal Rasmussen. *Interval Logic - Proof Theory and Theorem Proving*. PhD thesis, Informatics and Mathematical Modeling, Technical University of Denmark, january 2002.

[RCMP04] Jérôme Rocheteau, Samuel Colin, Georges Mariano, and Vincent Poirriez. Évaluation de l'extensibilité de phox : B/phox un assistant de preuves pour b. In *JFLA*, pages 139–153, jan 2004. `http://pauillac.inria.fr/jfla/2004/`.

[Ska94] Jens Ulrik Skakkebæk. *A Verification Assistant for a Real-Time Logic*. Phd-thesis, Department of Computer Science, Technical University of Denmark, 1994. Also available as Technical Report ID-TR : 1994-150.

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

Chapitre 6

Conclusions et perspectives

Sommaire

Nous résumons dans ce chapitre les buts que nous nous étions fixés, et les moyens mis en oeuvre ainsi que les difficultés rencontrées pour atteindre ces buts. Dans chaque cas, nous nous efforcerons de détailler les avantages et limitations des solutions que nous apportons. Lorsque c'est possible, nous indiquons des perspectives possibles pour dépasser ces limitations.

Cette thèse se compose de deux grandes parties : une description de l'extension de B au calcul des durées, et une mise en oeuvre dans un assistant de preuves de ce calcul. Nous reprenons cette décomposition, puis nous conjecturons d'autres résultats, plus indirects, découlant de nos travaux.

6.1 Apports d'une sémantique temporelle à B

L'idée initiale était d'offrir à B la possibilité d'exprimer et valider des systèmes comprenant des contraintes temporelles. Les questions pouvaient se résumer aux suivantes :

- Est-ce possible, et est-ce que ça a déjà été fait ? Si oui, dans quelle mesure ? Pouvons-nous aller plus loin ?
- Quelle est l'expressivité de l'extension que nous proposons ?
- Qu'est-ce qui change par rapport au B classique ?

Les premières questions trouvent leur écho au chapitre 3 où sont présentées diverses méthodologies étendant B vers la temporalité ou la concurrence. L'extension qui rassemble le plus de traits intéressants, celle de Lano et Dick[LD96], va jusqu'à la concurrence sans partage de variables et une temporalité discrète. D'autres extensions vont parfois plus loin, mais uniquement dans l'un des deux domaines (temporalité ou concurrence). Notre but était alors de pouvoir exprimer en B une concurrence avec partage de variables, et une temporalité qui soit au moins discrète, l'idéal étant de pouvoir exprimer des notions de continuité.

6.1.1 Aménagements nécessaires

Pour ce faire, nous avons exprimé la base *dynamique* de B, les substitutions, dans un formalisme temporel particulièrement expressif : WDC*, le calcul des durées avec itérations, faiblement monotone.

L'expression temporelle des substitutions se base sur une modélisation logique prenant en compte la notion d'exécution de processus et de séquencement d'instructions. Les substitutions qui ne correspondent pas à des comportements dynamiques (telles la précondition) y sont également intégrées. Puisque WDC* contient la logique classique, il est possible de vérifier ou modifier la sémantique ainsi produite pour s'assurer que la sémantique en transformateur de prédicats des substitutions de B est toujours valide.

Ensuite, WDC* a plutôt une vocation de modélisation que de validation. En effet, puisque nous en appelons à l'hypothèse du vrai synchronisme, nous ne sommes intéressés que par les opérations créant des délais *discernables*, comme des attentes ou des délais, spécifiés dans le cahier des charges du système à modéliser. Dans cette optique, nous extrayons donc par une opération de projection des formules DC* à partir des formules WDC* des substitutions. Ces formules exprimées en DC* permettent alors de vérifier simplement les contraintes du système.

La mise en oeuvre suit donc deux lignes directrices : la conservation des propriétés du B classique, et l'exploitation de la puissance d'expression obtenue via la sémantique temporelle.

Conservation des propriétés

La vérification de la sémantique temporelle des substitutions indique que la validité du calcul de plus faible précondition est conservée dans notre extension. Cela signifie donc que la validation *fonctionnelle* (par opposition à temporelle) des machines B doit aussi être réalisée.

De plus, la notion de raffinement est conservée : du côté fonctionnel, c'est le raffinement de B qui continue à être utilisé. Du côté temporel, le raffinement existe également, mais est différent du raffinement fonctionnel : là où celui-ci permet d'assurer que ce sont bien les mêmes calculs qui sont opérés au long de la chaîne de raffinements, le raffinement temporel permet d'assurer que la *manière* dont les calculs sont opérés reste la même. La continuité de (W)DC*, de par la continuité de son domaine de définition (les réels), fait se résumer le raffinement à une implication logique.

Ce dernier résultat constitue un avantage non négligeable par rapport à la plupart des méthodes discrètes. En effet celles-ci doivent disposer d'un mécanisme de bégaiment pour pouvoir introduire de nouveaux sous-événements au long du raffinement. C'est le cas par exemple la définition d'un raffinement pour les systèmes de transition temporisés de [HJMO03].

Puissance d'expression

Les substitutions de B, en tant que telles, contiennent peu de notions relatives à la temporalité. Pour pouvoir exploiter cette nouvelle expressivité, nous avons rajouté des substitutions qui complètent ce manque (illustrées par exemple au tableau 4.2 du chapitre 4) :

- Le délai, qui permet de spécifier une attente durant un certain nombre d'unités de temps
- La concurrence, grâce à laquelle il est désormais possible d'exprimer la simultanéité dans le fonctionnement de composants B
- L'attente réactive, qui met le processus courant en attente de la vérification d'une condition donnée. Il s'agit d'une substitution qui apporte la synchronisation à la notion de simultanéité apportée par la concurrence

Ces nouvelles substitutions, grâce à la manière dont leur sémantique temporelle a été spécifiée, permettent d'apporter la concurrence *avec variables partagées*, et l'expression de propriétés temporelles *selon un temps continu*. Elles sont utilisées comme le sont les substitutions classiques, et leur utilisation est transparente dans le cadre de la double validation (fonctionnelle et temporelle) que nous proposons :

- Comme pour les autres substitutions, nous spécifions une sémantique temporelle de ces substitutions. La nature continue de cette sémantique la rend simple à exprimer (la concurrence, par exemple, se résume quasiment à une conjonction logique)
- Nous donnons une sémantique en calcul de plus faible précondition à ces nouvelles substitutions, et nous vérifions que celle-ci est compatible avec leur sémantique temporelle. De cette manière nous sommes capables d'utiliser ces nouvelles substitutions dans un cadre fonctionnel.

En résumé, il est désormais possible de valider les contraintes fonctionnelles et temporelles de composants B faisant appel à la concurrence, et à des contraintes temporelles (y compris la non-terminaison).

6.1.2 Limitations

Une limitation conséquente à l'ajout d'expressivité est que la validation fonctionnelle de composants mis en concurrence peut être mise à mal. En effet, dans le cas où les composants ont peu de variables partagées, la transformation de la substitution concurrente en une substitution équivalente produit un terme de taille raisonnable. Cependant la technique de transformation utilisée montre clairement que beaucoup de synchronisations entre composants fait grandir exponentiellement la taille de ce terme. Ce problème est connu pour tout formalisme où la séquentialité le dispute à la compositionnalité (la composition de systèmes de transition, par exemple). Les formalismes qui évitent ce piège sont les formalismes de type *assumption/commitment*, mais un tel concept, bien qu'adaptable au B classique, semble s'adapter plus difficilement à B+DC. De ce point de vue, eventB semble être plus adéquat pour une telle transformation.

Une autre limitation concerne la logique temporelle retenue. En effet, nous avons vu que la logique fonctionnait à intervalle fermé : il est donc difficile de spécifier localement des propriétés sur le futur du système. Par exemple, il n'est pas possible de spécifier une contrainte sur l'ordre dans lequel seront appelées les opérations de la machine. Pour cela le contexte doit être celui de l'appelant. Pour résoudre ce problème, il serait possible de spécifier cette contrainte comme devant être vérifiée par l'appelant, mais dans ce cas nous nous dirigerions vers un formalisme de type *assumption/commitment*, ce qui rejoindrait nos commentaires plus haut. Une autre manière de faire pourrait être d'utiliser une logique temporelle continue (telle SIL [Ras02]) qui inclut la notion de futur. Comme les possibilités restent nombreuses, nous nous en tiendrons ici à de simples conjectures.

6.2 Apports de Coq au calcul des durées

Puisque nous utilisons DC* pour obtenir les propriétés temporelles de composants B, nous devions avoir à disposition un outil pour valider celles-ci. Les approches existantes nous semblaient insuffisantes selon des critères qui diffèrent selon les approches :

– Elles se limitent à une sous-classe de la logique
– Leur efficacité peut trop souvent être mise en défaut par la forme de la formule
– Elles peuvent être générales et efficaces, mais alors requièrent une implémentation de toute la logique et de ce sur quoi elle se base (logique classique, arithmétique des réels)

Toutes ces raisons nous ont décidés à essayer une mise en oeuvre qui s'efforcerait d'éviter ces pièges, en utilisant Coq.

6.2.1 Mise en lumière des particularités

La mise en oeuvre de DC* en Coq correspond à un plongement léger du système de preuve. Notre but était de suivre une approche similaire à l'implémentation en Isabelle/HOL de DC, mais en évitant les problèmes posés par les changements apportés par DC à la logique classique (le problème de l'égalité, par exemple).

Pour ce faire, nous avons rendu prédicatives les relations qui mettaient en oeuvre les éléments reliés à la longueur d'intervalle. Ceci nous a permis d'éviter de redéfinir la relation d'égalité de Coq, ce qui aurait eu pour conséquence de suivre la même voie que la mise en oeuvre Isabelle/HOL. Cette transformation met également en lumière la difficulté à mettre en oeuvre les mécanismes modaux des logiques de ce type. En effet ces mécanismes modaux font qu'une même sous-formule n'a pas la même valeur de vérité selon sa place dans la formule : c'est cela qui rend si difficile l'implémentation des logiques modales dans des prouveurs généraux (hors plongement profond, bien entendu).

Une autre difficulté, liée elle aussi à la modalité de la logique, est la gradation de l'expressivité d'une formule selon qu'elle est considérée sur un sous-intervalle, l'intervalle complet ou tout sous-intervalle. Nous avons reproduit cette gradation en rajoutant les axiomes manquants : en effet la logique d'intervalle (et par extension le calcul des durées) spécifie que tout axiome, tout théorème est également vérifié pour tous ses sous-intervalles. Les axiomes que nous avions rajouté à Coq pour DC n'étant pas suffisant pour obtenir ce résultat, nous avons complété l'axiomatique en rajoutant la forme des axiomes de Coq vraie pour tout sous-intervalle. Le résultat en est que tout théorème prouvé en Coq sera effectivement un théorème de DC.

Cependant, si la correction de notre système est vérifiée, nous n'avons pas de résultats de complétude. L'implémentation est assez proche de celle faite avec Isabelle/HOL, ce qui nous laisse à penser que l'implémentation en Coq est très proche de la complétude (tout théorème du système de preuve de DC* est aussi démontrable avec l'implémentation en Coq).

6.2.2 Intérêt pratique pour B

Cette implémentation a une conséquence particulièrement importante, non seulement pour B, mais également pour toute autre méthode formelle « étendue » de manière similaire.

D'un côté, nous avons pris soin d'étendre B avec un formalisme compatible avec sa logique. D'un autre côté, nous avons pris soin d'étendre un outil de preuve avec ce même formalisme. Puisque Coq est suffisant pour valider les propriétés *fonctionnelles* des composants B, la conséquence en est que Coq augmenté des librairies pour DC est également suffisant pour valider leurs propriétés *temporelles*.

Cela signifie que si l'outil de validation pour une méthode formelle n'est pas spécifique à cette méthode, mais qu'il s'agit de la spécialisation d'un outil plus générique, alors il est possible de réaliser une extension conservative de cette méthode formelle et de son outil de validation. Cela signifie, par exemple, qu'un outil trop spécialisé pour une méthode formelle sera trop difficile à faire évoluer pour prendre en compte une extension de cette méthode formelle. Cela pourra même nécessiter le développement d'un nouvel outil.

Cette dernière remarque trouve son écho dans les travaux sur les institutions, où la notion d'extensibilité du formalisme est intégrée dès le départ. Grâce aux institutions, il est en théorie possible de réunir plusieurs extensions du même formalisme à condition que ces extensions respectent certaines propriétés. Étant donné toutes les extensions de B existantes, il peut être intéressant d'observer celles-ci sous l'angle des institutions. Il existe des travaux [LMA$^+$01] qui mettent par exemple en parallèle deux modélisations d'un même système avec la méthode B et avec CASL [BM04, Ini04, CAS]. Le résultat de ces travaux peut être résumé de la manière suivante, au dire des auteurs : :

- Les capacités d'abstraction de CASL en font un outil particulièrement souple et concis pour la spécification d'opérations relativement complexe. Sa jeunesse et la particularité de son approche font néanmoins que l'obtention du modèle est difficile en premier lieu, bien que l'approche utilisée par CASL existe depuis longtemps
- Inversement, la spécification de ces opérations en B se fait naturellement, l'inconvénient étant la complexité (en taille) et la faible lisibilité du modèle obtenu.

L'article [LMA$^+$01] comparant les deux approches conclut que celles-ci sont de nature trop différentes pour pouvoir trancher en faveur de l'une ou de l'autre, et suggère de les utiliser ensemble (modélisation abstraite avec CASL, manipulation du modèle en B après une forme de raffinement de CASL vers B).

6.3 En résumé...

6.3.1 Conclusions

L'extension que nous avons proposée remplit son rôle : elle introduit la concurrence avec variables partagées en B, avec la possibilité d'exprimer des contraintes temporelles. La sémantique en transformateurs de prédicats des substitutions de B est conservée, ce qui offre la possibilité d'opérer une validation en deux temps des développements : une validation fonctionnelle, et une validation temporelle. Les limitations que nous avons pu observer sont pour la plupart inhérentes aux formalismes choisis.

La mise en oeuvre de bibliothèques pour le calcul des durées en Coq, selon un plongement léger, nous dote d'un prouveur pour cette logique réutilisant une grande partie de l'outil de départ (notamment les théorèmes sur l'arithmétique des réels). Cette mise en oeuvre illustre également les difficultés et les contournements pour réaliser un plongement d'une logique modale dans un outil de preuve générique, dans le cadre particulier d'un plongement léger.

6.3.2 Perspectives

Les limitations à l'extension B+DC que nous évoquons plus haut n'existent pas dans des formalismes orthogonaux, ou sont résolues par généralisation, comme nous l'allons voir :

- Les problèmes de compositionnalité (notamment concurrente) sont connus et disposent de solutions axiomatiques depuis longtemps [OG76]. Les approches passant le mieux à l'échelle, et donc éventuellement plus pertinentes pour des développements de grande taille, se basent plutôt sur les approches de type *assumption/commitment* (aussi appelées *rely/guarantee*). Ces approches apportent une compositionnalité naturelle en ce que chaque composant spécifie quelles propriétés il vérifie (*commitment–guarantee*), et quelles propriétés l'environnement doit vérifier (*assumption–rely*) pour que le composant fonctionne correctement. Une composition correcte découle donc de la vérification des propriétés d'environnement de composants avec les propriétés de fonctionnement des autres composants. Bien qu'eventB décompose les modèles B plutôt que de les composer, sa modularité évoque néanmoins des liens similaires à ceux qui se retrouvent dans les mécanismes cités plus haut.
  S'ajoute à cela le problème du raffinement des compositions, mais des travaux [Qiw96] indiquent qu'il est possible d'apporter un raffinement du même type que celui de B (*forward refinement*) à ces approches de type *assumption/commitment*. Il pourrait donc être intéressant de voir de quelle manière ces différents formalismes peuvent interagir. Il s'agirait là d'un changement complet de perspective par rapport à B cependant, puisque le B classique est plutôt orienté « flot d'instructions ».
- La logique temporelle que nous avons choisie est expressive, mais il existe des variantes dont les particularités peuvent s'avérer tout aussi intéressantes pour faciliter l'expression de certaines classes de contraintes temporelles : des intervalles de temps infinis [ZWR95], l'ordre supérieur[ZGN99],… Des travaux montrent que toutes ces variantes (y compris DC^*) peuvent être exprimées à l'aide d'un formalisme unique [HJ04]. Étant donné que les méthodes d'assumption/commitment décrites plus haut sont relativement indépendantes des logiques choisies, il est donc possible de réaliser des avancées intéressantes liant plus finement les raffinements fonctionnel et temporel à ces méthodes.
  Il est à noter qu'un article de Siewe [SHZC04] va dans ce sens : il suggère pour le développement un formalisme de type assumption/commitment intégrant DC^*.

Du côté de la preuve de formules temporelles, nous avons également commencé une implémentation sous forme de plongement profond. Le but est de disposer dans un même outil d'un mécanisme de validation (la version en plongement léger) et d'un mécanisme de raisonnement sur le formalisme (le plongement profond). De cette manière le même outil sert à prouver la correction de son extension en plus de mettre à disposition celle-ci. L'autre intérêt d'un plongement profond est de donner une base de départ pour la mise en oeuvre d'une version proche du formalisme implémenté, comme par exemple la version plus générale de DC cité plus haut.

Des travaux sur l'implémentation de logiques modales dans Coq , réalisés par de Wind [dW01], illustrent d'ailleurs une approche graduelle de ce type, en plongement profond. Le plongement profond peut également s'avérer une base de départ pour un calcul des durées basé sur la logique d'intervalle signée [Ras02] ou sur une logique de voisinage [ZH98].

Bibliographie

[ADE01] ADER/LORIA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, ADER/LORIA Nancy – France, June 2001. ADER/LORIA.

[AFA03] IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[CAS] CASL. `http://www.cofi.info/CASL.html`.

[dW01] Paulien de Wind. Modal logic in Coq. Master's thesis, Vrije Universiteit, Amsterdam, 2001. Supervised by dr. F. van Raamsdonk.

[HJ04] Jifeng He and Naiyong Jin. Integrating variants of DC. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004, First International Colloquium, Guiyang, China, September 20-24, 2004, Revised Selected Papers*, volume 3407 of *Lecture Notes in Computer Science*, pages 14–34. Springer-Verlag, 2004. ISBN :3-540-25304-1.

[HJMO03] Ahmed Hammad, Jacques Julliand, H. Mountassir, and Dieudonné Okalas Ossami. Expression en B et raffinement des systèmes réactifs temps réel. In AFADL2003 [AFA03], pages 211–226.

[Ini04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[LD96] Kevin Lano and Jeremy Dick. Development of concurrent systems in B AMN. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, july 1996.

[LMA+01] Franc Ledoux, Jean-Marc Mota, Agnès Arnould, Catherine Dubois, Pascale Le Gall, and Yves Bertrand. Spécification formelle du chanfreinage. In *AFADL'2001* [ADE01], pages 157–171.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6 :319–340, 1976.

[Qiw96] Xu Qiwen. On compositionality in refining concurrent systems. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, Berlin, Germany, 1996.

[Ras02] Thomas Marthedal Rasmussen. *Interval Logic - Proof Theory and Theorem Proving*. PhD thesis, Informatics and Mathematical Modeling, Technical University of Denmark, january 2002.

[SH01] François Siewe and Dan Van Hung. Deriving real-time programs from duration calculus specifications. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume LNCS 2144, pages 92–97, Livingston-Edinburgh, Scotland, september 2001. Springer-Verlag. (Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000).

[SHZC04] Francois Siewe, Dang Van Hung, Hussein Zedan, and Antonio Cau. A formal design technique for real-time embedded systems development using duration calculus. In A. Weitzenfeld and A. Barrera, editors, *1st IEEE Latin American Robotics Symposium*, pages 60–65, Mexico City, Mexico, october 2004.

[ZGN99] Chaochen Zhou, Dimitar P. Guelev, and Z. Naijun. A higher-order duration calculus. In *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).

[ZH98] Chaochen Zhou and Michael R. Hansen. An adequate first order interval logic. *Lecture Notes in Computer Science*, 1536 :584–608, 1998.

[ZWR95] Chaochen Zhou, J. Wang, and Anders P. Ravn. A duration calculus with infinite intervals. In H. Reichel, editor, *Fundamentals of Computing Theory*, volume 965 of LNCS, pages 16–41. Springer-Verlag, Lübeck, Germany, 1995.

Annexe A

Preuves de la sémantique temporelle de B en WDC*

Sommaire

Nous avons décrit au chapitre 4 la sémantique en DC* des substitutions. Nous allons décrire ici comment l'obtenir à partir d'une définition de la sémantique des substitutions en WDC*.

Nous reprenons cette approche de Siewe[SH01] : nous définissons des variables représentant les états de fonctionnement des processus composant le programme pour spécifier les évolutions possibles de celui-ci. Ces variables booléennes représentent le fait qu'un processus soit en fonctionnement, en ordonnancement ou bien autorise tout autre processus à fonctionner (lorsqu'il attend une condition particulière, par exemple). Les différentes formules obtenues reflètent le comportement des différentes substitutions : il s'agit d'une description des entrelacements des différentes exécutions possibles du programme. Cet entrelacement est caché par l'expressivité de la logique WDC^*. Cependant, si Siewe [SH01] montre effectivement le côté *temporel* de l'entrelacement, son côté *fonctionnel* est beaucoup moins visible. Celui-ci n'apparaît que dans les courts exemples présentés pour chaque élément du langage utilisé par Siewe [SH01]. L'exemple le plus visible est la définition du parallélisme qui montre les différents entrelacements d'un programme, mais pas comment les obtenir.

Nous allons donc en premier lieu introduire les définitions que nous utiliserons pour exprimer l'équivalent en WDC* des substitutions, puis nous démontrons que le calcul de plus faible précondition de B est conservé par cette sémantique. Nous déduisons ensuite de la sémantique en WDC* des substitutions leur sémantique en DC*. Nous terminons sur la preuve de validité de l'expansion du symbole de concurrence $|||$.

A.1 Définitions préliminaires

Le cadre proposé par Siewe [SH01] est celui de processus s'exécutant dans un « ordinateur abstrait » capable d'ordonnancer les processus ou de les mettre en attente. Cet ordinateur sait prendre en compte le fait qu'un processus est en attente réactive ou exécute un délai, et donc peut exécuter un autre processus à la place.

Ces processus sont modélisés par des états représentant le fait qu'ils soient en fonctionnement ou en attente. Des hypothèses sont ajoutées pour modéliser le comportement du système (deux processus ne peuvent pas fonctionner en même temps, il y a au moins un processus qui fonctionne,...). Finalement, la sémantique en WDC* des substitutions est définie à partir de ces variables, dans un cadre terminant et dans un cadre non-terminant. Un programme pourra donc par exemple traduit en une formule WDC* qui représente la manière dont il peut fonctionner.

A.1.1 Variables temporelles

Soient :

- $\mathcal{WO}(P_i)$ les variables modifiées par le processus P_i
- R_i et W_i des variables booléennes indiquant si P_i est en fonctionnement (respectivement fonctionne ou est déplacé dans la file d'attente) au micro-point de temps considéré. On a en outre $R_i \Rightarrow W_i$.
- Nous notons u_i les éléments de $\mathcal{WO}(P_i)$. Notons que nous manipulons indifféremment des variables seules ou des ensembles de variables par souci de lisibilité.
- SCH_i la variable booléenne indiquant que P_i n'est pas en fonctionnement et attend d'être traité. On a $SCH_i \equiv \lceil\!\lceil W_i \wedge \neg R_i \rceil^-$
- END_i l'état du processus P_i qui a terminé son exécution. On a $END_i \equiv \lceil\!\lceil \neg W_i \rceil\!\rceil^- \wedge \ell = 0$

Dans la sémantique en DC* des substitutions B, la notion de processus n'apparaît pas explicitement, sauf lors de l'utilisation de l'opérateur de concurrence. Mais même dans ce cas, les variables qui décrivent les états des processus n'apparaissent pas.

Dans la sémantique en WDC*, chaque partie de programme qui est délimitée par un opérateur, ou qui est une séquence délimitée par une mise en concurrence d'autres processus, peut être considéré comme un processus à part. Par exemple, dans la substitution de la forme suivante : $S_1; (S_2 \,|||\, S_3)$, chacun des S_i est un séquencement qui ne contient aucune substitution $|||$. Il est possible de considérer alors chacun des S_i comme un processus indépendant. Il est également possible de considérer S_1 et S_2 comme le même processus. Que l'une ou l'autre approche soit considérée, cela ne change rien à la manière dont la mise en oeuvre en WDC* fonctionne.

Ces considérations pourraient néanmoins avoir une incidence sur la génération de code, dans la manière dont sont créés les processus concurrents. Par exemple, le cas où S_1 et S_2 sont le même processus peuvent correspondre à un appel unix `fork` où S_3 est dérivé à la fin de S_1. Comme le propos n'est pas ici le problème de la génération de code, nous refermons ici la parenthèse.

A.1.2 Hypothèses de fonctionnement

Le système de processus est ensuite doté des hypothèses de fonctionnement suivantes :

– L'ordonnancement d'un processus ne prend aucun temps :

$$ASS \equiv SCH_i \Rightarrow \ell = 0$$

La formule signifie que les opérations d'ordonnancement se font en micro-temps (le temps que met un processeur à ordonnancer est suffisamment petit en regard des délais explicites du programme qu'il exécute, par exemple)

– Un seul processus peut fonctionner à la fois (l'exécution des processus est entrelacée) :

$$COND_0 \equiv \llbracket \bigwedge_{i,j,i \neq j} \neg (R_i \wedge R_j) \rrbracket^+$$

À tout moment, il est impossible que deux processus fonctionnent en même temps (*attention*, le fonctionnement n'inclut pas l'attente explicite, voir la sémantique temporelle des substitutions plus loin)

– S'il y a des processus en attente, alors il y a un processus en fonctionnement :

$$COND_1 \equiv \llbracket (\bigvee_i W_i) \Rightarrow (\bigvee_i R_i) \rrbracket^+$$

À tout moment, s'il y a au moins un processus en attente, alors il y a au moins un processus en fonctionnement. Cette formule garantit que les ordonnancements s'enchaînent sans « temps mort »

– Seul le processus P_i peut modifier la variable u_i :

$$COND_2 \equiv \llbracket \neg R_i \rrbracket \Rightarrow \neg (\mathbf{true}^\frown \lceil u_i = a \rceil^{0\frown} \mathbf{true}^\frown \lceil u_i = c \wedge a \neq c \rceil^{0\frown} \mathbf{true})$$

Cette formule introduit l'atomicité : pendant le temps qu'un processus fonctionne, ses variables ne peuvent pas être modifiées (sauf par le processus lui-même, la définition n'est pas incompatible avec la sémantique de l'affectation)

– Lorsqu'un programme (ensemble de processus) est terminé, cela signifie que tous ses processus sont terminés :

$$COND_3 \equiv \neg (\mathbf{true}^\frown \llbracket \bigwedge_i W_i \rrbracket^+)$$

Cette contrainte sert principalement à indiquer où se trouve la fin de l'exécution des processus, si fin il y a. La contrainte qui permet de réaliser cela est END_i, il faudra donc prendre soin de positionner celle-ci à la fin de chaque processus.

Maintenant que nous avons défini le cadre de fonctionnement, nous exprimons les formules en WDC* qui correspondent à chacune des substitutions.

A.2 Sémantique en WDC^* des substitutions

Sauf pour les substitutions qui diffèrent, la sémantique est la même que celle de Siewe [SH01].

1. Dans le cas terminant, le processus fonctionne, et à la fin du fonctionnement la variable x a pris la valeur E_0 telle qu'elle était au moment où l'exécution a débuté. Toute l'opération a pris un temps nul (hypothèse du vrai synchronisme). Dans le cas non terminant, il est impossible qu'une affectation ne termine pas.

$$\begin{array}{lcl} \mathcal{M}_{fin}(x := E) & \equiv & \lceil R_i \rceil^1 \frown \lceil x = E_0 \rceil^0 \wedge \ell = 0 \\ \mathcal{M}_{inf}(x := E) & \equiv & \textbf{false} \end{array}$$

2. Dans le cas terminant, le processus est en fonctionnement et ses variables ne changent pas de valeur. Il est impossible que le *skip* ne termine pas.

$$\begin{array}{lcl} \mathcal{M}_{fin}(skip) & \equiv & \mathcal{M}_{fin}(x := x) \text{ (x sont les variables modifiées par le processus)} \\ \mathcal{M}_{inf}(skip) & \equiv & \textbf{false} \end{array}$$

3. Dans le cas terminant, le processus n'est pas en fonctionnement (ce qui permet aux autres processus de fonctionner) et n'est pas en attente (ce qui permet d'éviter de le considérer pour traitement tant qu'il attend). Il laisse ensuite la main à l'ordonnanceur (d'où l'introduction du *Unit*). Le tout aura pris d unités de temps. Bien entendu, une attente durant un délai fixe termine forcément.

$$\begin{array}{lcl} \mathcal{M}_{fin}(\text{delay } d) & \equiv & [\![\neg W_i]\!]^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = d \\ \mathcal{M}_{inf}(\text{delay } d) & \equiv & \textbf{false} \end{array}$$

4. Cette substitution, qui n'apparaît pas dans l'article de Siewe [SH01], est un peu particulière du point de vue dynamique. En B, une substitution avec une précondition n'offre aucune garantie dynamique. Elle sert simplement à spécifier statiquement les états autorisés pour l'utilisation de cette substitution. Mais elle n'interdit en aucun cas d'exécuter cette substitution, à la différence de la garde. Si la précondition est fausse, il n'y a plus aucune garantie sur le bon fonctionnement de la substitution (style de programmation offensif). Nous considérons donc que dynamiquement, la précondition de B n'a pas d'impact sur les intervalles.

$$\begin{array}{lcl} \mathcal{M}_{fin}(g|S) & \equiv & \mathcal{M}_{fin}(S) \\ \mathcal{M}_{inf}(g|S) & \equiv & \mathcal{M}_{inf}(S) \end{array}$$

5. De la même manière que pour la précondition plus haut, nous devons supposer que la condition g est vraie au début de l'exécution du processus pour pouvoir exécuter S.

$$\begin{array}{lcl} \mathcal{M}_{fin}(g \Longrightarrow S) & \equiv & \lceil g \rceil^0 \frown \mathcal{M}_{fin}(S) \\ \mathcal{M}_{inf}(g \Longrightarrow S) & \equiv & \lceil g \rceil^0 \frown \mathcal{M}_{inf}(S) \end{array}$$

6. Deux instructions sont exécutées l'une à la suite de l'autre, avec l'opportunité de faire un ordonnancement après la fin de la première instruction pour permettre aux autres processus de fonctionner s'il est besoin. C'est la présence de cette phase d'ordonnancement au sein d'une séquence qui permet l'entrelacement des processus. Dans le cas où la séquence $S;T$ puisse ne pas terminer, plusieurs cas peuvent se présenter :
   – Soit S ne termine pas

– Soit un élément de S ne termine pas (et donc seul un préfixe de l'exécution de S est valide)
– Soit S termine effectivement mais c'est T qui ne termine pas.
$$\begin{array}{lcl} \mathcal{M}_{fin}(S;T) & \equiv & \mathcal{M}_{fin}(S)^\frown SCH_i{}^\frown \mathcal{M}_{fin}(T) \\ \mathcal{M}_{inf}(S;T) & \equiv & \mathcal{M}_{inf}(S) \vee PREF(\mathcal{M}_{fin}(S)^\frown SCH_i) \vee \mathcal{M}_{fin}(S)^\frown SCH_i{}^\frown \mathcal{M}_{inf}(T) \end{array}$$

7. L'exécution du programme peut prendre deux chemins différents.
$$\begin{array}{lcl} \mathcal{M}_{fin}(S[\!]T) & \equiv & \mathcal{M}_{fin}(S) \vee \mathcal{M}_{fin}(T) \\ \mathcal{M}_{inf}(S[\!]T) & \equiv & \mathcal{M}_{inf}(S) \vee \mathcal{M}_{inf}(T) \end{array}$$

8. La déclaration de variable locale n'a pas d'influence sur le comportement dynamique du processus. Il pourra arriver que nous omettions simplement les quantifications universelles lorsque cela ne pose pas de problème.
$$\begin{array}{lcl} \mathcal{M}_{fin}(@x.S) & \equiv & \exists x.\mathcal{M}_{fin}(S) \\ \mathcal{M}_{inf}(@x.S) & \equiv & \exists x.\mathcal{M}_{inf}(S) \end{array}$$

9. S'il y a terminaison, tous les processus sont ordonnancés au début, fonctionnent et se finissent. Dans ce cas, la fin de tous les processus est validée par END_i et $COND_3$.
S'il n'y a pas terminaison, alors d'un côté certains processus peuvent fonctionner indéfiniment, et les autres processus (ceux qui terminent) fonctionnent normalement et se finissent, mais doivent attendre indéfiniment les processus qui ne terminent pas.
$$\begin{array}{lcl} \mathcal{M}_{fin}(S_1 \,|||\, \ldots \,|||\, S_n) & \equiv & \bigwedge_{1\leq i\leq n}\left(SCH_i{}^\frown \mathcal{M}_{fin}(S_i)^\frown END_i\right) \\ & & \wedge COND_0 \wedge COND_1 \wedge COND_2 \wedge COND_3 \\ \mathcal{M}_{inf}(S_1 \,|||\, \ldots \,|||\, S_n) & \equiv & \bigvee_{\emptyset\neq J\subseteq\{1,\ldots,n\}}\left(\left(\bigwedge_{i\in J} SCH_i{}^\frown \mathcal{M}_{inf}(S_i)\right) \wedge \right. \\ & & \left.\left(\bigwedge_{i\in\{1,\ldots,n\}\setminus J} PREF(SCH_i{}^\frown \mathcal{M}_{fin}(S_i)^\frown [\![\neg W_i]\!])\right)\right) \\ & & \wedge COND_0 \wedge COND_1 \wedge COND_2 \wedge COND_3 \end{array}$$

10. De la même manière que le délai, l'attente réactive laisse le champ libre aux autres processus tant que la condition n'est pas vérifiée. Ensuite, dans le cas où l'attente se termine, la garde est vérifiée à la fin de l'intervalle, et dans le cas où l'attente ne se termine pas, elle continue indéfiniment.
$$\begin{array}{lcl} \mathcal{M}_{fin}(\text{await } g) & \equiv & [\![\neg W_i \wedge \neg g\rceil^{-\frown}\lceil g\rceil^0 \\ \mathcal{M}_{inf}(\text{await } g) & \equiv & [\![\neg W_i \wedge \neg g\rceil \end{array}$$

11. À chaque tour de boucle, la garde est vérifiée, et le corps de boucle exécuté. À la fin du tour de boucle, une phase d'ordonnancement a lieu (comme pour la séquence). Dans le cas terminant, la garde finit par ne plus être vraie. Dans le cas non terminant :
– soit le corps de la boucle finit par ne plus terminer (par exemple une attente réactive peut finir par ne plus être satisfiable par les autres processus)
– soit un des éléments du corps de la boucle ne termine plus.
$$\begin{array}{lcl} \mathcal{M}_{fin}(\mathcal{W}(g,S,I)) & \equiv & (\lceil g\rceil^{0\frown}\mathcal{M}_{fin}(S)^\frown SCH_i)^{*\frown}\lceil\neg g\rceil^0 \\ \mathcal{M}_{inf}(\mathcal{W}(g,S,I)) & \equiv & (\lceil g\rceil^{0\frown}\mathcal{M}_{fin}(S)^\frown SCH_i)^{*\frown}\lceil g\rceil^0 \\ & & {}^\frown(\mathcal{M}_{inf}(S) \vee PREF(\mathcal{M}_{fin}(S)^\frown SCH_i)) \end{array}$$

Pour vérifier que les formules des tableaux 4.2 et 4.3 sont correctes, les implications suivantes doivent être valides :

– Si le programme S est exécuté dans l'état pour lequel on est capable d'établir une post-condition P (prouvé par les mécanismes B habituels), alors P sera effectivement vérifié après exécution de S :

$$\lceil [S]P \rceil \frown \mathcal{M}_{fin}(S) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(S) \frown \lceil P \rceil$$

De plus, le programme doit vérifier la formule de durées correspondante (l'opérateur Π est décrit au chapitre 2) :

$$\Pi(\lceil [S]P \rceil \frown \mathcal{M}_{fin}(S)) \Rightarrow_{DC^*} dur(S,P)$$

– Dans le cas non-terminant, seule la vérification de la formule de durées est nécessaire (en effet, P n'est jamais atteignable) :

$$\Pi(\lceil [S]P \rceil \frown \mathcal{M}_{inf}(S)) \Rightarrow_{DC^*} dur_\infty(S)$$

– Il faut pouvoir relier la terminaison temporelle d'une substitution avec sa sémantique en WDC^* :
  – $Term(S) \equiv \mathcal{M}_{inf}(S) \Leftrightarrow$ **false**
  – $Nterm(S) \equiv \mathcal{M}_{fin}(S) \Leftrightarrow$ **false**

  Il y aura donc des cas où la terminaison, ou la non-terminaison, seront particulièrement difficiles à prouver.

A.3 Preuves de la sémantique de durées

Beaucoup de substitutions de B sont similaires à des instructions du langage proposé par Siewe [SH01]. Ces instructions ont là une présentation sous forme de triplets de Hoare, et nous savons que le calcul de plus faible précondition induit ces triplets. Nous nous limiterons donc aux substitutions qui n'apparaissent pas dans le langage proposé par Siewe.

Choix borné

1. $\lceil [S [] T]P \rceil \frown \mathcal{M}_{fin}(S [] T) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(S [] T) \frown \lceil P \rceil$

PROOF:

1.1. ASSUME: $\lceil [S]P \rceil \frown \mathcal{M}_{fin}(S) \Rightarrow \mathcal{M}_{fin}(S) \frown \lceil P \rceil$

1.2. ASSUME: $\lceil [T]P \rceil \frown \mathcal{M}_{fin}(T) \Rightarrow \mathcal{M}_{fin}(T) \frown \lceil P \rceil$

1.3. $\lceil [S [] T]P \rceil \frown \mathcal{M}_{fin}(S [] T)$

1.4. $\lceil [S]P \wedge [T]P \rceil \frown (\mathcal{M}_{fin}(S) \vee \mathcal{M}_{fin}(T))$ (application de $[]$, déf. de $\mathcal{M}_{fin}(S [] T)$)

1.5. $(\lceil [S]P \wedge [T]P \rceil \frown \mathcal{M}_{fin}(S)) \vee (\lceil [S]P \wedge [T]P \rceil \frown \mathcal{M}_{fin}(T))$ (WDC*)

1.6. $(\mathcal{M}_{fin}(S) \frown \lceil P \rceil) \vee (\mathcal{M}_{fin}(T) \frown \lceil P \rceil)$ (WDC*, 1.1, 1.2)

1.7. $(\mathcal{M}_{fin}(S) \vee \mathcal{M}_{fin}(T)) \frown \lceil P \rceil$ (WDC*)

1.8. $\mathcal{M}_{fin}(S [] T) \frown \lceil P \rceil$ (WDC*)

1.9. Q.E.D.

□

2. $\Pi(\lceil [S [] T]P \rceil \frown \mathcal{M}_{fin}(S [] T)) \Rightarrow_{DC^*} dur(S [] T, P)$

PROOF:

2.1. $\Pi(\lceil [S [] T]P \rceil \frown \mathcal{M}_{fin}(S [] T))$

2.2. $\Pi(\lceil [S]P \wedge [T]P \rceil \frown (\mathcal{M}_{fin}(S) \vee \mathcal{M}_{fin}(T)))$ (WDC*)

2.3. $\Pi(\lceil [S]P \wedge [T]P \rceil \frown (\mathcal{M}_{fin}(S))) \vee \Pi(\lceil [S]P \wedge [T]P \rceil \frown (\mathcal{M}_{fin}(T)))$ (WDC*, déf. de Π)

2.4. $\Pi(\lceil [S]P \rceil \frown \mathcal{M}_{fin}(S)) \vee \Pi(\lceil [T]P \rceil \frown \mathcal{M}_{fin}(T))$

2.5. $\Pi(\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S))$
$\vee \Pi(\lceil [T]P \rceil ^\frown \mathcal{M}_{fin}(T))$
$\vee (\mathcal{M}_{fin}(S) \Leftrightarrow \textbf{false}) \Rightarrow \Pi((\lceil [T]P \rceil ^\frown \mathcal{M}_{fin}(T)))$
$\vee (\mathcal{M}_{fin}(S) \Leftrightarrow \textbf{false}) \Rightarrow \Pi((\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S)))$ (WDC*)
2.6. $dur(S,P) \vee dur(T,P) \vee (NTerm(S) \Rightarrow dur(T,P)) \vee (NTerm(T) \Rightarrow dur(S,P))$ (déf. de $dur(\_,\_), NTerm$)
2.7. Q.E.D.
$\square$

3. $\Pi(\lceil [S \| T]P \rceil ^\frown \mathcal{M}_{inf}(S \| T)) \Rightarrow_{DC^*} dur_\infty(S \| T)$
PROOF:
3.1. La preuve est similaire à celle de l'étape 2

4. $Term(S \| T) \equiv \mathcal{M}_{inf}(S \| T) \Leftrightarrow \textbf{false}$
PROOF:
4.1. $\mathcal{M}_{inf}(S \| T) \Leftrightarrow \textbf{false}$ (WDC*)
4.2. $(\mathcal{M}_{inf}(S) \vee \mathcal{M}_{inf}(T)) \Leftrightarrow \textbf{false}$ (WDC*)
4.3. $(\mathcal{M}_{inf}(S) \Rightarrow \textbf{false}) \wedge (\mathcal{M}_{inf}(T) \Rightarrow \textbf{false})$ (WDC*)
4.4. $(\mathcal{M}_{inf}(S) \Leftrightarrow \textbf{false}) \wedge (\mathcal{M}_{inf}(T) \Leftrightarrow \textbf{false})$ (WDC*)
4.5. $Term(S) \wedge Term(T)$ (WDC*)
4.6. Q.E.D.
$\square$

5. $Nterm(S \| T) \equiv \mathcal{M}_{fin}(S \| T) \Leftrightarrow \textbf{false}$
PROOF:
5.1. $Nterm$ et $\mathcal{M}_{fin}(\_)$ sont symétriques à $Term$ et $\mathcal{M}_{inf}()$, la preuve est similaire à celle de l'étape 4

6. Q.E.D.
$\square$

Garde

1. $\lceil [g \Longrightarrow S]P \rceil ^\frown \mathcal{M}_{fin}(g \Longrightarrow S) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(g \Longrightarrow S) ^\frown \lceil P \rceil$
PROOF:
1.1. ASSUME: $\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S) \Rightarrow \mathcal{M}_{fin}(S) ^\frown \lceil P \rceil$
1.2. $\lceil [g \Longrightarrow S]P \rceil ^\frown \mathcal{M}_{fin}(g \Longrightarrow S)$
1.3. $\lceil g \Rightarrow [S]P \rceil ^\frown \lceil g \rceil ^\frown \mathcal{M}_{fin}(S)$ (WDC*)
1.4. $\lceil (g \Rightarrow [S]P) \wedge g \rceil ^\frown \mathcal{M}_{fin}(S)$ (WDC*)
1.5. $\lceil g \wedge [S]P \rceil ^\frown \mathcal{M}_{fin}(S)$ (WDC*)
1.6. $\lceil g \rceil ^\frown \mathcal{M}_{fin}(S) ^\frown \lceil P \rceil$ (WDC*,1.1)
1.7. $\mathcal{M}_{fin}(g \Longrightarrow S) ^\frown \lceil P \rceil$ (WDC*)
1.8. Q.E.D.
$\square$

2. $\Pi(\lceil [g \Longrightarrow S]P \rceil ^\frown \mathcal{M}_{fin}(g \Longrightarrow S)) \Rightarrow_{DC^*} dur(g \Longrightarrow S, P)$
PROOF:
2.1. $\Pi(\lceil [g \Longrightarrow S]P \rceil ^\frown \mathcal{M}_{fin}(g \Longrightarrow S))$
2.2. $\Pi(\lceil g \wedge [S]P \rceil ^\frown \mathcal{M}_{fin}(S))$ (WDC*, déf. de $\mathcal{M}_{fin}(g \Longrightarrow S)$)
2.3. $\Pi(\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S))$ (WDC*)
2.4. $dur(S,P)$
2.5. Q.E.D.
$\square$

3. $\Pi(\lceil [g \Longrightarrow S]P \rceil ^\frown \mathcal{M}_{inf}(g \Longrightarrow S)) \Rightarrow_{DC^*} dur_\infty(g \Longrightarrow S)$
PROOF:
3.1. Preuve similaire à celle de l'étape 2
3.2. Q.E.D.
$\square$

4. $Term(g \Longrightarrow S) \equiv \mathcal{M}_{inf}(g \Longrightarrow S) \Leftrightarrow \textbf{false}$
PROOF:
4.1. $\mathcal{M}_{inf}(g \Longrightarrow S) \Leftrightarrow \textbf{false}$
4.2. $(\lceil g \rceil ^\frown \mathcal{M}_{inf}(S)) \Rightarrow \textbf{false}$ (définition, WDC*)
4.3. $(\neg \lceil g \rceil \vee \neg \mathcal{M}_{inf}(S))$ (WDC*, propriétés de $\lceil \_ \rceil$)
4.4. $\lceil g \rceil \Rightarrow (\mathcal{M}_{inf}(S) \Leftrightarrow \textbf{false})$ (WDC*)
4.5. $g \Rightarrow Term(S)$ (passage au calcul des prédicats simple)
4.6. Q.E.D.
$\square$

5. $Nterm(g \Longrightarrow S) \equiv \mathcal{M}_{fin}(g \Longrightarrow S) \Leftrightarrow \textbf{false}$
PROOF:
5.1. $Nterm$ et $\mathcal{M}_{fin}(\_)$ sont symétriques à $Term$ et $\mathcal{M}_{inf}()$, la preuve est similaire à celle de l'étape 4
5.2. Q.E.D.
$\square$

6. Q.E.D.
$\square$

Nous pouvons remarque que la définition de Siewe pour l'alternative `if then else` découle naturellement de la définition pour le choix borné et la garde que nous venons de démontrer.

Précondition

1. $\lceil [g|S]P \rceil ^\frown \mathcal{M}_{fin}(g|S) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(g|S) ^\frown \lceil P \rceil$

PROOF:
- 1.1. ASSUME: $\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S) \Rightarrow \mathcal{M}_{fin}(S) ^\frown \lceil P \rceil$
- 1.2. $\lceil [g|S]P \rceil ^\frown \mathcal{M}_{fin}(g|S)$
- 1.3. $\lceil g \wedge [S]P \rceil ^\frown \mathcal{M}_{fin}(S)$ (précondition,WDC*)
- 1.4. $\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S)$ (WDC*)
- 1.5. $\mathcal{M}_{fin}(S) ^\frown \lceil P \rceil$ (WDC*,1.1)
- 1.6. Q.E.D.

□

2. $\Pi(\lceil [g|S]P \rceil ^\frown \mathcal{M}_{fin}(g|S)) \Rightarrow_{DC^*} dur(g|S,P)$

PROOF:
- 2.1. $\Pi(\lceil [g|S]P \rceil ^\frown \mathcal{M}_{fin}(g|S))$
- 2.2. $\Pi(\lceil g \wedge [S]P \rceil ^\frown \mathcal{M}_{fin}(S))$ (WDC*)
- 2.3. $dur(S,P)$ (WDC*)
- 2.4. Q.E.D.

□

3. $\Pi(\lceil [g|S]P \rceil ^\frown \mathcal{M}_{inf}(g|S)) \Rightarrow_{DC^*} dur_\infty(g|S)$

PROOF:
- 3.1. Preuve similaire à celle de l'étape 2
- 3.2. Q.E.D.

□

4. $Term(g|S) \equiv \mathcal{M}_{inf}(g|S) \Leftrightarrow$ **false**

PROOF:
- 4.1. $\mathcal{M}_{inf}(g|S) \Leftrightarrow$ **false**
- 4.2. $\mathcal{M}_{inf}(S) \Leftrightarrow$ **false** (WDC*, définitions)
- 4.3. $Term(S)$ (définition)
- 4.4. Q.E.D.

□

5. $Nterm(g|S) \equiv \mathcal{M}_{fin}(g|S) \Leftrightarrow$ **false**

PROOF:
- 5.1. $Nterm$ et $\mathcal{M}_{fin}(\_)$ sont symétriques à $Term$ et $\mathcal{M}_{inf}()$, la preuve est similaire à celle de l'étape 4
- 5.2. Q.E.D.

□

6. Q.E.D.

□

Choix non borné

1. $\lceil [@x.S]P \rceil ^\frown \mathcal{M}_{fin}(@x.S) \Rightarrow_{WDC^*} \mathcal{M}_{fin}(@x.S) ^\frown \lceil P \rceil$

PROOF:
- 1.1. ASSUME: $\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S) \Rightarrow \mathcal{M}_{fin}(S) ^\frown \lceil P \rceil$
- 1.2. $\lceil [@x.S]P \rceil ^\frown \mathcal{M}_{fin}(@x.S)$
- 1.3. $\lceil [\forall x.[S]P \rceil ^\frown \exists x.\mathcal{M}_{fin}(S)$ (définitions)
- 1.4. $\lceil [S]P \rceil ^\frown \exists y.(\mathcal{M}_{fin}(S)[y/x])$ (instanciation de x, renommage du second x en y)
- 1.5. $\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S)$ (instanciation de y par x)
- 1.6. $\mathcal{M}_{fin}(S) ^\frown \lceil P \rceil$ (WDC*, 1.1)
- 1.7. $\exists x.(\mathcal{M}_{fin}(S) ^\frown \lceil P \rceil)$ (existenciation)
- 1.8. Q.E.D.

□

2. $\Pi(\lceil [@x.S]P \rceil ^\frown \mathcal{M}_{fin}(@x.S)) \Rightarrow_{DC^*} dur(@x.S,P)$

PROOF:
- 2.1. $\Pi(\lceil [@x.S]P \rceil ^\frown \mathcal{M}_{fin}(@x.S))$
- 2.2. $\Pi(\lceil [S]P \rceil ^\frown \mathcal{M}_{fin}(S))$ (instanciations)
- 2.3. $\Pi(\mathcal{M}_{fin}(S) ^\frown \lceil P \rceil)$ (WDC*)
- 2.4. $\Pi(\exists x.(\mathcal{M}_{fin}(S) ^\frown \lceil P \rceil))$ (existenciation)
- 2.5. $\exists x.dur(S,P)$ (déf. de Π)
- 2.6. Q.E.D.

□

3. $\Pi(\lceil [@x.S]P \rceil ^\frown \mathcal{M}_{inf}(@x.S)) \Rightarrow_{DC^*} dur_\infty(@x.S)$

PROOF:

3.1. Preuve similaire à celle de l'étape 2
3.2. Q.E.D.
$\Box$

4. $Term(@x.S) \equiv \mathcal{M}_{inf}(@x.S) \Leftrightarrow \textbf{false}$

PROOF:
4.1. $\mathcal{M}_{inf}(@x.S) \Leftrightarrow \textbf{false}$
4.2. $\exists x.\mathcal{M}_{inf}(S) \Rightarrow \textbf{false}$ (définitions,WDC*)
4.3. $\forall x.(\mathcal{M}_{inf}(S) \Rightarrow \textbf{false})$ (WDC*)
4.4. $\forall x.Term(S)$ (WDC*, définitions)
4.5. Q.E.D.
$\Box$

5. $Nterm(@x.S) \equiv \mathcal{M}_{fin}(@x.S) \Leftrightarrow \textbf{false}$

PROOF:
5.1. $Nterm$ et $\mathcal{M}_{fin}(\_)$ sont symétriques à $Term$ et $\mathcal{M}_{inf}()$, la preuve est similaire à celle de l'étape 4
5.2. Q.E.D.
$\Box$

6. Q.E.D.
$\Box$

A.4 Preuve de la séquentialisation

A.4.1 Lemmes préliminaires

Nous introduisons dans cette section quelques lemmes utilisés dans les preuves des sections suivantes.

Lemme A.1 (Séquentialisation) *Deux processus réduits à une phase (une opération atomique) ordonnançables en cours d'exécution sont équivalents à une séquence où soit le premier processus s'exécute et l'autre attend, puis l'autre s'exécute, soit l'inverse :*

$$\begin{pmatrix} \wedge\, SCH_1 \frown \lceil R_1 \rceil^1 \frown SCH_1 \\ \wedge\, SCH_2 \frown \lceil R_2 \rceil^1 \frown SCH_2 \end{pmatrix} \Leftrightarrow \begin{pmatrix} \vee\, (\lceil\!\lceil W_1 \wedge \neg R_1 \rceil \wedge \lceil R_2 \rceil^1) \frown (\lceil R_1 \rceil^1 \wedge \lceil\!\lceil W_2 \wedge \neg R_2 \rceil) \\ \vee\, (\lceil R_1 \rceil^1 \wedge \lceil\!\lceil W_2 \wedge \neg R_2 \rceil) \frown (\lceil\!\lceil W_1 \wedge \neg R_1 \rceil \wedge \lceil R_2 \rceil^1) \end{pmatrix}$$

1. CASE: $\Rightarrow$

PROOF:

PROVE: $\begin{array}{l} \vee\, (\lceil\!\lceil W_1 \wedge \neg R_1 \rceil \wedge \lceil R_2 \rceil^1) \frown (\lceil R_1 \rceil^1 \wedge \lceil\!\lceil W_2 \wedge \neg R_2 \rceil) \\ \vee\, (\lceil R_1 \rceil^1 \wedge \lceil\!\lceil W_2 \wedge \neg R_2 \rceil) \frown (\lceil\!\lceil W_1 \wedge \neg R_1 \rceil \wedge \lceil R_2 \rceil^1) \end{array}$

1.1. ASSUME: $\begin{array}{l} \wedge\, SCH_1 \frown \lceil R_1 \rceil^1 \frown SCH_1 \\ \wedge\, SCH_2 \frown \lceil R_2 \rceil^1 \frown SCH_2 \end{array}$

1.2. $\begin{array}{l} \wedge\, \lceil\!\lceil W_1 \wedge \neg R_1 \rceil^- \frown \lceil R_1 \rceil^1 \frown \lceil\!\lceil W_1 \wedge \neg R_1 \rceil^- \\ \wedge\, \lceil\!\lceil W_2 \wedge \neg R_2 \rceil^- \frown \lceil R_2 \rceil^1 \frown \lceil\!\lceil W_2 \wedge \neg R_2 \rceil^- \end{array}$

(définition de SCH_i)

1.3. $\begin{array}{l} \vee \begin{cases} \wedge\, \lceil\!\lceil W_1 \wedge \neg R_1 \rceil \frown \lceil R_1 \rceil^1 \frown \lceil\!\lceil W_1 \wedge \neg R_1 \rceil \\ \wedge\, \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \frown \lceil R_2 \rceil^1 \frown \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \end{cases} \\ \vee \begin{cases} \wedge\, \lceil\rceil \frown \lceil R_1 \rceil^1 \frown \lceil\!\lceil W_1 \wedge \neg R_1 \rceil \\ \wedge\, \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \frown \lceil R_2 \rceil^1 \frown \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \end{cases} \\ \vee\, \ldots \\ \vee \begin{cases} \wedge\, \lceil\rceil \frown \lceil R_1 \rceil^1 \frown \lceil\rceil \\ \wedge\, \lceil\rceil \frown \lceil R_2 \rceil^1 \frown \lceil\rceil \end{cases} \end{array}$

(définition de $\lceil\!\lceil P \rceil^-$, distributivité du $\vee$)

1.4. $\begin{array}{l} \vee \begin{cases} \wedge\, \lceil\!\lceil W_1 \wedge \neg R_1 \rceil \frown \lceil R_1 \rceil^1 \frown \lceil\rceil \\ \wedge\, \lceil\rceil \frown \lceil R_2 \rceil^1 \frown \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \end{cases} \\ \vee \begin{cases} \wedge\, \lceil\rceil \frown \lceil R_1 \rceil^1 \frown \lceil\!\lceil W_1 \wedge \neg R_1 \rceil \\ \wedge\, \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \frown \lceil R_2 \rceil^1 \frown \lceil\rceil \end{cases} \end{array}$

($COND_0$,$COND_1$)

1.5. $\begin{array}{l} \vee \begin{cases} \wedge\, \lceil\!\lceil W_1 \wedge \neg R_1 \rceil \frown \lceil R_1 \rceil^1 \\ \wedge\, \lceil R_2 \rceil^1 \frown \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \end{cases} \\ \vee \begin{cases} \wedge\, \lceil R_1 \rceil^1 \frown \lceil\!\lceil W_1 \wedge \neg R_1 \rceil \\ \wedge\, \lceil\!\lceil W_2 \wedge \neg R_2 \rceil \frown \lceil R_2 \rceil^1 \end{cases} \end{array}$

(WDC^*)

1.6. $\vee ([\![W_1 \wedge \neg R_1]\!] \wedge \lceil R_2 \rceil^1)^\frown(\lceil R_1 \rceil^1 \wedge [\![W_2 \wedge \neg R_2]\!])$
$\vee (\lceil R_1 \rceil^1 \wedge [\![W_2 \wedge \neg R_2]\!])^\frown([\![W_1 \wedge \neg R_1]\!] \wedge \lceil R_2 \rceil^1)$

$(COND_0, COND_1, WDC^*)$

1.7. Q.E.D.

□

2. CASE: $\Leftarrow$

PROOF:

PROVE: $\wedge\, SCH_1{}^\frown\lceil R_1\rceil^1{}^\frown SCH_1$
$\wedge\, SCH_2{}^\frown\lceil R_2\rceil^1{}^\frown SCH_2$

2.1. ASSUME: $\vee ([\![W_1 \wedge \neg R_1]\!] \wedge \lceil R_2 \rceil^1)^\frown(\lceil R_1 \rceil^1 \wedge [\![W_2 \wedge \neg R_2]\!])$
$\vee (\lceil R_1 \rceil^1 \wedge [\![W_2 \wedge \neg R_2]\!])^\frown([\![W_1 \wedge \neg R_1]\!] \wedge \lceil R_2 \rceil^1)$

2.2. $$\vee \begin{cases} \wedge\, [\![W_1 \wedge \neg R_1]\!]^\frown\lceil R_1\rceil^1 \\ \wedge\, \lceil R_2\rceil^1{}^\frown[\![W_2 \wedge \neg R_2]\!] \end{cases} \quad \vee \begin{cases} \wedge\, \lceil R_1\rceil^1{}^\frown[\![W_1 \wedge \neg R_1]\!] \\ \wedge\, [\![W_2 \wedge \neg R_2]\!]^\frown\lceil R_2\rceil^1 \end{cases}$$

(WDC^*)

2.3. $\wedge\, [\![W_1 \wedge \neg R_1]\!]^{-\frown}\lceil R_1\rceil^1{}^\frown[\![W_1 \wedge \neg R_1]\!]^-$
$\wedge\, [\![W_2 \wedge \neg R_2]\!]^{-\frown}\lceil R_2\rceil^1{}^\frown[\![W_2 \wedge \neg R_2]\!]^-$

(Ajouts de $\lceil\rceil$ et de disjonctions, définition de $[\![P]\!]^-$)

2.4. $\wedge\, SCH_1{}^\frown\lceil R_1\rceil^1{}^\frown SCH_1$
$\wedge\, SCH_2{}^\frown\lceil R_2\rceil^1{}^\frown SCH_2$

2.5. Q.E.D.

□

3. Q.E.D.

□

A.4.2 Mise en parallèle

Nous prouvons ici que l'entrelacement présenté en section 4.4 est cohérent avec la sémantique en WDC* des substitutions. Nous reprenons l'ordre de présentation des tableaux pour ces preuves.

Celles-ci se font par récurrence sur la structure des substitutions considérées.

Substitutions atomiques

1. PROVE: $\lceil((S_a;S) \,|||\, (T_b;T))P\rceil^0{}^\frown\mathcal{M}_{fin}((S_a;S) \,|||\, (T_b;T)) \Rightarrow \mathcal{M}_{fin}((S_a;S) \,|||\, (T_b;T))^\frown\lceil P\rceil^0$
avec S_a, T_b des affectations, skip, ou attentes réactives

PROOF:

1.1. ASSUME: $skip \equiv (x := x)$

1.2. ASSUME: $\lceil(S \,|||\, (T_a;T))P\rceil^0{}^\frown\mathcal{M}_{fin}(S \,|||\, (T_a;T)) \Rightarrow \mathcal{M}_{fin}(S \,|||\, (T_a;T))^\frown\lceil P\rceil^0$

1.3. ASSUME: $\lceil((S_a;S) \,|||\, T)P\rceil^0{}^\frown\mathcal{M}_{fin}((S_a;S) \,|||\, T) \Rightarrow \mathcal{M}_{fin}((S_a;S) \,|||\, T)^\frown\lceil P\rceil^0$

1.4. CASE: $S_a \equiv x := A, T_b \equiv y := B$

1.4.1. $\lceil[(x := A;S) \,|||\, (y := B;T)]P\rceil^\frown\mathcal{M}_{fin}((x := A;S) \,|||\, (y := B;T))$

(hypothèses)

1.4.2. $$\begin{bmatrix} \wedge\, [x := A;(S \,|||\, (T_b;T))]P \\ \wedge\, [y := B;((S_a;S) \,|||\, T)]P \end{bmatrix}^0{}^\frown\mathcal{M}_{fin}((x := A;S) \,|||\, (y := B;T))$$

(définition)

1.4.3. $$\begin{bmatrix} \wedge\, [x := A;(S \,|||\, (T_b;T))]P \\ \wedge\, [y := B;((S_a;S) \,|||\, T)]P \end{bmatrix}^0{}^\frown\begin{pmatrix} \wedge\, SCH_1{}^\frown\mathcal{M}_{fin}(x := A)^\frown SCH_1{}^\frown\mathcal{M}_{fin}(S)^\frown END_1 \\ \wedge\, SCH_2{}^\frown\mathcal{M}_{fin}(y := B)^\frown SCH_2{}^\frown\mathcal{M}_{fin}(T)^\frown END_2 \end{pmatrix}$$

(définition)

1.4.4. $$\begin{bmatrix} \wedge\, [x := A;(S \,|||\, (T_b;T))]P \\ \wedge\, [y := B;((S_a;S) \,|||\, T)]P \end{bmatrix}^0{}^\frown\begin{pmatrix} \wedge\, SCH_1{}^\frown(\lceil R_1\rceil^1{}^\frown\lceil x = A_0\rceil^0 \wedge \ell = 0)^\frown SCH_1{}^\frown\mathcal{M}_{fin}(S)^\frown END_1 \\ \wedge\, SCH_2{}^\frown(\lceil R_2\rceil^1{}^\frown\lceil y = B_0\rceil^0 \wedge \ell = 0)^\frown SCH_2{}^\frown\mathcal{M}_{fin}(T)^\frown END_2 \end{pmatrix}$$

(définition)

1.4.5. $$\begin{bmatrix} \wedge\, [x := A;(S \,|||\, (T_b;T))]P \\ \wedge\, [y := B;((S_a;S) \,|||\, T)]P \end{bmatrix}^0{}^\frown\begin{pmatrix} \vee\, (SCH_1 \wedge \mathcal{M}_{fin}(y := B))^\frown\mathcal{M}_{fin}((S_a;S) \,|||\, T) \\ \vee\, (SCH_2 \wedge \mathcal{M}_{fin}(x := A))^\frown\mathcal{M}_{fin}(S \,|||\, (T_b;T)) \end{pmatrix}$$

(lemme de séquentialisation)

1.4.6. $$\vee \begin{bmatrix} \wedge\, ([S \,|||\, (T_b;T)]P)[x\backslash E] \\ \wedge\, ([(S_a;S) \,|||\, T]P)[y\backslash B] \end{bmatrix}^0{}^\frown(SCH_1 \wedge \mathcal{M}_{fin}(y := B))^\frown\mathcal{M}_{fin}((S_a;S) \,|||\, T)$$
$$\vee \begin{bmatrix} \wedge\, ([S \,|||\, (T_b;T)]P)[x\backslash A] \\ \wedge\, ([(S_a;S) \,|||\, T]P)[y\backslash B] \end{bmatrix}^0{}^\frown(SCH_2 \wedge \mathcal{M}_{fin}(x := A))^\frown\mathcal{M}_{fin}(S \,|||\, (T_b;T))$$

(déf. substitution, distributivité de la disjonction)

1.4.7. $\begin{array}{l}\vee \lceil(([(S_a;S) \,|||\, T]P)[y\backslash B]\rceil^0 \frown (SCH_1 \wedge \mathcal{M}_{fin}(y := B)) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \\ \vee \lceil([S \,|||\, (T_b;T)]P)[x\backslash A]\rceil^0 \frown (SCH_2 \wedge \mathcal{M}_{fin}(x := A)) \frown \mathcal{M}_{fin}(S \,|||\, (T_b;T))\end{array}$

(WDC^*)

1.4.8. $\begin{array}{l}\vee (SCH_1 \wedge \mathcal{M}_{fin}(y := B)) \frown \lceil[(S_a;S) \,|||\, T]P\rceil^0 \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \\ \vee (SCH_2 \wedge \mathcal{M}_{fin}(x := A)) \frown \lceil[S \,|||\, (T_b;T)]P\rceil^0 \frown \mathcal{M}_{fin}(S \,|||\, (T_b;T))\end{array}$

($COND_2$)

1.4.9. $\begin{array}{l}\vee (SCH_1 \wedge \mathcal{M}_{fin}(y := B)) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \frown \lceil P\rceil^0 \\ \vee (SCH_2 \wedge \mathcal{M}_{fin}(x := A)) \frown \mathcal{M}_{fin}(S \,|||\, (T_b;T)) \frown \lceil P\rceil^0\end{array}$

(par hypothèse d'induction)

1.4.10. $\begin{pmatrix}\wedge\, SCH_1 \frown \mathcal{M}_{fin}(x := A) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown \mathcal{M}_{fin}(y := B) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{pmatrix} \frown \lceil P\rceil^0$

(séquentialisation, WDC^*)

1.4.11. $\mathcal{M}_{fin}((S_a;S) \,|||\, (T_b;T)) \frown \lceil P\rceil$

(déf. $(S_a;S)$ et $(T_b;T)$)

1.4.12. Q.E.D.

□

1.5. CASE: $S_a \equiv \text{await } a, T_b \equiv y := B$
(ainsi que le cas symétrique)

PROOF SKETCH: L'attente peut être achevée à tout moment : immédiatement parce qu'un autre processus a rendu la condition vraie, ou plus tard. Si c'est plus tard, alors toute substitution qui ne prend aucun temps s'exécutera immédiatement.

1.5.1. $\left\lceil\left[\begin{array}{l}\text{await } a; (S \,|||\, (T_b;T)) \\ \llbracket y := B; ((S_a;S) \,|||\, T)\end{array}\right] P\right\rceil^0 \frown \begin{pmatrix}\wedge\, SCH_1 \frown (\lceil\!\lceil \neg W_1 \wedge \neg a\rceil^- \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{pmatrix}$

(définitions)

1.5.2. $\left\lceil\begin{array}{l}\wedge\, [\text{await } a; (S \,|||\, (T_b;T))]P \\ \wedge\, [y := B; ((S_a;S) \,|||\, T)]P\end{array}\right\rceil^0 \frown \begin{pmatrix}\wedge \begin{array}{l}\vee\, SCH_1 \frown (\lceil\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \vee\, SCH_1 \frown (\lceil \neg W_1 \wedge \neg a\rceil^0 \frown \lceil \neg W_1 \wedge \neg a\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1\end{array} \\ \wedge\, SCH_2 \frown (\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{pmatrix}$

1.5.3. $\begin{array}{l}\vee \left\lceil\begin{array}{l}\wedge\, a \Rightarrow [S \,|||\, (T_b;T)]P \\ \wedge\, ([(S_a;S) \,|||\, T]P)[y\backslash B]\end{array}\right\rceil^0 \frown \begin{cases}\wedge\, SCH_1 \frown (\lceil\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{cases} \\ \vee \left\lceil\begin{array}{l}\wedge\, a \Rightarrow [S \,|||\, (T_b;T)]P \\ \wedge\, ([(S_a;S) \,|||\, T]P)[y\backslash B]\end{array}\right\rceil^0 \frown \begin{cases}\wedge\, SCH_1 \frown (\lceil \neg W_1 \wedge \neg a\rceil^0 \frown \lceil \neg W_1 \wedge \neg a\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{cases}\end{array}$

(WDC^*)

1.5.4. $\begin{array}{l}\vee \left\lceil\begin{array}{l}\wedge\, a \Rightarrow [S \,|||\, (T_b;T)]P \\ \wedge\, ([(S_a;S) \,|||\, T]P)[y\backslash B]\end{array}\right\rceil^0 \frown \begin{cases}\vee\, (\lceil a\rceil^0 \wedge SCH_2) \frown \mathcal{M}_{fin}(S \,|||\, (T_b;T)) \\ \vee\, ((\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge SCH_1) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T)\end{cases} \\ \vee \left\lceil\begin{array}{l}\wedge\, a \Rightarrow [S \,|||\, (T_b;T)]P \\ \wedge\, ([(S_a;S) \,|||\, T]P)[y\backslash B]\end{array}\right\rceil^0 \frown \begin{cases}\vee\, SCH_1 \frown ((SCH_2 \frown \lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge \lceil \neg W_1 \wedge \neg a\rceil) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \\ \vee\, ((\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge SCH_1) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T)\end{cases}\end{array}$

(séquencialisation, parallélisme dû à $\neg W_1$)

1.5.5. $\begin{array}{l}\vee\, (\lceil a \wedge a \Rightarrow [S \,|||\, (T_b;T)]P\rceil^0 \wedge SCH_2) \frown \mathcal{M}_{fin}(S \,|||\, (T_b;T)) \\ \vee\, \lceil([(S_a;S) \,|||\, T]P)[y\backslash B]\rceil^0 \frown ((\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge SCH_1) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \\ \vee\, \lceil([(S_a;S) \,|||\, T]P)[y\backslash B]\rceil^0 \frown SCH_1 \frown ((SCH_2 \frown \lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge \lceil \neg W_1 \wedge \neg a\rceil) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \\ \vee\, \lceil([(S_a;S) \,|||\, T]P)[y\backslash B]\rceil^0 \frown ((\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge SCH_1) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T)\end{array}$

(WDC^*)

1.5.6. $\begin{array}{l}\vee\, (\lceil a\rceil^0 \wedge SCH_2) \frown \mathcal{M}_{fin}(S \,|||\, (T_b;T)) \frown \lceil P\rceil^0 \\ \vee\, ((\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge SCH_1) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \frown \lceil P\rceil^0 \\ \vee\, SCH_1 \frown ((SCH_2 \frown \lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge \lceil \neg W_1 \wedge \neg a\rceil) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \frown \lceil P\rceil^0 \\ \vee\, ((\lceil R_2\rceil^1 \frown \lceil y = B_0\rceil^0 \wedge \ell = 0) \wedge SCH_1) \frown \mathcal{M}_{fin}((S_a;S) \,|||\, T) \frown \lceil P\rceil^0\end{array}$

(WDC^*)

1.5.7. $\mathcal{M}_{fin}((S_a;S) \,|||\, (T_b;T)) \frown \lceil P\rceil^0$

(séquencialisation, parallélisme du $\neg W_1$, WDC^*)

1.5.8. Q.E.D.

□

1.6. CASE: $S_a \equiv \text{await } a, T_b \equiv \text{await } b$

1.6.1. $\left\lceil\left[\begin{array}{l}\text{await } a; (S \,|||\, (T_b;T)) \\ \llbracket \text{await } b; ((S_a;S) \,|||\, T)\end{array}\right] P\right\rceil^0 \frown \begin{pmatrix}\wedge\, SCH_1 \frown (\lceil\!\lceil \neg W_1 \wedge \neg a\rceil^- \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil\!\lceil \neg W_2 \wedge \neg b\rceil^- \frown \lceil a\rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{pmatrix}$

(définitions)

1.6.2. $\left\lceil\begin{array}{l}\wedge\, a \Rightarrow [S \,|||\, (T_b;T)]P \\ \wedge\, b \Rightarrow [(S_a;S) \,|||\, T]P\end{array}\right\rceil^0 \frown \begin{array}{l}\vee \begin{cases}\wedge\, SCH_1 \frown (\lceil\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil\rceil \frown \lceil b\rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{cases} \\ \vee \begin{cases}\wedge\, SCH_1 \frown (\lceil \neg W_1 \wedge \neg a\rceil^0 \frown \lceil \neg W_1 \wedge \neg a\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil\rceil \frown \lceil b\rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{cases} \\ \vee \begin{cases}\wedge\, SCH_1 \frown (\lceil\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil \neg W_2 \wedge \neg b\rceil^0 \frown \lceil \neg W_2 \wedge \neg b\rceil \frown \lceil b\rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{cases} \\ \vee \begin{cases}\wedge\, SCH_1 \frown (\lceil \neg W_1 \wedge \neg a\rceil^0 \frown \lceil \neg W_1 \wedge \neg a\rceil \frown \lceil a\rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\, SCH_2 \frown (\lceil \neg W_2 \wedge \neg b\rceil^0 \frown \lceil \neg W_2 \wedge \neg b\rceil \frown \lceil b\rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2\end{cases}\end{array}$

(expansion des $\lceil\!\lceil \neg W_i \wedge \neg \_\rceil^-$)

1.6.3.
$$\vee \begin{pmatrix} \wedge SCH_1 \frown (\lceil\rceil \frown \lceil a \rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge SCH_2 \frown (\lceil\rceil \frown \lceil b \rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix} \frown \lceil P \rceil^0$$
$$\vee \begin{pmatrix} \wedge SCH_1 \frown (\lceil \neg W_1 \wedge \neg a \rceil^0 \frown \lceil \neg W_1 \wedge \neg a \rceil \frown \lceil a \rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge SCH_2 \frown (\lceil\rceil \frown \lceil b \rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix} \frown \lceil P \rceil^0$$
$$\vee \begin{pmatrix} \wedge SCH_1 \frown (\lceil\rceil \frown \lceil a \rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge SCH_2 \frown (\lceil \neg W_2 \wedge \neg b \rceil^0 \frown \lceil \neg W_2 \wedge \neg b \rceil \frown \lceil b \rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix} \frown \lceil P \rceil^0$$
$$\vee \begin{pmatrix} \wedge SCH_1 \frown (\lceil \neg W_1 \wedge \neg a \rceil^0 \frown \lceil \neg W_1 \wedge \neg a \rceil \frown \lceil a \rceil^0) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge SCH_2 \frown (\lceil \neg W_2 \wedge \neg b \rceil^0 \frown \lceil \neg W_2 \wedge \neg b \rceil \frown \lceil b \rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix} \frown \lceil P \rceil^0$$
(Hypothèses de récurrence, séquentialisation)

1.6.4. Le passage de l'étape 1.6.2 à 1.6.3 s'effectue comme suit :
- Premier membre de la disjonction : le lemme de séquentialisation s'applique
- Deuxième membre : b est vérifié avant a, c'est donc $b \Rightarrow [(S_a;S) \mid\mid\mid T]P$ qui s'applique
- Troisième membre : a est vérifié avant b, c'est donc $a \Rightarrow [S \mid\mid\mid (T_b;T)]P$ qui s'applique
- Dernier membre : soit a est vérifié avant b, soit b est vérifié avant a, alors ces deux cas sont similaires aux deuxième et troisième membre. Soit a et b sont vérifiés en même temps, et donc ce cas est similaire au premier membre

1.6.5. Q.E.D.

□

1.7. Q.E.D.

□

2. PROVE: $\lceil [(\text{delay } a;S) \mid\mid\mid (y := B;T)]P \rceil^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid T) \Rightarrow \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid (y := B;T)) \frown \lceil P \rceil^0$

PROOF:

ASSUME: $\lceil [(\text{delay } a;S) \mid\mid\mid T]P \rceil^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid T) \Rightarrow \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid T) \frown \lceil P \rceil^0$

2.1. $\lceil [y := B; ((\text{delay } a;S) \mid\mid\mid T)]P \rceil^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid S_2)$

(hypothèse)

2.2. $\lceil [(\text{delay } a;S) \mid\mid\mid T]P)[y \backslash B] \rceil^0 \frown \begin{pmatrix} \wedge SCH_1 \frown (\lceil\lceil \neg W_1 \rceil\rceil^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = d) \frown SCH_1 \frown \mathcal{M}_{fin}(T_1) \frown END_1 \\ \wedge SCH_2 \frown \mathcal{M}_{fin}(y := B) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix}$

(définitions)

2.3. $\lceil [(\text{delay } a;S) \mid\mid\mid T]P)[y \backslash B] \rceil^0 \frown \begin{pmatrix} \wedge \lceil\lceil W_1 \wedge \neg R_1 \rceil^{-} \frown (\lceil\lceil \neg W_1 \rceil\rceil^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = d) \frown SCH_1 \frown \mathcal{M}_{fin}(T_1) \frown END_1 \\ \wedge \lceil\lceil W_2 \wedge \neg R_2 \rceil^{-} \frown (\lceil R_2 \rceil^1 \frown \lceil y = B_0 \rceil^0 \wedge \ell = 0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix}$

(définitions)

2.4. $\lceil [(((\text{delay } a;S) \mid\mid\mid T)]P)[y \backslash B] \rceil^0 \frown \begin{pmatrix} \wedge (\lceil\lceil \neg W_1 \rceil\rceil^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = d) \frown SCH_1 \frown \mathcal{M}_{fin}(T_1) \frown END_1 \\ \wedge (\lceil R_2 \rceil^1 \frown \lceil y = B_0 \rceil^0 \wedge \ell = 0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix}$

($COND_0, COND_1$: puisque R_1 n'attend pas, R_2 doit s'exécuter)

2.5. $\lceil [(((\text{delay } a;S) \mid\mid\mid T)]P)[y \backslash B] \rceil^0 \frown \begin{pmatrix} \wedge (\lceil\lceil \neg W_1 \rceil\rceil^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = d) \frown SCH_1 \frown \mathcal{M}_{fin}(T_1) \frown END_1 \\ \wedge (\lceil R_2 \rceil^1 \frown \lceil y = B_0 \rceil^0 \wedge \ell = 0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix}$

($COND_0, COND_1$: puisque (delay $a;S$) n'attend pas, S_2 doit s'exécuter)

2.6. $\begin{pmatrix} \wedge (\ell = 0 \wedge \lceil W_1 \rceil^1) \frown \lceil [((\text{delay } a;S) \mid\mid\mid T)]P \rceil^0 \frown (\lceil\lceil \neg W_1 \rceil\rceil^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = d) \frown SCH_1 \frown \mathcal{M}_{fin}(T_1) \frown END_1 \\ \wedge (\lceil R_2 \rceil^1 \frown \lceil y = B_0 \rceil^0 \wedge \ell = 0) \frown \lceil [((\text{delay } a;S) \mid\mid\mid T)]P \rceil^0 \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix}$

(propriétés de $\lceil\lceil P \rceil\rceil^+$, $COND_2$)

2.7. $\begin{pmatrix} \wedge (\ell = 0 \wedge \lceil W_1 \rceil^1) \\ \wedge (\lceil R_2 \rceil^1 \frown \lceil y = B_0 \rceil^0 \wedge \ell = 0) \end{pmatrix} \frown \lceil [((\text{delay } a;S) \mid\mid\mid T)]P \rceil^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid T)$

(définitions, WDC^*)

2.8. $\begin{pmatrix} \wedge (\ell = 0 \wedge \lceil W_1 \rceil^1) \\ \wedge (\lceil R_2 \rceil^1 \frown \lceil y = B_0 \rceil^0 \wedge \ell = 0) \end{pmatrix} \frown \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid T) \frown \lceil P \rceil^0$

(définitions, WDC^*)

2.9. $\mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid (y := B;T)) \frown \lceil P \rceil^0$

(définitions, WDC^*)

2.10. Q.E.D.

□

3. PROVE: $\lceil [(x := A;S) \mid\mid\mid (\text{delay } b;T)]P \rceil^0 \frown \mathcal{M}_{fin}((x := A;S) \mid\mid\mid (\text{delay } b;T)) \Rightarrow \mathcal{M}_{fin}((x := A;S) \mid\mid\mid (\text{delay } b;T)) \frown \lceil P \rceil^0$

PROOF:

3.1. La preuve est similaire à 2, à la symétrie des termes près

3.2. Q.E.D.

□

4. PROVE: $\lceil [(\text{delay } a;S) \mid\mid\mid (\text{await } b;T)]P \rceil^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid (\text{await } b;T)) \Rightarrow \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid (\text{await } b;T)) \frown \lceil P \rceil^0$

PROOF:

ASSUME: $\lceil [\text{delay } a; (S \mid\mid\mid (\text{await } b;T))]P \rceil^0 \frown \mathcal{M}_{fin}(\text{delay } a; (S \mid\mid\mid (\text{await } b;T))) \Rightarrow \mathcal{M}_{fin}(\text{delay } a; (S \mid\mid\mid (\text{await } b;T))) \frown \lceil P \rceil^0$

ASSUME: $\lceil [\text{await } b; ((\text{delay } a;S) \mid\mid\mid T)]P \rceil^0 \frown \mathcal{M}_{fin}(\text{await } b; ((\text{delay } a;S) \mid\mid\mid T)) \Rightarrow \mathcal{M}_{fin}(\text{await } b; ((\text{delay } a;S) \mid\mid\mid T)) \frown \lceil P \rceil^0$

4.1. $\begin{bmatrix} \wedge [\text{delay } a; (S \mid\mid\mid (\text{await } b;T))]P \\ \wedge [\text{await } b; ((\text{delay } a;S) \mid\mid\mid T)]P \end{bmatrix}^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \mid\mid\mid (\text{await } b;T))$

(définitions)

4.2. $\begin{bmatrix} \wedge [\text{delay } a; (S \mid\mid\mid (\text{await } b;T))]P \\ \wedge [\text{await } b; ((\text{delay } a;S) \mid\mid\mid T)]P \end{bmatrix}^0 \frown \begin{matrix} \wedge (\lceil\lceil \neg W_1 \rceil\rceil^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = a) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge SCH_2 \frown (\lceil\lceil \neg W_2 \wedge \neg b \rceil^{-} \frown \lceil a \rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{matrix}$

(définitions)

4.3. $\begin{bmatrix} \wedge [\text{delay } a; (S \mid\mid\mid (\text{await } b;T))]P \\ \wedge [\text{await } b; ((\text{delay } a;S) \mid\mid\mid T)]P \end{bmatrix}^0 \frown \begin{matrix} \wedge (\lceil\lceil \neg W_1 \rceil\rceil^+ \frown Unit \frown \lceil 1 \rceil^0 \wedge \ell = a) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge \begin{cases} \vee SCH_2 \frown (\lceil\rceil \frown \lceil b \rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \\ \vee SCH_2 \frown (\lceil \neg W_2 \wedge \neg b \rceil^0 \frown \lceil \neg W_2 \wedge \neg b \rceil \frown \lceil b \rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{cases} \end{matrix}$

(WDC*)

4.4. $\begin{bmatrix} \wedge\ [S \,|||\, (\text{await } b;T)]P \\ \wedge\ [\text{await } b;((\text{delay } a;S) \,|||\, T)]P \end{bmatrix}^0 \frown \begin{matrix} \vee \lceil b\rceil^0 \frown \begin{cases} \wedge\ (\llbracket\neg W_1\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = a) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\ SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{cases} \\ \vee \begin{cases} \wedge\ (\llbracket\neg W_1\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = a) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\ SCH_2 \frown (\lceil\neg W_2 \wedge \neg b\rceil^0 \frown \lceil\neg W_2 \wedge \neg b\rceil \frown \lceil b\rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{cases} \end{matrix}$

(WDC*)

4.5. $\vee \lceil[\text{await } b;((\text{delay } a;S) \,|||\, T)]P\rceil^0 \frown \lceil b\rceil^0 \frown \begin{cases} \wedge\ (\llbracket\neg W_1\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = a) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\ SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{cases}$

$\vee\ SCH_2 \frown (\lceil\neg W_2 \wedge \neg b\rceil^0 \frown \lceil\neg W_2 \wedge \neg b\rceil \wedge \ell = a) \frown \lceil[S \,|||\, (\text{await } b;T)]P\rceil^0 \frown \begin{cases} \wedge\ SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\ ((\lceil\neg W_2 \wedge \neg b\rceil \vee \lceil\rceil) \frown \lceil b\rceil^0) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{cases}$

(WDC*, l'attente continue car la condition n'est pas vérifiée : l'autre processus est en attente par un délai)

4.6. $\vee \lceil b \Rightarrow [(\text{delay } a;S) \,|||\, T]P\rceil^0 \frown \mathcal{M}_{fin}(b \Longrightarrow ((\text{delay } a;S) \,|||\, T))$

$\vee\ SCH_2 \frown (\lceil\neg W_2 \wedge \neg b\rceil^0 \frown \lceil\neg W_2 \wedge \neg b\rceil \wedge \ell = a) \frown \lceil[S \,|||\, (\text{await } b;T)]P\rceil^0 \frown \mathcal{M}_{fin}(S \,|||\, (\text{await } b;T))$

(WDC*)

4.7. $\vee\ \mathcal{M}_{fin}(b \Longrightarrow ((\text{delay } a;S) \,|||\, T)) \frown \lceil P\rceil^0$

$\vee\ SCH_2 \frown (\lceil\neg W_2 \wedge \neg b\rceil^0 \frown \lceil\neg W_2 \wedge \neg b\rceil \wedge \ell = a) \frown \mathcal{M}_{fin}(S \,|||\, (\text{await } b;T)) \frown \lceil P\rceil^0$

(WDC*)

4.8. $\vee\ \mathcal{M}_{fin}(b \Longrightarrow ((\text{delay } a;S) \,|||\, T)) \frown \lceil P\rceil^0$

$\vee\ SCH_2 \frown (\lceil\neg W_2 \wedge \neg b\rceil^0 \frown \lceil\neg W_2 \wedge \neg b\rceil \wedge \ell = a) \frown \mathcal{M}_{fin}(S \,|||\, (\text{await } b;T)) \frown \lceil P\rceil^0$

(WDC*)

4.9. $\vee \lceil b\rceil^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \,|||\, T) \frown \lceil P\rceil^0$

$\vee \frown \mathcal{M}_{fin}((\text{delay } a;S) \,|||\, (\text{await } b;T)) \frown \lceil P\rceil^0$

(WDC*, définitions)

4.10. $\vee \frown \mathcal{M}_{fin}((\text{delay } a;S) \,|||\, (\text{await } b;T)) \frown \lceil P\rceil^0$

(WDC*, le premier membre de la conjonction est un sous-cas du deuxième)

4.11. Q.E.D.

□

5. PROVE: $\lceil[(\text{await } a;S) \,|||\, (\text{delay } b;T)]P\rceil^0 \frown \mathcal{M}_{fin}((\text{await } a;S) \,|||\, (\text{delay } b;T)) \Rightarrow \mathcal{M}_{fin}((\text{await } a;S) \,|||\, (\text{delay } b;T)) \frown \lceil P\rceil^0$

PROOF:

5.1. La preuve est similaire à 4, à la symétrie des termes près

5.2. Q.E.D.

□

6. PROVE: $\lceil[(\text{delay } a;S) \,|||\, (\text{delay } b;T)]P\rceil^0 \frown \mathcal{M}_{fin}((\text{delay } a;S) \,|||\, (\text{delay } b;T)) \Rightarrow \mathcal{M}_{fin}((\text{delay } a;S) \,|||\, (\text{delay } b;T)) \frown \lceil P\rceil^0$

PROOF:

ASSUME: $a < b$

ASSUME: $\lceil[(S \,|||\, (\text{delay } (b-a);T)]P\rceil^0 \frown \mathcal{M}_{fin}(S \,|||\, (\text{delay } (b-a);T) \Rightarrow \mathcal{M}_{fin}((\text{delay } a;(S \,|||\, (\text{delay } (b-a);T)) \frown \lceil P\rceil^0$

6.1. $\lceil[(\text{delay } a;(S \,|||\, (\text{delay } (b-a);T))]P\rceil^0 \frown \mathcal{M}_{fin}(S_1 \,|||\, S_2)$

6.2. $\lceil[(\text{delay } a;(S \,|||\, (\text{delay } (b-a);T))]P\rceil^0 \frown \begin{pmatrix} \wedge\ (\llbracket\neg W_1\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = a) \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\ (\llbracket\neg W_2\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = b) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix}$

(définitions)

6.3. $(\llbracket\neg W_1\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = a) \frown \lceil[(\text{delay } a;(S \,|||\, (\text{delay } (b-a);T))]P\rceil^0$

$\frown \begin{pmatrix} \wedge \frown SCH_1 \frown \mathcal{M}_{fin}(S) \frown END_1 \\ \wedge\ (\llbracket\neg W_2\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = (b-a)) \frown SCH_2 \frown \mathcal{M}_{fin}(T) \frown END_2 \end{pmatrix}$

($b > a, WDC^*$)

6.4. $(\llbracket\neg W_1\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = a) \frown \lceil[(\text{delay } a;(S \,|||\, (\text{delay } (b-a);T))]P\rceil^0 \frown \mathcal{M}_{fin}(S \,|||\, (\text{delay } (b-a);T))$

(définitions)

6.5. $(\llbracket\neg W_1\rrbracket^+ \frown Unit \frown \lceil 1\rceil^0 \wedge \ell = a) \frown \lceil[(S \,|||\, (\text{delay } (b-a);T)]P\rceil^0 \frown \mathcal{M}_{fin}(S \,|||\, (\text{delay } (b-a);T))$

(définition de la substitution delay)

6.6. $\mathcal{M}_{fin}((\text{delay } a;S) \,|||\, (\text{delay } b;T)) \frown \lceil P\rceil^0$

(définitions, WDC^*)

6.7. Q.E.D.

□

Annexe B

Implémentation en Coq de DC*

Module PL_utils

Require Import *Classical_Type*.

Theorem *contraposition_not* : $\forall$ *A B* : *Prop*, $(A \rightarrow B) \rightarrow \neg\ B \rightarrow \neg\ A$.
unfold not in $\vdash \times$; *intros* ; *tauto*.
Qed.

Theorem *not_contraposition* : $\forall$ *A B* : *Prop*, $(\tilde{}\ A \rightarrow \neg\ B) \rightarrow B \rightarrow A$.
intros.
apply NNPP ; *unfold not in* $\vdash \times$; *intros* ; *apply H*.
assumption.
assumption.
Qed.

Theorem *or_false* : $\forall$ *A* : *Prop*, $A \vee False \rightarrow A$.
intros A AF.
elim AF ; [trivial | tauto].
Qed.

Theorem *false_or* : $\forall$ *A* : *Prop*, $False \vee A \rightarrow A$.
intros A AF.
elim AF ; [tauto | trivial].
Qed.

Theorem *or_false_r* : $\forall$ *A* : *Prop*, $A \rightarrow A \vee False$.
intros A HA ; *left* ; *assumption*.
Qed.

Theorem *false_or_r* : $\forall$ *A* : *Prop*, $A \rightarrow False \vee A$.
intros A HA ; *right* ; *assumption*.
Qed.

Theorem *and_true* : $\forall$ *A* : *Prop*, $A \wedge True \rightarrow A$.
intros A AT.
elim AT ; *intros* ; *assumption*.

Qed.

Theorem *true_and* : $\forall$ *A* : *Prop*, *True* $\wedge$ *A* $\rightarrow$ *A*.
intros A AT.
elim AT ; *intros* ; *assumption*.
Qed.

Theorem *and_true_r* : $\forall$ *A* : *Prop*, *A* $\rightarrow$ *A* $\wedge$ *True*.
intros A HA.
split ; [assumption | trivial].
Qed.

Theorem *true_and_r* : $\forall$ *A* : *Prop*, *A* $\rightarrow$ *True* $\wedge$ *A*.
intros A AT.
split ; [trivial | assumption].
Qed.

Theorem *and_comm* : $\forall$ *A B* : *Prop*, *A* $\wedge$ *B* $\rightarrow$ *B* $\wedge$ *A*.
intros A B AB ; *elim AB* ; *intros* ; *split* ; [assumption | assumption].
Qed.

Theorem *or_comm* : $\forall$ *A B* : *Prop*, *A* $\vee$ *B* $\rightarrow$ *B* $\vee$ *A*.
intros A B AB ; *elim AB* ;
  [intros ; right ; assumption | intros ; left ; assumption].
Qed.

Theorem *and_distr_or* : $\forall$ *A B C* : *Prop*, *A* $\wedge$ (*B* $\vee$ *C*) $\rightarrow$ *A* $\wedge$ *B* $\vee$ *A* $\wedge$ *C*.
intros A B C AaBoC.
elim AaBoC ; *intros HA HBoC*.
elim HBoC.
  intros ; *left* ; *split* ; [assumption | assumption].
  intros ; *right* ; *split* ; [assumption | assumption].
Qed.

Theorem *and_distr_or_r* :
  $\forall$ *A B C* : *Prop*, *A* $\wedge$ *B* $\vee$ *A* $\wedge$ *C* $\rightarrow$ *A* $\wedge$ (*B* $\vee$ *C*).
intros A B C AaBoAaC.
elim AaBoAaC ; *intros Aand*.
  elim Aand ; *intros* ; *split* ; [assumption | left ; assumption].
  elim Aand ; *intros* ; *split* ; [assumption | right ; assumption].
Qed.

Theorem *or_distr_and* :
  $\forall$ *A B C* : *Prop*, *A* $\vee$ *B* $\wedge$ *C* $\rightarrow$ (*A* $\vee$ *B*) $\wedge$ (*A* $\vee$ *C*).
intros A B C AoBaC.
elim AoBaC.
  intro HA ; *split* ; [left ; assumption | left ; assumption].
  intro BaC ; *elim BaC* ; *intros* ; *split* ;
    [right ; assumption | right ; assumption].
Qed.

Theorem $or\_distr\_and\_r$:
 $\forall A\ B\ C : Prop, (A \vee B) \wedge (A \vee C) \rightarrow A \vee B \wedge C$.
$intros\ A\ B\ C$; $apply\ not\_contraposition$.
$intro\ nAoBC$.
$apply\ or\_not\_and$.
cut (~ $A \wedge \neg\ B \vee \neg\ A \wedge \neg\ C$).
 $intro\ nAnBonAnC$; $elim\ nAnBonAnC$; $intro\ hypo$.
 $left$; $apply\ and\_not\_or$; $assumption$.
 $right$; $apply\ and\_not\_or$; $assumption$.
$apply\ and\_distr\_or$.
cut (~ $A \wedge \neg\ (B \wedge C)$).
 $intro\ nAnBC$; $elim\ nAnBC$; $intros$; $split$;
  [assumption | apply not_and_or ; assumption].
$apply\ not\_or\_and$; $assumption$.
Qed.

Module IL_base

Require Export $Reals$.

Parameter $length\_eq : R \rightarrow Prop$.
Parameter $length\_lt : R \rightarrow Prop$.

Definition $length\_le\ (x :R) :=(length\_eq\ x) \vee (length\_lt\ x)$.
Definition $length\_gt\ (x :R) :=$~$(length\_le\ x)$.
Definition $length\_ge\ (x :R) :=(length\_eq\ x) \vee (length\_gt\ x)$.

Parameter $chop : Prop \rightarrow Prop \rightarrow Prop$.

A few handy definitions

Definition $point := (length\_eq\ 0)$.
Definition $sometimes\ (P : Prop) := chop\ True\ (chop\ P\ True)$.
Definition $always\ (P : Prop) := \neg\ sometimes$ (~ P).

- "sometime" reads *for some subinterval*
- "always" reads *for all subintervals*

Infix "$\frown$" := $chop$ ($at\ level$ 79, $right\ associativity$).

Notation "'$l' = x$" := ($length\_eq\ x$) ($at\ level$ 65, $no\ associativity$).
Notation "'$l' \neq x$" := (~ ($length\_eq\ x$)) ($at\ level$ 65, $no\ associativity$).
Notation "'$l' \leq x$" := ($length\_le\ x$) ($at\ level$ 65, $no\ associativity$).
Notation "'$l' < x$" := ($length\_lt\ x$) ($at\ level$ 65, $no\ associativity$).
Notation "'$l' \geq x$" := ($length\_ge\ x$) ($at\ level$ 65, $no\ associativity$).
Notation "'$l' > x$" := ($length\_gt\ x$) ($at\ level$ 65, $no\ associativity$).

Notation "$\lceil\rceil$" :=$point$ ($at\ level$ 33, $no\ associativity$).

Notation "◇ *P*" :=(*sometimes P*) (*at level* 34, *no associativity*).

Notation "□ *P*" :=(*always P*) (*at level* 35, *no associativity*).

Module Common_functions

Require Import *IL_base*.

All functions in this file should be considered as "calculation" functions, because they are considered in corresponding proof systems as side-conditions.

Definition of the rigidity of a formula. Valid for both pure Interval Logic and Duration Calculus

Note that a state can be not rigid. Indeed S(1) = l, thus any state other than 0 will be flexible. For 0 states, as S(0)=0, replacing through equality allows finishing testing the rigidity of a term

Inductive *rigid_term* : *R* → *Prop* :=
  | *rig_r0* : *rigid_term* 0%*R*
  | *rig_r1* : *rigid_term* 1%*R*
  | *rig_rbin* :
    ∀ (*F* : *R* → *R* → *R*) (*x y* : *R*),
    rigid_term x ∧ *rigid_term y* → *rigid_term* (*F x y*)
  | *rig_opp* : ∀ *x* : *R*, *rigid_term x* → *rigid_term* (- *x*)%*R*
  | *rig_inv* : ∀ *x* : *R*, *rigid_term x* → *rigid_term* (/ *x*)%*R*
  | *rig_sqrt* : ∀ *x* : *R*, *rigid_term x* → *rigid_term* (*sqrt x*).

Inductive *rigid* : *Prop* → *Prop* :=
  | *rig_true* : *rigid True*
  | *rig_false* : *rigid False*
  | *rig_chop* : ∀ *p q* : *Prop*, *rigid p* ∧ *rigid q* → *rigid* (*chop p q*)
  | *rig_imp* : ∀ *p q* : *Prop*, *rigid p* ∧ *rigid q* → *rigid* (*p* → *q*)
  | *rig_and* : ∀ *p q* : *Prop*, *rigid p* ∧ *rigid q* → *rigid* (*p* ∧ *q*)
  | *rig_or* : ∀ *p q* : *Prop*, *rigid p* ∧ *rigid q* → *rigid* (*p* ∨ *q*)
  | *rig_not* : ∀ *p* : *Prop*, *rigid p* → *rigid* (~ *p*)
  | *rig_ex* : ∀ *p* : *R* → *Prop*, *rigid* (*p* 0%*R*) → *rigid* (*ex p*)
  | *rig_all* : ∀ *p* : *R* → *Prop*, *rigid* (*p* 0%*R*) → *rigid* (*all p*)
  | *rig_rel* :
    ∀ (*r* : *R* → *R* → *Prop*) (*x y* : *R*),
    rigid_term x ∧ *rigid_term y* → *rigid* (*r x y*).

What is important in this inductive definition of rigidity is what is not defined, i.e. all the *Var "constructions", and ℓ. They don't appear in the definition, so when using the tactic Rigidity defined below, you won't be able to state that a formula containing ℓ or a variable is rigid, which correspond to the definition of rigidity. Note that we don't define flexibility, as such a side-condition never appears in the axioms and inference rules of the proof system of IL or DC.

Ltac *Rigidity* := *repeat first* [constructor | split].

```
Inductive chop_free : Prop → Prop :=
  | cf_true : chop_free True
  | cf_false : chop_free False
  | cf_imp :
      ∀ p q : Prop, chop_free p ∧ chop_free q → chop_free (p → q)
  | cf_and :
      ∀ p q : Prop, chop_free p ∧ chop_free q → chop_free (p ∧ q)
  | cf_or :
      ∀ p q : Prop, chop_free p ∧ chop_free q → chop_free (p ∨ q)
  | cf_not : ∀ p : Prop, chop_free p → chop_free (~ p)
  | cf_ex : ∀ p : R → Prop, chop_free (p 0%R) → chop_free (ex p)
  | cf_all : ∀ p : R → Prop, chop_free (p 0%R) → chop_free (all p)
  | cf_rel : ∀ (F : R → R → Prop) (x y : R), chop_free (F x y).

Ltac ChopFree := repeat first [ constructor | split ].
```

Module IL_length

```
Require Import IL_base.
Require Import IL_axioms.
Require Import Common_functions.
Require Import Classical_Type.
Require Import PL_utils.
Require Import Fourier.

Theorem length_gt_not_eq :∀ x :R, l>x → ¬ l=x.
compute.
intros; apply H.
left; assumption.
Qed.

Theorem length_eq_not_gt :∀ x :R,l=x → ~(l>x).
compute; intros.
apply length_gt_not_eq with x.
assumption.
assumption.
Qed.

Theorem length_eq_not_lt :∀ x :R,l=x → ~(l<x).
compute; intros.
apply length_lt_not_eq with x.
assumption.
assumption.
Qed.

Theorem length_diff_lt_gt :∀ x :R,l≠x ↔ l<x ∨ l>x.
intros; split.
```

intros.
cut (˜(˜l<x ∧ ¬ l>x)).
intro.
apply NNPP.
intro.
elim H0.
apply not_or_and ; *assumption*.
intro.
elim H0 ; *intros*.
compute in H2.
tauto.
intros ; *elim H* ; *intros*.
apply length_lt_not_eq ; *assumption*.
apply length_gt_not_eq ; *assumption*.
Qed.

Theorem *length_lt_gt_false* :∀ *x* :*R*, *l*<*x* ∧ *l*>*x* → *False*.
compute ; *intros*.
elim H ; *intros*.
apply H1 ; *right* ; *assumption*.
Qed.

Theorem *total_length* :∀ *x* :*R*, *l*<*x* ∨ *l*=*x* ∨ *l*>*x*.
compute ; *intros*.
elim (*classic* (*l* < *x* ∨ *l* = *x*)).
tauto.
tauto.
Qed.

Theorem *length_lt_not_ge* :∀ *x* :*R*, *l*<*x* ↔ ¬ *l*≥*x*.
intros ; *split*.
intro.
unfold length_ge.
apply and_not_or ; *split*.
apply length_lt_not_eq ; *assumption*.
intro ; *apply H0* ; *right* ; *assumption*.
intro.
compute in H.
cut (*l*≠*x* ∧ ˜(*l* = *x* ∨ *l* < *x* → *False*)).
intros.
elim H0 ; *intros*.
cut (*l*≤*x*).
intros.
elim H3 ; *intros*.
contradiction.
assumption.
apply NNPP ; *assumption*.

apply not_or_and ; *assumption*.
Qed.

Theorem *length_le_not_gt* :$\forall$ *x* :*R*, $l \leq x \leftrightarrow \neg\ l > x$.
intros ; *split*.
intro.
unfold length_gt.
tauto.
intro.
apply NNPP.
intro.
apply H.
assumption.
Qed.

Theorem *length_eq_Rge* :$\forall$ *x y* :*R*,$l = x \rightarrow x \geq y \rightarrow l \geq y$.
intros.
elim H0 ; *intros*.
right ; *apply length_eq_Rgt with x* ; *assumption*.
left ; *rewrite* $\leftarrow$ *H1* ; *assumption*.
Qed.

Theorem *length_eq_Rle* :$\forall$ *x y* :*R*,$l = x \rightarrow x \leq y \rightarrow l \leq y$.
intros.
elim H0 ; *intros*.
right ; *apply length_eq_Rlt with x* ; *assumption*.
left ; *rewrite* $\leftarrow$ *H1* ; *assumption*.
Qed.

Theorem *length_eq_length_le* :$\forall$ *x y* :*R*,$l = x \rightarrow l \leq y \rightarrow x \leq y$.
intros.
elim H0 ; *intros*.
right ; *apply length_eq_length_eq* ; *assumption*.
left ; *apply length_eq_length_lt* ; *assumption*.
Qed.

Theorem *length_eq_length_ge* :$\forall$ *x y* :*R*,$l = x \rightarrow l \geq y \rightarrow x \geq y$.
intros.
elim H0 ; *intros*.
right ; *apply length_eq_length_eq* ; *assumption*.
left ; *apply length_eq_length_gt* ; *assumption*.
Qed.

Theorem *length_gt_Rge* :$\forall$ *x y* :*R*,$l > x \rightarrow x \geq y \rightarrow l > y$.
intros.
elim H0 ; *intros*.
apply length_gt_Rgt with x ; *assumption*.
rewrite $\leftarrow$ *H1* ; *assumption*.
Qed.

Module IL_length

Theorem *length_ge_Rgt* :$\forall$ *x y* :*R*,$l \geq x \rightarrow x > y \rightarrow l > y$.
intros.
elim H ; *intros*.
apply length_eq_Rgt with x ; *assumption*.
apply length_gt_Rgt with x ; *assumption*.
Qed.

Theorem *length_ge_Rge* :$\forall$ *x y* :*R*,$l \geq x \rightarrow x \geq y \rightarrow l \geq y$.
intros.
elim H ; *intros*.
apply length_eq_Rge with x ; *assumption*.
right ; *apply length_gt_Rge with x* ; *assumption*.
Qed.

Theorem *length_ge_length_lt* :$\forall$ *x y* :*R*,$l \geq x \rightarrow l < y \rightarrow x < y$.
intros.
elim H ; *intros*.
apply length_lt_length_eq ; *assumption*.
apply length_gt_length_lt ; *assumption*.
Qed.

Theorem *length_gt_length_le* :$\forall$ *x y* :*R*,$l > x \rightarrow l \leq y \rightarrow x < y$.
intros.
elim H0 ; *intros*.
apply length_gt_length_eq ; *assumption*.
apply length_gt_length_lt ; *assumption*.
Qed.

Theorem *length_ge_length_le* :$\forall$ *x y* :*R*,$l \geq x \rightarrow l \leq y \rightarrow x \leq y$.
intros.
elim H ; *intros*.
apply length_eq_length_le ; *assumption*.
left ; *apply length_gt_length_le* ; *assumption*.
Qed.

Theorem *length_ge_length_eq* :$\forall$ *x y* :*R*,$l \geq x \rightarrow l = y \rightarrow x \leq y$.
intros.
apply Rge_le.
apply length_eq_length_ge ; *assumption*.
Qed.

Theorem *length_le_Rlt* :$\forall$ *x y* :*R*,$l \leq x \rightarrow x < y \rightarrow l < y$.
intros.
elim H ; *intros*.
apply length_eq_Rlt with x ; *assumption*.
apply length_lt_Rlt with x ; *assumption*.
Qed.

Theorem *length_lt_Rle* :$\forall$ *x y* :*R*,$l < x \rightarrow x \leq y \rightarrow l < y$.

intros.
elim H0 ; *intros*.
apply length_lt_Rlt with x ; *assumption*.
rewrite ← *H1* ; *assumption*.
Qed.

Theorem *length_le_Rle* :∀ *x y* :*R*,*l*≤*x* → *x*≤*y* → *l*≤*y*.
intros.
elim H ; *intros*.
apply length_eq_Rle with x ; *assumption*.
right ; *apply length_lt_Rle with x* ; *assumption*.
Qed.

Theorem *length_le_length_eq* :∀ *x y* :*R*,*l*≤*x* → *l*=*y* → *y*≤*x*.
intros.
elim H ; *intros*.
apply length_eq_length_le ; *assumption*.
left ; *apply length_lt_length_eq* ; *assumption*.
Qed.

Theorem *length_le_length_gt* :∀ *x y* :*R*,*l*≤*x* → *l*>*y* → *y*<*x*.
intros.
elim H ; *intros*.
apply length_gt_length_eq ; *assumption*.
apply length_lt_length_gt ; *assumption*.
Qed.

Theorem *length_lt_length_ge* :∀ *x y* :*R*,*l*<*x* → *l*≥*y* → *y*<*x*.
intros.
elim H0 ; *intros*.
apply length_lt_length_eq ; *assumption*.
apply length_lt_length_gt ; *assumption*.
Qed.

Theorem *length_le_length_ge* :∀ *x y* :*R*,*l*≤*x* → *l*≥*y* → *y*≤*x*.
intros.
elim H ; *intros*.
apply length_ge_length_eq ; *assumption*.
left ; *apply length_lt_length_ge* ; *assumption*.
Qed.

Theorem *length_between* :∀ *x* :*R*, *l*≤*x* → *l*≥*x* → *l*=*x*.
intros.
elim H0 ; *intros*.
assumption.
contradiction.
Qed.

Module IL_axioms

Require Import *IL_base*.
Require Import *Common_functions*.
Require *Reals*.

Axiom *IL1* : $\forall\ p\ q : Prop, \Box\ (p \to q) \to \Box p \to \Box q$.
Axiom *IL2* : $\forall\ p : Prop, \Box p \to p$.

Open Scope R_scope.

Axiom *always_length_lt_not_eq* : $\Box\ (\ \forall\ x : R, \ell < x \to \ell \neq x\ )$.
Theorem *length_lt_not_eq* : $\forall\ x : R, \ell < x \to \ell \neq x$.
apply IL2 ; *apply always_length_lt_not_eq*. Qed.

Axiom *always_length_eq_Rgt* : $\Box\ (\ \forall\ x\ y : R, \ell = x \to x > y \to \ell > y\ )$.
Theorem *length_eq_Rgt* : $\forall\ x\ y : R, \ell = x \to x > y \to \ell > y$.
apply IL2 ; *apply always_length_eq_Rgt*. Qed.

Axiom *always_length_eq_Rlt* : $\Box\ (\ \forall\ x\ y : R, \ell = x \to x < y \to \ell < y\ )$.
Theorem *length_eq_Rlt* : $\forall\ x\ y : R, \ell = x \to x < y \to \ell < y$.
apply IL2 ; *apply always_length_eq_Rlt*. Qed.
Axiom *always_length_eq_length_lt* : $\Box\ (\ \forall\ x\ y : R, \ell = x \to \ell < y \to x < y\ )$.
Theorem *length_eq_length_lt* : $\forall\ x\ y : R, \ell = x \to \ell < y \to x < y$.
apply IL2 ; *apply always_length_eq_length_lt*. Qed.
Axiom *always_length_eq_length_eq* : $\Box\ (\ \forall\ x\ y : R, \ell = x \to \ell = y \to x = y\ )$.
Theorem *length_eq_length_eq* : $\forall\ x\ y : R, \ell = x \to \ell = y \to x = y$.
apply IL2 ; *apply always_length_eq_length_eq*. Qed.
Axiom *always_length_eq_length_gt* : $\Box\ (\ \forall\ x\ y : R, \ell = x \to \ell > y \to x > y\ )$.
Theorem *length_eq_length_gt* : $\forall\ x\ y : R, \ell = x \to \ell > y \to x > y$.
apply IL2 ; *apply always_length_eq_length_gt*. Qed.

Axiom *always_length_gt_Rgt* : $\Box\ (\ \forall\ x\ y : R, \ell > x \to x > y \to \ell > y\ )$.
Theorem *length_gt_Rgt* : $\forall\ x\ y : R, \ell > x \to x > y \to \ell > y$.
apply IL2 ; *apply always_length_gt_Rgt*. Qed.
Axiom *always_length_gt_length_lt* : $\Box\ (\ \forall\ x\ y : R, \ell > x \to \ell < y \to x < y\ )$.
Theorem *length_gt_length_lt* : $\forall\ x\ y : R, \ell > x \to \ell < y \to x < y$.
apply IL2 ; *apply always_length_gt_length_lt*. Qed.
Axiom *always_length_gt_length_eq* : $\Box\ (\ \forall\ x\ y : R, \ell > x \to \ell = y \to x < y\ )$.
Theorem *length_gt_length_eq* : $\forall\ x\ y : R, \ell > x \to \ell = y \to x < y$.
apply IL2 ; *apply always_length_gt_length_eq*. Qed.

Axiom *always_length_lt_Rlt* : $\Box\ (\ \forall\ x\ y : R, \ell < x \to x < y \to \ell < y\ )$.
Theorem *length_lt_Rlt* : $\forall\ x\ y : R, \ell < x \to x < y \to \ell < y$.
apply IL2 ; *apply always_length_lt_Rlt*. Qed.
Axiom *always_length_lt_length_eq* : $\Box\ (\ \forall\ x\ y : R, \ell < x \to \ell = y \to y < x\ )$.
Theorem *length_lt_length_eq* : $\forall\ x\ y : R, \ell < x \to \ell = y \to y < x$.
apply IL2 ; *apply always_length_lt_length_eq*. Qed.
Axiom *always_length_lt_length_gt* : $\Box\ (\ \forall\ x\ y : R, \ell < x \to \ell > y \to y < x\ )$.

Theorem *length_lt_length_gt* : $\forall$ *x y* :*R*,ℓ<*x* $\rightarrow$ ℓ>*y* $\rightarrow$ *y*<*x* .
apply IL2 ; *apply always_length_lt_length_gt*. Qed.

Axiom *always_length_exists* : $\Box$ ($\exists$ *x* :*R*, ℓ=*x*) .
Theorem *length_exists* : $\exists$ *x* :*R*, ℓ=*x*.
apply IL2 ; *apply always_length_exists*. Qed.

– Now for the original axioms

Axiom *always_pos_interval* : $\Box$ ($\ell \geq$ 0).
Theorem *pos_interval* : $\ell \geq 0$.
apply IL2 ; *apply always_pos_interval*. Qed.

Definition *A0* := *pos_interval*.

Axiom
always_chop_and_right :
$\forall$ *p q r* : *Prop*, $\Box$ (*chop p q* $\wedge$ $\neg$ *chop p r* $\rightarrow$ *chop p* (*q* $\wedge$ $\neg$ *r*)).
Axiom
always_chop_and_left :
$\forall$ *p q r* : *Prop*, $\Box$ (*chop p q* $\wedge$ $\neg$ *chop r q* $\rightarrow$ *chop* (*p* $\wedge$ $\neg$ *r*) *q*).
Theorem *chop_and_right* : $\forall$ *p q r* : *Prop*, *chop p q* $\wedge$ $\neg$ *chop p r* $\rightarrow$ *chop p* (*q* $\wedge$ $\neg$ *r*).
intros p q r ; *apply IL2* ; *apply always_chop_and_right*. Qed.
Theorem *chop_and_left* : $\forall$ *p q r* : *Prop*, *chop p q* $\wedge$ $\neg$ *chop r q* $\rightarrow$ *chop* (*p* $\wedge$ $\neg$ *r*) *q*.
intros p q r ; *apply IL2* ; *apply always_chop_and_left*. Qed.

Definition *A1_right* := *chop_and_right*.
Definition *A1_left* := *chop_and_left*.

Axiom
always_chop_assoc : $\forall$ *p q r* : *Prop*, $\Box$ (*chop* (*chop p q*) *r* $\leftrightarrow$ *chop p* (*chop q r*)).
Theorem *chop_assoc* : $\forall$ *p q r* : *Prop*, *chop* (*chop p q*) *r* $\leftrightarrow$ *chop p* (*chop q r*).
intros p q r ; *apply IL2* ; *apply always_chop_assoc*. Qed.
Definition *A2* := *chop_assoc*.

Theorem *chop_assoc_left_right* :
$\forall$ *P Q R* : *Prop*, *chop* (*chop P Q*) *R* $\rightarrow$ *chop P* (*chop Q R*).
intros P Q R chopleft.
cut (*chop* (*chop P Q*) *R* $\leftrightarrow$ *chop P* (*chop Q R*)).
intros H.
elim H ; *intros CLR CRL*.
apply CLR ; *assumption*.
apply chop_assoc.
Qed.

Theorem *chop_assoc_right_left* :
$\forall$ *P Q R* : *Prop*, *chop P* (*chop Q R*) $\rightarrow$ *chop* (*chop P Q*) *R*.
intros P Q R chopright.
cut (*chop* (*chop P Q*) *R* $\leftrightarrow$ *chop P* (*chop Q R*)).

```
intros H.
elim H ; intros CLR CRL.
apply CRL ; assumption.
apply chop_assoc.
Qed.

Ltac ChopAssoc :=
  match goal with
  | ⊢ (chop ?X1 (chop ?X2 ?X3)) ⇒ apply chop_assoc_left_right
  | ⊢ (chop (chop ?X1 ?X2) ?X3) ⇒ apply chop_assoc_right_left
  end.

Axiom always_rigid_chop_left : ∀ p q : Prop, rigid p → □ (chop p q → p).
Axiom always_rigid_chop_right : ∀ p q : Prop, rigid q → □ (chop p q → q).
Theorem rigid_chop_left : ∀ p q : Prop, rigid p → chop p q → p.
intros p q H ; apply IL2 ; apply always_rigid_chop_left ; assumption. Qed.
Theorem rigid_chop_right : ∀ p q : Prop, rigid q → chop p q → q.
intros p q H ; apply IL2 ; apply always_rigid_chop_right ; assumption. Qed.
Definition R_left := rigid_chop_left.
Definition R_right := rigid_chop_right.

Axiom
  always_exists_chop_left :
    ∀ (p : R → Prop) (q : Prop),
    □ (chop (∃ x : _, p x) q → ∃ x : _, chop (p x) q).
Axiom
  always_exists_chop_right :
    ∀ (p : Prop) (q : R → Prop),
    □ (chop p (∃ x : _, q x) → ∃ x : _, chop p (q x)).
Theorem
  exists_chop_left :
    ∀ (p : R → Prop) (q : Prop),
    chop (∃ x : _, p x) q → ∃ x : _, chop (p x) q.
intros p q ; apply IL2 ; apply always_exists_chop_left. Qed.
Theorem
  exists_chop_right :
    ∀ (p : Prop) (q : R → Prop),
    chop p (∃ x : _, q x) → ∃ x : _, chop p (q x).
intros p q ; apply IL2 ; apply always_exists_chop_right. Qed.
Definition B_left := exists_chop_left.
Definition B_right := exists_chop_right.

Axiom
  always_lx_chop_not_right :
    ∀ (x : R) (p : Prop), □ (chop (ℓ = x) p → ¬ chop (ℓ = x) (˜ p)).
Axiom
  always_lx_chop_not_left :
```

$\forall\ (x : R)\ (p : Prop), \Box\ (chop\ p\ (\ell = x) \rightarrow \neg\ chop\ (\tilde{}\ p)\ (\ell = x)).$

Theorem
$lx\_chop\_not\_right$:
$\forall\ (x : R)\ (p : Prop), chop\ (\ell = x)\ p \rightarrow \neg\ chop\ (\ell = x)\ (\tilde{}\ p).$

intros x p ; *apply IL2* ; *apply always_lx_chop_not_right*. Qed.

Theorem
$lx\_chop\_not\_left$:
$\forall\ (x : R)\ (p : Prop), chop\ p\ (\ell = x) \rightarrow \neg\ chop\ (\tilde{}\ p)\ (\ell = x).$

intros x p ; *apply IL2* ; *apply always_lx_chop_not_left*. Qed.

Definition *L1_right* := *lx_chop_not_right*.

Definition *L1_left* := *lx_chop_not_left*.

Axiom
$always\_interval\_chopping$:
$\forall\ x\ y : R,$
$\Box\ ((x \geq 0)\%R \wedge (y \geq 0)\%R \rightarrow (\ell = (x + y)\%R \leftrightarrow chop\ (\ell = x)\ (\ell = y))).$

Theorem
$interval\_chopping$:
$\forall\ x\ y : R,$
$(x \geq 0)\%R \wedge (y \geq 0)\%R \rightarrow (\ell = (x + y)\%R \leftrightarrow chop\ (\ell = x)\ (\ell = y)).$

intros x y ; *apply IL2* ; *apply always_interval_chopping*. Qed.

Definition *L2* := *interval_chopping*.

Axiom $always\_add\_point\_left : \forall\ p : Prop, \Box\ (p \rightarrow chop\ p\ (\ell = 0\%R)).$

Axiom $always\_add\_point\_right : \forall\ p : Prop, \Box\ (p \rightarrow chop\ (\ell = 0\%R)\ p).$

Theorem $add\_point\_left : \forall\ p : Prop, p \rightarrow chop\ p\ (\ell = 0\%R).$

intros p ; *apply IL2* ; *apply always_add_point_left*. Qed.

Theorem $add\_point\_right : \forall\ p : Prop, p \rightarrow chop\ (\ell = 0\%R)\ p.$

intros p ; *apply IL2* ; *apply always_add_point_right*. Qed.

Definition *L3_left* := *add_point_left*.

Definition *L3_right* := *add_point_right*.

Axiom
$quantification$:
$\forall\ (p : R \rightarrow Prop)\ (x : R),$
$chop\_free\ (p\ 0\%R) \vee rigid\_term\ x \rightarrow all\ (fun\ y \Rightarrow p\ y) \rightarrow p\ x.$

Ltac *Quantification term* := *apply quantification with* (*x* := *term*).

Axiom
$monotony\_left$:
$\forall\ p\ q : Prop, \Box\ (p \rightarrow q) \rightarrow \forall\ r : Prop, chop\ p\ r \rightarrow chop\ q\ r.$

Axiom
$monotony\_right$:
$\forall\ p\ q : Prop, \Box\ (p \rightarrow q) \rightarrow \forall\ r : Prop, chop\ r\ p \rightarrow chop\ r\ q.$

Definition *monotone_chop* := *monotony_left*.

Definition *chop_monotone* := *monotony_right*.

Ltac *Monotony* :=

```
match goal with
| _ :(chop ?X1 (chop ?X2 ?X3)) ⊢ (chop (chop ?X1 ?X4) ?X5) ⇒
      ChopAssoc ; Monotony
| _ :(chop (chop ?X1 ?X2) ?X3) ⊢ (chop ?X4 (chop ?X5 ?X3)) ⇒
      ChopAssoc ; Monotony
| hyp :(chop (chop ?X1 ?X2) ?X3) ⊢ (chop ?X1 (chop ?X4 ?X5)) ⇒
      cut (chop X1 (chop X2 X3)) ;
        [ clear hyp ; intros ; Monotony | ChopAssoc ; assumption ]
| hyp :(chop ?X1 (chop ?X2 ?X3)) ⊢ (chop (chop ?X4 ?X5) ?X3) ⇒
      cut (chop (chop X1 X2) X3) ;
        [ clear hyp ; intros ; Monotony | ChopAssoc ; assumption ]
| _ :(chop ?X1 ?X3) ⊢ (chop ?X2 ?X3) ⇒
      apply monotony_left with (p := X1) ; [ idtac | assumption ]
| _ :(chop ?X3 ?X1) ⊢ (chop ?X3 ?X2) ⇒
      apply monotony_right with (p := X1) ; [ idtac | assumption ]
end.
```

Finally we complete the axioms of classical logic with their necessitated counterpart

```
Axiom always_S : ∀ p q : Prop, □ (p→q→p).
Axiom always_K : ∀ p q r : Prop, □ ((p→q) → (p→q→r) → (p→r)).

Axiom always_and_ind : ∀ p q r : Prop, □ ( (p → q → r) → p ∧ q → r).
Axiom always_and : ∀ p q : Prop, □ (p → q → p ∧ q).

Theorem always_id : ∀ p : Prop, □ (p→p).
intros.
apply IL1 with (p->(p→p)->p).
2 :apply always_S.
apply IL1 with (p→p→p).
apply always_K.
apply always_S.
Qed.

Theorem always_Sprime : ∀ p q : Prop, □ (p → q → q).
intros.
apply IL1 with (q→q).
apply always_S.
apply always_id.
Qed.

Theorem always_proj1 : ∀ p q :Prop, □ (p ∧ q → p).
intros p q.
apply IL1 with (p→q→p).
apply always_and_ind.
apply always_S.
Qed.

Theorem always_proj2 : ∀ p q :Prop, □ (p ∧ q →q).
```

intros p q.
cut ([] ($q \wedge p \to q$)).
intros.
apply IL1 with ($p \to q \to q$).
apply always_and_ind.
apply always_Sprime.
apply IL1 with ($q \to p \to q$).
apply always_and_ind.
apply always_S.
Qed.

Theorem *always_false* : $False \to \Box False$.
apply False_ind ; *assumption*.
Qed.

Axiom *always_True_ind* : $\forall$ *p* : *Prop*, $\Box$ ($p \to True \to p$).
Axiom *always_True* : $\Box True$.

Axiom *always_or_ind* : $\forall$ *p q r* : *Prop*, $\Box$ (($p \to r$) $\to$ ($q \to r$) $\to p \vee q \to r$).
Axiom *always_or_introl* : $\forall$ *p q* : *Prop*, $\Box$ ($p \to p \vee q$).
Axiom *always_or_intror* : $\forall$ *p q* : *Prop*, $\Box$ ($q \to p \vee q$).

Module DC_base

Require Import *IL_base*.

Require Export *Reals*.

Inductive *DCState* : *Set* :=
| *SZero* : *DCState*
| *SOne* : *DCState*
| *SOr* : *DCState* $\to$ *DCState* $\to$ *DCState*
| *SNot* : *DCState* $\to$ *DCState*
| *SAnd* : *DCState* $\to$ *DCState* $\to$ *DCState*
| *SImply* : *DCState* $\to$ *DCState* $\to$ *DCState*
| *SIff* : *DCState* $\to$ *DCState* $\to$ *DCState*.

Axiom
SAnd_def : $\forall$ *S T* : *DCState*, *SAnd S T* = *SNot* (*SOr* (*SNot S*) (*SNot T*)).
Axiom *SImply_def* : $\forall$ *S T* : *DCState*, *SImply S T* = *SOr* (*SNot S*) *T*.
Axiom
SIff_def : $\forall$ *S T* : *DCState*, *SIff S T* = *SAnd* (*SImply S T*) (*SImply T S*).

Parameter *SVar* : *DCState* $\to$ *DCState*.

Parameter *duration_eq* :*DCState* $\to$ *R* $\to$ *Prop*.
Parameter *duration_lt* :*DCState* $\to$ *R* $\to$ *Prop*.

Definition *duration_le S x* :=(*duration_eq S x*) $\vee$ (*duration_lt S x*).

Definition *duration_gt S x* :=~(*duration_le S x*) .
Definition *duration_ge S x* :=(*duration_eq S x*) ∨ (*duration_gt S x*).

Fixpoint *proplogic* (*S* : *DCState*) : *Prop* :=
  match S with
  | *SZero* ⇒ *False*
  | *SOne* ⇒ *True*
  | *SOr S1 S2* ⇒ *proplogic S1* ∨ *proplogic S2*
  | *SNot S1* ⇒ ¬ *proplogic S1*
  | *SAnd S1 S2* ⇒ *proplogic S1* ∧ *proplogic S2*
  | *SImply S1 S2* ⇒ *proplogic S1* → *proplogic S2*
  | *SIff S1 S2* ⇒ *proplogic S1* ↔ *proplogic S2*
  end.

Delimit Scope states_scope with S.
Bind Scope states_scope with DCState.
Open Scope states_scope.
Notation "1" :=(*SOne*) (*at level* 70, *no associativity*) : *states_scope*.
Notation "0" :=(*SZero*) (*at level* 70, *no associativity*) : *states_scope*.

Notation "¬ *A*" :=(*SNot A*) (*at level* 75, *right associativity*) : *states_scope*.
Notation "*A* ∧ *B*" :=(*SAnd A B*) (*at level* 80, *right associativity*) : *states_scope*.
Notation "*A* ∨ *B*" :=(*SOr A B*) (*at level* 85, *right associativity*) : *states_scope*.
Notation "*A* –> *B*" :=(*SImply A B*) (*at level* 90, *right associativity*) : *states_scope*.
Notation "*A* ↔ *B*" :=(*SIff A B*) (*at level* 95, *no associativity*) : *states_scope*.

Notation " ! *s* !" :=(*proplogic s*) (*at level* 0, *no associativity*).

Notation "# *s* #= *x*" :=(*duration_eq s x*) (*at level* 0, *no associativity*).
Notation "# *s* #< *x*" :=(*duration_lt s x*) (*at level* 0, *no associativity*).
Notation "# *s* #<= *x*" :=(*duration_le s x*) (*at level* 0, *no associativity*).
Notation "# *s* #> *x*" :=(*duration_gt s x*) (*at level* 0, *no associativity*).
Notation "# *s* #>= *x*" :=(*duration_ge s x*) (*at level* 0, *no associativity*).
Notation "# *s* #<> *x*" :=(~(*duration_eq s x*))(*at level* 0, *no associativity*).

Close Scope states_scope.

Module DC_axioms

Require Import *IL_base*.
Require Import *DC_base*.
Require Import *Common_functions*.

Definition *pr* (*S* : *DCState*) := ∀ (*x* :*R*), #*S*#=*x* ∧ (*l*= *x*) ∧ (*x*>0)%*R*.
Notation "[[*s*]]" :=(*pr s*) (*at level* 0, *no associativity*).

Open Scope states_scope.

Axiom *DCA1* : #0#= 0.

```
Axiom DCA2 : ∀ (x :R), #1#= x ↔ l=x.
Axiom DCA3 : ∀ s : DCState, #s#>= 0.
Axiom
  DCA4 :
    ∀ (s1 s2 : DCState) (x1 x2 and12 or12 :R),
    #s1#= x1 → #s2#=x2 → #s1 ∧ s2#=and12 → #s1 ∨ s2#=or12 →
    (x1 + x2 = and12 + or12)%R.
Axiom
  DCA5 :
    ∀ (x y : R) (s : DCState),
    #s#=x ^^ #s#=y → #s#= (x+y).
Axiom
  DCA6 :
    ∀ s1 s2 : DCState,
    (proplogic s1 ↔ proplogic s2) → (∀ (x :R), #s1#=x ↔ #s2#=x).

Close Scope states_scope.
Require Export List.

Fixpoint disjunction (LS : list DCState) : DCState :=
  match LS with
  | state : : sequel ⇒
        match sequel with
        | nil ⇒ state
        | _ ⇒ SOr state (disjunction sequel)
        end
  | nil ⇒ SZero
  end.

Fixpoint chop_sharing_left (X : Prop) (LS : list DCState) {struct LS} :
 Prop :=
  match LS with
  | state : : sequel ⇒
        match sequel with
        | nil ⇒ chop X (pr state)
        | _ ⇒ chop X (pr state) ∨ (chop_sharing_left X sequel)
        end
  | nil ⇒ False
  end.

Fixpoint chop_sharing_right (X : Prop) (LS : list DCState) {struct LS} :
 Prop :=
  match LS with
  | state : : sequel ⇒
        match sequel with
        | nil ⇒ chop (pr state) X
        | _ ⇒ chop (pr state) X ∨ chop_sharing_right X sequel
```

```
      end
  | nil ⇒ False
  end.
Axiom
  IR1 :
    ∀ (H : Prop → Prop) (LS : list DCState),
    (proplogic (disjunction LS) ↔ True) →
    H point →
    (∀ X : Prop, H X → H (X ∨ chop_sharing_left X LS)) → H True.
Axiom
  IR2 :
    ∀ (H : Prop → Prop) (LS : list DCState),
    (proplogic (disjunction LS) ↔ True) →
    H point →
    (∀ X : Prop, H X → H (X ∨ chop_sharing_right X LS)) → H True.
Require Import Classical.
Theorem IR1_simple :
 ∀ (H : Prop → Prop) (S : DCState),
 H point →
 (∀ X : Prop, H X → H (X ∨ chop X (pr S) ∨ chop X (pr (SNot S)))) →
 H True.
intros H S Hpoint IR1hypo.
apply IR1 with (LS := S : : SNot S : : nil) ;
 [ simpl in ⊢ × ; split ; [ tauto | intros ; apply classic ]
 | assumption
 | assumption ].
Qed.
Theorem IR2_simple :
 ∀ (H : Prop → Prop) (S : DCState),
 H point →
 (∀ X : Prop, H X → H (X ∨ chop (pr S) X ∨ chop (pr (SNot S)) X)) →
 H True.
intros H S Hpoint IR2hypo.
apply IR2 with (LS := S : : SNot S : : nil) ;
 [ simpl in ⊢ × ; split ; [ tauto | intros ; apply classic ]
 | assumption
 | assumption ].
Qed.
```

Bibliographie

[Abr74] Jean-Raymond Abrial. Data semantics. In *IFIP Working Conference Data Base Management*, pages 1–60, 1974.

[Abr84a] Jean-Raymond Abrial. The mathematical construction of a program. *Science of Computer Programming*, 4(1), April 1984.

[Abr84b] Jean-Raymond Abrial. Specification or how to give reality to abstraction. *Technique et Science Informatiques*, 1984.

[Abr88] J. R. Abrial. The B tool. In R. Bloomfield ; L. Marshall ; R. Jones, editor, *Proceedings of the 2nd VDM-Europe Symposium*, volume 328 of *LNCS*, pages 86–87, Berlin, September 1988. Springer.

[Abr96a] Jean-Raymond Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[Abr96b] Jean-Raymond Abrial. Extending B without changing it (for developing distributed systems). In Habrias [Hab96], pages 169–191.

[Abr96c] Jean-Raymond Abrial. A steam-boiler control specification problem. In Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors, *Formal Methods for Industrial Applications : Specifying and Programming the Steam Boiler*, volume 1165 of *Lecture Notes in Computer Science*, pages 500–509. Springer-Verlag, Berlin, Germany, 1996, 1996.

[Abr00] Jean-Raymond Abrial. Event driven sequential program construction. MATISSE project, October 2000.

[ACM03a] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal aspects of computing*, 14(3), April 2003.

[ACM03b] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Formal derivation of spanning trees algorithms. In Bert et al. [BBKW03], pages 457 – 476.

[ACM05] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in event_b. In Helen Treharne, Steve King, and Martin Henson, editors, *ZB 2005 : Formal Specification and Development in Z and B : 4th International Conference of B and Z Users, Guilford, UK*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer Verlag, Apr 2005.

[AD94] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[ADE01] ADER/LORIA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, ADER/LORIA Nancy – France, June 2001. ADER/LORIA.

[AFA03] IRISA. *Approches Formelles dans l'Assistance au Développement de Logiciels*, IRISA Rennes – France, January 2003. IRISA.

[AGM92] Samson Abramsky, Dov M. Gabbay, and T.S.E Maibaum, editors. *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[AM97] Jean-Raymond Abrial and Louis Mussat. Specification and design of a transmission protocol by successive refinements using B. In Manfred Broy and Birgit Schieder, editors, *Mathematical Methods in Program Development*, volume 158 of *NATO ASI Series F : Computer ans Systems Sciences*, pages 129–200. Springer, 1997.

[AM98] Jean-Raymond Abrial and L. Mussat. Introducing dynamic constraints in B. In Bert [Ber98], pages 83–128.

[AO97] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of sequential and concurrent programs (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.

[Bar92] Henk Barendregt. *Handbook of Logic in Computer Science*, chapter Lambda Calculi with Types, pages 117–309. Clarendon Press, 1992. (Technical Report 91-19, Catholic University Nijmegen, 1991).

[BB99] Martin Büchi and Ralph Back. Compositional symmetric sharing in B. In Wing et al. [WWD99], pages 431–451.

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Meteor : A successful application of B in a large project. In *World Congress on Formal Methods 1999*, number 1709 in Lecture Notes in Computer Science, pages 369–387. Springer Verlag, september 1999.

[BBKW03] Didier Bert, Jonathan P. Bowen, S. King, and M. Waldén, editors. *ZB'2003 – Formal Specification and Development in Z and B,International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science (Springer-Verlag)*, Turku, Finland, June 2003. Springer.

[BDM97] Patrick Behm, Pierre Desforges, and Fernando Mejia. *Application de la méthode B dans l'industrie ferroviaire*, chapter III, pages 59–88. Volume 20 of OFTA [OFT97], 1997.

[Ber98] Didier Bert, editor. *B'98 : The 2nd International B Conference, Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science (Springer-Verlag)*, Montpellier, April 1998. LIRRM Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier, Springer Verlag.

[Ber00] Gérard Berry. The foundations of esterel. *Proof, language, and interaction : essays in honour of Robin Milner*, pages 425–454, 2000.

[BF02] Michael Butler and Jerome Falampin. An approach to modelling and refining timing properties in B. In *RCS02 (Refinement of Critical Systems)*, Grenoble, January 2002. CERT-ONERA.

[BFM99] Jean-Paul Bodeveix, Mamoun Filali, and César Munoz. A formalization of the B method in Coq and PVS. In BUGM99 [BUG99], pages 32–48.

[BFR04] Jean-Paul Bodeveix, Mamoun Filali, and Miloud Rached. Méthodes de spécification de systèmes temps réel en B. In *FAC2004 (Formalisation des Activités Concurrentes)*, Toulouse, Mars 2004. CERT-ONERA. `http://www.cert.fr/feria/svf/FAC/2004/actes.html`.

[BHB+89] Dines Bjørner, C.A.R. Hoare, Jonathan Bowen, He Jifeng, Hans Langmaack, Ernst-Rüdiger Olderog, Ursula Martin, Victoria Stavridou, Fleming Nielson, Hanne Riis Nielson, Howard Barringer, Doug Edwards, Hans Henrik Løvengreen, Anders Ravn, and Hans Rischel. A ProCoS project description : ESPRIT BRA 3104. In *Bulletin of the European Association for Theoretical Computer Science*, volume 39, pages 60–73. EATCS, october 1989.

[BM04] Michel Bidoit and Peter D. Mosses. CASL *User Manual*. LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[BPR96] D. Bert, M.-L. Potet, and Y. Rouzaud. A study on components and assembly primitives in B. In Habrias [Hab96], pages 47–62.

[Büc98] Martin Büchi. The B bank : A complete case study. In *Proceedings of ICFEM98, the Second International Conference on Formal Engineering Methods*, pages 190–199. IEEE Press, December 1998.

[BUG99] *FM'99 – B Users Group Meeting – Applying B in an industrial context : Tools, Lessons and Techniques*. Springer-Verlag, 1999.

[Bur97] Rainer Burkhardt. *UML : Unified Modeling Language*. Addison-Wesley, 1997.

[CAS] CASL. `http://www.cofi.info/CASL.html`.

[CH85] Thierry Coquand and Gérard Huet. Constructions : A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, volume 203 of *Lecture Notes in Computer Science*, Linz, 1985. Springer-Verlag.

[CH87] Thierry Coquand and Gérard Huet. Concepts mathématiques et informatiques formalisés dans le calcul des constructions. In The Paris Logic Group, editor, *Logic Colloquium'85*, pages 123–146. Elsevier, 1987. ISBN :0-444-70211-3.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3) :95–120, february 1988.

[CMP04a] Samuel Colin, Georges Mariano, and Vincent Poirriez. Duration calculus : A real-time semantic for B. In *First International Colloquium on Theoretical Aspects of Computing*, Guiyang, China, september 2004. UNU-IIST.

[CMP04b] Samuel Colin, Georges Mariano, and Vincent Poirriez. A natural extension of B substitutions : postconditions. Technical report, LAMIH/ROI, 2004. `http://www.univ-valenciennes.fr/ROI/WP/`.

[Coq85] Thierry Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris 7, january 1985.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[CPM03] Samuel Colin, Vincent Poirriez, and Georges Mariano. Thoughts about the implementation of the duration calculus with coq. In *4th International Workshop on the Implementation of Logics*, volume Technical report ULCS-03-018. University of Liverpool, september 2003. `http://www.csc.liv.ac.uk/research/techreports/`.

[CPR+05] Samuel Colin, Dorian Petit, Jérôme Rocheteau, Rafaël Marcano, Georges Mariano, and Vincent Poirriez. BRILLANT : An open source and XML-based platform for rigourous software development. In *SEFM (Software Engineering and Formal Methods)*, Koblenz, Germany, september 2005. AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press.

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

[Dij71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1 :115–138, 1971. `http://www.cs.utexas.edu/users/EWD/ewd03xx/EWD310.PDF`.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[Dut95a] Bruno Dutertre. Complete proof systems for first order interval temporal logic. In *Logic in Computer Science*, pages 36–43, 1995.

[Dut95b] Bruno Dutertre. On first order interval temporal logic. Technical Report CSD-TR-94-3, University of London, Department of computer science, Egham, Surrey TW20 0EX, England, february 1995.

[dW01] Paulien de Wind. Modal logic in Coq. Master's thesis, Vrije Universiteit, Amsterdam, 2001. Supervised by dr. F. van Raamsdonk.

[DY00] DCS-York, editor. *ZB'2000 – International Conference of B and Z Users*, volume 1878 of *Lecture Notes in Computer Science (Springer-Verlag)*, Helsington, York, UK YO10 5DD, August 2000.

[EH85] E. A. Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. *J. Comput. Syst. Sci.*, 30(1) :1–24, 1985.

[EN05] Jacob Enslev and Anne-Sofie Nielsen. Bounded model construction for duration calculus. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby, 2005. Supervised by Assoc. prof. Martin Franzle and Assoc. prof. Michael R. Hansen.

[EST] `http://www.esterel-technologies.com/`.

[Frä02] Martin Fränzle. Take it NP-easy : Bounded model construction for duration calculus. In Springer Verlag, editor, *International Symposium on Formal Techniques in Real-Time and Fault-Tolerant systems (FTRTFT 2002)*, volume 2469, pages 245–264, 2002.

[GH99] Dimitar P. Guelev and Dan Van Hung. Completeness and decidability of a fragment of duration calculus with iteration. In *Asian Computing Science Conference (ASIAN'99)*, volume 1742 of *LNCS*, pages 139–150, Phuket, Thailand, December 1999. Springer-Verlag. Also presented at International Conference on Mathematical Foundation of Informatics, Hanoi, October 25-28, 1999.

[Gil62] Arthur Gill. *Introduction to the Theory of Finite-state Machines*. McGraw-Hill, 1962.

[Gue98] Dimitar P. Guelev. Iteration of simple formulas in duration calculus. Technical Report 141, UNU-IIST, P.O. Box 3058, Macau, June 1998.

[Hab96] Henri Habrias, editor. *Proceedings of the 1st Conference on the B method*, Putting into Practice methods and tools for information system design, 3 rue du Maréchal Joffre, BP 34103, 44041 Nantes Cedex 1, November 1996. IRIN Institut de recherche en informatique de Nantes.

[HB95] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Series in Computer Science. Prentice Hall International, 1995.

[Hei99] Søren T. Heilmann. *Proof Support for Duration Calculus*. Phd-thesis, Department of Information Technology, Technical University of Denmark, Januar 1999.

[HJ04] Jifeng He and Naiyong Jin. Integrating variants of DC. In Zhiming Liu and Keijiro Araki, editors, *Theoretical Aspects of Computing - ICTAC 2004, First International Colloquium, Guiyang, China, September 20-24, 2004, Revised Selected Papers*, volume 3407 of *Lecture Notes in Computer Science*, pages 14–34. Springer-Verlag, 2004. ISBN :3-540-25304-1.

[HJMO03] Ahmed Hammad, Jacques Julliand, H. Mountassir, and Dieudonné Okalas Ossami. Expression en B et raffinement des systèmes réactifs temps réel. In AFADL2003 [AFA03], pages 211–226.

[HMM83] Joseph Y. Halpern, Zohar Manna, and Ben C. Moszkowski. A hardware semantics based on temporal intervals. In *10th International Colloquium on Automata, Languages and Programming*, volume LNCS 154, pages 278–291, London, UK, 1983. Springer-Verlag. ISBN :3-540-12317-2.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[HP98] Dang Van Hung and Paritosh K. Pandya. Duration calculus with weakly monotonic time. In Anders P. Ravn and Hans Rischel, editors, *5th symposium of Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98)*, volume LNCS 1486, pages 55–64, Lyngby, Denmark, september 1998. Springer-Verlag. Technical Report 122, UNU-IIST, P.O. Box 3058, Macau, September 1997.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[HZ97] Michael R. Hansen and Chaochen Zhou. Duration calculus : logical foundations. *Formal Aspects of Computing*, 9 :283–330, 1997.

[HZ04] Michael R. Hansen and Chaochen Zhou. *Duration Calculus, a formal approach to real-time systems*. Number ISBN : 3-540-40823-1 in Monographs in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[Ini04] CoFI (The Common Framework Initiative). CASL *Reference Manual*. LNCS 2960 (IFIP Series). Springer, 2004.

[Isa93] Isabelle/HOL, 1993. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[Lam90] Leslie Lamport. *win* and *sin* : Predicate transformers for concurrency. *ACM Trans. Program. Lang. Syst.*, 12(3) :396–428, 1990.

[Lam02] Leslie Lamport. *Specifying Systems : The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, July 2002.

[LBS96] K. Lano, J. Bicarregui, and A. Sanchez. Using B to design and verify controllers for chemical processing. In Habrias [Hab96], pages 237–270.

[LD96] Kevin Lano and Jeremy Dick. Development of concurrent systems in B AMN. In He Jifeng, John Cooke, and Peter Wallis, editors, *BCS-FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, july 1996.

[LFD96] Kevin Lano, J. Fiadeiro, and Jeremy Dick. Extending B AMN with concurrency. Technical report, Dept. of Computing, Imperial College, 1996.

[LL32] C. I. Lewis and C. H. Langford. *Symbolic logic*. Dover Publications, 1932. 1959 reprint.

[LMA+01] Franc Ledoux, Jean-Marc Mota, Agnès Arnould, Catherine Dubois, Pascale Le Gall, and Yves Bertrand. Spécification formelle du chanfreinage. In *AFADL'2001* [ADE01], pages 157–171.

[LUS] `http://www-verimag.imag.fr/SYNCHRONE`.

[MCM04] Rafaël Marcano, Samuel Colin, and Georges Mariano. A formal framework for uml modelling with timed constraints : Application to railway control systems. In *SVERTS : Specification and Validation of UML models for Real Time and Embedded Systems*, Lisbon, Portugal, October 2004. (in conjunction with 7th International Conference on the Unified Modeling Language, UML 2004).

[MM95] Abdelillah Mokkedem and Dominique Méry. On using temporal logic for refinement and compositional verification of concurrent systems". *Theoretical Computer Science*, 140(1) :95–138, 1995.

[Mos85] Ben C. Moszkowski. Temporal logic for multilevel reasoning about hardware. *IEEE Computer*, 18(2) :10–19, 1985.

[Mos94] Ben C. Moszkowski. Some very compositional temporal properties. In *Programming concepts, methods and calculi*, pages 307–326. Elsevier Science B.V. (North-Holland), 1994.

[Mos03] Till Mossakowski. Foundations of heterogeneous specification. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Recent Trends in Algebraic Development Techniques, 16th International Workshop, WADT 2002, Frauenchiemsee, Germany, 2002, Revised Selected Papers*, Lecture Notes in Computer Science, pages 359–375. Springer Verlag, London, 2003.

[OFT97] OFTA, editor. *Application des techniques formelles au logiciel*, volume 20. Observatoire Français des Techniques Avancéees & Lavoisier TEC & DOC, 1997.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6 :319–340, 1976.

[Pan01] Paritosh K. Pandya. Specifying and deciding quantified discrete-time duration calculus formulae using dcvalid. In *RT-TOOLS'2001*, Aalborg, August 2001. (affiliated with CONCUR 2001). Technical report TCS-00-PKP-1, Tata Institute of Fundamental Research, Mumbai, 2000.

[Pet03] Dorian Petit. *Génération automatique de composants logiciels sûrs à partir de spécifications formelles B*. Thèse de doctorat, Université de Valenciennes et du Hainaut-Cambrésis, December 2003.

[PHS03] Henrik Pilegaard, Michael R. Hansen, and Robin Sharp. An approach to analyzing availability properties of security protocols. *Nord. J. Comput.*, 10(4) :337–, 2003.

[PR95] Paritosh K. Pandya and Y. Ramakrishna. A recursive duration calculus. Technical report, TIFR, Mumbai, 1995. Technical Report CS-95/3.

[PR98] Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the B method. In Bert [Ber98], pages 46–65.

[Qiw96] Xu Qiwen. On compositionality in refining concurrent systems. In He Jifeng, John Cooke, and Peter Wallis, editors, *Proceedings of the BCS FACS 7th Refinement Workshop*, Electronic Workshops in Computing. Springer-Verlag, Berlin, Germany, 1996.

[RAI85] The RAISE typechecker, 1985. `http://www.iist.unu.edu/home/Unuiist/newrh/III/3/1/page.html`.

[Ras02] Thomas Marthedal Rasmussen. *Interval Logic - Proof Theory and Theorem Proving*. PhD thesis, Informatics and Mathematical Modeling, Technical University of Denmark, january 2002.

[RCMP04] Jérôme Rocheteau, Samuel Colin, Georges Mariano, and Vincent Poirriez. Évaluation de l'extensibilité de phox : B/phox un assistant de preuves pour b. In *JFLA*, pages 139–153, jan 2004. `http://pauillac.inria.fr/jfla/2004/`.

[ROD05] 2005. `http://rodin.cs.ncl.ac.uk/`.

[Sai96] Hossein Saiedian. An invitation to formal methods. *Computer*, 29(4) :16–17, 1996.

[SH01] François Siewe and Dan Van Hung. Deriving real-time programs from duration calculus specifications. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001)*, volume LNCS 2144, pages 92–97, Livingston-Edinburgh, Scotland, september 2001. Springer-Verlag. (Technical Report 222, UNU-IIST, P.O. Box 3058, Macau, December 2000).

[SHZC04] Francois Siewe, Dang Van Hung, Hussein Zedan, and Antonio Cau. A formal design technique for real-time embedded systems development using duration calculus. In A. Weitzenfeld and A. Barrera, editors, *1st IEEE Latin American Robotics Symposium*, pages 60–65, Mexico City, Mexico, october 2004.

[Ska94] Jens Ulrik Skakkebæk. *A Verification Assistant for a Real-Time Logic*. Phd-thesis, Department of Computer Science, Technical University of Denmark, 1994. Also available as Technical Report ID-TR : 1994-150.

[SL00] Denis Sabatier and Pierre Lartigue. The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. In *Formal Method in System Design*, volume 17, pages 245–272. Kluwer academic publisher, december 2000.

[SM02] Lutz Schröder and Till Mossakowski. HasCASL : towards integrated specification and development of functional programs. In Hélène Kirchner and Christophe Ringeissen, editors, *Algebraic Methodology And Software Technology (AMAST 2002)*, volume 2422 of *Lecture Notes in Computer Science*, pages 99–116. Springer ; Berlin ; http ://www.springer.de, 2002.

[ST02] Steve Schneider and Helen Treharne. Communicating B machines. In ZB02 [ZB002], pages 416–435.

[TS99a] Helen Treharne and Steve Schneider. Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, Royal Holloway, Department of Computer Science, University of London, Egham, Surrey TW20 0EX, England, June 1999.

[TS99b] Helen Treharne and Steve Schneider. Using a process algebra to control B OPERATIONS. Technical Report CSD-TR-99-01, Royal Holloway, Department of Computer Science, University of London, Egham, Surrey TW20 0EX, England, June 1999.

[TS00] Helen Treharne and Steve Schneider. How to drive a B machine. In DCS-York [DY00], pages 188–208.

[UML] UML. `http://www.uml.org`.

[Var96] Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency : Structure versus Automata*, volume 1043 of LNCS, pages 238–265. Springer-Verlag, 1996.

[WWD99] Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors. *Proceedings of FM'99 : World Congress on Formal Methods*, number 1709 in Lecture Notes in Computer Science (Springer-Verlag). Springer Verlag, September 1999.

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

[ZB002] LSR-IMAG. *ZB'2002 – Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science (Springer-Verlag)*, Grenoble, France, January 2002.

[ZGN99] Chaochen Zhou, Dimitar P. Guelev, and Z. Naijun. A higher-order duration calculus. In *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).

[ZH98] Chaochen Zhou and Michael R. Hansen. An adequate first order interval logic. *Lecture Notes in Computer Science*, 1536 :584–608, 1998.

[ZHR91] Chaochen Zhou, C.A.R. Hoare, and Anders P. Ravn. A calculus of durations. In *Information Processing Letters*, volume 10(5), pages 269–276. Elsevier, December 1991.

[ZL94] Chaochen Zhou and Xiaoshan Li. A mean value duration calculus. In A.W. Roscoe, editor, *A Classical Mind, Essays in Honour of C.A.R. Hoare*, pages 432–451. Prentice-Hall International, 1994.

[ZM95] Ying Zhang and Alan K. Mackworth. Synthesis of hybrid constraint-based controllers. In *Hybrid Systems II*, volume 999 of *LNCS*, pages 552–567, London, UK, 1995. Springer-Verlag.

[ZWR95] Chaochen Zhou, J. Wang, and Anders P. Ravn. A duration calculus with infinite intervals. In H. Reichel, editor, *Fundamentals of Computing Theory*, volume 965 of LNCS, pages 16–41. Springer-Verlag, Lübeck, Germany, 1995.

Résumé

Dans le domaine des systèmes informatisés où la fiabilité est la première priorité, les méthodes formelles ont prouvé leur efficacité pour la conception de logiciels sûrs. La dépendance à de tels systèmes augmente, et les contraintes rencontrées se font plus diverses et précises, en particulier les contraintes temporelles. Certaines méthodes formelles, notamment la méthode B, rendent la conception malaisée sous de telles contraintes, puisqu'elles n'ont pas été prévues pour cela à l'origine.

Nous nous proposons donc d'étendre la méthode B pour lui permettre de spécifier et valider des systèmes à contraintes temporelles complexes. Nous utilisons pour ce faire des calculs de durées pour exprimer la sémantique du langage B et en déduire une extension conservative qui permet de l'utiliser à la fois dans son cadre d'origine et dans le cadre de systèmes à contraintes temporelles.

Nous nous penchons également sur le problème de l'utilisation d'un outil de preuve générique pour valider des formules de calcul des durées. La généricité de ce type d'outil répond à la multiplication des méthodes formelles, mais pose le problème de l'intégration des fondations mathématiques de ces méthodes à un outil générique. Nous proposons donc d'étudier la mise en oeuvre en plongement léger du calcul des durées dans l'assistant de preuve Coq. Nous en déduisons un retour sur expérience de la définition d'une logique modale particulière dans un outil à vocation générique.

Mots-clés: méthode formelle, méthode B, logique temporelle, calcul des durées, assistant de preuve, Coq

Abstract

In the field of automated systems where reliability is the first requirement, formal methods proved to be efficient for the design of safe software. The dependency towards such systems is increasing, and the constraints to be met by these systems become more and more various and precise, particularly timed constraints. Some formal methods, notably the B method, make designing difficult under these constraints, because this is not what they have been made for in the first place.

We therefore propose to extend the B method so that it can help specifying and validating systems with complex timed constraints. We use calculi of durations in order to express the semantics of the B language and deduce a conservative extension allowing its use in both its original context and in the context of time-constrained systems.

We also study the problem of using a generic proof tool for validating duration calculus formulas. The genericity of this kind of tool helps answering the growing number of formal methods, but introduces the problem of adapting the mathematical foundations of such formal methods to a generic tool. We thus propose to study the implementation of duration calculus as a shallow embedding in the Coq proof assistant. We draw from it more general conclusions about the implementation of a particular modal logic in a generic-oriented tool.

Keywords: formal method, B method, temporal logic, duration calculus, proof assistant, Coq