La logique temporelle

La logique temporelle ou modale trouve de nombreuses applications en informatique : traitement du langage naturel, représentation des connaissances [16], spécification de systèmes [19], etc. Cette logique étend la logique conventionnelle (propositionnelle) en introduisant de nouveaux opérateurs qui permettent de référencer plusieurs instants différents de l'exécution d'un système. La logique temporelle permet ainsi d'exprimer des propriétés dynamiques portant sur plusieurs états des traces d'exécution d'un système. Dans la littérature, nous distinguons les deux classes principales de logique temporelle suivantes :

- La logique linéaire comme LTL (*Linear Temporal Logic*) permettant d'exprimer des propriétés portant sur des chemins individuels issus de l'état initial du programme.
- La logique arborescente comme CTL (*Computation Tree Logic*) permettant d'exprimer des propriétés portant sur les arbres d'exécution issus de l'état initial du programme.

Il existe aussi la logique CTL* qui permet de décrire à la fois des propriétés linéaires et arborescentes. La logique CTL* est souvent considérée comme une fusion des logiques LTL et CTL puisque les formules LTL et CTL sont des sous-ensembles des formules CTL*.

2.1.1 La logique LTL

LTL est définie sur un ensemble de variables propositionnelles $\{p, q, \dots\}$, de connecteurs logiques propositionnels habituels $\neg, \wedge, \vee, \rightarrow$ et de connecteurs temporels (**G**, **F**, **U** et **X**). La grammaire associée à la logique LTL est comme suit :

- les formules atomiques :

$$\psi, \phi := p \mid q \mid \dots \mid true \mid false$$

- les formules construites à partir de connecteurs propositionnels :

$$\psi := \phi \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \Rightarrow \phi \mid \phi \Leftrightarrow \phi$$

- les formules construites à partir de connecteurs temporels :

- $\mathbf{G}\phi$: (**G** comme Globally) spécifie que ϕ est toujours vrai
- $\mathbf{F}\phi$: (**F** comme Futur) indique que ϕ sera vrai dans le futur
- $\phi\mathbf{U}\psi$: (**U** comme Until) spécifie que ϕ reste vrai jusqu'à ce que ψ devienne vrai. Cet opérateur impose que le système atteigne un état où ψ devient vrai.
- $\mathbf{X}\phi$: (**X** comme Next) spécifie que ϕ sera vrai à l'étape suivante de l'exécution

2.1.2 La logique CTL

Contrairement à la logique LTL qui considère une vue linéaire du système, la logique CTL (*Computation Tree Logic*) permet de prendre en compte plusieurs futurs possibles à partir d'un état donné du système. La logique CTL a donc une vision arborescente de l'évolution du système. Elle étend la grammaire de LTL en incluant deux quantificateurs de chemins : l'un universel (*A*) et un second existentiel (*E*).

- **A** signifie : "pour tous les chemins possibles de calcul à partir de l'état courant"
- **E** signifie : "pour un certain chemin de calcul à partir de l'état courant"

Les formules CTL sont donc toutes précédées des quantificateurs **A** ou **E** et sont définies par la grammaire suivante :

$$\phi, \psi := \mathbf{E}\mathbf{G}\phi \mid \mathbf{E}\mathbf{F}\phi \mid \mathbf{E}\phi\mathbf{U}\psi \mid \mathbf{E}\mathbf{X}\phi \mid \mathbf{A}\mathbf{G}\phi \mid \mathbf{A}\mathbf{F}\phi \mid \mathbf{A}\phi\mathbf{U}\psi \mid \mathbf{A}\mathbf{X}\phi$$

avec ϕ, ψ désignant des formules LTL. La logique CTL introduit également quelques notions d'équité afin de ne considérer que les chemins où aucun événement du système ne prend indéfiniment le contrôle. Par exemple, dans un système d'allocation de ressources partagées par plusieurs utilisateurs, on ne voudrait pas s'intéresser aux chemins où un utilisateur garde la ressource indéfiniment. Pour ce faire, CTL introduit deux opérateurs $\mathbf{A}_f$ et $\mathbf{E}_f$ pour préciser qu'on ne considère que les chemins jugés équitables.

2.1.3 La logique CTL*

La logique CTL* est la logique temporelle la plus générale. Comme en LTL et CTL, une formule CTL* est donc construite à partir de propositions atomiques, de connecteurs propositionnels, de connecteurs temporels et de quantificateurs de chemins. Les opérateurs propositionnels et les quantificateurs de chemins restent les mêmes que dans CTL. Ajouté à ces différents connecteurs, CTL* introduit un cinquième connecteur temporel noté **W** et dont la sémantique est comme suit : étant donné deux prédicats p et q , $p\mathbf{W}q$ signifie que p doit rester vrai jusqu'à ce que q soit établi. Contrairement au connecteur **U**, le connecteur **W** n'impose pas que le prédicat q devienne vrai durant l'exécution du programme. En CTL*, on distingue les formules d'états de celles de chemins :

- Une formule d'état ϕ est définie comme suit :

$$\phi := p \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid A\psi \mid E\psi$$

avec ψ désignant une formule de chemin.

- Une formule de chemin ψ est quant à elle définie par :

$$\psi := \phi \mid \neg\psi \mid \psi \wedge \psi \mid \psi \vee \psi \mid \mathbf{X}\psi \mid \mathbf{F}\psi \mid \mathbf{G}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{W}\psi$$

Notons, qu'une nouvelle notation des connecteurs temporels **F** et **G**, très utilisée dans la littérature, a été introduite dans la logique CTL*. Les connecteurs **F** et **G** sont notés $\diamond$ et $\square$ respectivement. Ces nouvelles notations s'expriment comme suit :

- $\diamond p = \top\mathbf{U}p$ signifie que p sera vrai dans le futur.
- $\square p = \neg\diamond\neg p$ signifie que p est toujours vrai. Par conséquent, l'opposé de p ne sera jamais vrai dans le futur.

2.2 Vérification de propriétés de vivacité d'UNITY avec Event B

Dans [4], les auteurs proposent une méthode de vérification de propriétés de vivacité sur un système spécifié en Event B [2]. Pour ce faire, ils se basent sur l'opérateur *LeadsTo* défini dans la logique UNITY [18] en distinguant deux classes de propriétés :

1. Propriétés basiques : étant donnés deux prédicats p et q , la propriété p *ensures* q signifie que le prédicat p reste vérifié tant que le prédicat q ne l'est pas encore. De

plus, elle impose que q soit vérifié dans le futur. Cette propriété se décline en deux cas :

- (a) Propriété basique avec progrès minimum : dans ce cas, on exige que le prédicat q soit vérifié immédiatement sur l'état suivant l'état où le prédicat p est vrai. Cette propriété exige également la possibilité d'évolution du système en tout état qui satisfait p , *i.e.*, il doit toujours exister un événement dont la précondition est vraie. Formellement :

$$p \text{ ensures } q \equiv (\forall t.(p \wedge \neg q \wedge \text{grd}.t \Rightarrow \text{wp}.t.q)) \wedge (\exists s.(p \wedge \neg q \Rightarrow \text{grd}.s))$$

avec :

- t et s deux événements quelconques du système.
 - $\text{grd}.t$ et $\text{grd}.s$ les gardes de t et s respectivement.
 - $\text{wp}.t.q$ représente la plus faible précondition où t termine dans q .
- (b) Propriété basique avec équité faible : dans ce cas, on n'exige pas que q soit vrai immédiatement sur l'état suivant l'état vérifiant p . Cependant, tous les événements potentiellement exécutables (dont la précondition est vraie) doivent établir q ou préserver p . De plus, il doit toujours exister un événement exécutable qui permet d'établir q . Formellement :

$$p \text{ ensures } q \equiv (\forall t.(p \wedge \neg q \Rightarrow \text{wp}.t.(p \vee q))) \wedge (\exists s.(p \wedge \neg q \Rightarrow \text{wp}.s.q \wedge \text{grd}.s))$$

2. Propriétés générales : contrairement aux propriétés de base, les propriétés générales n'imposent pas que le prédicat p reste vrai tant que le prédicat q n'est pas vérifié. La preuve d'une propriété de vivacité générale "*LeadsTo*", notée $p \rightsquigarrow q$, doit être établie en appliquant les règles suivantes :

- Règle basique (BRL) : si la propriété basique $p \text{ ensures } q$ est vérifiée alors, nous pouvons déduire que la propriété $p \rightsquigarrow q$ est aussi vérifiée :

$$\frac{p \text{ ensures } q}{p \rightsquigarrow q}$$

- Règle de transitivité (TRA) : une propriété générale $p \rightsquigarrow q$ est vérifiée si cette dernière peut être déduite à partir de plusieurs propriétés plus élémentaires en appliquant la transitivité :

$$\frac{p \rightsquigarrow r \quad r \rightsquigarrow q}{p \rightsquigarrow q}$$

Afin donc de pouvoir prouver ces mêmes propriétés en Event B, les auteurs proposent une ré-expression des différentes règles suscitées en termes d'obligations de preuve B. La présentation de ces obligations de preuve fait l'objet des sections suivantes.

2.2.1 Preuve des propriétés basiques de vivacité

Pour prouver les propriétés basiques de vivacité, les auteurs ont défini quatre obligations de preuve qui correspondent à la définition formelle, vue précédemment, de ces propriétés. Pour cela, soient I et E l'invariant et l'ensemble des événements du système exprimé en Event B. On considère également la substitution de choix non déterministe F définie par :

$$F \hat{=} \coprod_{e \in E} e$$

Les obligations de preuve permettant d'établir une propriété basique de vivacité p *ensures* q sont définies comme suit :

- Avec progrès minimum : il suffit d'établir les deux obligations de preuve suivantes :

$$(MP0) \quad I \wedge p \wedge \neg q \Rightarrow [F]q^1$$

$$(MP1) \quad I \wedge p \wedge \neg q \Rightarrow \text{grad}(F)^2$$

L'obligation de preuve (MP0) permet de vérifier que pour tout état potentiel du système (vérifiant l'invariant I) et satisfaisant le prédicat $(p \wedge \neg q)$, alors l'exécution de chaque événement, dont la garde est vraie, mène bien à un état qui vérifie q . L'obligation de preuve (MP1), quand à elle, assure que tout état potentiel du système vérifiant le prédicat $(p \wedge \neg q)$ satisfait une des gardes des événements de l'ensemble E . Ceci permet de garantir qu'un des événements du système est exécutable. On en déduit donc que pour tous états σ_i et σ_{i+1} d'une trace d'exécution du système, si $(p \wedge \neg q)$ est vrai à l'état σ_i alors q devient vrai à l'état σ_{i+1} .

- Avec équité faible : il suffit d'établir les deux obligations de preuve suivantes :

$$(WF0) \quad I \wedge p \wedge \neg q \Rightarrow [F](p \vee q)$$

$$(WF1) \quad I \wedge p \wedge \neg q \Rightarrow \neg[F]\neg q$$

1. $[F]q \hat{=} [e_1 \square \dots \square e_n]q = \forall e. (e \in E \wedge \text{grad}(e) \Rightarrow [e]q)$

2. $\text{grad}(F) \hat{=} \text{grad}(e_1 \square \dots \square e_n) = \bigvee_{e \in E} \text{grad}(e)$

L'obligation de preuve (*WF0*) assure que pour tout état potentiel du système (vérifiant l'invariant I) et satisfaisant le prédicat $(p \wedge \neg q)$, tout événement exécutable préserve p ou établit q . De même, l'obligation de preuve (*WF1*) vérifie qu'il existe au moins un événement exécutable qui établit q . Ces deux obligations de preuve assurent donc que si le système est dans un état qui satisfait p alors il évoluera pour atteindre un état où q devient vrai. En termes de séquence d'exécution, si un état σ_i satisfait $(p \wedge \neg q)$, alors il existe un état σ_j , avec $(j > i)$, qui vérifie q . De plus, pour tout état σ_k , avec $(i \leq k \leq j)$, le prédicat $(p \wedge \neg q)$ est vrai et la garde d'un événement, au moins, qui établit q est vraie.

2.2.2 Preuve de propriétés générales de vivacité

Rappelons qu'une propriété générale de vivacité p *LeadsTo* q , notée $p \rightsquigarrow q$, signifie qu'à partir de tout état s vérifiant p , il est possible d'atteindre un état s' qui vérifie q sans exiger pour autant que p demeure vrai tant qu'on n'a pas atteint s' . Pour prouver une telle propriété, les auteurs ont repris le théorème suivant, défini dans la logique UNITY, et l'ont adapté à Event B :

$$\frac{\forall v. (p \wedge V = v \rightsquigarrow (p \wedge (V < v)) \vee q)}{p \rightsquigarrow q}$$

Ce théorème signifie que pour prouver la propriété générale de vivacité $p \rightsquigarrow q$ est vraie, il suffit d'exhiber une expression entière décroissante et strictement positive V appelée variant et de prouver que chaque transition du système décroît ce variant, tout en préservant p , ou établit q . Pour prouver la prémisse de ce théorème, on utilise la propriété basique de vivacité $((p \wedge V = v) \text{ ensures } ((p \wedge (V < v)) \vee q))$ qu'on prouve grâce aux différentes règles présentées dans la section précédente.

2.2.3 Exemple illustratif

Dans cette section, nous allons illustrer l'approche proposée sur un système de gestion de processus demandant une ressource critique qui est le processeur. Nous supposons que le nombre de processeurs est inférieur au nombre de processus en attente. à la création, un processus se trouve dans l'état NT (Nouvelle Tâche) et doit donc s'insérer dans une file, à

taille limitée, si cette dernière n'est pas pleine. Dans ce cas, le processus passe dans l'état EF (En File) en attendant qu'un processeur se libère. Dès qu'un processeur devient libre, ce dernier est alloué, par l'ordonnanceur, à un processus en attente et qui passera donc à l'état EL (Élu). Sur ce système de gestion de processus, on souhaiterait s'assurer qu'un processus en attente dans la file finira toujours par être élu.

Afin de modéliser ce système en Event B, on considère les ensembles et variables suivants :

- l'ensemble abstrait **PROCS** pour représenter l'ensemble de tous les processus possibles, et l'ensemble énuméré **ETATS** pour modéliser les trois états possibles d'un processus :

PROCS ;

ETATS = {NT, EF, EL}

- les variables **Procs** et **Etat** représentant respectivement l'ensemble des processus existants à tout instant et l'état de chaque processus :

Procs $\subseteq$ **PROCS**

Etat $\in$ **Procs** $\rightarrow$ **ETATS**

- la variable **File** représentant, à tout instant, l'état de la file :

File $\in$ **seq**(**Procs**)

Pour faciliter l'écriture de la spécification Event B du système, nous considérons aussi les définitions suivantes :

- **New** pour représenter l'ensemble des processus nouveaux :

New $\hat{=}$ **Etat**⁻¹[{NT}]

- **EnFile** pour représenter l'ensemble des processus présents dans la file :

EnFile $\hat{=}$ **Etat**⁻¹[{EF}]

Elu pour représenter l'ensemble des processus élus :

Elu $\hat{=}$ **Etat**⁻¹[{EL}]

Le système de gestion de processus est défini en Event B comme suit :

MACHINE

GestionProcessus

DEFINITIONS

EnFile $\hat{=}$ **Etat**⁻¹[{EF}];

New $\hat{=}$ **Etat**⁻¹[{NT}];

$\text{Elu} \hat{=} \text{Etat}^{-1}\{\{\text{EL}\}\}$

SETS

PROCS;
 ETATS = {NT, EF, EL}

CONCRETE_CONSTANTS

TAIL

PROPERTIES

TAIL $\in$ NAT

ABSTRACT_VARIABLES

Procs, Etat, File

INVARIANT

Procs $\subseteq$ PROCS $\wedge$
 Etat $\in$ Procs $\rightarrow$ ETATS $\wedge$
 File $\in$ seq(Procs)

INITIALISATION

Etat := $\emptyset$ || Procs : (Procs $\subseteq$ PROCS) || File := $\emptyset$

EVENTS

eluEven $\hat{=}$
ANY pr **WHERE** pr $\in$ Procs $\wedge$ pr $\in$ EnFile $\wedge$ File(1) = pr
THEN
 Etat(pr) := EL ||
 File := file $\downarrow$ 1
END

END;

fileEven $\hat{=}$
ANY pr **WHERE** pr $\in$ procs $\wedge$ pr $\in$ New $\wedge$ size(File) < TAIL
THEN
 Etat(pr) := EF ||
 File := File $\leftarrow$ pr
END

END;

supEven $\hat{=}$

```

ANY pr WHERE pr ∈ Procs ∧ pr ∈ Elu
THEN
    Procs := Procs - {pr} ||
    Etat := {pr} ≪ Etat
END
END
END

```

Sur cette spécification Event B, nous allons illustrer l'approche présentée pour prouver deux propriétés, l'une sous l'hypothèse du progrès minimum et une seconde sous l'équité.

2.2.3.1 Vérification sous l'hypothèse du progrès minimum

Sur le système de gestion de processus précédent, nous souhaitons vérifier qu'un processus se trouvant en tête de la file se verra allouer un processeur. Une telle propriété est exprimée comme suit :

$$(p \in \text{EnFile} \wedge \text{File}(1) = p) \rightsquigarrow p \in \text{Elu} \quad (2.1)$$

En appliquant la règle BRL, prouver la propriété (2.1) revient à établir la propriété suivante :

$$(p \in \text{EnFile} \wedge \text{File}(1) = p) \textit{ ensures } (p \in \text{Elu}) \quad (2.2)$$

Nous devons donc prouver les deux obligations de preuve suivantes :

MP0 :

$$\begin{aligned}
 & (p \in \text{EnFile}) \wedge (\text{File}(1)=p) \wedge (\neg p \in \text{Elu}) \\
 & \quad \Rightarrow \\
 & [\text{eluEven} \ [] \ \text{fileEven} \ [] \ \text{supEven}] (p \in \text{Elu})
 \end{aligned}$$

MP1 :

$$\begin{aligned}
 & (p \in \text{EnFile}) \wedge (\text{File}(1)=p) \wedge (\neg p \in \text{Elu}) \\
 & \quad \Rightarrow \\
 & \text{grd}(\text{eluEven} \ [] \ \text{fileEven} \ [] \ \text{supEven})
 \end{aligned}$$

L'obligation de preuve (*MP0*) spécifie que si un processus *p* est en tête de la file, alors l'exécution de l'un des événements de la spécification rend ce processus élu. Ceci est vrai

pour l'événement `eluEven` qui permet donc de prouver (*MP0*).

Prouvons à présent l'obligation de preuve (*MP1*) qui exprime qu'il doit exister au moins un événement du système qui soit exécutable. Cette obligation de preuve est évidente pour l'événement `eluEven` dont la garde est vraie et qui prouve aussi (*MP0*).

à présent, nous allons prouver la propriété générale de vivacité suivante :

$$p \in \text{EnFile} \rightsquigarrow p \in \text{Elu} \quad (2.3)$$

Cette propriété garantit que chaque processus en attente dans la file finira par être élu. Pour établir une telle propriété, considérons $V(p)$ une variable qui donne la position du processus p dans la file. Cette variable est définie par :

$$V(p) = \text{File}^{-1}(p)$$

En appliquant la règle citée précédemment, nous devons établir :

$$\forall n. ((p \in \text{EnFile} \wedge V(p) = n) \rightsquigarrow ((p \in \text{EnFile} \wedge V(p) < n) \vee p \in \text{Elu})) \quad (2.4)$$

En utilisation la règle BRL, cela revient à prouver :

$$(p \in \text{EnFile} \wedge V(p) = n) \textit{ ensures } (((p \in \text{EnFile}) \wedge (V(p) < n)) \vee (p \in \text{Elu})) \quad (2.5)$$

pour chaque processus p et entier n . Pour cela, nous utilisons les obligations de preuve *MP0* et *MP1* suivantes :

MP0 :

$$\begin{aligned} & p \in \text{EnFile} \wedge V(p) = n \\ & \Rightarrow \\ & [\text{eluEven}[]\text{fileEven}[]\text{supEven}]((p \in \text{EnFile} \wedge V(p) < n) \vee p \in \text{Elu}) \end{aligned}$$

MP1 :

$$\begin{aligned} & p \in \text{EnFile} \wedge V(p) = n \\ & \Rightarrow \\ & \text{grd}(\text{eluEven}) \vee \text{grd}(\text{supEven}) \vee \text{grd}(\text{fileEven}) \end{aligned}$$

Pour prouver $MP0$, il suffit de montrer qu'après chaque extraction du premier élément de la file, les positions des processus dans la file décroissent. Ce qui se déduit de la définition de la variable `File` qui est une séquence de processus et par conséquent $MP0$ est prouvée. Pour prouver $MP1$, l'invariant suivant a été défini :

$$\text{size}(\text{File}) \neq 0 \Rightarrow \exists p. (p \in \text{EnFile} \wedge \text{File}(1)=p)$$

Cet invariant exprime que si la file n'est pas vide, alors il existe toujours un processus qui est en tête. Dans ce cas, l'événement `eluEvent` devient donc exécutable. Par conséquent, tant qu'on a $(V(p)=n)$ avec $n > 0$ (ce qui veut dire que la file n'est pas vide), il existe un événement dont la précondition est toujours vérifiée si $(V(p)=n)$. L'obligation de preuve $MP1$ est donc vraie.

2.2.3.2 Vérification sous l'hypothèse d'équité faible

Considérons toujours le même système de gestion de processus défini précédemment sous l'hypothèse d'équité faible. L'objectif est de montrer comment une telle hypothèse suffit pour prouver la propriété :

$$(p \in \text{EnFile}) \rightsquigarrow (p \in \text{Elu}) \quad (2.6)$$

Pour ce faire, il suffit de prouver :

$$(p \in \text{EnFile}) \text{ ensures } (p \in \text{Elu}) \quad (2.7)$$

en appliquant la règle basique BRL. Pour prouver la propriété (2.7), il faut démontrer que les deux obligations de preuve $WF0$ et $WF1$ suivantes sont vérifiées :

$WF0$:

$$(p \in \text{EnFile} \wedge \neg p \in \text{Elu}) \Rightarrow [\text{fileEven} \ [] \ \text{eluEven} \ [] \ \text{supEven}](p \in \text{EnFile} \vee p \in \text{Elu})$$

$WF1$:

$$(p \in \text{EnFile} \wedge \neg p \in \text{Elu}) \Rightarrow \neg[\text{fileEven} \ [] \ \text{eluEven} \ [] \ \text{supEven}] \neg(p \in \text{Elu})$$

D'après ($MP0$), on sait que $(p \in \text{EnFile} \Rightarrow [\text{fileEven} \ [] \ \text{eluEven} \ [] \ \text{supEven}](p \in \text{EnFile} \vee p \in \text{Elu}))$. Nous en déduisons donc que l'obligation de preuve ($WF0$) est vérifiée. Pour prouver ($WF1$), il suffit de prouver que $(p \in \text{EnFile} \Rightarrow [\text{eluEven}] p \in \text{Elu})$. Ce prédicat est équivalent à :

$$p \in \text{EnFile} \Rightarrow \exists x. (x \in \text{Procs} \wedge x \in \text{EnFile} \wedge p \in ((\text{Etat} \triangleleft \{x \mapsto \text{EL}\})^{-1}\{\{\text{EL}\}\}))$$

Ce prédicat est vrai pour ($x=p$), l'obligation de preuve (*WF1*) est donc vérifiée.

2.3 Vérification de propriétés dynamiques en Event B

Dans [28], les auteurs proposent un ensemble de règles de vérification de propriétés dynamiques sur un système exprimé en Event B, et plus précisément des propriétés d'existence et de progression. Pour ce faire, ils ont défini un ensemble de règles d'inférence pour prouver des propriétés plus élémentaires comme la convergence, la divergence et le non-blocage.

2.3.1 Preuve de propriétés élémentaires

Dans cette section, nous décrivons les trois propriétés élémentaires qui ont été définies et qui permettent la preuve de propriétés plus élaborées comme l'existence et la progression.

2.3.1.1 Propriété LeadsTo

Soient P_1 et P_2 deux prédicats quelconques. Un système S vérifie la propriété " P_1 LeadsTo³ P_2 " ssi tout état du système qui satisfait P_1 est suivi d'un état qui vérifie P_2 . Formellement, pour tout événement du système dont la garde et la postcondition sont G et $Post_{v,v'}$ respectivement :

$$P_1 \wedge G \wedge Post_{v,v'} \Rightarrow P_2$$

avec les prédicats P_1 et P_2 dépendant des variables v et v' respectivement. La notation : $S \vdash P_1 \curvearrowright P_2$ représente l'expression P_1 LeadsTo P_2 .

2.3.1.2 Propriété de convergence

La convergence d'un système vers un prédicat P signifie qu'il n'existe pas de traces d'exécution qui se terminerait avec une suite infinie d'états qui vérifie le prédicat P . Pour

3. Notons que cette propriété, bien que portant portant le même nom, est différente de celle présentée à la section 2.2.

montrer qu'un système converge vers le prédicat P , il suffit d'exhiber une expression entière $V(v)$, dite variant et définie sur les variables v du système, et d'établir les obligations de preuve suivantes pour tout événement du système dont la garde et la postcondition sont G et $Post_{v,v'}$ respectivement :

1. l'expression $V(v)$ désigne toujours une expression entière :

$$P \wedge G \Rightarrow V(v) \in \mathbb{N}$$

2. l'exécution de tout événement décroît le variant :

$$P \wedge G \wedge Post_{v,v'} \Rightarrow V(v') < V(v)$$

La notation : $S \vdash \downarrow P$ indique que le système converge vers P .

2.3.1.3 Propriété de divergence

La divergence d'un système sur un prédicat P signifie que toutes ses traces d'exécution se terminent avec une suite infinie d'états qui vérifie le prédicat P . Pour montrer qu'un système diverge sur le prédicat P , il suffit d'exhiber une expression entière $V(v)$, dite variant et définie sur les variables v du système, et d'établir les obligations de preuve suivantes pour tout événement du système dont la garde et la postcondition sont G et $Post_{v,v'}$ respectivement :

1. l'expression $V(v)$ désigne toujours une expression entière :

$$\neg P \wedge G \Rightarrow V(v) \in \mathbb{N}$$

2. l'exécution de tout événement sur un état qui vérifie $\neg P$ décroît le variant :

$$\neg P \wedge G \wedge Post_{v,v'} \Rightarrow V(v') < V(v)$$

3. l'exécution de tout événement sur un état qui vérifie P ne fait pas accroître le variant :

$$P \wedge G \wedge Post_{v,v'} \wedge V(v) \in \mathbb{N} \Rightarrow V(v') \leq V(v)$$

En effet, comme le variant doit toujours rester positif et décroît à chaque exécution d'événement, alors on est sûr d'atteindre un état où P est vérifié. La notation : $S \vdash \nearrow P$ indique que le système diverge vers P .

2.3.1.4 Propriété de non-blocage

Le non-blocage d'un système sur un prédicat P signifie qu'il existe toujours un événement du système dont la garde est vraie :

$$P \Rightarrow \bigvee_i G_i$$

avec G_i désignant la garde d'un événement e_i du système. La notation : $S \vdash \circlearrowleft P$ indique que le système n'est pas bloqué sur P .

2.3.2 Propriété d'existence

Une propriété d'existence spécifie qu'une situation, désignée par le prédicat P , se produira toujours dans le futur. Cette propriété est notée $(\Box\Diamond P)$. L'idée intuitive pour établir une telle propriété est de montrer que les traces du système S ne se terminent pas par une séquence infinie d'états vérifiant $\neg P$. Formellement

$$\frac{S \vdash \downarrow \neg P \quad S \vdash \circlearrowleft \neg P}{S \vdash \Box\Diamond P}$$

2.3.3 Propriété de progression

Cette propriété exprime que si le système S est dans un état satisfaisant un prédicat P_1 , alors il évoluera pour atteindre un état qui vérifie un prédicat P_2 : $S \vdash \Box(P_1 \Rightarrow \Diamond P_2)$. Pour établir une telle propriété, les auteurs ont défini une règle intermédiaire qui permet de prouver la propriété $S \vdash \Box(P_1 \Rightarrow P_1 \mathbf{U} P_2)$ qui exprime que si le système est dans un état qui satisfait P_1 alors ce prédicat reste vrai tant qu'un état satisfaisant P_2 n'est pas encore atteint. De plus, cette propriété exige qu'un tel état soit atteignable. Pour prouver cette propriété, la règle suivante a été définie :

$$\frac{S \vdash (P_1 \wedge \neg P_2) \curvearrowright (P_1 \vee P_2) \quad S \vdash \Box\Diamond(\neg P_1 \vee P_2)}{S \vdash \Box(P_1 \Rightarrow (P_1 \mathbf{U} P_2))}$$

Cette règle est ensuite utilisée pour prouver la propriété de progression :

$$\frac{M \vdash \Box(P_1 \wedge \neg P_2) \Rightarrow P_3 \quad M \vdash \Box(P_3 \Rightarrow P_3 \mathbf{U} P_2)}{M \vdash \Box(P_1 \Rightarrow \Diamond P_2)}$$

Cette dernière règle signifie qu'on doit exhiber un prédicat P_3 (invariant) qui doit rester vrai tant que P_2 n'est pas encore vérifié.

2.3.4 Application à un cas d'étude

Pour illustrer cette approche, soit le modèle du lecteur/rédacteur représenté par deux variables l (pour lecteur) et r (pour rédacteur). Le tampon partagé est modélisé par l'intervalle $[l+1, r]$ et dénote les données écrites mais pas encore lues. On pose comme hypothèse que la taille du tampon est de 3 :

$$0 \leq r - l \leq 3$$

nous obtenons alors la spécification B suivante :

MACHINE

LectRedac

ABSTRACT_VARIABLES

l, r

INVARIANT

$l \in \text{NAT} \wedge r \in \text{NAT} \wedge$

$r-l \geq 0 \wedge r-l \leq 3$

EVENTS

Lire $\hat{=}$

WHEN $l < r$ **THEN**

$l := l+1$

END ;

Ecrire $\hat{=}$

WHEN $r < l+3$ **THEN**

$r := r+1$

END

END

Sur ce système on se propose de prouver une propriété d'existence et une seconde de progression.

Propriété d'existence On souhaiterait prouver sur le système *LectRedac* précédent la propriété d'existence suivante : $\Box\Diamond(l \geq L)$ avec L désignant un entier naturel positif. Pour ce faire, nous devons donc établir :

$$LectRedac \vdash \Downarrow (l < L) \quad (a)$$

$$LectRedac \vdash \Updownarrow (l < L) \quad (b)$$

Preuve de (a) : pour établir (a), il suffit de choisir comme variant $V = (L - l) + (L + 3 - r)$. En effet, ce variant décroît à chaque lecture ou écriture et est toujours positif ou nul.

Preuve de (b) : on doit prouver la formule $(l < L \Rightarrow l < r \vee r < l + 3)$. Cette formule se déduit directement de l'invariant du système.

Propriété de progression On se propose de prouver que toute donnée écrite sera lue dans le futur $\Box(r = L \Rightarrow \Diamond l = L)$. Pour ce faire, nous devons établir :

$$LectRedac \vdash \Box((r = L \wedge l \neq L) \Rightarrow l < L) \quad (c)$$

$$LectRedac \vdash \Box(l < L \Rightarrow ((l < L)\mathbf{U}(l = L))) \quad (d)$$

Preuve de (c) : pour prouver cette formule, la règle suivante est utilisée avec ($J = r - l \geq 0$) et I désignant l'invariant de la machine B :

$$\frac{\vdash J \Rightarrow I \quad S \vdash \Box J}{S \vdash \Box I}$$

Preuve de (d) : pour établir cette formule, nous devons prouver :

$$LectRedac \vdash \Box(l < L \wedge l \neq L) \curvearrowright (l < L \vee l = L) \quad (d.1)$$

$$LectRedac \vdash \Box\Diamond(l \geq L \vee l = L) \quad (d.2)$$

Preuve de (d.1) : cette formule est équivalente à $LectRedac \vdash \Box(l < L) \curvearrowright (l \leq L)$.

Il faut donc prouver que les événements d'écriture et de lecture conduisent d'un état satisfaisant $(r < L)$ vers un état qui satisfait $(r \leq L)$, ce qui est trivialement prouvé.

Preuve de (d.2) : cette formule est équivalente à $LectRedac \vdash \Box\Diamond(l \geq L)$ qu'on a déjà prouvée en tant que propriété d'existence.