Modélisation et vérification des systèmes informartiques

Fondements et techniques

17 septembre 2016

Dominique Méry

Université de Lorraine (Telecom Nancy)
LORIA, BP 70239,Campus Scientifique
54506 Vandœuvre-lès-Nancy Cedex, FRANCE
dominique.mery@loria.fr
http://www.loria.fr/∼mery

written on the 17 septembre 2016, please send bug reports to mery@loria.fr

Ce document est édité le 17 septembre 2016 au début du cours MVSI de l'année d'apprentissage 2A de Telecom Nancy et sera, sans doute, modifié et enrichi en fonction des cours et des séances d'exercices. Deux mots dans le titre caractérisent les objectifs :

— *modèles* : nous allons donner des éléments sur les techniques de modélisation formelle avec une vue sur les applications en génie logiciel, notamment en vérification et validation de logiciels, de programmes, de systèmes.

— *algorithmes* : nous allons considérer les questions de construction des algorithmes corrects et nous aborderons les questions de calculabilité, de décidabilité et d'indécidabilité.

Deux mots caractérisent les concepts que nous allons introduire pour les modèles et les algorithmes :

— *abstraction* : une abstraction est une notion très importante quand on veut modéliser un système ; une abstraction désigne à la fois une action et l'objet de cette action. Abstraire un système, c'est en donner une vue détachée de la réalité mais conforme à la réalité du système. Cette notion d'abstraction permet de donner des vues simples mais fidèles des systèmes.

— *Raffinement* : un raffinement est une action inverse de l'abstraction et consiste à préciser le modèle en cours d'examen afin de lui donner des articifices identifiés sur le système en cours de modélisation.

Ces deux notions peuvent avoir d'autres sens mais sont essentielles pour modéliser des systèmes. Nous allons donc envisager des techniques mettant en œuvre ces deux notions et nous allons les utiliser dans ce cours mais aussi dans les cours où la sûreté et la sécurité sont requises dans la mesure où elles sont critiques et peuvent poser des problèmes critiques par rapport aux personnes et aux biens.

Cette nouvelles édition ajoute la pratique d'un environnement, PAT [42], permettant d'analyser les systèmes informatiques dans le cadre de langages de programmation comme C# ou encore des formalismes spécifiquement développés pour permettre la construction d'outils de vérification.

Table des matières

Première partie

Modélisation et vérification

Chapitre 1

Modélisation et vérification des programmes et des systèmes

Ce chapitre envisage la question de vérification des programmes et des systèmes ; cette vérification repose sur des données claires de ce que l'on veut vérifier et de ce que l'on souhaite effectivement mettre en évidence (correction partielle, correction totale, absence d'erreurs à l'exécution, bon typage, . . .). Le besoin de justifier les programmes et les systèmes notamment répartis apparaît avec les premiers modèles de calcul comme les machines de Turing [44], repris par Floyd [20] pour les *flowcharts* ; Hoare [24] apporte une expression axiomatique des principes de preuve et une vue orientée vers la méthodologie de conception. La vérification et la construction se trouvent liées par la notion de *asserted programs* ; cette notion a permis de développer une démarche rigoureuse de conception fondée sur les pré-conditions et les post-conditions, notamment le langage de spécification VDM [27, 26, 8, 28]. Un certain nombre de langages de programmation [34, 7, 22] intègrent des éléments permettant d'exprimer à la fois l'algorithmique mais aussi les *annotations* comme les pré-conditions, les post-conditions, les invariants de classe, . . .
Au préalable, il faut définir un cadre sémantique pour le système à vérifier et être capable de modéliser un système. Nous utiliserons les notations ensemblistes de B [1, 3, 12] et de TLA$^+$ [30, 32], ainsi que les notations nécessaires pour utiliser l'environnement PAT [42], et nous donnons une explication de ce qui est en jeu dans le cadre de la modélisation.

1.1 Modélisation d'un système

Un modèle abstrait relationnel $\mathcal{AM}$ ($\mathcal{AM}_{\mathrm{P}}$ d'un programme P ou $\mathcal{AM}_{\mathrm{P}}$ d'un système P) est la donnée d'un espace d'états Σ, d'un ensemble d'états initiaux $Init_{\mathrm{P}}$, d'un ensemble d'états terminaux $Term_{\mathrm{P}}$ et d'une relation binaire $\mathcal{R}$ sur Σ. L'ensemble des états terminaux peut être vide et, dans ce cas, le programme ne termine pas ; cet aspect peut être utilisé pour modéliser les programmes des systèmes d'exploitation qui ne terminent pas et qui ne doivent pas terminer. Nous allons utiliser l'expression système plutôt que programme, dans la mesure où nous pouvons décrire des objets plus généraux que des programmes au sens informatiques mais aussi que ce formalisme est aussi utilisable pour les applications répartis.
Un système est caractérisé par ses traces d'exécution construites à l'aide du modèle abstrait comme suit :
$s_0 \underset{R}{\longrightarrow} s_1 \underset{R}{\longrightarrow} s_2 \underset{R}{\longrightarrow} s_3 \underset{R}{\longrightarrow} \ldots \underset{R}{\longrightarrow} s_i \underset{R}{\longrightarrow} \ldots$ est une trace engendrée par le modèle abstrait.
L'observation d'un système peut donc être réalisée par l'analyse des traces du système ; Θ_{S} est l'ensemble de toutes les traces de S. Pour exprimer des propriétés, un langage d'assertions ou un langage de formules est important ; nous notons un langage d'assertions $\mathcal{L}$. Pour simplifier, nous pouvons prendre comme langage d'assertions $\mathcal{P}(\Sigma)$ (l'ensemble des parties de Σ) et $\varphi(s)$ (ou $s \in \{u | u \in \Sigma \wedge \varphi(u)\}$) signifie que φ est vraie en s. Le langage d'assertions permet d'exprimer des propriétés mais il se peut que le langage considéré ne soit pas suffisamment expressif. Dans le cadre de la correction des programmes, nous supposerons que les langages d'assertions sont suffisamment complets (au sens de Cook) et cela veut dire que les propriétés requises pour la complétude sont exprimables dans le langage considéré.
Les propriétés d'un système S sont, en particulier, les propriétés de sûreté et les propriétés de fatalité. Les propriétés de sûreté sont, par exemple, la correction partielle d'un système S par rapport à ses spécifications, l'absence d'erreurs à l'exécution ; les propriétés de fatalité sont, par exemple, la terminaison d'un programme P par rapport à ses spécifications ou la correction totale de P par rapport à ses spécifications.

Nous pourrions nous intéresser à d'autres propriétés de programmes comme ses performances mais cela impliquerait des modèles pour exprimer des propriétés dites non fonctionnelles.
Les propriétés sont exprimées dans un langage $\mathcal{L}$ dont les éléments sont combinés par des connecteurs logiques ou par des instanciation de variables ; la relation d'implication modulo l'équivalence définit une relation d'ordre partiel. Pour être plus précis, on peut définir une relation de satisfaction d'une propriété en utilisant la notation ensembliste :

1. $s \models \alpha(c)$, si $s \in c$: α est une fonction d'abstraction.
2. $s \in \gamma(a)$, si $s \models a$: γ est une fonction de concrétisation.

Ces deux fonctions sont liées par la propriété suivante :
pour toute assertion x de $(\mathcal{L}, \Rightarrow)$, pour toute partie y de $(\mathcal{P}, \subseteq)$, $x \subseteq \alpha(y)$ si, et seulement si, $x \subseteq \alpha(y)$
Cela signifie que l'on peut utiliser une notation ensembliste mais qu'il faut ensuite utiliser une fonction de transfert des résultats du domaine concret vers le domaine abstrait. Dans le cas précédent, le domaine abstrait était celui des assertions et le domaine concret était celui des parties de l'ensemble des états.

Nous supposons qu'un système S est modélisé par un ensemble d'états Σ_{S}, noté Σ, et que $\Sigma \stackrel{def}{=} \text{VARIABLES} \longrightarrow \text{VALS}$. L'écriture $s \in A$ se traduit en une expression de la forme $s[\![\varphi(x)]\!]$ où x est une liste dont les éléments sont toutes les variables de VARIABLES et exclusivement ces variables ; cela signifie que $s \in A$ signifie de manière équivalente que $\varphi(x)$ est vraie en s. La définition de la validité d'une formule ou d'un prédicat peut être donnée sous une forme inductive de $s[\![\varphi(x)]\!]$. La définition inductive sera expliquée en détail dans les chapitres **??** et **??**.

Exemple 1.1

1. $s[\![x]\!]$ est la valeur de s en x c'est-à-dire $s(x)$ ou encore la valeur de x en s.
2. $s[\![\varphi(x) \wedge \psi(x)]\!] \stackrel{def}{=} s[\![\varphi(x)]\!]$ et $s[\![\psi(x)]\!]$.
3. $s[\![x = 6 \wedge y = x{+}8]\!] \stackrel{def}{=} s[\![x]\!] = 6$ et $s[\![y]\!] = s[\![x]\!]{+}8$.

Nous utilisons des notations simplifiant la référence aux états ; ainsi, $s[\![x]\!]$ est la valeur de x en s et le nom de la variable x et sa valeur ne seront pas distinguées. $s'[\![x]\!]$ est la valeur de x en s' et sera notée x'. Ainsi, $s[\![x = 6]\!] \wedge s'[\![y = x{+}8]\!]$ se simplifiera en $x = 6 \wedge y' = x'{+}8$. La conséquence est que l'on pourra écrire la relation de transition comme une relation liant l'état des variables en s et l'état des variables en s'.
Avec TLA [30], Lamport introduit la notion d'action sous la fome d'une formule logique comprenant des occurrences primées et non-primées de variables. Une action A est une expression booléenne ayant des occurences libres des variables non-primées, primées et des symboles de constantes ; une action désigne une relation entre une paire d'états consécutifs, un état avant (exprimé l'aide des variables non-primés et des symboles de constantes) et un nouvel état (exprimé à l'aide des variables primées et des symboles de constantes).

☼**Définition 1.1**
Soient $\langle s, t\rangle$ une paire d'états consécutifs, x la liste des variables ; $s[\![x]\!]$ est la valeur des variables en s et $t[\![x]\!]$ la valeur des variables en t. Le couple $\langle s, t\rangle$ satisfait l'action A si et seulement si $s[\![A]\!]t$ est vraie. $s[\![A]\!]t$ est la valeur obtenue en remplaçant les variables non-primées x par leurs valeurs $s[\![x]\!]$ en s et celle primées (x') par les valeurs de x, $t[\![x]\!]$ en t :

$$s[\![A]\!]t \triangleq A(\forall \text{`}x' : s[\![x]\!]/x, t[\![x]\!]/x')$$

On peut ainsi définir des pas de calcul comme des relations entre des variables primées et non-primées :
— $s[\![x \leq 0 \wedge x'{+}y = y']\!]t$ est équivalent à $s[\![x]\!] \leq 0 \wedge t[\![x]\!]{+}s[\![y]\!] = t[\![y]\!]$.
— $s[\![x' = y']\!]t$ est équivalent à $t[\![x]\!] = t[\![y]\!]$.
Si la condition $s[\![A]\!]t$ est vaie, on dit que la paire $\langle s, t\rangle$ est un A pas. Nous avons introduit la notion de variable primée de la logique temporelle des actions de Lamport [30] et nous pourrions utiliser aussi le système de règles d'inférence et d'axiomes pour modéliser les systèmes concurrents.
x' est la valeur après la transition considérée et x est la valeur avant la transition considérée.

Ainsi, la condition $\exists y.\mathrm{A}(x, y)$ définit la condition de transition ou la garde ; nous nous intéressons aux expressions particulières de la forme suivante $cond(x) \wedge x' = f(x)$ où $cond$ est une condition sur x et f est une fonction. Ceyye fonction et cette condition pourront avoir des formes très générale mais on recherchera les formes *exécutables* ou *codables* dans un langage de programmation. On peut exprimer les principes d'induction en utilisant les relations entre les variables non-primées et les variables primées. Les conditions initiales sont définies par un prédicat caractérisant les valeurs des variables initialement. Nous proposons donc de définir plus généralement ce qu'est un modèle relationnel d'un système dont on a observé les variables flexibles. Nous avons noté que l'ensemble des états était Σ pour un système donné et nous identifions cet ensemble à l'ensemble des valeurs possibles des variables flexibles x. Nous utiliserons donc la même notation mais VALS sera en fait l'ensemble des valeurs possibles de x .

☼**Définition 1.2** Modèle relationnel d'un système
Un modèle relationnel $\mathcal{MS}$ pour un système $\mathcal{S}$ est une structure

$$(Th(s, c), x, \textsc{Vals}, \textsc{Init}(x), \{r_0, \ldots, r_n\})$$

où

- $Th(s, c)$ est une théorie définissant les ensembles, les constantes et les propriétés statiques de ces éléments.
- x est une liste de variables flexibles.
- VALS est un ensemble de valeurs possibles pour x.
- $\{r_0, \ldots, r_n\}$ est un ensemble fini de relations reliant les valeurs avant x et les valeurs après x'.
- $\textsc{Init}(x)$ définit l'ensemble des valeurs initiales de x.

Un modèle relationnel $\mathcal{MS} = (Th(s, c), x, \textsc{Vals}, \textsc{Init}(x), \{r_0, \ldots, r_n\})$ pour un système $\mathcal{S}$ est une structure permettant d'étudier un système au travers d'un modèle opérationnel. Nous supposons que la relation r_0 est la relation $Id[\textsc{Vals}]$, identité sur VALS.

☼**Définition 1.3**
Soit $(Th(s, c), x, \textsc{Vals}, \textsc{Init}(x), \{r_0, \ldots, r_n\})$ un modèle relationnel d'un système $\mathcal{S}$. La relation NEXT associée à ce modèle est définie par la disjonction des relations r_i :
$\textsc{Next} \stackrel{def}{=} r_0 \vee \ldots \vee r_n$

La modélisation d'un système comprend la donnée des variables x, du prédicat caractériant les valeurs initiales des variables et une relation NEXT modélisant la relation entre les valeurs avant et les valeurs après. Les principes d'induction sont transcrits dans le cadre des modèles relationnels et nous introduisons la défintion de la sûreté dans un modèle relationnel.
Soit $(Th(s, c), x, \textsc{Vals}, \textsc{Init}(x), \{r_0, \ldots, r_n\})$ un modèle relationnel d'un système $\mathcal{S}$. La théorie $Th(s, c)$ est définie dans un langage d'assertions qui permet de décrire un certain nombre de propriétés et de définir des ensembles. Un exemple est la théorie des ensembles du langage B ou du langage TLA$^+$. Nous apporterons des éléments plus précis dans les parties suivantes de ce cours, quand nous introduirons les notations de B et de TLA$^+$. Quand nous parlerons d'une propriété φ, il s'agira de ce langage implicite dans notre exposé. Pour être précis et ne pas confondre les différentes notations, il est important de bien définir les valeurs du système étudié ; ainsi, pour une variable x, nous définissons les valeurs suivantes :

- x est la valeur courante de la variable x.
- x' est la valeur suivante de la variable x.
- x_0 ou $\underline{x}$ sont la valeur initiale de la variable x.
- $\overline{x}$ est la valeur finale de la variable x, quand cette notion a du sens.

Nous avons utilisé un style différent pour désigner la variable x et sa valeur x et la plupart du temps ces deux notations sont confondues. Dans la section suivante, nous allons définir les propriétés de sûreté pour les modèles relationnels de système.

1.2 Propriétés de sûreté et d'invariance dans un modèle relationnel

Parmi les propriétés d'état, les propriétés de sûreté désignent les propriétés de programme ou de système suivantes : la correction partielle d'un programme par rapport à ses pré et post conditions, l'absence d'erreurs à l'exécution ou encore l'exclusion mutuelle dans le cas des algorithmes ou systèmes répartis. Nous verrons comment ces propriétés de correction sont dérivées de la définition générale suivante.

☼**Définition 1.4**
Soit $(Th(s,c), x, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ un modèle relationnel M d'un système $\mathcal{S}$. Une propriété A est une propriété de sûreté pour le système $\mathcal{S}$, si

$$\forall x_0, x \in \Sigma . Init(x_0) \wedge \text{NEXT}^\star(x_0, x) \Rightarrow A(x).$$

Si on considère la propriété $\forall x_0, x \in \Sigma . Init(x_0) \wedge \text{NEXT}^\star(x_0, x) \Rightarrow A(x)$, on notera qu'on peut appliquer une transformation formelle de l'expression logique et produire l'expression logique suivante équivalente au sens sémantique : $\forall x \in \Sigma.(\exists x_0 . x_0 \in \Sigma \wedge Init(x_0) \wedge \text{NEXT}^\star(x_0, x)) \Rightarrow A(x)$. Cette observation est importante car l'ensemble suivant : $\{u | u \in \Sigma \wedge (\exists x_0 . x_0 \in \Sigma \wedge Init(x_0) \wedge \text{NEXT}^\star(x_0, x))\}$ est l'ensemble des états accessibles à partir des états initiaux que nous noterons $\text{REACHABLE}(M)$.

☺ **Propriété 1.1**
Les deux expressions suivantes sont équivalentes :
— $\forall x_0, x \in \Sigma . Init(x_0) \wedge \text{NEXT}^\star(x_0, x) \Rightarrow A(x)$
— $\forall x \in \Sigma.(\exists x_0 . x_0 \in \Sigma \wedge Init(x_0) \wedge \text{NEXT}^\star(x_0, x)) \Rightarrow A(x)$

A partir de cette propriété, nous en déduison le résulta suivant constituant un principe d'induction pour démontrer les propriététs de sûreté des systèmes modélisés par des modèles relationnels.

☺ **Propriété 1.2** (Principe d'induction)
Soit $(Th(s,c), x, \text{VALS}, Init(x), \{r_0, \ldots, r_n\})$ un modèle relationnel M d'un système S. Une propriété $A(x)$ est une propriété de sûreté pour le système S, si et seulement s'il existe une propriété d'état $I(x)$, telle que :

$$\forall x, x' \in \text{VALS} \ : \ \begin{cases} (1)\ Init(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow P(x) \\ (3)\ I(x) \wedge \text{NEXT}(x, x') \Rightarrow I(x') \end{cases}$$

La propriété $I(x)$ est appelée un invariant inductif de S et est une propriété de sûreté particulière plus forte que les autres propriétés de sûreté. Nous justifions maintenant ce principe d'induction.

PREUVE:
⟨1⟩1. SUPPOSONS QUE: il existe une propriété $I(x)$ telle que :

$$\forall x, x' \in \text{VALS} \ : \ \begin{cases} (1)\ Init(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow A(x) \\ (3)\ I(x) \wedge \text{NEXT}(x, x') \Rightarrow I(x') \end{cases}$$

PROUVONS QUE: $A(x)$ est une propriété de sûreté pour pour le système S modélisé par M.
PREUVE:
Soient x et $x' \in \text{VALS}$ tels que $INIT(x) \wedge \text{NEXT}^{\cdot}(x, x')$. On peut construire une suite telle que : $(x = x_0) \xrightarrow[\text{NEXT}]{} x_1 \xrightarrow[\text{NEXT}]{} x_2 \xrightarrow[\text{NEXT}]{} \ldots \xrightarrow[\text{NEXT}]{} (x_i = x')$. L'hypothèse (1) nous permet de déduire $I(x_0)$. L'hypothèse (3) nous permet de déduire $I(x_1), I(x_2), I(x_3), \ldots, I(x_i)$. En utilisant l'hypothèse (2) pour x', nous en déduisons que x' satisfait A. □
⟨1⟩2. SUPPOSONS QUE: $\forall x, y \cdot x, y \in \text{VALS} \wedge Init(x) \wedge \text{NEXT}^{\cdot}(x, y) \Rightarrow A(y)$

PROUVONS QUE: PROUVONS QUE : il existe une propriété $I(x)$ telle que :

$$\forall x, x' \in \text{VALS} \;:\; \begin{cases} (1)\ Init(x) \Rightarrow I(x) \\ (2)\ I(x) \Rightarrow A(x) \\ (3)\ I(x) \wedge \text{NEXT}(x,x') \Rightarrow I(x') \end{cases}$$

PREUVE:Nous considérons la propriété suivante : $I(x) \;\hat{=}\; \exists y \in \text{VALS} \cdot Init(y) \wedge \text{NEXT}^{*}(y,x)$. $I(x)$ exprime que la valeur x est accessible à partir d'une valeur initiale y. Les trois propriétés sont simples à vérifier pour $I(x)$. $I(x)$ est appelé le plus fort invariant de l'algorithme $\mathcal{A}$. □

$\langle 1\rangle 3$. Q.E.D.

PREUVE:On en déduit l'équivalence par les pas $\langle 1\rangle 1$ et $\langle 1\rangle 2$.□

□

Si on transforme la propriété (3), on obtient une forme plus proche de ce que nous utiliserons dans la suite.

☺ **Propriété 1.3**

Les deux énoncés suivants sont équivalents :

(I) Il existe une propriété d'état $\text{I}(x)$ telle que :

$$\forall x, x' \in \text{VALS} : \begin{cases} (1)\ \ \text{INIT}(x) \Rightarrow \text{I}(x) \\ (2)\ \ \text{I}(x) \Rightarrow \text{A}(x) \\ (3)\ \ \text{I}(x) \wedge \text{NEXT}(x,x') \Rightarrow \text{I}(x') \end{cases}$$

(II) Il existe une propriété d'état $\text{I}(x)$ telle que :

$$\forall x, x' \in \text{VALS} : \begin{cases} (1)\ \ \text{INIT}(x) \Rightarrow \text{I}(x) \\ (2)\ \ \text{I}(x) \Rightarrow \text{A}(x) \\ (3)\ \ \forall i \in \{0,\ldots,n\} : \text{I}(x) \wedge x\ r_i\ x' \Rightarrow \text{I}(x') \end{cases}$$

PREUVE:La preuve est immédiate en appliquant la règle suivante : $\forall i \in \{0,\ldots,n\} : A \wedge x\ r_i\ x' \Rightarrow B \equiv (A \wedge (\exists i \in \{0,\ldots,n\} : x\ r_i\ x')) \Rightarrow B$ et la définition de $\text{NEXT}(x,x')$.□

La propriété $I(x)$ est appelée un invariant inductif de S et est une propriété de sûreté particulière plus forte que les autres propriétés de sûreté.

☼**Définition 1.5** invariant d'un système S

Une propriété $A(x)$ est un invariant inductif d'un système S défini par un modèle M, si

$$\forall x, x' \in \text{VALS} : \begin{cases} (1)\ \ \text{INIT}(x) \Rightarrow \text{I}(x) \\ (2)\ \ \forall i \in \{0,\ldots,n\} : \text{I}(x) \wedge x\ r_i\ x' \Rightarrow \text{I}(x') \end{cases}$$

Enfin, nous avons une propriété importante permettant de comprendre le ien entre la méthode de Floyd [20] et la méthode de Hoare [24].

☺ **Propriété 1.4**

Les deux énoncés suivants sont équivalents :

— $\forall x, x' \in \text{VALS} : \text{I}(x) \wedge x\ r_i\ x' \Rightarrow \text{I}(x')$ (Hoare)
— $\forall x' \in \text{VALS} : (\exists x \in \text{VALS} : \text{I}(x) \wedge x\ r_i\ x') \Rightarrow \text{I}(x')$ (Floyd)

PREUVE:La preuve est immédiate en appliquant la règle suivante : $\forall u.P(u) \Rightarrow Q \equiv (\exists u.P(u)) \Rightarrow Q$. □

Nous appliquerons cette transformation plus tard et nous allons maintenant utiliser ce principe d'induction dans le cas d'un langage de programmation généraliste qui pourrait s'apparenter au langage C. L'objectif est de comprendre comment fonctionnent les outils de vérification dans le cas de langage comme Spec# avec le système `rise4fun` [41] et de comprendre comment fonctionnent les environnements comme `TLAPS` [43] ou encore `Rodin` [4]

1.3 Conception d'une méthode de preuves de propriétés d'invariance et de sûreté pour un langage de programmation

1.3.1 Spécialisation des principes d'induction pour le cas des programmes

Dans la cas de programmes ou d'algorithmes, la méthode de correction partielle peut être spécialisée en fonction du contexte très particulier des programmes. On considère un langage de programmation classique noté PROGRAMS et nous supposons que ce langage de programmation dispose de l'affectation, de la conditionnelle, de l'itération bornée, de l'itération non-bornée, de variables simples ou structurées comme les tableaux et de la définition de constantes.
On se donne un programme P de PROGRAMS ; ce programme comprend

- des variables notées globalement v,
- des constantes notées globalement c,
- des types associés aux variables notés globalement VALSet identifiés à un ensemble de valeurs possibles des variables,
- des instructions suivant un ordre défini par la syntaxe du langage de programmation.

Nous donnons trois exemples de programmes ou d'algorithmes que nous pouvons écrire et nous introduisons pour chacun de ces exemples la spécification des données et des résultats sous la forme de prédicates placés dans l'entête des algorithmes.

Exemple 1.2
L'$addition$ est définie comme suit :

$$\forall x, y \in \mathbb{N} : \begin{cases} addition(x,0) & = & {\pi_1}^1(x) \\ addition(x,y{+}1) & = & \sigma(addition(x,y)) \end{cases},$$

precondition : $x, y \in \mathbb{N}$
postcondition : $result = ADDITION(x,y)$
local variables : $cx, cy, cresult \in \mathbb{N}$

$cx := x$;
$cy := 0$;
$cresult := {\pi_1}^1(x)$;
while $cy < y$ **do**
  Invariant : $0 \leq cy \wedge cy < y \wedge cx = x \wedge$
    $cresult = addition[cx, cy]$
  $cresult := \sigma[cresult]$;
  $cy := cy{+}1$;
;
$result := cresult$;

Algorithme 1: Iterative algorithm $ADDITION$ for computing the primitive recursive function $addition$

Exemple 1.3
La $multiplication$ est définie comme suit :

$$\forall x, y \in \mathbb{N} : \begin{cases} multiplication(x,0) & = & \zeta() \\ multiplication(x,y{+}1) & = & addition(x, multiplication(x,y)) \end{cases},$$

Exemple 1.4
L'$exponentiation$ est définie comme suit/

precondition : $x, y \in \mathbb{N}$
postcondition : $result = multiplication(x, y)$
local variables : $cx, cy, cresult \in \mathbb{N}$

```
cx := x;
cy := 0;
cresult := ζ();
while cy < y do
    Invariant : 0 ≤ cy ∧ cy < y ∧ cx = x∧
                cresult = multiplication[cx, cy]
    cresult := addition[cx, cresult];
    cy := cy+1;
;
result := cresult;
```

Algorithme 2: Iterative algorithm $MULTIPLICATION$ for computing the primitive recursive function $multiplication$

$$\forall x, y \in \mathbb{N} : \begin{cases} exponentiation(x, 0) & = & \sigma(\zeta()) \\ exponentiation(x, y{+}1) & = & multiplication(x, exponentiation(x, y)) \end{cases},$$

precondition : $x, y \in \mathbb{N}$
postcondition : $result = exponentiation(x, y)$
local variables : $cx, cy, cresult \in \mathbb{N}$

```
cx := x;
cy := 0;
cresult := σ(ζ());
while cy < y do
    Invariant : 0 ≤ cy ∧ cy < y ∧ cx = x∧
                cresult = exponentiation[cx, cy]
    cresult := MULTIPLICATION[cx, cresult];
    cy := cy+1;
;
result := cresult;
```

Algorithme 3: Iterative algorithm $EXPONENTIATION$ for computing the primitive recursive function $exponentiation$

Chaque exemple est commenté et contient des informations qui permettent de comprendre comment fonctionne chacun des algorithmes. Cette approche vise à annoter de manière systématique les algorithmes ou les programmes, afin de permettre une meilleure compréhension des calculs réalisés et des instructions.
Les annotations sont donc intégrées à la définition des algorithmes ou des programmes et font partie de ces structures. En fait, on définit des points de contrôle pour chaque algorithme et on associe à chaque point de contrôle une information sur ce que vérifient les valeurs des variables à ce point. Plus généralement, on définit un ensemble de points de contrôle LOCATIONS pour chaque programme ou algorithme P. LOCATIONS est un ensemble fini de valeurs et une variable cachée notée ℓ parcourt cet ensemble selon l'enchaînement.
Cela signifie que l'espace des valeurs possibles VALS est un produit cartésien de la forme LOCATIONS×MEMORY et que les variables x du système se décomposent en deux entités indépendantes $x = (pc, v)$ avec comme conditions $pc \in$ LOCATIONS et $v \in$ MEMORY.

$$x = (pc, v) \wedge pc \in \text{Locations} \wedge v \in \text{Memory} \tag{1.1}$$

precondition : $x, y \in \mathbb{N}$
postcondition : $result = multiplication(x, y)$
local variables : $cx, cy, cresult \in \mathbb{N}$

$\ell_0 : \{x, y \in \mathbb{N} \wedge cx, cy, cresult \in \mathbb{N}\}$
$cx := x$;
$cy := 0$;
$cresult := \zeta()$;
$\ell_1 : \{cx = x \wedge cy = 0 \wedge cresult = \zeta() \wedge x, y \in \mathbb{N} \wedge cx, cy, cresult \in \mathbb{N}\}$
while $cy < y$ **do**
  $\ell_2 : \{0 \leq cy \wedge cy < y \wedge cx = x \wedge cresult = multiplication[cx, cy] \wedge x, y \in \mathbb{N} \wedge cx, cy, cresult \in \mathbb{N}\}$
  $cresult := addition[cx, cresult]$;
  $cy := cy{+}1$;
  $\ell_3 : \{0 \leq cy \wedge cy \leq y \wedge cx = x \wedge cresult = multiplication[cx, cy] \wedge x, y \in \mathbb{N} \wedge cx, cy, cresult \in \mathbb{N}\}$
;
$\ell_4 : \{0 \leq cy \wedge cy = y \wedge cx = x \wedge cresult = multiplication[cx, cy] \wedge x, y \in \mathbb{N} \wedge cx, cy, cresult \in \mathbb{N}\}$
$result := cresult$;
$\ell_5 : \{resulte = cresult \wedge 0 \leq cy \wedge cy = y \wedge cx = x \wedge cresult = multiplication[cx, cy] \wedge x, y \in \mathbb{N} \wedge cx, cy, cresult \in \mathbb{N}\}$

Algorithme 4: $MULTIPLICATION$ annotée

On considère un programme P annoté ; on se donne un modèle relationnel $\mathcal{MP} = (Th(s, c), x, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ où

- $Th(s, c)$ est une théorie définissant les ensembles, les constantes et les propriétés statiques de ce programme
- x est une liste de variables flexibles et x comprend une partie contrôle et une partie mémoire.
- LOCATIONS$\times$MEMORY est un ensemble de valeurs possibles pour x.
- $\{r_0, \ldots, r_n\}$ est un ensemble fini de relations reliant les valeurs avant x et les valeurs après x' et conformes à la relation de succession $\longrightarrow$ entre les points de contrôle.
- INIT(x) définit l'ensemble des valeurs initiales de (pc_0, v) et $x = (pc_0, v)$.

On suppose qu'il existe un graphe sur l'ensemble des valeurs de contrôle définissant la relation de flux et nous notons cette structure (LOCATIONS, $\longrightarrow$). Par exemple, le graphe de contrôle de notre précédent exemple est $(\{pc_i | i \in 0..4\}, \{pc_0 \longrightarrow pc_1, pc_1 \longrightarrow pc_2, pc_2 \longrightarrow pc_3, pc_3 \longrightarrow pc_2, pc_2 \longrightarrow p_4, pc_3 \longrightarrow pc_4\}$.

☼**Définition 1.6**
$\ell_1 \longrightarrow \ell_2 \stackrel{def}{=} pc = \ell_1 \wedge pc' = \ell_2$

Un exemple d'annotation d'un algorithme ou d'un programme est donné dans l'algorithme 1.3.1. L'enchainement des étiquettes suit un ordre dérivé du texte du programme. En règle générale, il faudra placer au moins une étiquette à l'intérieur d'une boucle et le point privilégiée est en début de corps de boucle et cette annotation est l'invariant de boucle.

☼**Définition 1.7** Annotation d'un point de contrôle
Soit une structure (LOCATIONS, $\longrightarrow$) et une étiquette $\ell \in$ LOCATIONS. Une annotation d'un point de contrôle ℓ est un prédicat $P_\ell(v)$.

Nous considérons une propriété de sûreté notée A(x) et se rapportant au programme P.

Soit $(Th(s,c), x, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ un modèle relationnel pour ce programme. Une propriété A(x) est une propriété de sûreté pour P, si $\forall y, x \in \text{LOCATIONS}\times\text{MEMORY}.Init(y) \wedge \text{NEXT}^\star(y,x) \Rightarrow A(x)$. On sait que cette propriété implique qu'il existe une propriété d'état I(x) telle que :

$$\forall x, x' \in \text{LOCATIONS}\times\text{MEMORY} : \begin{cases} (1) & \text{INIT}(x) \Rightarrow \text{I}(x) \\ (2) & \text{I}(x) \Rightarrow \text{A}(x) \\ (3) & \forall i \in \{0,\ldots,n\} : \text{I}(x) \wedge x\, r_i\, x' \Rightarrow \text{I}(x') \end{cases}$$

La propriété I(x) est un invariant du programme P et nous pouvons déduire qu'il existe une décomposition de cette propriété selon les points de contrôle du programme :

Propriété 1 *Il existe une famille de propriétés* $\{P_\ell(v) : \ell \in \text{LOCATIONS}\}$ *satisfaisant l'équivalence suivante :* $\text{I}(x) \equiv \bigvee_{\ell\in\text{LOCATIONS}} \left(\bigvee_{v\in\text{MEMORY}} x = (\ell, v) \wedge P_\ell(v) \right)$.

PREUVE:Pour chaque étiquette ℓ, on définit $P_\ell(v) \stackrel{def}{=} \exists x.x = (\ell, v) \wedge \text{I}(x)$ et $\text{I}(x) \equiv \bigvee_{\ell\in\text{LOCATIONS}} \left(\bigvee_{v\in\text{MEMORY}} x = (\ell, v) \wedge P_\ell(v) \right)$ est évidemment vérifiée.□

Sous la relation $x = (\ell, v)$, nous avons donc les relations suivantes entre $I(x)$ et les annotations $P_\ell(v)$:

- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \text{LOCATIONS} \wedge v \in \text{MEMORY} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v) \stackrel{def}{=} \exists x.(x \in \text{VALS} \wedge x = (\ell, v) \wedge I(x))$

Nous pouvons maintenant *adapter* le principe d'induction, en transformant les expressions logiques. La transformation est fondée la relation de transition définie pour chaque couple d'étiquettes de contrôle qui se suivent et exprimée très simplement par la forme relationnelle suivante : $cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)$ et signifie que la transition de ℓ à ℓ' est possible quand la condition $cond_{\ell,\ell'}(v)$ est vraie pour v et quand elle a lieu, les variables v sont transformées comme suit $v' = f_{\ell,\ell'}(v)$. On a donc la relation suivante $r_{\ell,\ell'}$ de transition :

$$x\, r_{\ell,\ell'}\, x' \stackrel{def}{=} (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$$

Le modèle relationnel $M(P)$ pour le programme P annoté est donc défini comme suit :
$M(P) \stackrel{def}{=} (Th(s,c), (pc, v), \text{LOCATIONS}\times\text{MEMORY}, Init(\ell, v), \{r_{\ell,\ell'} | \ell, \ell' \in \text{LOCATIONS} \wedge \ell \longrightarrow \ell'\})$.
La définition de $Init(\ell, v)$ est dépendante de la précondition de P :

$$Init(\ell, v) \stackrel{def}{=} \ell \in \text{INPUTS} \wedge \textbf{pre}(\text{P})(v).$$

☺ **Propriété 1.5** Conditions initiales
Les deux propriétés suivantes sont équivalentes :

- $\forall x \in \text{VALS} : \text{INIT}(x) \Rightarrow \text{I}(x)$
- $\forall \ell \in \text{INPUTS}, v \in \text{MEMORY}.\textbf{pre}(\text{P})(v) \Rightarrow P_\ell(v)$

PREUVE:

- $\forall x \in \text{VALS} : \text{INIT}(x) \Rightarrow \text{I}(x)$
- appliquons la transformation $x = (\ell, v)$
- $\forall \ell, v \in \text{LOCATIONS}\times\text{MEMORY} : \text{INIT}(\ell, v) \Rightarrow \text{I}(\ell, v)$
- $\forall \ell, v \in \text{LOCATIONS}\times\text{MEMORY} : \ell \in \text{INPUTS} \wedge \textbf{pre}(\text{P})(v) \Rightarrow \text{I}(\ell, v)$
- $\forall \ell \in \text{LOCATIONS}, v \in \text{MEMORY} : \ell \in \text{INPUTS} \wedge \textbf{pre}(\text{P})(v) \Rightarrow \text{I}(\ell, v)$
- $\forall v \in \text{MEMORY}, \ell \in \text{INPUTS} : \textbf{pre}(\text{P})(v) \Rightarrow \text{I}(\ell, v)$
- $\forall v \in \text{MEMORY}, \ell \in \text{INPUTS} : \textbf{pre}(\text{P})(v) \Rightarrow (\exists \ell', v'.(\ell' \in \text{LOCATIONS} \wedge v' \in \text{MEMORY} \wedge x = (\ell', v') \wedge P_{\ell'}(v')))$
- $\forall v \in \text{MEMORY}, \ell \in \text{INPUTS} : \textbf{pre}(\text{P})(v) \Rightarrow (\exists \ell', v'.(\ell' \in \text{LOCATIONS} \wedge v' \in \text{MEMORY} \wedge (\ell, v) = (\ell', v') \wedge P_{\ell'}(v')))$
- $\forall v \in \text{MEMORY}, \forall \ell \in \text{INPUTS}.\textbf{pre}(\text{P})(v) \Rightarrow P_\ell(v)$

□

☺ **Propriété 1.6** Pas d'induction
Les deux propriétés suivantes sont équivalentes :
- $\forall i \in \{0, \ldots, n\} : \mathrm{I}(x) \wedge x\, r_i\, x' \Rightarrow \mathrm{I}(x')$
- $\forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$

PREUVE:
- $\forall \ell, \ell' \in \textsc{Locations} : \mathrm{I}(x) \wedge x\, r_{\ell,\ell'}\, x' \Rightarrow \mathrm{I}(x')$
- appliquons la transformation $x = (\ell, v)$.
- $\forall \ell, \ell' \in \textsc{Locations} : \mathrm{I}(\ell, v) \wedge (\ell, v)\, r_{\ell,\ell'}\, (\ell', v') \Rightarrow \mathrm{I}(\ell', v')$
- $\forall \ell, \ell' \in \textsc{Locations} : \mathrm{I}(\ell, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \Rightarrow \mathrm{I}(\ell', v')$
- $\forall \ell, \ell' \in \textsc{Locations} :$ $\begin{array}{l} \exists \ell_1, v_1.(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1)) \\ \wedge\, (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \\ \Rightarrow \\ \exists \ell_2, v_2.(\ell_2 \in \textsc{Locations} \wedge v_2 \in \textsc{Memory} \wedge (\ell', v') = (\ell_2, v_2) \wedge P_{\ell_2}(v_2)) \end{array}$
- $\forall \ell, \ell', \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory} :$
  $(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
  $\wedge\, (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
  $\Rightarrow$
  $\exists \ell_2, v_2.(\ell_2 \in \textsc{Locations} \wedge v_2 \in \textsc{Memory} \wedge (\ell', v') = (\ell_2, v_2) \wedge P_{\ell_2}(v_2))$
- $\forall \ell, \ell', \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory} :$
  $(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
  $\wedge\, (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
  $\Rightarrow$
  $(P_{\ell'}(v'))$
  $pc = \ell \wedge \wedge pc' = \ell'$ est remplacé par $\ell \longrightarrow \ell'$.
- $\forall \ell, \ell' \in \textsc{Locations}, v_1 \in \textsc{Memory} : \ell \longrightarrow \ell' :$
  $(P_\ell(v)) \;\wedge\; cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)$
  $\Rightarrow$
  $(P_{\ell'}(v'))$
- $\forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$

□

☺ **Propriété 1.7** Conclusion
Les deux propriétés suivantes sont équivalentes :
- $\mathrm{I}(x) \Rightarrow \mathrm{A}(x)$
- $\forall \ell \in \textsc{Locations}.P_\ell(v) \Rightarrow \mathrm{A}(\ell, v)$

PREUVE:
- $\mathrm{I}(x) \Rightarrow \mathrm{A}(x)$
  appliquons la transformation $x = (\ell, v)$.
- $\mathrm{I}(\ell, v) \Rightarrow \mathrm{A}(\ell, v)$
  $\exists \ell_1, v_1.(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
- $\Rightarrow$
  $\mathrm{A}(\ell, v)$
- $\forall \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory} :$ $\begin{array}{l} (\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1)) \\ \Rightarrow \\ \mathrm{A}(\ell, v) \end{array}$
- $\forall \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory} :$ $\begin{array}{l} (\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1)) \\ \Rightarrow \\ \mathrm{A}(\ell_1, v_1) \end{array}$

— $\forall \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory} : \begin{array}{l}(P_{\ell_1}(v_1))\\ \Rightarrow\\ \mathrm{A}(\ell_1, v_1)\end{array}$

— $\forall \ell \in \textsc{Locations}.P_\ell(v) \Rightarrow \mathrm{A}(\ell, v)$

□

Les transformations utilisées dans ces trois propriétés sont assez simples et seront justifoées dans les chapitres sur les énoncés logiques **??**. Nous avons la propriété suivante rassemblant les trois propriétés précédentes.

☺ **Propriété 1.8**
Les conditions de vérification suivantes sont équivalentes :

— $\forall x, x' \in \textsc{Locations} \times \textsc{Memory} : \begin{cases} (1)\ \ \textsc{Init}(x) \Rightarrow \mathrm{I}(x) \\ (2)\ \ \mathrm{I}(x) \Rightarrow \mathrm{A}(x) \\ (3)\ \ \forall i \in \{0, \ldots, n\} : \mathrm{I}(x) \wedge x\ r_i\ x' \Rightarrow \mathrm{I}(x') \end{cases}$

— $\forall v, v' \in \textsc{Memory} : \begin{cases} (1)\ \ \forall \ell \in \textsc{Inputs}.\mathbf{pre}(\mathrm{P})(v) \Rightarrow P_\ell(v) \\ (2)\ \ \forall \ell \in \textsc{Locations}.P_\ell(v) \Rightarrow \mathrm{A}(\ell, v) \\ (3)\ \ \forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v') \end{cases}$

On en déduit une méthode de preuve de correction de propriétés de sûreté générale.

☺ **Propriété 1.9** Méthode de correction de propriétés de sûreté
Soit $\mathrm{A}(\ell, v)$ une propriété d'un programme P. Soit une famille d'annotations famille de propriétés $\{P_\ell(v) : \ell \in \textsc{Locations}\}$ pour ce programme. Si les conditions suivantes sont vérifiées :

$\forall v, v' \in \textsc{Memory} : \begin{cases} (1)\ \ \forall \ell \in \textsc{Inputs}.\textsc{Precondition}(v) \Rightarrow P_\ell(v) \\ (2)\ \ \forall \ell \in \textsc{Locations}.P_\ell(v) \Rightarrow \mathrm{A}(\ell, v) \\ (3)\ \ \forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v') \end{cases}$,

alors $\mathrm{A}(\ell, v)$ est une propriété de sûreté pour le programme P.

PREUVE:La preuve est évidente en reprenant les propriétés précédentes et les arguments donnés.□

La propriété précédente nous donne une technique générale pour montrer qu'une propriété est une propriété de sûreté pour un programme donné. De plus, la méthode est complète, puisqu'on a montré qu'on peut toujours construire une famille qui convient. Cela étant, il est clair que le plus difficile est de trouver cette famille de prédicats de contrôle de manière à ce que toutes les conditions de vérification ou obligations de preuve soient satisfaites.

☼**Définition 1.8** Condition de vérification
L'expression $P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$ où ℓ, ℓ' sont deux étiquettes liées par la relation $\longrightarrow$, est appelée une condition de vérification.

☺ **Propriété 1.10** Floyd and Hoare

— $\forall v, v' \in \textsc{Memory}.\forall \ell, \ell' \in \textsc{Locations}.\ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$ est équivalent à $\forall \ell, \ell' \in \textsc{Locations}.\ell \longrightarrow \ell' \Rightarrow \forall v' \in \textsc{Memory}.P_\ell(v) \wedge cond_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v \mapsto f_{\ell,\ell'}(v))$

— $\forall v, v' \in \textsc{Memory}.\forall \ell, \ell' \in \textsc{Locations}.\ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$ est équivalent à
$\forall \ell, \ell' \in \textsc{Locations}.\ell \longrightarrow \ell' \Rightarrow \forall v' \in \textsc{Memory}.\,(\exists v \in \textsc{Memory}.P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)) \Rightarrow P_{\ell'}(v')$

Ces deux équivalences montrent que l'axiome de Hoare[24] est différent de la condition de vérification choisie par Floyd[20].

Nous pouvons resumer les deux formes possibles de l'affectation suivante :

$\ell : P_\ell(v)$
$v := f_{\ell,\ell'}(v)$
$\ell' : P_{\ell'}(v)$

Algorithme 5: Affectation

- $\forall v' \in \text{MEMORY}.P_\ell(v) \Rightarrow P_{\ell'}(v \mapsto f_{\ell,\ell'}(v))$ correspond à l'axiomatique de Hoare.
- $\forall v' \in \text{MEMORY}.\ (\exists v' \in \text{MEMORY}.P_\ell(v) \wedge v' = f_{\ell,\ell'}(v)) \Rightarrow P_{\ell'}(v')$ correspond à la règle d'affectation de Floyd.

Les deux formes sont équivalentes et se trouvent dans la littérature. Nous allons maintenant proposer des techniques de preuve de correction pour certaines propriétés classiques.
On peut maintenant définir des conditions de vérification pour chaque structure de programme.

Algorithme 6: Structure d'itération

Pour la structure d'itération, les conditions de vérification sont les suivantes :
- $P_{\ell_1}(v) \wedge B(v) \Rightarrow P_{\ell_2}(v)$
- $P_{\ell_1}(v) \wedge \neg B(v) \Rightarrow P_{\ell_4}(v)$
- $P_{\ell_3}(v) \wedge B(v) \Rightarrow P_{\ell_2}(v)$
- $P_{\ell_3}(v) \wedge \neg B(v) \Rightarrow P_{\ell_4}(v)$

$\ell_1 : P_{\ell_1}(v)$;
if $B(v)$ **then**
  $\ell_2 : P_{\ell_2}(v)$;
  $\ldots$;
  $\ell_3 : P_{\ell_3}(v)$;
else
  $m_2 : P_{\ell_2}(v)$;
  $\ldots$;
  $m_3 : P_{\ell_3}(v)$;
;
$\ell_4 : P_{\ell_4}(v)$;

Algorithme 7: Structure de conditionnelle

Pour la structure de conditionnelle, les conditions de vérification sont les suivantes :
- $P_{\ell_1}(v) \wedge B(v) \Rightarrow P_{\ell_2}(v)$
- $P_{\ell_3}(v) \Rightarrow P_{\ell_4}(v)$
- $P_{\ell_1}(v) \wedge \neg B(v) \Rightarrow P_{m_2}(v)$
- $P_{m_3}(v) \Rightarrow P_{\ell_4}(v)$

On peut définir des conditions de vérification pour toutes les structures algorithmiques mais il faut définir clairement la relation $\longrightarrow$.

1.3.2 Propriétés de correction des programmes

Soit v une variable d'état de P. **pre**(P)(v) est la précondition de P pour v ; elle caractérise les valeurs initiales de v. **post**(P)(v) est la postcondition de P pour v ; elle caractérise les valeurs finales de v.

Exemple 1.5

1. **pre**(P)(x, y, z)=$x, y, z \in \mathbb{N}$ et **post**(P)(x, y, z)=$z = x \cdot y$
2. **pre**(Q)(x, y, z)=$x, y, z \in \mathbb{N}$ et **post**(Q)(x, y, z)=$z = x + y$

Exemple 1.6 [18]
Nous reprenons un exemple simple de P. et R. Cousot pour illustrer les notions de préconditions et de post-conditions assertionnelles ou relationnelles. Le document analyse en profoindeur les différentes techniques de preuves de propriétés d'invariance et en particulier leur principle d'induction.

variables : X,Y,Q,R
valeurs : $\underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}, x, y, r, q$
précondition : $\underline{x}, \underline{y}, \underline{r}, \underline{q} \in \mathbb{N}$
postcondition : $\overline{x}, \overline{y}, \overline{r}, \overline{q} \in \mathbb{N} \wedge \overline{x} = \underline{x} \wedge \overline{y} = \underline{y} \wedge \overline{x} = \overline{q} \cdot \underline{y} + \overline{r} \wedge \overline{r} < \underline{y}$

$Q = 0;$
$R = X;$
while $R \geq Y$ **do**
  $Q = Q+1;$
  $R := R-Y;$

Algorithme 8: Division euclidienne de deux nombres et spécification relationnelle

pre(P)($\underline{x}, \underline{y}, \underline{r}, \underline{q}$) est définie par $\underline{x}, \underline{y}, \underline{r}, \underline{q} \in \mathbb{N}$.
post(P)($\underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}$) est définie par $\overline{x}, \overline{y}, \overline{r}, \overline{q} \in \mathbb{N} \wedge \overline{x} = \underline{x} \wedge \overline{y} = \underline{y} \wedge \overline{x} = \overline{q} \cdot \underline{y} + \overline{r} \wedge \overline{r} < \underline{y}$
Nous utilisons les variables soulignées pour désigner les valeurs initiales de ces variables et les variables surlignées pour désigner les valeurs finales de ces variables. Ainsi, on peut écrire la relation de calcul de P de la manière suivante :

$$\forall \underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}. \mathbf{pre}(\mathrm{P})(\underline{x}, \underline{y}, \underline{r}, \underline{q}) \wedge (\underline{x}, \underline{y}, \underline{r}, \underline{q}) \xrightarrow{\mathrm{P}} (\overline{x}, \overline{y}, \overline{r}, \overline{q}) \Rightarrow \mathbf{post}(\mathrm{P})(\underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}) \quad (1.2)$$

Notre spécification de la correction partielle par rapport à la précondition et à la postcondition est relationnelle. On peut utiliser un style assertionnel qui s'intègre dans la démarche de la logique de Hoare [24] en introduisant les programmes assertés ou annotés (*asserted programs*). Par exemple, on écrira les précondition et postcondition comme suit :
— **pre**(P)(x, y, r, q) est définie par $x, y, r, q \in \mathbb{N}$.
— **post**(P)(x, y, r, q) est définie par $x, y, r, q \in \mathbb{N} \wedge x = q \cdot y + r \wedge r < y$
La spécification est notée {**pre**(P)(x, y, r, q)} **P** {**post**(P)(x, y, r, q)} et est appelée triplet de Hoare.

variables : X,Y,Q,R
valeurs : x, y, q, r
précondition : $x, y \in \mathbb{N}$
postcondition : $x, y, r, q \in \mathbb{N} \wedge x = q \cdot y + r \wedge r < y$

$Q = 0;$
$R = X;$
while $R \geq Y$ **do**
  $Q = Q+1;$
  $R := R-Y;$

Algorithme 9: Division euclidienne de deux nombres et spécification assertionnelle

L'algorithme 9 peut aussi être spécifié par une expression de la forme suivante :

$$\text{X,Y,Q,R} : [x, y \in \mathbb{N}\ ,\ x, y, r, q \in \mathbb{N} \wedge x = q \cdot y + r \wedge r < y]$$

qui exprime que les variables X, Y, Q, R peuvent être modifiées dans le calcul débutant avec la précondition $x, y \in \mathbb{N}$ et se terminant par la postconditin $x, y, r, q \in \mathbb{N} \wedge x = q{\cdot}y{+}r \wedge r < y$. Cette écriture revient aussi à écrire de manière équivalente :

$$\{x, y \in \mathbb{N}\}\ \mathbf{P}_{X,Y,Q,R}\ \{x, y, r, q \in \mathbb{N} \wedge x = q{\cdot}y{+}r \wedge r < y\}$$

où $\mathrm{P}_{X,Y,Q,R}$ est un algorithme modifiant les variables X,Y,Q,R de manière à respecter la précondition et la postcondition. La notation : [,] est dûe à Morgan [35] qui développe un calcul de raffinement pour développer un programmae ou un algorithme en se conformant aux spécifications.
La spécification d'un problème donné est une étape essentielle dans la conception de solutions algorithmiques ; partant de la spécification on peut construire de façon systématique une solution algorithmique en introduisant des variables ou des constructions algorithmiques intermédiaires. Le calcul de raffinement de Morgan [35] propose un ensemble de règles de transformations permettant d'obtenir un programme conforme à la spécification w : $[P\ ,\ Q]$; la méthode B [12] peut aussi être utilisée pour mettre en œuvre ce calcul de raffinement constituant un calcul de développement.

Correction partielle d'un programme

La correction partielle vise à établir qu'un programme P est partiellement correct par rapport à sa précondition et à sa postcondition. On ne prend pas en compte la terminaison et on demande que quand le programme se termine, les valeurs des variables satisfont la postcondition. On note les éléments suivants :
— la spécification des données de P **pre**(P)(v)
— la spécification des résultats de P **post**(P)(v)
— une famille d'annotations de propriétés $\{P_\ell(v) : \ell \in \textsc{Locations}\}$ pour ce programme.
— une propriété de sûreté définissant la correction partielle $\ell = output \Rightarrow \mathbf{post}(\mathrm{P})(v)$ où $output$ est l'étiquette marquant la fin du programme P

☼**Définition 1.9**
Le programme P est partiellement correct par rapport à **pre**(P)(v) et **post**(P)(v), si la propriété $\ell = output \Rightarrow \mathbf{post}(\mathrm{P})(v)$ est une propriété de sûreté pour ce programme.

☺ **Propriété 1.11**
Si les conditions suivantes sont vérifiées :
— $\forall \ell \in \textsc{Inputs}.\forall v, v' \in \textsc{Memory}.\mathbf{pre}(\mathrm{P})(v) \Rightarrow P_\ell(v)$
— $\forall \ell \in \textsc{Outputs}.\forall v, v' \in \textsc{Memory}.P_\ell(v) \Rightarrow \mathbf{post}(\mathrm{P})(v)$
— $\forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' : \forall v, v' \in \textsc{Memory}.\ (P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v'))$,
alors le programme P est partiellement correct par rapport à **pre**(P)(v) et **post**(P)(v).

Absence d'erreurs à l'exécution

Pour la preuve de l'absence d'erreurs à l'exécution, nous devons définir ce que signifie une erreur à l'exécution. Pour cela, nous nouis plaçons dans le cas où la transition à exécuter est celle allant de ℓ à ℓ' et caractérisée par la condition ou garde $cond_{\ell,\ell'}(v)$ sur v et une transformation de la variable v, $v' = f_{\ell,\ell'}(v)$. Une condition d'absence d'erreur est définie par $\mathbf{DOM}(\ell, \ell')(v)$ pour la transition considérée. $\mathbf{DOM}(\ell, \ell')(v)$ signifie que la transition $\ell \longrightarrow \ell'$ est possible et ne conduit pas à une erreur. Une erreur est un débordement arithmétique, une référence à un élément de tableau qui 'existe pas, une référence à un pointeur nul, ...

Exemple 1.7

1. La transition correspond à une affectation de la forme $x := x{+}y$ ou $y := x{+}y$:
$\mathbf{DOM}(x{+}y)(x, y) \stackrel{def}{=} \mathbf{DOM}(x)(x, y) \wedge \mathbf{DOM}(y)(x, y) \wedge x{+}y \in int$
2. La transition correspond à une affectation de la forme $x := x{+}1$ ou $y := x{+}1$:
$\mathbf{DOM}(x{+}1)(x, y) \stackrel{def}{=} \mathbf{DOM}(x)(x, y) \wedge x{+}2 \in int$

Parmi les cas d'erreurs, on pourra citer le débordement arithmétique, la référence à une adresse non définie, la division par zéro, ... Le prédicat $\mathbf{DOM}(\ell,\ell')(v)$ doit intégrer ces éléments pour chaque cas possible qui se ramène à une affectation ou un test en C par exemple.
L'absence d'erreurs à l'exécution vise à établir qu'un programme P ne va pas produire des erreurs durant son exécution par rapport à sa précondition et à sa postcondition. On ne prend pas en compte la terminaison et le programme peut naturellement boucler. Nous définissons donc les éléments suivants :

- la spécification des données de P **pre**(P)(v)
- la spécification des résultats de P **post**(P)(v)
- une famille d'annotations de propriétés $\{P_\ell(v) : \ell \in \text{LOCATIONS}\}$ pour ce programme.
- une propriété de sûreté définissant l'absence d'erreurs à l'exécution :

$$\bigwedge_{\ell\in \text{LOCATIONS}-\{output\},n\in\text{LOCATIONS},\ell\longrightarrow n} (\mathbf{DOM}(\ell,n)(v))$$

☼**Définition 1.10**
Le programme P ne produira pas d'erreurs à l'exécution par rapport à **pre**(P)(v) et **post**(P)(v), si la propriété $\bigwedge_{\ell\in \text{LOCATIONS}-\{output\},n\in\text{LOCATIONS},\ell\longrightarrow n} (\mathbf{DOM}(\ell,n)(v))$ est une propriété de sûreté pour ce programme.

☺ **Propriété 1.12**
Si les conditions suivantes sont vérifiées :

$$\begin{cases} (1)\ \forall \ell \in \text{INPUTS}.\forall v,v' \in \text{MEMORY}.\mathbf{pre}(\mathrm{P})(v) \Rightarrow P_\ell(v) \\ (2)\ \forall m \in \text{LOCATIONS}-\{output\}, n \in \text{LOCATIONS}, \forall v,v' \in \text{MEMORY} : \\ \qquad m \longrightarrow n : P_m(v) \Rightarrow \mathbf{DOM}(m,n)(v) \\ (3)\ \forall \ell,\ell' \in \text{LOCATIONS}, \forall v,v' \in \text{MEMORY} : \ell \longrightarrow \ell' : (P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')) \end{cases},$$

alors le programme P ne produira pas d'erreurs à l'exécution par rapport à **pre**(P)(v) et **post**(P)(v).

Pour démontrer l'absence d'ereurs à l'xécution, on utilisera les annotations faites pour la correction partielle et on remplacera les ensembles abstraits (par exemple, $\mathbb{N}$) par des ensembles concrets (par exemple, $\mathbb{N}_{computer}$). Pour la mécanisation de la preuve des conditions de vérification, nous avons utilisé la plateforme `Rodin` [4]. Cette mécanisation est fondée sur la preuve d'énoncés appelés *séquents de Gentzen* [21] et sur l'utilisation de procédures permettant de `vérifier` que le séquent est valide ou non.

1.4 Vue ensembliste

Les propriétés de sûreté expriment que *rien de mauvais ne peut arriver [29]*. Par exemple, la valeur de la variable x est toujours comprise entre 0 et 567 ; la somme des valeurs courantes des variables x et y est égale à la valeur courante de la valeur z. On se donne un langage d'assertions $(\mathcal{P}(\Sigma),\subseteq)$ et une interprétation ensembliste de la relation de satisfaction. Ainsi, on écrira $s \in \{u|u \in \Sigma \wedge \varphi(u)\}$ pour exprimer que s est élément d'un ensemble d'états satisfaisant φ. Cela veut dire qu'on peut utiliser $\mathcal{P}(\Sigma)$ comme un langage pour exprimer des propriétés en tant qu'ensemble d'états ou propriétés d'états. Dans ce cas, les *assertions* sont les parties de Σ et l'expressivité du lanage est donc limitée à ces ensembles. Une question fondamentale est de savoir, si toutes les propriétés exprimables dans un langage $\mathcal{L}$ sont exprimables dans un autre langage $\mathcal{M}$. Nous définissons ce que nous entendons par *propriété de sûreté.*

☼**Définition 1.11** Structure de Kripke d'un système
Une structure de Kripke $\mathcal{MK}$ pour un système $\mathcal{S}$ est une structure $(\Sigma, \text{INIT}, \longrightarrow)$ où

- Σ est l'ensemble de tous les états.
- $\text{INIT} \subseteq \Sigma$ définit l'ensemble des états initiaux de S.
- $\longrightarrow$ est une relation binaire

1.4.1 Propriétés de sûreté

☼**Définition 1.12**
Soit un modèle ensembliste MK pour S. REACHABLE$(\mathrm{MK}) = \{u | u \in \Sigma_S \wedge \exists s \in \Sigma_S . s \in Init_S \wedge s \xrightarrow{\star} u\}$ est l'ensemble des tous les états accessibles pour le modèle ensembliste MK.

☼**Définition 1.13**
Une propriété A est une propriété de sûreté pour le système S, si

$$\forall s, s' \in \Sigma . s \in Init_S \wedge s \xrightarrow{\star} s' \Rightarrow s' \in A.$$

L'expression $\xrightarrow{\star}$ désigne la fermeture réflexive transitive de la relation $\longrightarrow$. La propriété de sûreté utilise une expression universelle pour quantifier les états. Pour démontrer une telle propriété, on peut soit vérifier la propriété pour chaque état possible de Σ_S, à condition que cet ensemble soit fini, ou bien trouver un principe d'induction. Pour ce qui est de la vérification pour chaque état, on utilise un algorithme de calcul des états accessibles à partir d'un état initial. Cette technique de calcul des états accessibles est souvent utilisée et est à la base des techniques de vérification automatique comme le *model checking* [33, 25, 14] Nous donnons maintenant un principe d'induction, pour prouver les propriétés de sûreté ; l'objectif est de montrer comment les techniques de preuves de propriétés de programmes sont construites et sur quels principes elles reposent.

☺ **Propriété 1.13** (Principe d'induction)
Une propriété A est une propriété de sûreté pour le système S si, et seulement si, il existe une propriété INV telle que

$$\begin{cases} (1)\ Init_S \subseteq INV \\ (2)\ INV \subseteq A \\ (3)\ \forall s, s' \in \Sigma_S . s \in INV \wedge s \longrightarrow s' \Rightarrow s' \in INV \end{cases}$$

La propriété INV est appelée un invariant du programme et est une propriété de sûreté particulière plus forte que les autres propriétés de sûreté. La justification de ce principe d'induction est assez simple. La condition (3) peut aussi s'écrire plus simplement $\longrightarrow [INV] \subseteq INV$ où $\longrightarrow [INV] = \{s | s \in \Sigma \wedge \exists u . u \in \Sigma \wedge u \in INV\}$ ou l'application de la relation $\longrightarrow$ sur l'ensemble INV.

PREUVE:
$\langle 1 \rangle 1$. SUPPOSONS QUE: Il existe une propriété INV telle que

$$\begin{cases} (1)\ Init_S \subseteq INV \\ (2)\ INV \subseteq A \\ (3)\ \forall s, s' \in \Sigma_S . s \in INV \wedge s \longrightarrow s' \Rightarrow s' \in INV \end{cases}$$

PROUVONS QUE: A est une propriété de sûreté pour le programme S

PREUVE:Soient deux états s, s' tels $s \in Init_S \wedge s \xrightarrow{\star} s'$. On peut construire une suite d'états $s = s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow \ldots \longrightarrow s_i = s'$. Par l'hypothèse (1), on en déduit que INV est vraie en s. En utilisant (3) pour tous les couples de la trace, on en déduit que INV est vraie en $s_1, s_2, \ldots s_i$. Puis, nous appliquons (2) pour l'état s' et nous en déduisons que s' satisfait A.$\square$

$\langle 1 \rangle 2$. SUPPOSONS QUE: $\forall s, s' \in \Sigma . s \in Init_S \wedge s \xrightarrow{\star} s' \Rightarrow s' \in A$

PROUVONS QUE: Il existe une propriété INV telle que

$$\begin{cases} (1)\ Init_S \subseteq INV \\ (2)\ INV \subseteq A \\ (3)\ \forall s, s' \in \Sigma_S . s \in INV \wedge s \longrightarrow s' \Rightarrow s' \in INV \end{cases}$$

PROUVONS QUE: A est une propriété de sûreté pour le système S

PREUVE:On considère la propriété suivante $INV \stackrel{def}{=} \{u | u \in \Sigma \wedge \exists s \in \Sigma . s \in Init_S \wedge s \xrightarrow{\star} u\}$. INV exprime que l'état s' est un état accessible à partir d'un état initial de S. $\mathcal{R}^\star$ est la fermeture réflexive transitive de $\mathcal{R}$. Les trois propriétés sont simples à vérifier pour INV. INV est appelé le plus fort invariant du système S. □

$\langle 1 \rangle 3$. Q.E.D.

PREUVE:On en déduit l'équivalence par les pas $\langle 1 \rangle 1$ et $\langle 1 \rangle 2$.□

□

Cette propriété justifie la méthode de preuve par induction plus connue comme la méthode de Floyd/Hoare [20, 24], imaginée par Turing en 1949 [44]. La propriété énoncée donne une forme de l'induction qu'il faut ramener à des formes plus connues.

La propriété prop :induction définit un principe d'induction et nous donnons dans la section suivante des variations sur ce principe.

1.4.2 Principes d'induction

P. et R. Cousot [19] donne une synthèse complète sur les principes d'induction équivalents à ce principe d'induction :

☺ **Propriété 1.14**

Les propriétés suivantes sont équivalentes :

1. $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq i \\ (2)\ \ i \subseteq A \\ (3)\ \ \forall s, s' \in \Sigma_{\mathrm{P}}. s \in i \wedge s \longrightarrow s' \Rightarrow s' \in i \end{array}\right.$

2. $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq i \\ (2)\ \ i \subseteq A \\ (3)\ \ \forall s' \in \Sigma_{\mathrm{P}}. \left(\exists s \in \Sigma_{\mathrm{P}}. s \in i \wedge s \longrightarrow s'\right) \Rightarrow s' \in i \end{array}\right.$

3. $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq i \\ (2)\ \ i \subseteq A \\ (3)\ \ \forall s' \in \Sigma_{\mathrm{P}}. s \in i \Rightarrow \neg\left(\exists s' \in \Sigma_{\mathrm{P}}. s \longrightarrow s' \wedge s' \notin i\right) \end{array}\right.$

4. $\exists\ i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq \neg i \\ (2)\ \ \neg A \subseteq i \\ (3)\ \ \forall s \in \Sigma_{\mathrm{P}}. \left(\exists s' \in \Sigma_{\mathrm{P}}. s \longrightarrow s' \wedge s' \in i\right) \Rightarrow s \in i \end{array}\right.$

5. $\exists\ i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq \neg i \\ (2)\ \ \neg A \subseteq i \\ (3)\ \ \forall s' \in \Sigma_{\mathrm{P}}. s' \in i \Rightarrow \neg\left(\exists s \in \Sigma_{\mathrm{P}}. s \longrightarrow s' \wedge s \notin i\right) \end{array}\right.$

PREUVE:

$\langle 1 \rangle 1$. $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq i \\ (2)\ \ i \subseteq A \\ (3)\ \ \forall s, s' \in \Sigma_{\mathrm{P}}. s \in i \wedge s \longrightarrow s' \Rightarrow s' \in i \end{array}\right.$

si, et seulement si, $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq i \\ (2)\ \ i \subseteq A \\ (3)\ \ \forall s' \in \Sigma_{\mathrm{P}}. \left(\exists s \in \Sigma_{\mathrm{P}}. s \in i \wedge s \longrightarrow s'\right) \Rightarrow s' \in i \end{array}\right.$

PREUVE:$\forall s, s' \in \Sigma_{\mathrm{P}}. s \in i \wedge s \longrightarrow s' \Rightarrow s' \in i$ est équivalent à

$\forall s' \in \Sigma_{\mathrm{P}}. \left(\exists s \in \Sigma_{\mathrm{P}}. s \in i \wedge s \longrightarrow s'\right) \Rightarrow s' \in i$, par application de la règle suivante : $\forall x.(P(x) \Rightarrow Q) \equiv ((\exists x.P(x)) \Rightarrow Q)$, si x n'a pas d'occurence libre dans Q. □

$\langle 1 \rangle 2$. $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l} (1)\ \ Init_{\mathrm{P}} \subseteq i \\ (2)\ \ i \subseteq A \\ (3)\ \ \forall s' \in \Sigma_{\mathrm{P}}. \left(\exists s \in \Sigma_{\mathrm{P}}. s \in i \wedge s \longrightarrow s'\right) \Rightarrow s' \in i \end{array}\right.$

si, et seulement si, $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l}(1)\ \ Init_{\mathrm{P}} \subseteq i\\(2)\ \ i \subseteq A\\(3)\ \ \forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow \neg\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin i\Big)\end{array}\right.$

PREUVE:$\forall s' \in \Sigma_{\mathrm{P}}.\Big(\exists s \in \Sigma_{\mathrm{P}}.s \in i \wedge s \longrightarrow s'\Big) \Rightarrow s' \in i$ est équivalent à $\forall s, s' \in \Sigma_{\mathrm{P}}.s \in i \wedge s \longrightarrow s' \Rightarrow s' \in i$, d'après la propriété précédente. $\forall s, s' \in \Sigma_{\mathrm{P}}.s \in i \wedge s \longrightarrow s' \Rightarrow s' \in i$ peut s'écrire de façon équivalente sous la forme $\forall s, s' \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow (s \longrightarrow s' \Rightarrow s' \in i)$, en appliquant la règle suivante : $P \wedge Q \Rightarrow R \equiv P \Rightarrow (Q \Rightarrow R)$.
$\forall s, s' \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow (s \longrightarrow s' \Rightarrow s' \in i)$ est équivalent à $\forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow (\forall s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \Rightarrow s' \in i)$, puisque s' n'est pas libre dans $s \in i$.
$\forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow (\forall s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \Rightarrow s' \in i)$ est équivalent à $\forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow (\neg\exists s' \in \Sigma_{\mathrm{P}}.\neg(s \longrightarrow s' \Rightarrow s' \in i))$
$\forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow \neg\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin i\Big)$ peut s'écrire de façon équivalente en $\forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow \neg\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin i\Big)$, par application de la règle : $\neg(P \Rightarrow Q) \equiv (P \wedge \neg Q)$.
$\square$

$\langle 1\rangle 3.$ $\exists i \in \mathcal{P}(\Sigma). \left[\begin{array}{l}(1)\ \ Init_{\mathrm{P}} \subseteq i\\(2)\ \ i \subseteq A\\(3)\ \ \forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow \neg\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin i\Big)\end{array}\right.$

si, et seulement si, $\exists\ i \in \mathcal{P}(\Sigma). \left[\begin{array}{l}(1)\ \ Init_{\mathrm{P}} \subseteq \neg i\\(2)\ \ \neg A \subseteq i\\(3)\ \ \forall s \in \Sigma_{\mathrm{P}}.\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \in i\Big) \Rightarrow s \in i\end{array}\right.$

PREUVE:Soit i une assertion telle que $\left[\begin{array}{l}(1)\ \ Init_{\mathrm{P}} \subseteq i\\(2)\ \ i \subseteq A\\(3)\ \ \forall s \in \Sigma_{\mathrm{P}}.s \in i \Rightarrow \neg\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin i\Big)\end{array}\right.$.
Notons j l'assertion définie par $\neg i$. Si nous substituons $\neg j$ à i, nous obtenons les propriétés suivantes :
$\left[\begin{array}{l}(1)\ \ Init_{\mathrm{P}} \subseteq \neg j\\(2)\ \ \neg j \subseteq A\\(3)\ \ \forall s \in \Sigma_{\mathrm{P}}.s \in \neg j \Rightarrow \neg\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin \neg j\Big)\end{array}\right.$.
$\neg j \subseteq A$ est équivalent à $\neg A \subseteq j$.
$\forall s \in \Sigma_{\mathrm{P}}.s \in \neg j \Rightarrow \neg\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin \neg j\Big)$ est équivalent à $\forall s \in \Sigma_{\mathrm{P}}.\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin \neg j\Big) \Rightarrow s \in j$, en appliquant la règle de la contraposée. Enfin, $\forall s \in \Sigma_{\mathrm{P}}.\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \notin \neg j\Big) \Rightarrow s \in j$ est équivalent à $\forall s \in \Sigma_{\mathrm{P}}.\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \in j\Big) \Rightarrow s \in j$. Nous avons utilisé des transformations par équivalence et cela conduit à la preuve.$\square$

$\langle 1\rangle 4.$ $\exists\ i \in \mathcal{P}(\Sigma). \left[\begin{array}{l}(1)\ \ Init_{\mathrm{P}} \subseteq \neg i\\(2)\ \ \neg A \subseteq i\\(3)\ \ \forall s \in \Sigma_{\mathrm{P}}.\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \in i\Big) \Rightarrow s \in i\end{array}\right.$

si, et seulement si, $\exists\ i \in \mathcal{P}(\Sigma). \left[\begin{array}{l}(1)\ \ Init_{\mathrm{P}} \subseteq \neg i\\(2)\ \ \neg A \subseteq i\\(3)\ \ \forall s' \in \Sigma_{\mathrm{P}}.s' \in i \Rightarrow \neg\Big(\exists s \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s \notin i\Big)\end{array}\right.$

PREUVE:$\forall s \in \Sigma_{\mathrm{P}}.\Big(\exists s' \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s' \in i\Big) \Rightarrow s \in i$ est équivalent à $\forall s, s' \in \Sigma_{\mathrm{P}}.\Big(s \longrightarrow s' \wedge s' \in i\Big) \Rightarrow s \in i$, en appliquant la règle : $\forall x.(P(x) \Rightarrow Q) \equiv ((\exists x.P(x)) \Rightarrow Q)$, si x n'a pas d'occurence libre dans Q. $\forall s, s' \in \Sigma_{\mathrm{P}}.\Big(s \longrightarrow s' \wedge s' \in i\Big) \Rightarrow s \in i$ est équivalent à $\forall s' \in \Sigma_{\mathrm{P}}.s' \in i \Rightarrow \Big(\forall s \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \Rightarrow s \in i\Big)$, qui est ensuite tranformé en $\forall s' \in \Sigma_{\mathrm{P}}.s' \in i \Rightarrow \neg\Big(\exists s \in \Sigma_{\mathrm{P}}.s \longrightarrow s' \wedge s \notin i\Big)$. $\square$

$\langle 1\rangle 5.$ Q.E.D.

PREUVE:Les étapes $\langle 1\rangle 1,\ \langle 1\rangle 2,\ \langle 1\rangle 3, \langle 1\rangle 4$ constituent la preuve complète. $\square$

$\square$

Nous appliquons ces résultats au cas des modèles relationnels de système et nous obtenons une expression de la définition d'une propriété de sûreté dans le cas d'un modèle relationnel de système.

1.5 Notes bibliographiques

Les techniques de spécification et de vérification se sont imposées très tôt. En effet, le problème est de montrer pourquoi un programme satisfait la spécification. Turing [44] a proposé une méthode de preuves de machines de Turing, reprise plus tard par Floyd [20] et par Hoare [24]. P. et R. Cousot [18] analysent les différents principes d'induction que l'on peut construire et que l'on peut utiliser pour concevoir des méthodes de preuves de propriétés de programmes séquentiels. Pour le cas des programmes parallèles, les techniques ont été étendues par Owicki et Gries [40] et montrent que le nombre de preuves à réaliser devient trés important dans la mesure où il faut prendre en compte la notion d'interaction.

Chapitre 2

Environnements pour la modélisation et la vérification de systèmes informatiques

DANS le chapitre 1, nous avons examiné les propriétés de sûreté des systèmes modélisés comme des systèmes de transition discrète. Un des objectifs est de montrer qu'une méthode de correction de propriétés de sûreté peut être facilement définie pour n'importe quel langage de programmation. Certes, cela nécessite un peu de travail notamment pour définir correctement les questions des passages de paramètres ou des spécificités propres à certains langages. De plus, nous n'avons jamais limité cette technique au seul cas des algorithmes ou programmes séquentiels et elle pourra être étendue aux cas des programems parallèles ou celui des programmes répartis. Dans ce chapitre, nous introduisons (au moins) deux environnements mettant en œuvre la méthode du chapitre 1 pour deux langages de modélisation *Event-B* [6, 2, 12] avec Rodin [4] et TLA/TLA$^+$ [30, 31] avec TLAPS [43]. Ces deux langages de modélisation utilisent la théorie des ensembles [10] pour décrire les données manipulées avec deux points de vue différents : en *Event-B*, les données sont des ensembles (ensembles finies, relations, fonctions totales ou partielles) et en TLA$^+$, les données sont des fonctions totales et des ensembles.

2.1 Le langage de modélisation *Event-B*

La méthode *Event-B* [2, 12] s'appuie sur un langage de modélisation permettant de décrire des modèles à états et les propriétés de sûreté de ces modèles à états. *Event-B* s'attache à exprimer des modèles de système caractérisés par leur invariant et par des propriétés de sûreté. On peut néanmoins considérer les propriétés de fatalité comme UNITY [13] ou TLA$^+$ [31, 30] mais dans un cadre limité. Nous utilisons le langage *Event-B* et ses environnements pour vérifier des algorithmes et les conditions de vérifications qui sont associées. Nous verrons en troisième année que ce langage propose une voie pour vérifier mais aussi surtout pour développer des systèmes corrects par construction avec la relation de raffinement de modèles Event-B.

Event-B désigne à la fois le langage et la méthode utilisant ce langage. Les concepts de ce langage sont limités et permettent à l'utilisateur de gérer une palette réduite : axiome, théorème, théorie, événement, machine, raffinement. Nous allons présenter ces notions de manière plus détaillée dans ce qui suit mais il est assez clair que les exemples constituent des moyens opérationnels pour comprendre par le jeu avec les outils mettant en œuvre ce langage.

La construction d'un modèle *Event-B* repose sur des constructions syntaxiques comme les ensembles, les constantes, les axiomes, les théorèmes, les variables, les invariants, les événements ; ces constructions syntaxiques sont organisées dans des structures de deux types :

— les contextes rassemblent les informations statiques du domaine d'étude : les ensembles, les constantes, les axiomes, les théorèmes ; ces contextes sont extensibles et réutilisables au travers d'une clause spécifique permettant de les étendre ; ils sont utilisés pour définir la théorie mathématqiue du problème et apportent aux outils de preuve les informations nécessaires pour assister l'utilisateur dans la preuve des obligations de preuve.

— les machines organisent la définition des transitions du système en cours de modélisation et des changements des variables (d'état) du système ; elles utilisent les contextes pour faire référence à des informations statiques requises. Ces transitions d'état sont définies par des événements qui *réagissent* selon l'état courant des variables. Les variables d'une machine sont caractérisées par une liste de propriétés appelées *invariants* et peuvent aussi satisfaire des propriétés appelées *théorèmes*. Enfin, comme tout contexte peut être étendus par un autre contexte, une machine peut être raffinée par une autre machine.

— Que cela soit un contexte ou une machine *Event-B*, leur cohérence doit être démontrée par la preuve des *obligations de preuve* engendrées par les outils et conformes aux résulatats exposés dans la section précédente. Si ces obligations de preuve sont démontrées, alors la structure est valide au moins du moins de vue du typage. En effet, le point délicat est l'explicitation et la preuve des propriétés d'invariance d'une machine donnée mais le raffinement est un outil très important pour assurer une progression dans la définition d'un modèle d'un système.

Nous allons donc décrire chaque élément mentionné ci-dessus sans entrer dans des détails de justification mais la section précédente apporte une explication des structures de modélisation en *Event-B*. L'important est aussi d'avoir des illustrations pratiques de la modélisation en *Event-B*. Nous résumons dans le diagramme ci-contre les relations entre les structures de *Event-B*et la forme d'un développement d'une modélisation d'un système. Cette modélisation est très rarement à un seul niveau et doit bénéficier au maximum du raffinement.

2.1.1 Eléments de base d'un modèle *Event-B*

Nous allons commencer par définir les événements qui sont au cœur de cette méthode et qui réagissent à une condition appelée une garde. Un événement est donc caractérisé par une condition et par une action. On retiendra trois uniques formes possibles pour les événements et cela suffira pour modéliser les systèmes au sens général. La première forme constitue une forme normale dans le sens où on peut réduire les deux suivantes sous cette forme. Le second cas correspond à un événement gardé et le troisième cas correspond à un événement quantifié qardé. Intuitivement, l'observation d'un événement est faite dans le cas où la garde est vraie mais le fait que la garde soit vraie ne permet pas d'en déduire que l'événement est ou sera observé. Chaque événement peut être défini par une relation before-after notée $BA(x, x')$.

Un événement est caractérisé par sa garde qui est déterminée au moment de la modélisation et il ne peut être déclenché que si cette garde est vraie. D'une certaine mesure, cela signifie qu'une condition apparaît et que la partie transformation associée est exécutée.

Nous allons détailler les obligations de preuve engendrées pour un événement donné e et expliquer la signification de ces obligations de preuve. Dans notre exposé, nous avons souligné le rôle du raffinement qui est défini sur les événements. La forme générale d'un événement est la suivante :

EVENT e
  ANY t
  WHERE
    $G(c, s, t, x)$
  THEN
    $x :|(P(c, s, t, x, x'))$
  END

— c et s désignent les constantes et les ensembles visibles par l'événement e et sont définis dans un contexte.
— x est la variable d'état ou une liste de variables.
— $G(c, s, t, x)$ est la condition d'activation de e.
— $P(c, s, t, x, x')$ est le prédicat établissant la relation entre la valeur avant de x, notée x, et la valeur après de x, notée x'.
— $BA(e)(c, s, x, x')$ est la relation *before-after* associée à e et définie par $\exists t.G(c, s, t, x) \ \wedge\ P(c, s, t, x, x')$.

Pour chaque événement e, les obligations de preuve sont désignées selon le format suivant : $e/inv/ < type >$ où $< type >$ est soit INV, soit FIS, GRD, SIM, THM, $WFIS$, WD, ...et correspondent à des obligations de preuves engendrées pour garantir l'invariance, le renforcement de la garde, la simulation, une propriété de sûreté, une définition d'une valeur, ... Nous ne recherchons pas l'exhaustivité et renvoyons le lecteur au livre de J.-R. Abrial [2] pour un exposé complet, ainsi qu'à la plateforme Rodin. Nous allons donner quelques éléments pour comprendre comment sont engendrées ces obligations de preuves.

2.1.2 Propriétés d'invariance en *Event-B*

L'invariant $I(x)$ d'un modèle est une propriété invariante pour tous les événements du système modélisé y compris l'événement initial. Si e est un événement du modèle, alors la condition de préservation de cet invariant par cet événement est la suivante : $I(x) \ \wedge\ BA(e)(c, s, x, x') \ \Rightarrow\ I(x') \quad (INV)$. $I(x)$ est écrit sous la forme d'une liste de prédicats étiquettés $inv_1, \ldots inv_n$ et est interprétée comme une conjonction. La condition sur les conditions initiales est la suivante : $Init(x, s, c) \;\Rightarrow\; I(x) \quad (INIT)$.

Quand un événement$e$ définit le prédicat before-after $BA(e)(c, s, x, x')$, la faisabilité de cet événement signifie que sous l'hypothèse définie par l'invariant $I(x)$ et par la garde $\mathsf{grd}\,(e)$, de l'événement, il existe toujours x' tel que $BA(e)(c, s, x, x')$; en d'autres termes cela veut dire que cet événement, quand il est observé, n'induit pas des comportements non souhaités et nous donnons une condition pour chaque événement : $I(x) \;\wedge\; \mathsf{grd}\,(e) \;\Rightarrow\; \exists x' \cdot BA(e)(c, s, x, x') \quad (FIS)$.

Les propriétés de sûreté sont déduites par la preuve que l'invariant du système implique la propriété de sûreté $A(x)$ et nous ajoutons en plus, le contexte $C(s, c)$ de cette preuve. Le contexte de cette preuve est donné par les propriétés $C(s, c)$ des ensembles s et des constantes c définies dans le modèle : $C(s, c) \;\wedge\; I(x) \;\Rightarrow\; A(s, c, x) \quad (THM)$.

Pour conclure ce point des obligations de preuve, elles ont été dérivées du théorème 13 et nous pouvons donc en conclure la propriété suivante.

☺ **Propriété 2.15**

Soit $Th(s, c)$ une théorie définie par les ensembles s, les constantes c et les propriétés $C(s, c)$ et soit une liste finie E d'événements modifiant x et définis dans le contexte de la théorie $Th(s, c)$. On considère les points suivants :

— VALSest un ensemble de valeurs possibles pour x.
— $\{r_0, \ldots, r_n\}$ est l'ensemble des relations $BA(e)(s, c, x, x')$ définies pour les événements e de E et l'un des événements correspond à l'événement $skip$.
— $\textsc{Init}(x)$ est le prédicat définissant les conditions initiales de x.

Si les obligations de preuves (INIT) et (INV) sont satisfaites, alors le modèle relationnel $(Th(s, c), x, \textsc{Vals}, \textsc{Init}(x), \{r_0, \ldots,$
satisfait l'invariant $I(x)$ et les propriétés de sûreté $A(s, c, x)$.

Nous allons maintenant donner les différentes obligations de preuves engendrées à partir de la forme générale donnée plus haut. On suppose que le contexte de la théorie est $C(s, c)$ et nous utiliserons la notation $C(s, c) \vdash P$ pour exprimer l'obligation de preuve dans le contexte $C(s, c)$. Nous avons donc les reformulations suivantes :

— *INIT/I/INV* : $C(s, c), INIT(c, s, x) \vdash I(c, s, x)$
— *e/I/INV* : $C(s, c), I(c, s, x), G(c, s, t, x), P(c, s, t, x, x') \vdash I(c, s, x')$
— *e/act/FIS* : $C(s, c), I(c, s, x), G(c, s, t, x) \vdash \exists x'.P(c, s, t, x, x')$

Nous avons donc instancié les principes d'induction pour assurer l'invariance de I. Le générateur d'obligations de preuve effectue aussi des simplifications importantes dans les énoncés produits. Il reste à définir les structures de machines et de contextes.

2.1.3 Structures des machines et des contextes en *Event-B*

Un contexte rassemble les définitions des objets statiques du modèle du système à développer. Les ensembles de base sont définis dans la clause SETS et son *a priori* quelconques ; on peut déclarer qu'ils sont finis par le prédicat $finite(S)$.

CONTEXT $\mathcal{D}$
EXTENDS $\mathcal{AD}$
SETS
  $S_1, \dots S_n$
CONSTANTS
  $C_1, \dots, C_m$
AXIOMS
  $ax_1 : P_1(S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $ax_p : P_p(S_1, \dots S_n, C_1, \dots, C_m)$
THEOREMS
  $th_1 : Q_1(S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $th_q : Q_q(S_1, \dots S_n, C_1, \dots, C_m)$

— Les constantes sont déclarées dans la clause CONSTANTS.
— Les axiomes sont énumérés dans la clause AXIOMS et définissent les propriétés des constantes.
— Les théorèmes sont des propriétés déclarées dans la clauseTHEOREMS et doivent être démontrées valides en fonction des axiomes.
— Le contexte définit une théorie logico-mathématique qui doit être consistante.
— La clause EXTENDS étend le contexte mentionné et étend donc la théorie définie par le contexte de cette clause.

Un environnement de preuve $\Gamma(\mathcal{D})$ permet de formaliser le contexte dans un cadre logique et les propriétés suivantes doivent être prouvées logiquement :

$$\text{for any } j \text{ in } \{1..q\}, \Gamma(\mathcal{D}) \vdash th_j : Q_j(S_1, \dots S_n, C_1, \dots, C_m).$$

MACHINE $\mathcal{M}$
REFINES $\mathcal{AM}$
SEES $\mathcal{D}$
VARIABLES x
INVARIANTS
  $inv_1 : I_1(x, S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $inv_r : I_r(x, S_1, \dots S_n, C_1, \dots, C_m)$
THEOREMS
  $th_1 : SAFE_1(x, S_1, \dots S_n, C_1, \dots, C_m)$
  $\dots$
  $th_s : SAFE_s(x, S_1, \dots S_n, C_1, \dots, C_m)$
EVENTS
  EVENT initialisation
    BEGIN
      $x : |(P(x'))$
    END
  $\dots$
  EVENT e
    ANY t
    WHERE
      $G(x, t)$
    THEN
      $x : |(P(x, x', t))$
    END
  $\dots$
END

— Une machine est un modèle décrivant un ensemble d'événements modifiant des variables déclarées dans la clause VARIABLES.
— Un événement particulier définit l'initialisation des variables : EVENT Initialisation
— Une clause INVARIANTS décrit l'invariant que cette machine est supposée respecter à condition que les conditions de vérification associées soient démontrées valides dans la théorie induite par le contexte mentionné par la clause SEES.
— Enfin, la clause THEOREMS introduit la liste des propriétés de sûreté dérivées dans la théorie induite par le contexte et l'invariant; ces propriétés portent sur les variables et doivent être démontrées valides.

Les conditions de vérification sont les suivantes :

— For any j in $\{1..r\}$,

$$\Gamma(\mathcal{D}, M) \vdash INITIALISATION(x') \Rightarrow I_j(x', S_1, \dots S_n, C_1, \dots, C_m)$$

— For any j in $\{1..r\}$, for any event e of M,

$$\Gamma(\mathcal{D}, M) \vdash \begin{pmatrix} \left(\bigwedge_{j \in \{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m) \right) \wedge BA(e)(x, x') \\ \Rightarrow \\ I_j(x', S_1, \ldots S_n, C_1, \ldots, C_m) \end{pmatrix}$$

— For any k in $\{1..s\}$,

$$\Gamma(\mathcal{D}, M) \vdash \begin{array}{c} \left(\bigwedge_{j \in \{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m) \right) \\ \Rightarrow \\ SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m) \end{array}$$

— For any j in $\{1..r\}$,
$\mathcal{D}, M \longrightarrow \Box I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m)$.

— For any k in $\{1..s\}$,
$\mathcal{D}, M \longrightarrow \Box SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m)$.

$$\mathcal{D}, M \longrightarrow \Box \begin{pmatrix} \left(\bigwedge_{j \in \{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m) \right) \\ \left(\bigwedge_{k \in \{1..s\}} SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m) \right) \end{pmatrix}$$

Ces conditions contiennent des notations logiques et mathématiques de la déduction qui seront explicitées dans le chapitre de la logique. Il faut simplement retenir que l'expression $\Gamma(\mathcal{D}, M) \vdash \varphi$ signifie que la formule φ est valide dans la théorie définie par $\mathcal{D}$ et M.

2.1.4 Vérification d'un algorithme annoté

L'annotation [44, 20, 24] est donc une technique pour vérifier la correction d'un algorithme par rapport à sa précondition et sa postconditions. Plus généralement les annotations sont des asseryions placées dans le texte d'un programme ou d'un algorithme. Par exemple, l'algorithme2.1.4 calcule la valeur de l'indice d'un tableau contenant la valeur maximale. Les nnotations sont définies par l'utilisateur qui veut expliquer ce que fait l'algorithme. Dans notre cas, nous nnotons l'algorithme en vue de vérifier des conditions de vérification parfois appelées *obligations de preuve* . Les annotations dépendetn des n pre/post spécifications et sont assez difficiles à trouver. En fait, le problème principal est d'identifier la théorie logico-mathèùatique ou le contexte qui soutient le processus de correction de l'algorithme.
Un algroithme annoté ALG peut être tarduit simplement en un modèle *Event-B*et Rodin [4] fournit gratuitement un environnement pour valider les annotations et pour vérifier la correction partelle d'un algorithme par rapport à sa préconditon et à sa postcondition, ainsi que l'absence d'erreurs à lexécution ou plus géérae-ment les propriététs de sûreté.

2.1.5 Transformations des algorithmes annotés en modèles à états

Un algorithme annoté est un algorithm avec une précondition, postcp,ndition et un ensemble d'annotations : une annotaton est une paire étiquette ℓ et assertion $P_\ell(v)$ écrite sous la forme $\ell : \{P_\ell(v)\}$. v modélise les variables d'état de ALG.
Pur un algorithme annoté ALG, LOCATIONS désgne l'ensemble des étidquettes et l'ensemble des valeurs possibles de v est MEMORY. L'ensemble des valeurs possibles VALS est défini par l'ensemble LOCATIONS$\times$MEMORY. L sémantique d'un algorithme annoté est donnée sous la forme d'une sémantqiue relationnelle :

☼**Définition 2.14**
Un modèle relationnel $\mathcal{RM}$(A) est un a quintuple $(Th(s, c), x, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ où
- $Th(s, c)$ est une théorie logico-mathématique definissant des ensembles, des constants, des axiomes pour les ensembles et les onstnates et énonçant des théorèmes relativment à ces ensembles, ces constnates et ces axiomes.
- x est une paire (ℓ, v) où ℓ est la variable modélisant le contrôle par l'usage de points de contrôle de l'agorithme appeléaussi le compteur ordinal et v désigne les variables d'état qui sont effectivement utilisées dans l'algorithme, principalement les variables contenant le résultat et les variabls locales.
- LOCATIONS$\times$MEMORY est l'ensemble des valeurs possibles de x.
- $\{r_0, \ldots, r_n\}$ est un esnemble fini de relations entre les variables primées et non-primées de ALG :

precondition : $\left(n \in \mathbb{N} \wedge n \neq 0 \wedge f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \right)$

postcondition : $\left(m \in \mathbb{N} \wedge m \in ran(f) \wedge (\forall j \cdot j \in 0 \,..\, n-1 \Rightarrow f(j) \leq m) \right)$

local variables : $i \in \mathbb{Z}$

$\ell_0 : \{\left(n \in \mathbb{N} \wedge n \neq 0 \wedge f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \right) \wedge i \in \mathbb{Z}\}$
$m := f(0);$
$\ell_1 : \{\left(n \in \mathbb{N} \wedge n \neq 0 \wedge f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \right) \wedge i \in \mathbb{Z} \wedge m = f(0)\}$
$i := 1;$
$\ell_2 : \{\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \end{pmatrix} \wedge i = 1 \wedge \begin{pmatrix} m \in \mathbb{N} \wedge m \in ran(f[0..i-1]) \wedge \\ (\forall j \cdot j \in 0 \,..\, i-1 \Rightarrow f(j) \leq m) \end{pmatrix}\}$

while $i < n$ **do**

$\ell_3 : \{\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \end{pmatrix} \wedge \begin{pmatrix} i \in 1..n-1 \wedge \\ m \in \mathbb{N} \wedge m \in ran(f[0..i-1]) \wedge \\ (\forall j \cdot j \in 0 \,..\, i-1 \Rightarrow f(j) \leq m) \end{pmatrix}\}$

if *f(i)>m* **then**

$\ell_4 : \{\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \end{pmatrix} \wedge \begin{pmatrix} i \in 1..n-1 \wedge f(i) > m \\ m \in \mathbb{N} \wedge m \in ran(f[0..i-1]) \wedge \\ (\forall j \cdot j \in 0 \,..\, i-1 \Rightarrow f(j) \leq m) \end{pmatrix}\}$

$m := f(i);$

$\ell_5 : \{\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \end{pmatrix} \wedge \begin{pmatrix} i \in 1..n-1 \wedge \\ m \in \mathbb{N} \wedge m \in ran(f[0..i]) \wedge \\ (\forall j \cdot j \in 0 \,..\, i \Rightarrow f(j) \leq m) \end{pmatrix}\}$

;

$\ell_6 : \{\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \end{pmatrix} \wedge \begin{pmatrix} i \in 1..n-1 \wedge \\ m \in \mathbb{N} \wedge m \in ran(f[0..i]) \wedge \\ (\forall j \cdot j \in 0 \,..\, i \Rightarrow f(j) \leq m) \end{pmatrix}\}$

$i{+}{+};$

$\ell_7 : \{\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \end{pmatrix} \wedge \begin{pmatrix} i \in 1..n-1 \wedge \\ m \in \mathbb{N} \wedge m \in ran(f[0..i-1]) \wedge \\ (\forall j \cdot j \in 0 \,..\, i-1 \Rightarrow f(j) \leq m) \end{pmatrix}\}$

;

$\ell_8 : \{\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0 \,..\, n-1 \rightarrow \mathbb{N} \end{pmatrix} \wedge \begin{pmatrix} i = n \wedge m \in \mathbb{N} \wedge m \in ran(f) \wedge \\ (\forall j \cdot j \in 0 \,..\, n-1 \Rightarrow f(j) \leq m) \end{pmatrix}\}$

Algorithme 10: Annotated Algorithm $MAXIMUM$

pour tout $i \in \{0, \ldots, n\}$, $r_i(x, x')$ exprime la modification de x dont la valeur courante est x et les valeurs suivantes notées x'.

La notation primée des variables d'état appelées aussi variables flexibles est empruntée à TLA [30]. L'algorithme annoté induit un graphe sur les étiquettes. Dans la figure 2.1.4, ℓ_0 est relié à ℓ_1, ℓ_2 à ℓ_3, ℓ_0 à ℓ_1, ℓ_2 à ℓ_8, …Le graphe des étiquettes est noté ($\textsc{Locations}, \longrightarrow$). La relation peut être définie par induction sur la syntaxe et nous laissons cela non spécificé. Les relations $r_0, \ldots, r_n$ sont définiées sur les paires étiquettes et elles simulent le calcul de chaque pas élémentaire de l'algorithme.
Plus précisèment, si nous considérons deux étiquettes ℓ et ℓ' telles que $\ell \longrightarrow \ell'$, nous exprimons la condition possible de transition (voir par exemple les itérations while, conditionnelles if et les mises à jour de variables, tandis que le contrôle se déplace de ℓ à ℓ' : $cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)$ est la relation simulant la pas ℓ–ℓ'.

☼**Définition 2.15**
Une propriété de sûreté $\mathrm{S}(x)$ pour ALG est une assertion satisfaisant :
$\forall y, x \in \textsc{Locations} \times \textsc{Memory}. Init(y) \wedge (r_0 \vee \ldots r_n)^\star(y, x) \Rightarrow S(x)$.

Le principe d'induction sur la structure par les relations relie les propriétés de sûréte et les propriétés d'invariant ce comme suit :

Propriété 2 $\mathrm{S}(x)$ *est une propriété de sûreté pour* ALG*, si, et seulement si, il existe une assertion* $I(x)$ *sur les états telle que :*
$\forall x, x' \in \textsc{Locations} \times \textsc{Memory}$:
$$\begin{cases} (1)\ \textsc{Init}(x) \Rightarrow \mathrm{I}(x) \\ (2)\ \mathrm{I}(x) \Rightarrow \mathrm{S}(x) \\ (3)\ \forall i \in \{0, \ldots, n\} : \mathrm{I}(x) \wedge x\ r_i\ x' \Rightarrow \mathrm{I}(x') \end{cases}$$

La propriété $\mathrm{I}(x)$ est un invariant de l'algorithme annoté ALG. Puiswue x est défini selon deux dimensions, le point de contrôle ℓ et l'état mémoire v, toute assertion peut être décomposée pr rapport à la variable de contrôle. L'invariant $I(x)$ est décomposé en une famille de prédicats d'états exprimant les propriétés de la variable de mémoire v comme suit : $\mathrm{I}(x) \equiv \bigvee_{\ell \in \textsc{Locations}} \left(\bigvee_{v \in \textsc{Memory}} x = (\ell, v) \wedge P_\ell(v) \right)$. Les conditions de vérification sont exprimées de l façon suivante :
$\forall x, x' \in \textsc{Locations} \times \textsc{Memory}$:
$$\begin{cases} (1)\ \textsc{Init}(x) \Rightarrow \mathrm{I}(x) \\ (2)\ \mathrm{I}(x) \Rightarrow \mathrm{S}(x) \\ (3)\ \forall i \in \{0, \ldots, n\} : \mathrm{I}(x) \wedge x\ r_i\ x' \Rightarrow \mathrm{I}(x') \end{cases}$$
et elles sont transformées dans les conditions équivalentes suivantes :
$\forall v, v' \in \textsc{Memory}$:
$$\begin{cases} (1)\ \forall \ell \in \textsc{Inputs}.\textsc{Precondition}(v) \Rightarrow P_\ell(v) \\ (2)\ \forall \ell \in \textsc{Locations}.P_\ell(v) \Rightarrow \mathrm{S}(\ell, v) \\ (3)\ \forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v') \end{cases}$$

Propriété 3 *Considérons une propriété* $\mathrm{S}(\ell, v)$ *pour un algorithme* ALG *et une famille d'annotations* $\{P_\ell(v) : \ell \in \textsc{Locations}\}$ *pour l'algorithme. Si les conditions de vérification suivantes sont vérifiées :*
$\forall v, v' \in \textsc{Memory}$:
$$\begin{cases} (1)\ \forall \ell \in \textsc{Inputs}.\textsc{Precondition}(v) \Rightarrow P_\ell(v) \\ (2)\ \forall \ell \in \textsc{Locations}.P_\ell(v) \Rightarrow \mathrm{S}(\ell, v) \\ (3)\ \forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v') \end{cases},$$
alors $\mathrm{S}(\ell, v)$ *est une propriété de sûreté satisfaite par l'algorithme* ALG.

La dernière propriétét donne une technique générale pour vérifier que l'annotation d'un algorithme est correcte pour la correction partielle ou pour l'absence d'erreurs à l'exécution or toute autre propriété de sûreté. La technique générale est fondée sur une traduction de l'algorithme annoté en une liste de paires d'étiquettes. Nous appliquons notre transformation pour notre algorithme and nous obtenons les conditions de vérification suivantes :

— $\ell_0 \longrightarrow \ell_1 : \begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0\,..\,n{-}1 \rightarrow \mathbb{N} \\ \wedge i \in \mathbb{Z} \end{pmatrix} \wedge m' = f(0) \wedge i' = i \Rightarrow \begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0\,..\,n{-}1 \rightarrow \mathbb{N} \wedge \\ i' \in \mathbb{Z} \wedge m' = f(0) \end{pmatrix}$

— $\ell_3 \longrightarrow \ell_4$:

$$\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge \\ f \in 0\,..\,n{-}1 \rightarrow \mathbb{N} \wedge i \in 1..n{-}1 \wedge \\ \begin{pmatrix} m \in \mathbb{N} \wedge m \in ran(f[0..i{-}1]) \wedge \\ (\forall j \cdot j \in 0\,..\,i{-}1 \Rightarrow f(j) \leq m) \end{pmatrix} \end{pmatrix} \wedge f(i) > m \wedge (m', i') = (m, i) \Rightarrow$$

$$\begin{pmatrix} n \in \mathbb{N} \wedge n \neq 0 \wedge f \in 0\,..\,n{-}1 \rightarrow \mathbb{N} \wedge i' \in 1..n{-}1 \wedge \\ \begin{pmatrix} m' \in \mathbb{N} \wedge m' \in ran(f[0..i'{-}1]) \wedge \\ (\forall j \cdot j \in 0\,..\,i'{-}1 \Rightarrow f(j) \leq m') \end{pmatrix} \wedge f(i) > m' \end{pmatrix}$$

P. Cousot et R. Cousot [18]ont étudié les différents principes d'induction et les méthodes de preuves dérivées pour les propriétés d'invariance des programmes parallèles. Nous avons simplement appliqué leurs résultats dans le cas spécial des algorithmes annotés en utilisant *Event-B*/Rodin pour valider les conditions de vérification de l'algorithme annoté. Nous avons obtenu une caractérisation des conditions de vérification pour une propriété de sûreté pour une propriété de sûreté donnée et nous allons montrer comment Rodin peut être simplement utiliser pour vérifier les conditions de vérification mécaniquement.

2.1.6 Vérifier les conditions de vérification avec Rodin

La question est de truver un moyen pour mécaniser la vérification ds condition de vérification et de développer une mécanique générale pour vérifier les propriététs de sûreté d'algorithme annoté. Nous utilisons le langage *Event-B*pour exprimer la relation entre deux points de contrôle successifs et nous définissons deux composants : un contexte C et une machine M. Tout d'abord, nous définissons un événement $\mathcal{E}(\ell, \ell')$ lié à la transformation débutant en ℓ et terminant à ℓ'. Il est définit comme suit :

$\mathcal{E}(\ell, \ell')$
WHEN
$c = \ell$
$cond_{\ell,\ell'}(v)$
THEN
$c := \ell'$
$v := f_{\ell,\ell'}(v)$
END

— v est la variable de l'état mémoire ou la liste des variables de l'tat mémoire ; v inclut les variables locales et les variables résultat.
— c est une nouvelle variable qui modélise le flôt de contrôle de type LOCATIONS.
— $\mathcal{E}(\ell, \ell')$ simule le calcul débutant en ℓ et terminant en o ℓ' ; v est mise à jour.

Nous définissons l'invariant et ajoutons les informations sur le typage pour les variables, comme définis dans le contexte C. C contient les définitions des points de contrôle (LOCATIONS), des données définies dans la préconditions et des fonctions mathématiques requises pour exprimer des propriétés mathématiques.

INVARIANTS
$inv_i : c \in$ LOCATIONS
$inv_j : v \in Type$
...
$inv_\ell : c = \ell \Rightarrow P_\ell(v)$
$inv_{\ell'} : c = \ell' \Rightarrow P_{\ell'}(v)$
...
$th_n : S(c, v)$

— $Type$ est le type des variables v et est un ensemble de valeurs possibles définies dans le contexte C.
— L'annotation donne gratuitement les conditions satisfaites par v qyand le contrôle est en ℓ, (resp. en ℓ').
— $S(c, v)$ est une propriété de sûreté à vérifier et est une théorème dans le cas de *Event-B*.

Nou avons obtenu une machine M qui utilise les données de C. La iste des conditions de vérification produites pour M correspond aux conditions de vérification pour les propriétés de sûreté.

Propriété 4 *Pour toute paire d'étiquettes successives ℓ, ℓ', les trois énncés suivants sont équivalents :*
— $P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$
— $I(c, v) \wedge c = \ell \wedge cond_{\ell,\ell'}(v) \wedge c' = \ell' \wedge v' = f_{\ell,\ell'}(v) \Rightarrow (c' = \ell' \Rightarrow P_{\ell'}(v'))$
— $I(c, v) \wedge BA(\mathcal{E}(\ell, \ell'))(c, v, c', v') \Rightarrow (c' = \ell' \Rightarrow P_{\ell'}(v'))$

La preuve est facile et utilse des manipulations des connecteurs logiques. Deuxièmement, nous observons que la propriété suivante est vraie pour chaque étiquette : $m \neq \ell'$, $I(c,v) \wedge BA(\mathcal{E}(\ell,\ell'))(c,v,c',v') \Rightarrow (c' = m \Rightarrow P_m(v'))$. Cela veur fire que, si les conditions de vérifications de la machine M sont vérifiées, alors l'algorithme annoté satisfait la propriété de sûreté $S(c,v)$.

Propriété 5 *soit* ALG *un algorithme annoté avec comme précondition* ***pre(*** ALG *)(v) et postcondition* ***post(*** ALG *)(v₀,v). Soit le contexte C et la machine M engendrée à partir de* ALG *en utilisant la construction présentée précédemment. Nous supposons que ℓ_0 est l'étiquette input et ℓ_e est l'étiquette output. Nous ajoutons les propriétés de sûreté suivantes dans la machine M :*

— $c = \ell_0 \wedge \textbf{\textit{pre}}(\text{ALG})(v) \Rightarrow P_{\ell_0}(v)$

— $c = \ell_e \Rightarrow (P_{\ell_e}(v) \Rightarrow \textbf{\textit{post}}(\text{ALG})(v_0,v))$

Si les conditions de vérification de M sont vérifiées et démontrées, alors l'algorithme annoté ALG *est partiellement correct par rapport à la pré/post spécification.*

La justification est fondée sur la définition de la correction partielle et la traduction de l correction partielle en tant que propriété de sûreté. Nous appliquons la technique sur l'algorithme annoté et nous obtenons l'invariant suivant et les événements :

$$
\begin{array}{l}
inv1 : l \in C, inv2 : m \in \mathbb{N}, inv3 : i \in \mathbb{N}, inv4 : i \in 0\,..\,n \\
inv5 : l = l0 \Rightarrow m \in \mathbb{N} \wedge i \in \mathbb{N} \\
inv6 : l = l1 \Rightarrow m = f(0) \\
inv7 : l = l2 \Rightarrow \left(\begin{array}{l} i = 1 \wedge m = f(0) \wedge i \leq n \wedge 0\,..\,i-1 \subseteq dom(f) \\ (\forall j \cdot j \in 0\,..\,i-1 \Rightarrow f(j) \leq m) \wedge m \in ran(f) \end{array}\right) \\
inv8 : l = l3 \Rightarrow \left(\begin{array}{l} i < n \wedge 0\,..\,i \subseteq dom(f) \\ (\forall j \cdot j \in 0\,..\,i-1 \Rightarrow f(j) \leq m) \wedge m \in ran(f) \end{array}\right) \\
inv9 : l = l4 \Rightarrow \left(\begin{array}{l} i < n \wedge 0\,..\,i \subseteq dom(f) \\ (\forall j \cdot j \in 0\,..\,i-1 \Rightarrow f(j) \leq m) \wedge f(i) > m \wedge m \in ran(f) \end{array}\right) \\
inv10 : l = l5 \Rightarrow \left(\begin{array}{l} i < n \wedge 0\,..\,i \subseteq dom(f) \\ (\forall j \cdot j \in 0\,..\,i-1 \Rightarrow f(j) \leq m) \\ (\forall j \cdot j \in 0\,..\,i \Rightarrow f(j) \leq m) \wedge m \in ran(f) \end{array}\right) \\
inv11 : l = l6 \Rightarrow \left(\begin{array}{l} i < n \wedge 0\,..\,i \subseteq dom(f) \\ (\forall j \cdot j \in 0\,..\,i \Rightarrow f(j) \leq m) \wedge m \in ran(f) \end{array}\right) \\
inv12 : l = l7 \Rightarrow \left(\begin{array}{l} i \leq n \wedge 0\,..\,i-1 \subseteq dom(f) \\ (\forall j \cdot j \in 0\,..\,i-1 \Rightarrow f(j) \leq m) \wedge m \in ran(f) \end{array}\right) \\
inv13 : l = l8 \Rightarrow \left(\begin{array}{l} i = n \wedge dom(f) \subseteq 0\,..\,i-1 \\ (\forall j \cdot j \in 0\,..\,i-1 \Rightarrow f(j) \leq m) \wedge m \in ran(f) \end{array}\right) \\
post : l = l8 \Rightarrow \left(\; (\forall j \cdot j \in 0\,..\,n-1 \Rightarrow f(j) \leq m) \wedge m \in ran(f) \; \right) \\
pre : f \in 0\,..\,n-1 \rightarrow \mathbb{N} \wedge i \in 0\,..\,n \wedge m \in \mathbb{N} \Rightarrow m \in \mathbb{N} \wedge i \in \mathbb{N}
\end{array}
$$

EVENT Event-B al2l8
WHEN
$grd1 : l = l2$
$grd2 : i \geq n$
THEN
$act1 : l := l8$
END

EVENT Event-B el3l6
WHEN
$grd1 : l = l3$
$grd2 : f(i) \leq m$
THEN
$act1 : l := l6$
END

— Event-B al2l8 modélise la transition de $l2$ à $l8$ et correspond au cas $i \geq n$.

— Event-B el3l6 modélise la transition de $l3$ à $l6$ et correspond au cas $i \geq n$.

Le nombre de conditions de vérification prouvées est une mesure de la complexité du processus de preuve. Le sommaire des conditions de vérification montre que e processus est principalement automatique : 5 sur 134 (4 %) sont prouvées après quelques interactions et 129 sur 134 (96 %) sont complètement automatiques.

2.1.7 Notations ensemblistes *Event-B*

Cette section présente les notations ensemblistes du langage B [1, 3, 12]. Ce langage est supporté par un outil disponible sous le nom de plate-forme `Rodin` [4] mais aussi est disponible à partir du site de la compagnie ClearSy `Atelier-B` [15]. Ces deux versions vous permettront de vérifier vos écrits et de démontrer des propriétés à partir d'une liste d'axiomes.

Prédicats logiques

Nom	Expression ASCII	Expression logique	Signification intuitive
conjonction	`P & Q`	$P \wedge Q$	P et Q sont vrais tous les deux
disjunction	`P or Q`	$P \vee Q$	P ou Q est vrai
implication	`P => Q`	$P \Rightarrow Q$	P et Q sont vrais ou bien P est faux
équivalenece	`P <=> Q`	$P \Leftrightarrow Q$	P et Q sont vrais tous les deux ou bine P et Q sont faux tous les deux
négation	`not Q`	$\neg P$	P est faux
quantification universelle	`!x.P => Q`	$\forall x.P \Rightarrow Q$	pour toute valeur v de x, $P(v)$ est vrai
quantification existencielle	`#x.P=>Q`	$\exists x.P \Rightarrow Q$	il existe une valeur v telle que $P(v)$ est vrai
égalité de termes	`E = F`	$E = F$	la valeur de E et celle de F sont identiques
inégalité de termes	`E /= F`	$E \neq F$	la valeur de E et celle de F sont différentes

Ensembles

Nom	Expression ASCII	Expression formatée	signification intuitive
ensemble vide	`{}`	$\varnothing$	valeur désignant l'ensemble vide
ensemble singleton	`{E}`	$\{E\}$	ensemble contenant l'unique élément E
ensemble énuméré	`{E1,E2}`	$\{E1, E2\}$	ensemble défini en extension ou en énumération avec deux éléments
ensemble en compréhension	`{x|P}`	$\{x \| P\}$	ensemble défini en compréhension par P
ensemble des parties	`POW(A)`	$\mathbb{P}(A)$	ensemble des parties de l'ensemble A
ensemble des parties non-vides	`POW1(A)`	$\mathbb{P}_1(A)$	ensemble des parties non-vides de l'ensemble A
appartenance	`x:A`	$x \in A$	appartenance de x à A
non appartenance	`x /: A`	$x \notin A$	non appartenance de x à A
inclusion	`S<:T`	$A \subseteq B$	A est inclus dans B
inclusion stricte	`S«T`	$A \subset B$	A est inclus strictement dans B
non-inclusion	`S /<: T`	$A \not\subseteq B$	A n'est pas inclus dans B
cardinalité	`card(A)`	$card(A)$	cardinalité de l'ensemble A
produit cartésien	`A**B`	$A \times B$	produit cartésien de deux ensembles A et B contenant des paires
paire d'éléments	`a|->b`	$a \mapsto b$	paires des éléments $a \in A$ et $b \in B$ et $a \mapsto b \in A \times B$
union	`A \/ B`	$A \cup B$	ensemble union de A et de B
intersection	`A /\ B`	$A \cap B$	ensemble intersection de A et de B
différence	`A \ B`	$A \setminus B$	ensemble différence de A par B

Relations

<table>
<tr><th>Nom</th><th>Expression ASCII</th><th>Expression formatée</th><th>signification intuitive</th></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td>ensemble des parties sur $S{\times}T$</td><td><code>S<->T</code></td><td>$S \leftrightarrow T$</td><td>ensemble des relations entre S et T</td></tr>
<tr><td>domaine de définition de r</td><td>dom(r)</td><td>$dom(r)$</td><td>domain of relation</td></tr>
<tr><td>image de r</td><td>ran(r)</td><td>$ran(r)$</td><td>range of relation</td></tr>
<tr><td>identité sur S</td><td>id(S)</td><td></td><td>$id(S)$ identity relation</td></tr>
<tr><td>restriction de r sur S</td><td><code>S<|r</code></td><td>$S \lhd r$</td><td>domain restriction</td></tr>
<tr><td>soustraction au domaine de r de S</td><td><code>S«|r</code></td><td>$S ⩤ r$</td><td>domain subtraction</td></tr>
<tr><td>restriction de l'image de r à S</td><td><code>r|>S</code></td><td>$r \rhd S$</td><td>range restriction</td></tr>
<tr><td>soustraction de S à l'image de r</td><td><code>r|»S</code></td><td>$r ⩥ S$</td><td>range subtraction</td></tr>
<tr><td>relation inverse de r</td><td><code>r~</code></td><td>r^{-1}</td><td>inverse of relation</td></tr>
<tr><td>image de r des éléments de S</td><td><code>r[S]</code></td><td>$r[S]$</td><td>relational image</td></tr>
<tr><td></td><td><code>r1<+r2</code></td><td>$r1 \lhd\!\!\!+ r2$</td><td>right overriding</td></tr>
<tr><td></td><td><code>r1><r2</code></td><td>$r1 \otimes r2$</td><td>direct product <code>{x,(y,z) | x,y:r1 & x,z:r2}</code></td></tr>
<tr><td></td><td><code>r1;r2</code></td><td>$r1;r2$</td><td>relational composition <code>{x,y| x|->z:r1 & z|->y:r2}</code></td></tr>
<tr><td></td><td><code>prj1(S,T)</code></td><td></td><td>projection</td></tr>
<tr><td></td><td><code>prj2(S,T)</code></td><td></td><td>projection</td></tr>
</table>

Fonctions

Nom	Expression ASCII	Expression formatée	signification intuitive
fonctions totales	`A -> B`	$A \rightarrow B$	valeur désignant l'ensemble des fonctions totales de A dans B
fonctions partielles	`A +-> B`	$A \nrightarrow B$	valeur désignant l'ensemble des fonctions totales de A dans B
Injections partielles	`A >+> B`	$A \rightarrowtail\!\!\!\!\!+ B$	valeur désignant l'ensemble des fonctions partielles injectives de A dans B
Injections totales	`A >-> B`	$A \rightarrowtail B$	valeur désignant l'ensemble des fonctions totales injectives de A dans B
Surjections partielles	`A +-» B`	$A \twoheadrightarrow\!\!\!\!\!+ B$	valeur désignant l'ensemble des fonctions partielles surjectives de A dans B
Surjections totales	`A -» B`	$A \twoheadrightarrow B$	valeur désignant l'ensemble des fonctions totales surjectives de A dans B
Bijections totales	`A >-» B`	$A \rightarrowtail\!\!\!\!\twoheadrightarrow B$	valeur désignant l'ensemble des fonctions totales surjectives de A dans B

2.2 Modélisation de systèmes concurrents et répartis avec TLA/TLA$^+$

La logique temporelle des actions TLA est introduite par Leslie Lamport en 1989 et la présentation dans un document publié date de 1994 [30]. L'idée de cette logique est de fournir un cadre simplifié aux concepteurs ou aux spécifieurs, pour modéliser des systèmes et leurs propriétés dans la même logique. La spécification des propriétés est réduite aux propriétés de sûreté et aux propriétés de fatalité sous hypothèse d'équité ; l'hypothèse d'équité faite sur certains composants est formalisable en TLA. A partir de 1994, Leslie Lamport développe progressivement un langage de spécifications étendant TLA par ds notations de la théorie des ensembles et une structuration sous forme de modules. Le lanage de spécification TLA$^+$est présenté de manière complète dans l'ouvrage de Leslie Lamport [32].

2.2.1 La logique temporelle des actions TLA

Prédicats et actions

La logique temporelle des actions TLA repose sur la logique classique comprenant les connecteurs logiques et les quantificateurs logiwues classiques et sur une extension temporelle reposant sur des variables primées ou non-primées.
Soit un ensemble de variables Var et une ensemble de variables primées $pVar$. Une variable x de Var correspond à une variable priéme x' de $pVar$. Intuitivement, une variable primée correspond à la valeur suivante d'une variable flexible et une variable non-primée correspond à la valeur d'une variable flexible.
Une variable flexible est une variable au sens informatique ; elle admet une valeur dans un état donné et peut en changer ; elle a un nom qui appartient à Var.
On se donne un ensemble d'états $States$ définis comme des fonctions totales de l'ensemble des variables flexibles Var dans l'ensemble des valeurs possibles noté Val :

$$States \triangleq Var \longrightarrow Val \tag{2.1}$$

Soit une variable flexible x de Var et un état s de $States$. La valeur de x en s est notée $s[\![x]\!]$ et est définie comme suit :

⟨*formula*⟩	$\triangleq$	⟨*predicate*⟩ | $\Box$[⟨*action*⟩]$_{\langle \textit{state function} \rangle}$ | $\neg$⟨*formula*⟩
		| ⟨*formule*⟩ $\wedge$ ⟨*formule*⟩ | $\Box$ ⟨*formula*⟩
⟨*action*⟩	$\triangleq$	expression à valeur booléenne contenat des constantes,
		variables, et variables primées
⟨*predicate*⟩	$\triangleq$	boolean-valued ⟨*state function*⟩ | *Enabled* ⟨*action*⟩
⟨*fonction d'état*⟩	$\triangleq$	expression avec des constantes et des variables

FIGURE 2.1 – Syntaxe des formules TLA

$$s[\![x]\!] \triangleq s(x) \tag{2.2}$$

A partie de cette définition, on peut en déduire la définition de la valeur d'une expression incluant des variables flexibles en utilisant une induction sur la syntaxe. Considérons quelques exemples d'expressions :

1. $s[\![E{+}F]\!] \triangleq s[\![E]\!]{+}_m s[\![F]\!]$ où $+_m$ est l'opération mathématique d'addition.
2. $s[\![E > F]\!] \triangleq s[\![E]\!] >_m s[\![F]\!]$ où $>_m$ est la comparaison dans les nombres entiers.

Par conséquent, la valeur d'une expression quelconque E est définie pour chaque état s et l'écriture $s[\![E]\!]$ a un sens. Il s epeut néanmoins que cette interprétation conduise à des expressions interprétes comme $1 > \text{“}bidule''$ mais nous considérons que les expressions sont bien écrites. La vérification de la bonne écriture des expressions pourrait être faite à l'aide d'un algorithme de typage, par exemple. On peut ensuite définir ce qu'est un prédicat et déterminer la validité d'un prédicat, en étendant les règles sur les expressions aux prédicats. Nous avons déjà expliqué comment défini la valeur d'une expression relationnelle, on simpelment étendre aux connecteurs logiques et aux quantificateurs :

1. $s[\![P \wedge Q]\!] \triangleq s[\![P]\!]\textbf{et}s[\![Q]\!]$ où P et Q sont deux prédicats.
2. $s[\![P \vee Q]\!] \triangleq s[\![P]\!]\textbf{ou}s[\![Q]\!]$ où P et Q sont deux prédicats.

On a construit la classe des prédicats $Pred$ et on peut donner une valeur de vérité à un prédicat P en un état s ; on écrira :

$$s \models P \triangleq (s[\![P]\!]\textbf{est vrai}). \tag{2.3}$$

Les expressions traitées jusqu'à présent n'ont pas de variables primées ; si on ajoute la possibilité d'avoir des variables primées dans les expressions, on construit une autre classe d'objets appelés des actions. Une action A sur Var est un prédicat ayant des occurences de variables primées et de variables non-primées. On note Act la classe des actions sur Var et on suppose que la classe des actions étend la classe des prédicats. La sémantique d'une action $A(x, x')$ sur Var est définie très simplement comme suit :

$$(s, s') \models A(x, x') \triangleq [\![A]\!][s(x)/x, s'(x)/x'] \tag{2.4}$$

L'idée est de remplacer les occurences libres des variables flexibles non-primées par les valeurs en s c'est-à-dire avant et les occurences libres des varaiables primées par les valeurs en s' c'est-à-dire après. La paire (s, s') est appelée un pas. On pourra écrire un pas entre deux états s et s' comme suit :

$$s \xrightarrow{A} s' \tag{2.5}$$

$s[\![f]\!]$	$\triangleq f(\forall$ 'v' $: s[\![v]\!]/v)$	$\sigma[\![F \wedge G]\!]$	$\triangleq \sigma[\![F]\!] \wedge \sigma[\![G]\!]$
$s[\![\mathcal{A}]\!]t$	$\triangleq \mathcal{A}(\forall$ 'v' $: s[\![v]\!]/v,\ t[\![v]\!]/v')$	$\sigma[\![\neg F]\!]$	$\triangleq \neg\sigma[\![F]\!]$
$\models \mathcal{A}$	$\triangleq \forall s, t \in \mathbf{St} : \models s[\![\mathcal{A}]\!]t$	$\models F$	$\triangleq \forall \sigma \in \mathbf{St}^{\infty} : \models \sigma[\![F]\!]$

$$s[\![\mathit{Enabled}\ \mathcal{A}]\!] \triangleq \exists t \in \mathbf{St} : s[\![\mathcal{A}]\!]t$$

$$\langle s_0, s_1, \ldots \rangle[\![\Box F]\!] \triangleq \forall n \in \mathbb{N} : \langle s_n, s_{n+1}, \ldots \rangle[\![F]\!]$$

$$\langle s_0, s_1, \ldots \rangle[\![\mathcal{A}]\!] \triangleq s_0[\![\mathcal{A}]\!]s_1$$

FIGURE 2.2 – Sémantique de la logique temporelle TLA

Traces et formules temporelles

Très naturellement, un système peut être modélisé par une suite d'états ou de pas que nous appelons la trace du système. Une trace du système est définie comme un mot infini $\sigma = (s_0, s_1, \ldots, s_i, \ldots)$ sur $States$ satisfaisant les conditions suivantes :

1. s_0 est un état initial ($s_0 \in IStates$ où $IStates \subseteq States$).
2. $\forall i \geq 0.\exists A \in Act.s_i \xrightarrow{A} s_{i+1}$

Nous avons introduit un ensemble non-vide dont les éléments sont les états initiaux du système courant et nous le notons $IStates$ ($IStates \subseteq States$). Nous avons aussi utilisé un ensemble Act dont les éléments sont les actions possibles du système et cet ensemble est fini. Nous noterons $Traces$ l'ensemble de toutes les traces possibles et nous supposerons qu'il existe une action particulière appelée τ dont l'action est l'indentité sur l'ensemble des variables :

$$\tau(x, x') \triangleq x = x' \tag{2.6}$$

Ainsi, une trace peut étendue indéfiniment et un état peut être répété autant de fois qu'il faut pour produire une trace infinie. Un système S peut être modélisé par la donnée des variables Var, des valeurs des variables Val et un ensemble d'actions Act.
Le concept central de TLA est la notion de bégaiement ou stuttering ; en fait, on va considérer que deux traces sont identiques modulo le bégaiement et les variables d'observation. En fait, un ensemble de traces caractérise un système donné mais on peut être intéressé par la formulation sous forme temporelle de cet ensemble de traces. TLA définit un système par un prédicat qui définit les états initiaux, une relation entre les variales primées et non-primées du système et des contraintes d'équité.

Spécification temporelle

2.2.2 Le langage de spécifications TLA$^+$

La modélisation d'un système repose sur la définition d'objets formel comme par exemple un ensemble de valeurs entières comprises entre 0 et 12 ou bien encore un graphe connexe acyclique. Cette modélisation requiert la possibilité de défінri des strucrures de données. TLA$^+$offre cette possibilité de définir des objets formels modélisant des éléments du problème à résoudre. Nous donnons un premier exemple d'un tel élément de ce langage.

Exemple 2.8 (tiré de [32])
TLA$^+$structure les spéciifications ou les modèles de système sous forme de module. Un module $HourClock$ décrit une montre ou une horloge d'heures. Une variable hr est déclaré ; une action $HCnext$ est définie, ainsi que le prédicat d'initialisation de cette variable $HCini$. HC est la défintion en TLA du système. Enfin, un théorème est énoncé pour donner la propriété satisfaite par ce système.

MODULE *HourClock*

$$
\begin{array}{llll}
f' & \triangleq f(\forall \text{ `}v\text{' } : v'/v) & \Diamond F & \triangleq \neg\Box\neg F \\
[\mathcal{A}]_f & \triangleq \mathcal{A} \vee (f' = f) & F \leadsto G & \triangleq \Box(F \Rightarrow \Diamond G) \\
\langle\mathcal{A}\rangle_f & \triangleq \mathcal{A} \wedge (f' \neq f) & \mathrm{WF} f \mathcal{A} & \triangleq \Box\Diamond\langle\mathcal{A}\rangle_f \vee \Box\Diamond\neg \mathit{Enabled}\ \langle\mathcal{A}\rangle_f \\
\mathit{Unchanged}\ f & \triangleq f' = f & \mathrm{SF} f \mathcal{A} & \triangleq \Box\Diamond\langle\mathcal{A}\rangle_f \vee \Diamond\Box\neg \mathit{Enabled}\ \langle\mathcal{A}\rangle_f
\end{array}
$$

where f est une ⟨*fonction d'état*⟩ — $s, s_0, s_1, \ldots$ are states

$\mathcal{A}$ est une ⟨*action*⟩ — σ est un comportement

F, G sont ⟨ *des formules*⟩ — $(\forall \text{ `}v\text{' } : \ldots/v, \ldots/v')$ denotes substitution for all variables v

FIGURE 2.3 – Notations de la logique temporelle TLA

STL1. $\dfrac{F \text{ provable by propositional logic}}{F}$

STL4. $\dfrac{F \Rightarrow G}{\Box F \ \Rightarrow \ \Box G}$

STL2. $\vdash \Box F \ \Rightarrow \ F$

STL5. $\vdash \Box(F \wedge G) \ \equiv \ (\Box F) \wedge (\Box G)$

STL3. $\vdash \Box\Box F \ \equiv \ \Box F$

STL6. $\vdash (\Diamond\Box F) \wedge (\Diamond\Box G) \ \equiv \ \Diamond\Box(F \wedge G)$

ATTICE. $\succ$ well-founded partial order on nonempty set S

$$\dfrac{F \wedge (\mathsf{c} \in \mathsf{S}) \ \Rightarrow \ (H_{\mathsf{c}} \leadsto (G \vee \exists \mathsf{d} \in \mathsf{S} : (\mathsf{c} \succ \mathsf{d}) \wedge H_{\mathsf{d}}))}{F \ \Rightarrow \ ((\exists \mathsf{c} \in \mathsf{S} : H_{\mathsf{c}}) \leadsto G)}$$

TLA1. $\vdash \Box P \ \equiv \ P \wedge \Box[P \Rightarrow P']_P$

TLA2. $\dfrac{P \wedge [\mathcal{A}]_f \ \Rightarrow \ Q \wedge [\mathcal{B}]_g}{\Box P \wedge \Box[\mathcal{A}]_f \ \Rightarrow \ \Box Q \wedge \Box[\mathcal{B}]_g}$

FIGURE 2.4 – Règles de la logique TLA

INV1. $\dfrac{I \wedge [\mathcal{N}]_f \Rightarrow I'}{I \wedge \Box[\mathcal{N}]_f \Rightarrow \Box I}$

INV2. $\vdash \Box I \;\Rightarrow\; (\Box[\mathcal{N}]_f \;\equiv\; \Box[\mathcal{N} \wedge I \wedge I']_f)$

WF1.

$$\dfrac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q' \\ P \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF} f \mathcal{A} \;\Rightarrow\; (P \leadsto Q)}$$

WF2.

$$\dfrac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{\mathcal{M}} \rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg \mathcal{B}]_f \wedge \mathrm{WF} f \mathcal{A} \wedge \Box F \;\Rightarrow\; \Diamond \Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{WF} f \mathcal{A} \wedge \Box F \;\Rightarrow\; \overline{\mathrm{WF} g \mathcal{M}}}$$

SF1.

$$\dfrac{\begin{array}{l} P \wedge [\mathcal{N}]_f \;\Rightarrow\; (P' \vee Q') \\ P \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; Q' \\ \Box P \wedge \Box[\mathcal{N}]_f \wedge \Box F \;\Rightarrow\; \Diamond \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{SF} f \mathcal{A} \wedge \Box F \;\Rightarrow\; (P \leadsto Q)}$$

SF2.

$$\dfrac{\begin{array}{l} \langle \mathcal{N} \wedge \mathcal{B} \rangle_f \;\Rightarrow\; \langle \overline{M} \rangle_{\overline{g}} \\ P \wedge P' \wedge \langle \mathcal{N} \wedge \mathcal{A} \rangle_f \;\Rightarrow\; \mathcal{B} \\ P \wedge \overline{\mathit{Enabled}\ \langle \mathcal{M} \rangle_g} \;\Rightarrow\; \mathit{Enabled}\ \langle \mathcal{A} \rangle_f \\ \Box[\mathcal{N} \wedge \neg \mathcal{B}]_f \wedge \mathrm{SF} f \mathcal{A} \wedge \Box F \;\Rightarrow\; \Diamond \Box P \end{array}}{\Box[\mathcal{N}]_f \wedge \mathrm{SF} f \mathcal{A} \wedge \Box F \;\Rightarrow\; \overline{\mathrm{SF} g \mathcal{M}}}$$

where F, G, H_c are TLA formulas $\quad$ P, Q, I are predicates

$\mathcal{A}$, $\mathcal{B}$, $\mathcal{N}$, $\mathcal{M}$ are actions $\quad$ f, g are state functions

FIGURE 2.5 – Règles additionnelles

EXTENDS *Naturals*
VARIABLE *hr*
$\mathit{HCini} \;\triangleq\; hr \;\in\; (1\,..\,12)$
$\mathit{HCnxt} \;\triangleq\; hr' \;=\; \text{IF } hr \;\neq\; 12 \text{ THEN } hr + 1 \text{ ELSE } 1$
$\mathit{HC} \;\triangleq\; \mathit{HCini} \;\wedge\; \Box[\mathit{HCnxt}]_{hr}$

THEOREM $\mathit{HC} \;\Rightarrow\; \Box \mathit{HCini}$

La structure de base est le module ; un module a un nom et rassemble des défitions selon certaines règles de cohérence au niveau de l''apparition des éléments utilisés. Dans le texte qui suit, le module inclut les défintions du module $Integers$ contenant les opérateurs, les fonctions et les prédicats sur les entiers. Le mot-clé $VARIABLES$ introduit des nouvelles variables qui pourront mainetnat être utilisées dans toute la suite du module et dans tous les modules qui seront étendus par la clause EXTENDS. Enfin, $ICini$ est une définition nouvelle et le texte de cette définition se trouve à droite du signe $\triangleq$; elle définit des conditions satisfaites par les variables définie précédemment.

EXTENDS *Integers*
VARIABLES *hr*, *min*, *chg*
$\mathit{ICini} \;\triangleq\; \wedge\ hr \;=\; 1$
$\qquad\qquad \wedge\ min \;=\; 3$
$\qquad\qquad \wedge\ chg \;=\; \mathit{FALSE}$

Dans la suite, deux nouvelles défintions sont introduites et elles définissent deux nouvelles actions au sens TLA.

$$
\begin{array}{ll}
ICmin \triangleq & \land \neg ((min = 0) \land chg) \\
& \land\, min' = (min + 1)\ 60 \\
& \land\, chg' = ((min = 59) \land \neg chg) \\
& \land\, hr' = hr \\
IChr \triangleq & \land\, min = 0 \\
& \land \lor (min = 59) \land \neg chg \\
& \quad \lor (min = 0) \land chg \\
& \land\, hr' = (hr\ 12) + 1 \\
& \land\, chg' = \neg chg \\
& \land\, min' = min
\end{array}
$$

Enfin, on trouve des définitions de relation de transition à l'aide des actions précédemment définies.

$$
\begin{array}{l}
ICnxt \triangleq ICmin \lor IChr \\
IIC \triangleq ICini \land \Box[ICnxt]_{\langle hr,\ min,\ chg \rangle} \\
ICinv \triangleq chg \Rightarrow (min \in \{0,\ 59\}) \\
ICtyp \triangleq \land\, hr \in (1..12) \\
\qquad\quad \land\, min \in (0..59) \\
\qquad\quad \land\, chg \in \{TRUE,\ FALSE\}
\end{array}
$$

2.2.3 Construction d'un modèle en TLA$^+$

A ce point de l'exposé, nous sommes en face du problème de la construction d'un modèle d'un système donné et il faut envisager des questions sur la validation du modèle obtenu. Pour cela, nous allons envisager une étude de cas complète liée aux services de télécommunications et reprenant un problème posé dans un concours. Nous voulons modéliser les activités liées à la téléphonie, en particulier, les opérations décrocher, raccrocher, numéroter, ... Cet exemple est important car pourra être utilisé pour développer des services et les ajouter à ce modèle. Une question va se poser et elle est en relation avec une validation de la spécification produite. Il est clair qu'il faille s'assurer de certaines propriétés du modèle construit. Nous allons envisager quelques propriétés pour chacun de nos modèles.
Une première propriété concerne la vivacité des actions du modèle ; la vivacité d'une action signifie que cette action est activable dans un état accessible du modèle ; il est clair qu'il s'agit d'une propriété de sûreté et que la vérification de cette vivacité d'une action passe par un calcul d'un invariant modélisant l'ensemble des états accessibles.

2.2.4 Validation d'un modèle construit avec TLA$^+$

Modélisation d'un automate

Le premier exemple de modélisation est le cas d'un automate dont les transitions sont les suivantes :

- Initialement, la valeur d'une variable value est 0.
- La valeur de la variable value est augmentée d'une valeur égale à 1, à condition que le contrôle soit dans la boucle.
- Si le contrôle sort de la boucle, la valeur est augmentée d'une valeur égale à 1.

Deux variables sont utilisées : l'une modélise le contrôle $state$ et l'autre modélise la valeur $value$ calculée. La relation de transition comprend trois disjonctions correspondant aux cas suivants : le premier cas $state$ correspond au cas oû $state$ vaut $dbut$; le second cas correspond au cas où $state$ vaut $boucle$ et enfin le troisième cas correspond au cas oû $state$ vaut $boucl$ mais oû la transition va transformer $state$ en fin.

MODULE *automate*

EXTENDS *Naturals*, *Strings*

VARIABLE *state*, *value*

$Initial \triangleq (state = \text{"début"}) \land (value = 0)$

$Invstate \triangleq state \in \{\text{"début"}, \text{"fin"}, \text{"boucle"}\}$

$$
\begin{aligned}
Next \triangleq\ & \lor\ \land\ state = \text{"début"} \land state' = \text{"boucle"} \land value' = value \\
& \lor\ \land\ state = \text{"boucle"} \land value' = value+1 \land state' = state \\
& \lor\ \land\ state = \text{"boucle"} \land value' = value+1 \land state' = \text{"fin"}
\end{aligned}
$$

$SpecAutomate \triangleq Initial \land [Next]_{\langle state, value \rangle}$

THEOREM $SpecAutomate \Rightarrow \Box Invstate$

L'invariant choisi correspond au typage de la variable $state$. Le fichier de configuration retenu est le suivant :

```
NEXT      Next
INVARIANT Invstate
INIT      Initial
```

Pour tester cette spécification, on peut naturellement modifier l'invariant de manière à mettre en évidence des états possibles accessibles. Par exemple, on peut tester si la valeur 37 est calculable par ce système et on remplace l'invariant par :

$Invstate \triangleq state \in \{\text{"début"}, \text{"fin"}, \text{"boucle"}\} \land value \neq 10$

L'analyse avec TLC conduit à tester si la propriété $value \neq 10$ est toujours vraie ; la conséquence est que l'outil produit un contre-exemple et donne le chemin des actions pour y parvenir.

```
TLC Version 1.01 of 18 Jun 1999
Model-checking
automate
automate
 size of list of intended modules 2
Finished computing initial states: 1 distinct state generated.
Error: Invariant Invstate is violated. The behavior up to this point is:
STATE 1:
/\ value = 0
/\ state = "début"

STATE 2:
/\ value = 0
/\ state = "boucle"

STATE 3:
/\ value = 1
/\ state = "boucle"

STATE 4:
```

```
/\ value = 2
/\ state = "boucle"

STATE 5:
/\ value = 3
/\ state = "boucle"

STATE 6:
/\ value = 4
/\ state = "boucle"

STATE 7:
/\ value = 5
/\ state = "boucle"

STATE 8:
/\ value = 6
/\ state = "boucle"

STATE 9:
/\ value = 7
/\ state = "boucle"

STATE 10:
/\ value = 8
/\ state = "boucle"

STATE 11:
/\ value = 9
/\ state = "boucle"

STATE 12:
/\ value = 10
/\ state = "boucle"

29 states generated, 21 distinct states found. 1 states left on queue.
```

Modélisation et vérification d'un algorithme annoté

On peut utiliser TLA$^+$ pour vérifier un algorithme annoté. Considérons l'exemple très simple d'un algorithme.

precondition : $x = x_0 \wedge x_0 \in \mathbb{N}$
postcondition : $x = 0$

ℓ_0 : $\{ x = x_0 \wedge x_0 \in \mathbb{N}\}$
while $0 < x$ **do**
  $\ell_1 : \{O < x \leq x_0 \wedge x_0 \in \mathbb{N}\}$
  $x := x-1$;
  $\ell_2 : \{0 \leq x \leq x_0 \wedge x_0 \in \mathbb{N}\}$
;
$\ell_3 : \{x = 0\}$

Algorithme 11: $EX1$ annotée

MODULE $tp1$

EXTENDS *Integers*
VARIABLES x, pc
CONSTANTS x_0, min, max
ASSUME $x_0 \in Nat \cap min \leq x_0 \cap x_0 \leq max$

Define actions from the text of annotated algorithm

$al0l_1 \triangleq$
$\quad \wedge pc = \text{"l0"}$
$\quad \wedge 0 < x$
$\quad \wedge pc' = \text{"l1"}$
$\quad \wedge x' = x$

$al0l_3 \triangleq$
$\quad \wedge pc = \text{"l0"}$
$\quad \wedge \neg(0 < x)$
$\quad \wedge pc' = \text{"l3"}$
$\quad \wedge x' = x$

$al1l_2 \triangleq$
$\quad \wedge pc = \text{"l1"}$
$\quad \wedge TRUE$
$\quad \wedge pc' = \text{"l2"}$
$\quad \wedge x' = x-1$

$al2l_1 \triangleq$
$\quad \wedge pc = \text{"l2"}$
$\quad \wedge 0 < x$
$\quad \wedge pc' = \text{"l1"}$
$\quad \wedge x' = x$

$al2l_3 \triangleq$
$\quad \wedge pc = \text{"l2"}$
$\quad \wedge \neg(0 < x)$
$\quad \wedge pc' = \text{"l3"}$
$\quad \wedge x' = x$

Define the computation relation

$Next \triangleq$
$\quad \vee al0l_1$
$\quad \vee al0l_3$
$\quad \vee al1l_2$
$\quad \vee al2l_1$
$\quad \vee al2l3$

Define the initial conditions

$Init \triangleq pc = \text{"l0"} \wedge x = x0$

Define the invariant from the annotation

$i \triangleq$
$\quad \wedge pc = \text{"l0"} \Rightarrow x = x0$
$\quad \wedge pc = \text{"l1"} \Rightarrow 0 < x \wedge x \leq x0$
$\quad \wedge pc = \text{"l2"} \Rightarrow 0 \leq x \wedge x < x0$
$\quad \wedge pc = \text{"l3"} \Rightarrow x = 0$

Define safety

$Q1 \triangleq x \geq 0$ ok

$Q2 \triangleq x \leq -6$ ko

$Qpost \triangleq pc = \text{"l3"} \Rightarrow x = 0$ correction partielle : ok

$Qoverflowfree \triangleq min \leq x \wedge x \leq max$

Modification History

Last modified Mon Dec 14 21 :19 :44 CET 2015 by mery

Created Wed Sep 09 17 :02 :47 CEST 2015 by mery

2.3 Outils et plateformes

Les outils mettant en œuvre le langage de modélisation *Event-B* sont d'une part l'Atelier B [15] et d'autre part Rodin [4]. Pour TLA$^+$, il s'agit de TLAPS [43] avec aussi un outil partagé avec la communauté B et celle de CSP, ProB [23].

2.3.1 Atelier B

L'Atelier B [15] est diffusé gratuitement par la société ClearSy qui le propose pour les quatre plateformes Windows, Linux, MacOs, Solaris ; une diffusion sous licence est proposée et permet d'avoir accès à un certain nombre de documentation et d'études de cas. Cette plateforme propose dans un même cadre la méthode B Classique avec des outils de traduction mais aussi *Event-B* avec une syntaxe légèrement différente. Les fonctionnalités offertes incluent la génération d'obligations de preuve, l'assistance à la preuve interactive, le raffinement automatique avec l'outil Bart [17] et des outils de traductions vers C ou ADA. Cette même société poursuit la diffusion gratuiite d'une plateforme appelée B4Free[16] basée sur les travaux communs de J.-R. Abrial et de D. Cansell sur la balbulette [5]. L'idée de la balbulette était de fournir une interface avec les composants de l'Atelier B comme le générateur d'obligations de preuve, le prouvuer ou des traducteurs, afin de faciliter la tâche du développeur en l'assistant dans la démarche de preuve interactive et lma gestion des projets. Une des difficultés dans l'utilisation d'outil comme l'Atelier B réside dans l'utilisation interactive de l'assistant pour prouver des obligations de preuve qui n'ont pu être traitées par les procédures automatiques. B4Free offre donc des aides durant le processus de preuve et propose des règles à appliquer et ces règles sont ensuite sélectionnées par l'utilisateur. Cet outil a rencontré un grand succès auprès des partenaires académiques et ses fonctionnalités sont intégrées à la plateforme Rodin sous Eclipse.

2.3.2 La plateforme Rodin

La plateforme Rodin est la mise en œuvre de la méthode *Event-B* dans l'environnement Eclipse ; elle fait suite aux travaux menés dans le cadre des outils comme Click'N'Prove [11]. Elle est dédiée uniquement à Event B mais propose des fonctionnalités sous forme de plugins (traduction vers des langages de programmation à partir de modèles *Event-B* ou encore intégration de méthodologies comme UML). La plateforme Rodin a été utilisée pour développer les études de cas illustrant ce texte et nous [36, 39, 37, 38] avons utilisé des outils complémentaires comme ProB [23] qui offre les fonctionnalités à l'assistant de preuve, comme l'animation et le model-checking.

2.3.3 Vérification automatique avec TLAPS

TLAPS [43] est un outil en cours de développement. Il comprend d'une part un éditeur de modules TLA$^+$ et d'autre part des outils permettant de définir et d'analyser les modèles de système dans le langage TLA$^+$. Il fait suite à une première génération d'outils notamment TLC.

TLC est un outil développé par Yun Yan, Jean-Charles Grégoire et Leslie Lamport ; cet outil comprend des composants pour l'analyse syntaxique, le formatage, la simulation et l'analyse exhaustive. TLC est met en œuvre les techniques de model checking et permet de valider les spécifications écrites. On peut utiliser TLC pour réaliser des tâches liées à la validation de spécifications temporelles TLA$^+$; on dispose de plusieurs options :

— `java tlatk.TLC -syntax <module.tla>` vérifie la syntaxe du module dont le nom est $< module.tla >$.

— `java tlatk.TLC -config <fichier.cfg> <module.tla>` engendre les états correspondant à la spécification donnée dans le fichier de configuration et trouve les défintions des éléments du fichier de configuration dans le fichier $< module.tla >$

2.3.4 L'environnement ProB

ProB [23] est un animateur, un solveur de contrainte et un model cjecker pour la méthode B. Il permet une animation automatqieu des modèles B et peut être utilise pour vérifier des spécifications pour une variatét assez large d'erreurs. Il peut être utilisé pour trouver des modèles, pour vérifier la présence de blocages ou pour la génération de tests. ProB propose des modes d'utilisarion pour les lanages *Event-B*, CSP-M, TLA^+ et Z.

2.4 Notes bibliographiques

TLA^+ est un langage de spécifications construit par Leslie Lamport et est un enrichissement de la logique temporelle des actions TLA. TLA et TLA^+ sont présentés dans l'ouvrage «Specifying Systems» [32] de Leslie Lamport ; en outre, Leslie Lamport a mené des développments avec un certain nombre de personnes pour fournir un environnment logiciel permettant de manipuler les spécifications et de les analyser sur des modèles finis et l'outil TLC est le fruit de cette collaboration qui se poursuit. La page `http://lamport.org` contient toutes les références et les renseignements nécessaires sur TLA/TLA^+.

Bibliographie

[1] J.-R. Abrial. *The B book - Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, 2010.

[3] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[4] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin : an open toolset for modelling and reasoning in event-b. *STTT*, 12(6) :447–466, 2010.

[5] Jean-Raymond Abrial and Dominique Cansell. Click'n prove : Interactive proofs within set theory. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.

[6] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[7] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. Spec♯. http ://research.microsoft.com/specsharp/.

[8] D. Bjoener and C. B. Jones. *Formal Specification and Software Development*. Prentice-Hall International, 1982.

[9] Dines Bjørner and Martin C. Henson, editors. *Logics of Specification Languages*. EATCS Textbook in Computer Science. Springer, 2007.

[10] N. Bourbaki. *Théorie des ensembles*. Hermes, 1966.

[11] Dominique Cansell. Click'N'Prove. http ://plateforme-qsl.loria.fr/click

[12] Dominique Cansell and Dominique Méry. *The event-B Modelling Method : Concepts and Case Studies*, pages 33–140. Springer, 2007. See [9].

[13] K. M. Chandy and J. Misra. *Parallel Program Design A Foundation*. Addison-Wesley Publishing Company, 1988. ISBN 0-201-05866-9.

[14] E. M. Clarke, O. Grunberg, and D. A. Peled. *Model Checking*. The MIT Press, 2000.

[15] ClearSy, Aix-en-Provence (F). *Atelier B, Version 3.6*, 2001.

[16] ClearSy, Aix-en-Provence (F). *B4FREE*, 2004. http ://www.b4free.com.

[17] ClearSy, Aix-en-Provence (F). *BART*, 2010. http ://www.atelierb.eu.

[18] P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. In D. Néel, editor, *Tools and Notions for Program Construction*, pages 75–119, 1982.

[19] P. Cousot and R. Cousot. Induction principles for proving invariance properties of programs. In D. Néel, editor, *Tools & Notions for Program Construction : an Advanced Course*, pages 75–119. Cambridge University Press, Cambridge, UK, August 1982.

[20] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Proc. Symp. Appl. Math. 19, Mathematical Aspects of Computer Science*, pages 19 – 32. American Mathematical Society, 1967.

[21] G. Gentzen. *Untersuchungen Uber das Logische Schliessen ou Recherches sur la déduction loqique*. Presses Universitaires de France, 1955. Traduction de Feys et Ladrière.

[22] JML Group. The java modeling language (jml). JML Home Page.

[23] Heinrich-Heine-Universität Düsseldorf. *The ProB Animator and Model Checker*. http ://www.stups.uni-duesseldorf.de/ProB.

[24] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12 :576–580, 1969.

[25] G. Holzmann. The spin model checker. *IEEE Trans. on software engi.*

[26] C. B. Jones. *Software Development : A Rigorous Approach*. Prentice-Hall International, 1980.

[27] C. B. Jones. *Sytematic Software Development Using VDM*. Prentice-Hall International, 1986.

[28] C. B. Jones and R. C. Shaw. *Case Studies in Systematic Software Development*. Prentice-Hall International Series in Computer Science. Prentice-Hall, 1990. ISBN0-13-116088-5.

[29] L. Lamport. Sometime is sometimes Not never : A tutorial on the temporal logic of programs. In *Proceedings of the Seventh Annual Symposium on Principles of Programming Languages*, pages 174–185. ACM SIGACT-SIGPLAN, ACM, 1980.

[30] L. Lamport. A temporal logic of actions. *Transactions On Programming Languages and Systems*, 16(3) :872–923, May 1994.

[31] L. Lamport. *Specifying Systems : The TLA$^{+}+$ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

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

[33] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[34] B. Meyer. *Eiffel : The Language*. Prentice Hall International Ltd., 1992.

[35] C. Morgan. *Programming from Specifications*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[36] Dominique Méry and Neeraj Kumar Singh. Pacemaker's Functional Behaviors in Event-B. Research report, 2009.

[37] Dominique Méry and Neeraj Kumar Singh. Functional Behavior of a Cardiac Pacing System. *International Journal of Discrete Event Control Systems (IJDECS)*, Digital Equipment Corporation 2010.

[38] Dominique Méry and Neeraj Kumar Singh. Technical Report on Formal Development of Two-Electrode Cardiac Pacing System. Research report, February 2010.

[39] Dominique Méry and Neeraj Kumar Singh. Trustable Formal Specification for Software Certification. In T. Margaria and B. Ste, editors, *4th International Symposium On Leveraging Applications of Formal Methods - ISOLA 2010*, volume 6416 of *Lecture Notes in Computer Science*, pages 312–326, Heraklion, Crete, Greece, October 2010. Springer.

[40] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6 :319–340, 1976.

[41] Rise4fun, a community of software engineering tools. http ://rise4fun.com/.

[42] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat : Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[43] The TLA+ Proof System (TLAPS). https ://tla.msr-inria.inria.fr/tlaps/content/Home.html.

[44] A. Turing. On checking a large routine. In *Conference on High-Speed Automatic Calculating Machines*. University Mathematical Laboratory,Cambridge, 1949.