Modèles et algorithmes

Modélisation et vérification de systèmes

28 février 2021

Dominique Méry

Université de Lorraine (Telecom Nancy)
LORIA, BP 239,Campus Scientifique
54506 Vandœuvre-lès-Nancy Cedex, FRANCE
dominique.mery@loria.fr
http://members.loria.fr/mery
version 28 février 2021, commentaires à dominique.mery@loria.fr
Année universitaire 2020-2021

Ce document est édité le 28 février 2021 pendant le cours MALG 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 artifices 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 édition ajoute la pratique des environnements PAT [29] (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) et Frama-C [16] (permettant l'analyse et la preuve *mécanisée* de programmes C réduit). Elle aborde toujours le langage TLA$^+$ [23, 24, 30] et les outils, ainsi que le langage Event-B [9, 2] et les environnements Rodin [4] et Atelier-B [10].

Les outils et les environnements associés comme Frama-C, PAT, Rodin, Atelier-B, TLA$^+$ Toolbox ... qui sont des environnements formels (F-IDE). Dans la mesure où ces environnements contiennent plus ou moins de fonctionnalités avancées comme une procédure de décision ou un assistant de preuve, nous avons ajouté des éléments de logique dans le chapitre *Modélisation et vérification des programmes et des systèmes*.

TABLE DES MATIÈRES

1 Modélisation et vérification des programmes et des systèmes **5**
1.1 Formules logiques 6
1.1.1 Syntaxe de formules propositionnelles 6
1.1.2 Sémantique des propositions 8
1.1.3 Syntaxe de formules du premier ordre 9
1.1.4 Interprétation des formules du calcul des prédicats du premier ordre 11
1.1.5 Sémantique des formules 13
1.2 Modélisation Opérationnelle d'un Système 14
1.3 Propriétés de sûreté et d'invariance dans un modèle relationnel 19
1.4 Conception d'une méthode de preuves de propriétés d'invariance et de sûreté pour un langage de programmation 22
1.4.1 Spécialisation des principes d'induction pour le cas des programmes 22
1.5 Propriétés de correction des programmes 33
1.5.1 Correction partielle d'un programme 35
1.5.2 Absence d'erreurs à l'exécution 36
1.6 Notes bibliographiques 38

CHAPITRE 1

MODÉLISATION ET VÉRIFICATION DES PROGRAMMES ET DES SYSTÈMES

Sommaire

1.1 Formules logiques 6
1.1.1 Syntaxe de formules propositionnelles 6
1.1.2 Sémantique des propositions 8
1.1.3 Syntaxe de formules du premier ordre 9
1.1.4 Interprétation des formules du calcul des prédicats du premier ordre 11
1.1.5 Sémantique des formules 13
1.2 Modélisation Opérationnelle d'un Système 14
1.3 Propriétés de sûreté et d'invariance dans un modèle relationnel . . . 19
1.4 Conception d'une méthode de preuves de propriétés d'invariance et de sûreté pour un langage de programmation 22
1.4.1 Spécialisation des principes d'induction pour le cas des programmes 22
1.5 Propriétés de correction des programmes 33
1.5.1 Correction partielle d'un programme 35
1.5.2 Absence d'erreurs à l'exécution 36
1.6 Notes bibliographiques 38

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 [31], repris par Floyd [15] pour les *flowcharts* ; Hoare [19] 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 [21, 20, 7, 22]. Un certain nombre de langages de programmation [25, 5, 18, 16, 11] 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 le système. Nous utiliserons les notations ensemblistes de B [1, 3, 9] et de TLA$^+$ [23, 24] et nous donnons une explication de ce qui est en jeu dans le cadre de la modélisation. La section suivante présente la définition de ce qui est appelée une formule logique dans la suite.

1.1 Formules logiques

Le calcul des propositions est un premier cadre dans lequel nous allons définir les notions relatives à la syntaxe et à la sémantique comme les modèles, la consistance, la complétude, la déduction. .... Il parait à première vue rudimentaire mais néanmoins permet de bien poser les problèmes des systèmes formels.

1.1.1 Syntaxe de formules propositionnelles

Nous donnons dans cette partie quelques notations très utiles par la suite. Nous définissons la notion de formules qui sont les termes manipulés dans ce système. On note $\mathcal{V}$ un ensemble infini dénombrable d'éléments appelés symboles de variables propositionnelles :
$\mathcal{V} = \{A, B, C, D, \ldots, A_i, B_i, \ldots, P, Q, R, S, \ldots\}$.
On note $\mathcal{C}$ un ensemble d'éléments appelés symboles de connection et de séparation :
$\mathcal{C} = \{\vee, \wedge, \sim, \Rightarrow, \equiv, (,), \ldots\}$
On suppose que $\mathcal{C} \cap \mathcal{P} = \varnothing$.

☼**Définition 1.1** Syntaxe des formules
Une formule est une suite finie de symboles de $\mathcal{C} \cup \mathcal{P}$ obtenue par application finie des règles suivantes :

1. tout symbole de variable propositionnelle est une formule dite atomique.
2. si φ est une formule, alors $\sim (\varphi)$ est une formule.
3. si φ et ψ sont deux formules, alors $(\varphi) \wedge (\psi)$, $(\varphi) \Rightarrow (\psi)$, $(\varphi) \vee (\psi)$, $(\varphi) \equiv (\psi)$ sont des formules.

Une formule sur $\mathcal{C} \cup \mathcal{P}$ est appelée une formule propositionnelle. Dans ce chapitre, une formule sera implicitement propositionnelle. Le parenthésage permet de rendre l'analyse non ambigue. Nous pourrions aussi utiliser une grammaire pour définir l'ensemble des formules.

Exercice 1 *Construire une grammaire définissant les formules ci-dessus.*

Une suite finie de formules $(\varphi_0, \varphi_1, \ldots, \varphi_n)$ est une construction si elle vérifie pour tout i de $\{0, \ldots, n\}$, l'une des conditions suivantes :

1. φ_i est un symbole de variable propositionnelle.
2. il existe $j < i$ tel que φ_i est $\sim (\varphi_j)$.
3. il existe j et k tels que $j < i$ et $k < i$ et φ_i est (φ_j) *op* (φ_k) où *op* $\in \{\wedge, \vee, \Rightarrow, \equiv\}$

La construction $(\varphi_0, \varphi_1 \ldots \varphi_n)$ est une construction de φ_n. Toute sous-suite $(\varphi_0, \varphi_1 \ldots \varphi_i)$ est une construction de φ_i, pour tout i de $\{0, \ldots, n\}$. Une formule φ admet une infinité dénombrable de constructions, il suffit d'ajouter des variables propositionnelles de $\mathcal{P}$ dans une construction de φ. Le nombre d'application des règles (2) ou (3) est l'ordre de complexité de la construction.

Exemple 1.1

1. $(P, (P) \wedge (P), P)$ est une construction de P d'ordre 1.
2. $(P, Q, R, (P) \wedge (Q), ((P) \wedge (Q)) \Rightarrow (R))$ est une construction de $((P) \wedge (Q)) \Rightarrow (R)$ d'ordre 2.
3. $(P, ((P) \wedge (P)) \Rightarrow (P), P)$ n'est pas une construction.

Exercice 2 *Quelle est la relation entre la notion d'ordre d'une construction et la hauteur de l'arbre associé à une formule donnée ?*

Une formule φ se décompose suivant les règles de construction. On définit parallèlement la notion d'arbre de décomposition d'une formule φ. Cette décomposition correspond à la construction d'un arbre syntaxique associé à φ en utilisant les règles de grammaire adéquates.

☼**Définition 1.2** Décomposition d'une formule φ
La décomposition d'une formule φ et la construction d'un arbre de décomposition associé obéissent aux règles suivantes :

1. si φ est un symbole de variable propositionnelle, alors φ ne se décompose pas et son arbre de décomposition est réduit à φ.
2. si φ est obtenue par la règle 2, alors φ se décompose en ψ où φ est $\sim (\psi)$ et son arbre de décomposition est :
   $\sim$
   |
   ψ

3. si φ est obtenue par la règle 3, alors φ se décompose en ψ_1 et ψ_2 où φ est (ψ_1) *op* (ψ_2) et son arbre de décomposition est :
   op

   $\psi_1 \quad \psi_2$

On peut ajouter aux formules une formule de base qui est la constante false. Son statut est celui d'un connecteur d'arité 0. Nous le définirons ainsi $FAUX \stackrel{def}{=} \sim (P) \wedge (P)$. De même, nous pouvons définir une constante true comme suit : $VRAI \stackrel{def}{=} \sim (P) \vee (P)$. le propositioon P est quelconque. Cette définition sera justifiée par la suite.

Exercice 3 *Ecrire une procédure ou une fonction retournant l'arbre de décomposition d'une formule.*

On note $\mathcal{P}rop(\mathcal{P})$, l'ensemble des formules sur $\mathcal{P} \cup \mathcal{C}$.

1.1.2 Sémantique des propositions

Les formules de $\mathcal{P}rop(\mathcal{P})$ ont une existence purement syntaxique. Nous leur associons un sens permettant de signifier les connecteurs. On note $\mathcal{B} = \{0, 1\}$ et $\mathcal{B}_n$, les fonctions n-aires à valeur dans $\mathcal{B}$. Soit $f \in \mathcal{B}_n$ une table de vérité de f est une structure ayant n+1 colonnes et 2^n lignes.

Exemple 1.2

$f \in B_1 :$

0	ϵ_0
1	ϵ_1

$f \in B_2 :$

0	0	ϵ_0
0	1	ϵ_1
1	0	ϵ_2
1	1	ϵ_3

A tout symbole de $\mathcal{C}$, on associe une fonction de $\bigcup_{i \geq 0} \mathcal{B}_i$ ou une table de vérité. Soit $\mathcal{B} = (\mathcal{B}, \bigcup_{i \geq 0} \mathcal{B}_i$. $\mathcal{B}$ est une algèbre. $\mathcal{B}$ est l'algèbre booléenne. Soit φ une formule de $\mathcal{P}rop(\mathcal{P})$. Une fonction booléenne $[\![\varphi]\!]$, dont l'arité est le nombre de variables figurant dans φ, peut être associée à φ. On dit que $[\![\varphi]\!]$ est la fonction de vérité de φ.
Donnons une justification à cette définition. Pour justifier cette définition, nous en donnons une version constructive.
$[\![\ ]\!] \ : \ \mathcal{P}rop(\mathcal{P}) \longrightarrow \bigcup_{i \geq 0} \mathcal{B}_i$

1. Pour toute variable propositionnelle P de $\mathcal{P}$, on associe la fonction notée $[\![P]\!]$ et définie par : $[\![P]\!](\varepsilon) = \varepsilon$, pour tout $\varepsilon \in \mathcal{B}$.
2. Si la formule φ est de la forme $\sim (\psi)$, alors $[\![\varphi]\!] = \overline{[\![\psi]\!]}$.

3. Si la formule φ est de la forme $(\varphi_1)\ op\ (\varphi_2)$, alors $[\![\varphi]\!] = [\![\varphi_1]\!]\ fb(op)\ [\![\varphi_2]\!]$ où $fb(op)$ est la fonction booléenne associée à op.

Une formule φ a donc une sémantique complètement définie.
Une formule φ est une tautologie, si $[\![\varphi]\!]$ est la fonction booléenne uniformément égale à 1. Une formule φ est une antilogie, si $[\![\varphi]\!]$ est la fonction booléenne uniformément égale à 0. Deux formules ayant la même table de vérité sont dites synonymes.

► **Théorème 1.1**

Pour toute fonction booléenne f de $\mathcal{B}^n$, il existe une formule φ écrite avec $\sim, \wedge, \vee$, à partir de n symboles de proposition et telle que $[\![\varphi]\!] = f$.

PREUVE:
1. $f = 0$:
   $f = [\![(A_1 \wedge \sim A_1)) \vee \ldots (A_n \wedge \sim A_n)]\!]$
2. $f \neq 0$:
   Soit $(\varepsilon_1, \ldots, \varepsilon_n)$ telle que $f(\varepsilon_1, \ldots, \varepsilon_n) = 1$. On associe à $(\varepsilon_1, \ldots, \varepsilon_n)$ la formule suivante :
   $\alpha_1 \wedge \ldots \wedge \alpha_n$ où $\alpha_i = \sim A_i$ si $\varepsilon_i = 0$ et $\alpha_i = A_i$, si $\varepsilon_i = 1$
   Alors $f = [\![\bigvee_{j=\{1,\ldots,2^n\}} \bigwedge_{i\in\{1,n\}} \alpha_i^j]\!]$. On suppose que les termes non nuls figurent dans la sommation.

□

Corollaire 1 *Toute formule φ est synonyme d'une forme dite en forme normale disjonctive ou conjonctive.*

On appelle valuation ou réalisation, une application de $\mathcal{P}$ dans $\mathcal{B}$, qui associe à tout symbole de $\mathcal{P}$ une valeur 0 ou 1. On appelle valeur d'une formule φ pour une valuation δ donnée et on note $[\![\varphi]\!](\delta)$, la valeur de $[\![\varphi]\!]$ pour δ.

☼**Définition 1.3** modèle d'une formule propositionnelle
Une valuation δ est un modèle pour φ, si $[\![\varphi]\!](\delta) = 1$, que l'on notera aussi sous la forme "$\delta \models \varphi$".

Soit φ une formule. $\models \varphi$ dénotera le fait que φ est une tautologie c'est-à-dire que toutes les valuations possibles sont modèles de cette formule. Soient deux ensembles de formules Σ_1 et Σ_2 telles que $\Sigma_1 \subseteq \Sigma_2$. Toute valuation modèle des formules de Σ_2 est un modèle des formules de Σ_1.

1.1.3 Syntaxe de formules du premier ordre

Nous avons vu les propriétés de la relation " $\longrightarrow$ " pour le cas des formules propositionnelles. Nous allons introduire la notion de variable d'individu et la notion de quantificateurs. Les classes suivantes sont définies :

1. Symboles logiques : $\sim, \Rightarrow, \forall, \wedge, \vee, \equiv, \exists, \ldots$

2. Symboles de variables : $\mathcal{V} = \{x, y, z, \ldots\}$
3. Symboles de constantes : $\mathcal{C} = \{a, b, c, \ldots\}$
4. Symboles de fonctions : $\mathcal{F} = \{f, g, h, \ldots\}$
5. Symboles de relations : $\mathcal{R} = \{P, Q, R, \ldots\}$

Exemple 1.3

1. $\forall x(\exists y\ (P(x, y) \wedge R(x, y))\ \Rightarrow\ Q(z, x))$
2. $\forall x((>(x, 0))\ \Rightarrow \exists y\ (>(y, 0)\ \Rightarrow\ (-(x, y) < 0)))$

Remarque
la quantification porte sur les variables de constantes. Il n'y a pas de quantification sur les fonctions.

Tout triplet $\mathcal{L} = (\mathcal{C}; \mathcal{R}; \mathcal{F})$ est appelé un langage du premier ordre. Deux catégories d'objets sont à définir :

$\longrightarrow$ les termes de $\mathcal{L}$
$\longrightarrow$ les formules de $\mathcal{L}$

Les termes de $\mathcal{L}$, notés $\mathcal{T}[\mathcal{L}]$, sont les éléments de la $\mathcal{F} \cup \mathcal{C}$-algèbre $M(\mathcal{F}, \mathcal{V})$.
Les formules de $\mathcal{L}$ sont obtenues à partir des formules atomiques de $\mathcal{L}$, noté $\mathcal{A}[\mathcal{L}]$:
$\mathcal{A}[\mathcal{L}] = \{R(t_1, \ldots, t_n)/t_1, \ldots, t_n \in M(\mathcal{F}, \mathcal{V})\}$ et $R \in \mathcal{R}\}$
Les formules atomiques permettent de construire les formules de $\mathcal{L}$. Il faut noter qu'il y a une formule particulière atomique construite à l'aide de la relation d'égalité ; dans ce cas, on parle de calcul des prédicats avec égalité.

Une formule de f de $\mathcal{L}$, ou un élément de $\mathcal{F}(\mathcal{L})$, est :

1. une formule atomique de $\mathcal{A}[\mathcal{L}]$
2. la négation d'une formule : $\sim f$ où $f \in \mathcal{F}(\mathcal{L})$
3. la forme implicative $f_1 \Rightarrow f_2$ où $f_1, f_2 \in \mathcal{F}(\mathcal{L})$
4. la forme quantifiée $\forall x(f)$ où $x \in \mathcal{V}$ et $f \in \mathcal{F}(\mathcal{L})$
5. le résultat de l'application finie des règles 1,2,3,4 et uniquement celles-ci.

L'occurence d'une variable x d'une formule f est liée, si elle se trouve dans la partie d'un quantificateur $\forall x$, sinon on dit qu'elle est libre.

Exemple 1.4
$\forall x(\exists y(P(f(x, y)) \wedge Q(x, z))) \wedge R(x, z)$

— les occurences de z sont libres
— les 2 premières occurences de x sont liées
— la dernière occurence de x est libre
— les occurences de y sont liées

Une variable peut avoir des occurences libres et des occurences liées. Une variable est libre, s'il y a au moins une occurence libre de celle-ci dans la formule considérée. On note $\mathcal{V}(\varphi)$, l'ensemble des variables libres de φ.

Exercice 4 *Donner une définition structurelle de la notion de variable libre dans une formule.*

Soit x une variable, t un terme et φ une formule. On dit que t est libre pour x dans φ, quand aucune occurence libre de x dans φ n'est dans le champ d'un quantificateur liant une variable de t.

Exemple 1.5
$\varphi = \forall y\ (\exists z\ (R(x, y, z)))$

(1) $t = f(x, u)$: t est libre pour x dans φ
(2) $t = f(x, y)$: t n'est pas libre pour x dans φ car y est lié par $\forall$.

Une formule φ est ouverte, s'il y a au moins une occurence libre d'une variable. Sinon elle est close. Nous noterons $\varphi(x_1, x_2, \ldots, x_n)$ une formule ouverte avec $x_1, x_2, \ldots, x_n$ les variables ayant des occurences libres. Une formule libre peut être close par quantification $\forall$:

$\forall x_1(\forall x_2(\ldots(\forall x_n(\varphi(x_1, \ldots, x_n))\ldots))$, c'est la cloture universelle de φ. La formule φ peut être close par $\exists$, c'est la cloture existencielle.

Une formule φ est dite sous forme prénexe, quand elle s'écrit
$Q_1x_1(Q_2x_2(\ldots(Q_nx_n(\varphi(x_1, \ldots, x_n, y_1, \ldots, y_p))$ où $Q_i \in \{\forall, \exists\}$ et $\varphi(x_1, \ldots, y_p)$ sans quantificateur.

Soit t un terme, φ une formule et x une variable de φ. $\varphi[t/x]$ ou $\varphi_x[t]$ est la formule obtenue en remplaçant les occurences libres de x par t où t est libre pour x dans φ. Avant de substituer une variable par un terme, on prendra soin de vérifier cette condition d'application.

Exemple 1.6
Soit $\varphi = \exists y.(x = 2y)$
- $\varphi_x[y{+}1] = \exists y.\ (y{+}1 = 2y)$ n'a pas de sens car $y{+}1$ n'est pas libre pour y dans ϕ.
- $\varphi_x[z{+}5] = \exists y.\ (z{+}5 = 2y)$.

La substitution est donc à manipuler avec soin !

1.1.4 Interprétation des formules du calcul des prédicats du premier ordre

Une interprétation $\mathcal{I}$ de $\mathcal{L}$ ou une réalisation de $\mathcal{L}$ est la donnée :

1. d'un ensemble non-vide D, appelé domaine de $\mathcal{I}$.
2. pour chaque symbole de fonction f de $\mathcal{F}$, d'une fonction ou application $f_{\mathcal{I}}$: $|\mathcal{I}|^n \longrightarrow |\mathcal{I}|$.
3. pour chaque symbole de relation R de $\mathcal{R}$, d'une relation $R_{\mathcal{I}}$ sur D.

Une interprétation $\mathcal{I}$ de $\mathcal{L}$ est une $\mathcal{F}$-algèbre où les relations sont des fonctions à valeurs dans les booléens BOOL. Le symbole d'égalité est interprété par l'égalité. Les constantes seront interprétées par des éléments de D. Le langage étendu pour une interprétation sera obtenu en ajoutant des symboles de constantes pour les valeurs des éléments de D.
Soit une formule close φ et $\mathcal{I}$ une interprétation.

1. φ est atomique :
   φ s'écrit $R(t_1, \ldots, t_n)$ où $R \in \mathcal{R}$ et $t_1 \ldots t_n$ des termes clos (sans variables).
   $\mathcal{I} \models \varphi$ ssi $(t_{1\mathcal{I}}, \ldots, t_{n\mathcal{I}}) \in R_{\mathcal{I}}$
2. φ est $\sim \Psi$:
   $\mathcal{I} \models \varphi$ ssi $non(\mathcal{I} \models \Psi)$
3. φ est $\alpha \Rightarrow \beta$:
   $\mathcal{I} \models \varphi$ ssi $(\mathcal{I} \models \beta \; ou \; \mathcal{I} \models\sim \alpha)$
4. φ est $\forall x(\psi(x))$:
   $\mathcal{I} \models \varphi$ ssi , pour tout i de D, $\mathcal{I} \models \psi(\bar{i}/x)$.

Le point (4) suppose que les quantificateurs lient des variables libres ayant des occurences sinon l'interprétation est lue même celle sans le quantificateur. "$\mathcal{I} \models \varphi$" se lit "I satisfait φ"

Soit φ une formule ouverte. Soit δ une application de $\mathcal{V}$ dans D est une valuation.
On dit qu'une formule ouverte $\varphi(x_1, \ldots, x_n)$ est satisfaite dans une interprétation $\mathcal{I}$, si elle est satisfaite pour toutes les valuations possibles :

$\mathcal{I} \models \varphi(x_1 \ldots x_n)$ ssi $\mathcal{I} \models \forall x_1(\forall x_2 \ldots (\forall x_n \varphi(x_1 \ldots x_n)) \ldots)$.

On note $\mathcal{I}, \delta \models \varphi(x_1 \ldots x_n)$ la satisfaction de φ dans $\mathcal{I}$ pour δ. Cela veut aussi signifier que les occurences de x_i sont substitués par $\delta(x_i)$ dans la, formule $\varphi(x_1 \ldots x_n)$.

☺ **Propriété 1.1**
Soit φ une formule close.

1. φ est satisfaite dans $\mathcal{I}$ ssi $\sim \varphi$ n'est pas satisfaite dans $\mathcal{I}$.
2. Soit $\mathcal{I}$ une interprétation. φ est soit satisfaite, soit non satisfaite.
3. Si $\mathcal{I}$ satisfait φ et $\varphi \Rightarrow \psi$, alors $\mathcal{I}$ satisfait ψ.

Une formule φ est satisfaisable, s'il existe une interprétation $\mathcal{I}$ telle que $\mathcal{I} \models \varphi$.
Une formule φ est valide, si elle est satisfaite dans toutes les interprétations : on note $\models \varphi$.

Deux formules φ et ψ sont équivalentes, si, pour toute interprétation $\mathcal{I}$, $\mathcal{I} \models \varphi$ ssi $\mathcal{I} \models \psi$.

Exemple 1.7

1. $\forall x \varphi(x, x_1, \ldots, x_n)$ équivaut à $\forall y\ \varphi(y, x_1, \ldots, x_n)$.
2. $\forall x\ \varphi(x)$ équivaut à $\sim \exists x \sim \varphi(x)$

Une interprétation $\mathcal{I}$ satisfaisant une formule φ est un modèle de φ. Une formule φ qui admet un modèle est non contradictoire. Une formule valide est une thèse.
Soit Σ un ensemble de formules. Un modèle pour Σ est une interprétation $\mathcal{I}$ satisfaisant chaque formule de Σ :

1. Soit φ et ψ deux formules closes. ψ se déduit sémantiquement de φ, si tout modèle de φ est un modèle de ψ : $\varphi \models \psi$.
2. Soit Σ un ensemble de formules closes et ψ une formule close. $\Sigma \models \psi$, si tout modèle de Σ est un modèle de ψ.

► **Théorème 1.2** Théorème de la déduction sémantique
Soient φ, ψ des formules closes. Soit Σ un ensemble de formules closes.

1. $\varphi \models \psi$ ssi $\models \varphi \Rightarrow \psi$
2. $\Sigma \cup \{\varphi\} \models \psi$ ssi $\Sigma \models \varphi \Rightarrow \psi$

Exercice 5 *Démontrer les 2 résultats du théorème.*

Un ensemble Σ de formules closes est consistant, si Σ a un modèle.
Proposition 1.1
Soit Σ ensemble de formules closes et φ closes. $non(\Sigma \models \varphi)$ ssi $\Sigma \cup \{\sim \varphi\}$ est consistant.
Preuve $\Sigma \cup \sim \varphi$ consistant ssi il existe $\mathcal{I}$ modèle de Σ et de $\sim \varphi$ ssi il existe $\mathcal{I}$ modèle de Σ et non modèle de Σ ssi $non(\Sigma \models \varphi)$. *fin de la preuve*
Cette présentation est empruntée aux logiciens et on peut en donner une version qui se rapproche d'une vue plus proche de la sémantique des langages de programmation en proposant une reformulation assez simple de la sémantique des formules.

1.1.5 Sémantique des formules

Dans la sous-section précédente, nous avons défini les points suivants :

- $\mathcal{L} = (\mathcal{C}; \mathcal{R}; \mathcal{F})$ est un langage du premier ordre où
  - $\mathcal{C}$ désigne l'ensemble des symboles de constantes.
  - $\mathcal{R}$ désigne l'ensemble des des symboles de relations
  - $\mathcal{F}$ désigne l'ensemble des des symboles de fonctions
- $\mathcal{E}(\mathcal{L})$ désigne les énoncés ou les formules construites pour le langage $\mathcal{L}$.
- $\mathcal{V}$ désigne l'ensemble des symboles de variables.
- $\mathcal{O}$ désigne l'ensemble des symboles d'opérateurs logiques ($\wedge$, $\vee$, $\neg$, $\Rightarrow$, ...).

— $\mathcal{I}$ désigne une interprétation pour les formules $\mathcal{E}(\mathcal{L})$ définie par une paire $(D, [\![\bullet]\!])$ où $[\![\bullet]\!]$ est un opérateur associant à toute formule un élément défini sur le domaine D.
— $\mathcal{C}$ désigne l'ensemble des symboles de constantes.
— $\mathcal{S}$ désigne l'ensemble des valuations c'est-à-dire $\mathcal{V} \longrightarrow D$.

L'opérateur $[\![\bullet]\!]$ est défini par induction sur la structure des termes sur $\mathcal{L}$ et sur la structure des fomules de $\mathcal{E}(\mathcal{L})$. La défintion est donnée selon les différents cas syntaxiques des termes et des formules ; on note $s \in \mathcal{S}$ une valuation quelconque et $BOOL = \{ff, tt\}$:

— $[\![c]\!](s) \in D$ où c est un symbole de constantes ($c \in \mathcal{C}$).
— $[\![x]\!](s) = s(x)$ où x est un symbole de variables ($x \in \mathcal{V}$).
— $[\![f(t_1, \ldots, t_n)]\!](s) = f_{[\![I]\!]}([\![t_1]\!], \ldots, [\![t_n]\!])(s)$ où $f_{[\![I]\!]}$ est l'interprétation de f pour l'interprétation $[\![I]\!]$ et $f_{[\![I]\!]} \in D^n \longrightarrow D$.
— $[\![R(t_1, \ldots, t_n)]\!](s) = R_{[\![I]\!]}([\![t_1]\!], \ldots, [\![t_n]\!])(s) \in BOOL$ où $R_{[\![I]\!]}$ est l'interprétation de R pour l'interprétation $[\![I]\!]$ et $R_{[\![I]\!]} \in D^n \longrightarrow BOOL$.
— $[\![TRUE]\!](s) = ttet$[[FALSE]](s)=ff
— $[\![\varphi_1 \; op \; \varphi_2]\!](s) = op_{[\![I]\!]}([\![\varphi_1]\!](s), [\![\varphi_2]\!](s))$ où op est un symbole d' opérateur de $\mathcal{O}$ et leur interprétation est canonique pour les connecteurs logiques usuels.
— $[\![\forall x.\varphi]\!](s) = tt$, si pour tout valeur d de D, $[\![\varphi]\!](s[x \mapsto d]]) = tt$.
— $[\![\exists x.\varphi]\!](s) = tt$, si pour une valeur d de D, $[\![\varphi]\!](s[x \mapsto d]]) = tt$.

Nous avons utilisé la notation $s[x \mapsto d]$ pour désigner la fonction identique à s sauf en x où elle vaut d. Enfin,

Dans notre section précédente, nous avons défini la notation suivante $\mathcal{I}, \delta \models \varphi(x_1 \ldots x_n)$ et la relation avec la définition de $[\![\bullet]\!]$ est la suivante : $\mathcal{I}, \delta \models \varphi$ si, et seulement, $[\![\varphi]\!](\delta) = tt$.

Avant de poursuivre, il est important de noter que la notion de variable logique et celle de variable informatique sont différentes. En effet, une variable logique désigne un emplacement dans une formule pour désigner une valeur constante. Pour une variable informatique, une variable informatique a un nom qui identifie une mémoire dont le contenu peut changer. Le lien entre les deux notions existent, puisque nous pouvons utiliser une variable logique pour représenter la valeur de la variable informatique à un instant donné. Nous pouvons donc utiliser les formules de la logique du premier ordre pour caractériser ce qui est vrai à un instant donné et aussi utiliser les variables logiques pour représenter les valeurs des variables informatiques. On définira des conventions de nommage des variables que nous utiliserons.

1.2 Modélisation Opérationnelle d'un Système

Dans un premier temps, nous allons donner une notation simple pour modéliser ce que l'on observe d'un système selon un niveau d'abstraction. Nous donnons un exemple de système de gestion de l'accès à une salle.

Exemple 1.8 (gestion de l'accès à une salle)
Une salle de classe est observée en tenant compte des personnes qui sont à l'intérieur et deux *modifications* sont possibles soit une ou plusieurs personnes sont entrées, soit une ou plusieurs personnes sont sorties ; de plus, l'occupation de cette salle obéit à deux *propriétés* : il y a aun plus un nombre de 19 personnes à la fois et il y a un nombre entier naturel de personnes. On peut donc identifier à ce niveau d'abstraction une variable d'état

qui contient les personnes présentes au moment où on observe le système. On a ainsi une suite d'observations des états de la salle en question :

$$R_0 = \{\} \xrightarrow{enter(p)} R_1 = \{p\} \xrightarrow{enter(q)} R_2 = \{p, q\} \xrightarrow{exit(p)} R_3 = \{q\} \xrightarrow{skip} R_4 = \{q\} \ldots$$

Les deux modifications observées *enter(p)* et *exit(p)* sont visibles à ce niveau d'abstraction et une modification *skip* est observée. *skip* signifie que les deux configurations R_3 et R_4 sont observées à deux *instants* successifs avec l'observation de *skip*. Le rôle de *skip* est fondamental, puisqu'il permet de modéliser qu'il se passe éventuellement *autre chose* sur des variables non-visibles. Enfin, du point de vue des propriétés attendues de ce système, il est requis par les règles que le nombre de personnes est au plus 19 et cette propriété dite de sûreté est énoncée comme suit :

$$saferoom : 0 \leq card(R) \wedge card(R) \leq 19 \tag{1.1}$$

Si $(R_i : i \in \mathbb{N})$ une suite des valeurs de R construite par application des trois *modifications skip*, *enter(p)* ou *exit(p)*, alors on souhaite que cette suite vérifie la propriété suivante : $\forall i \in \mathbb{N}.0 \leq card(R_i) \wedge card(R_i) \leq 19$. Nous nous posons donc une question simple de la vérification de la propriété pour toutes les valeurs possibles de la variable d'état *R*. Dans ce cas, les valeurs possibles sont les valeurs entières mais restent limitées à celles de l'intervalle entier $0..19$ mais cette exploration des valeurs possibles doit être systématique et le plus simplement automatique.

L'exemple 8 modélise un fonctionnement d'un système de gestion d'une salle et nous pouvons aussi décrire comment fonctionne ou comment évoluent les variables d'un algorithme.
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 informatique 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 :
$s_0 \xrightarrow[R]{} s_1 \xrightarrow[R]{} s_2 \xrightarrow[R]{} s_3 \xrightarrow[R]{} \ldots \xrightarrow[R]{} s_i \xrightarrow[R]{} \ldots$ 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 $s \in c : \alpha$ est une fonction d'abstraction.
2. $s \in \gamma(a)$, si $s \models a : \gamma$ 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 $\Sigma \stackrel{def}{=}$ VARIABLES $\longrightarrow$ 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 par $[\![\varphi(x)]\!]$. La définition inductive a été expliquée au début de ce chapitre et sera reprise dans les chapitres **??** et **??**. L'ensemble des variables logiques est $\mathcal{V}$ et en fait, nous allons utiliser une partie de cet ensemble pour représenter les valeurs des variables d'un système VARIABLES représente la liste des variables qui sera aussi simplement notée x. La notation X désignera les variables mais en tant que variables du système. Cette idée de séparer les variables informatiques et les variables logiques est empruntée à P. Cousot et à L. Lamport. Les variables flexibles de TLA sont des variables représentant des valeurs des variables informatiques et les présentations l'interprétation abstraite utilise des fontes différentes selon la nature des variables.
Pour résumer, X désignera la liste des variables informatiques du système en cours de modélisation ; x désignera les variables logiques correspondant aux valeurs courantes des variables informatiques X. On pourra utiliser d'autres variables que x pour décrire les éléments d'un système notamment ses propriétés mais $\mathcal{V}$ est supposé être assez riche. Σ est l'ensemble des états du système dont les variables sont notées X et dont un élément

est noté $s \in \Sigma$. Nous donnons un exemple d'application des techniques d'interprétation de formules logiques.

Exemple 1.9
Soit $s \in X \longrightarrow D$. X est une variable flexible dont l'instance logique est x.

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

Nous utilisons des notations simplifiant la référence aux états ; ainsi, $[\![x]\!](s)$ est la valeur de x en s et le nom de la variable X et sa valeur ne seront pas distinguées. $[\![X]\!](s')$ est la valeur de X en s' et sera notée x'. Ainsi, $[\![x = 6]\!](s) \wedge [\![y = x{+}8]\!](s')$ 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 [23], 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 1.4** ([23])
Soient $\langle s, t\rangle$ une paire d'états, x la liste des variables ; $[\![x]\!](s)$ est la valeur des variables en s et $]\!]x]\!](t)$ 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 $[\![x]\!]$ en s et celle primées (x') par les valeurs de x, $[\![x]\!]$ en t :

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

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 [23] 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. Cette 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 .
Dans la définition des formules du premier ordre $\mathcal{E}(\mathcal{L})$, nous utilisons des symboles de $\mathcal{V}$ et nous avons mentionné que parmi les variables utilisées dans la définition des actions, deux types d'occurence pouvaient être présentes :

— x et x' désignent deux valeurs de la même variable flexible.
— $[\![x]\!](s) = s(x)$ et on écrit simplement x.

On aboutit à une convention qui est la suivante : parmi les symboles de variables $\mathcal{V}$, il y a des symboles de *variables flexibles* que l'on retrouve dans $\mathcal{W}$ qui est le « double V » et qui signifie que si une variable x est dans $\mathcal{W}$, alors x' est aussi dans $\mathcal{W}$. Enfin, on peut avoir besoin d'ajouter des variables flexibles et, pour cela, on va considérer que les variables flexibles comme x et x' sont représentées par un symbole de variable X. En résumé, pour toute variable flexible X de $\mathcal{W}$, x et x' sont dans $\mathcal{V}$. On peut aussi étendre la notation $[\![\bullet]\!]$ sur $\mathcal{W}$ par $[\![X]\!](s) = s(X)$.
Dans le cas suivant $x \leq 0 \wedge x'{+}y = y'$, l'expression $s[\![x \leq 0 \wedge x'{+}y = y']\!]t$ ou $[\![x \leq 0 \wedge x'{+}y = y']\!](s,t)$ est définie par $[\![x]\!](s) \leq 0 \wedge [\![x]\!](t){+}[\![y]\!](s) = [\![y]\!](t)$. Une convention est que les symboles de variables flexibles appartiennent à $\mathcal{W}$ et sont notés par des lettres majuscules X, Y, Z, ... Dans ce cas, x, y', y, y', z, z' sont dans $\mathcal{V}$.

☼**Définition 1.5** Modèle relationnel d'un système
Un modèle relationnel $\mathcal{MS}$ pour un système $\mathcal{S}$ est une structure

$$(Th(s,c), X, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$$

où

— $Th(s,c)$ est une théorie définissant les ensembles s, les constantes c et les propriétés statiques de ces éléments.
— X est une liste de variables flexibles de $\mathcal{W}$.
— 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'.
— INIT(x) définit l'ensemble des valeurs initiales de x.

Un modèle relationnel $(Th(s,c), X, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ 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 $Id[\text{VALS}]$, identité sur VALS.

☼**Définition 1.6**
Soit $(Th(s,c), X, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ un modèle relationnel d'un système $\mathcal{S}$. La relation NEXT associée à ce modèle est définie par la disjonction des relations r_i :
$\text{NEXT} \stackrel{def}{=} r_0 \vee \ldots \vee r_n$

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 $(Th(s,c), X, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ 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 flexible 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.
- $\overline{x}$ (ou x_f) est la valeur finale de la variable X, quand cette notion a du sens.

x, x', x_0 et x_f sont des variables de $\mathcal{V}$. 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.

1.3 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 1.7**
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 pro-

duire 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 REACHABLE(M).

☺ **Propriété 1.2**
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 déduisons le résultat suivant constituant un principe d'induction pour démontrer les propriétés de sûreté des systèmes modélisés par des modèles relationnels.

☺ **Propriété 1.3** (Principe d'induction d'un modèle relationnel)
Soit $(Th(s, c), X, \text{VALS}, Init(x), \{r_0, \ldots, r_n\})$ un modèle relationnel 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 :

$$\forall x_0, x, x' \in \text{VALS} \ : \ \begin{cases} (1)\ Init(x_0) \Rightarrow I(x_0) \\ (2)\ I(x) \wedge \text{NEXT}(x, x') \Rightarrow I(x') \\ (3)\ I(x) \Rightarrow A(x) \end{cases}$$

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 :

$$\forall x_0, x, x' \in \text{VALS} \ : \ \begin{cases} (1)\ Init(x_0) \Rightarrow I(x_0) \\ (2)\ I(x) \wedge \text{NEXT}(x, x') \Rightarrow I(x') \\ (3)\ I(x) \Rightarrow A(x) \end{cases}$$

PROUVONS QUE: $A(x)$ est une propriété de sûreté pour pour le système S modélisé par M.

PREUVE:
Soient x_0 et $x \in$ VALS tels que $Init(x_0) \wedge \text{NEXT}^\star(x_0, x)$. On peut construire une suite telle que : $x_0 \xrightarrow[\text{NEXT}]{} x_1 \xrightarrow[\text{NEXT}]{} x_2 \xrightarrow[\text{NEXT}]{} \ldots \xrightarrow[\text{NEXT}]{} (x_i = x)$. L'hypothèse (1) nous permet de déduire $I(x_0)$. L'hypothèse (3) nous permet de déduire $I(x_1)$, $I(x_2)$, $I(x_3)$, ..., $I(x_i)$. En utilisant l'hypothèse (2) pour x, nous en déduisons que x satisfait A. □

⟨1⟩2. SUPPOSONS QUE: $\forall x_0, x \cdot x_0, x \in \text{VALS} \wedge Init(x) \wedge \text{NEXT}^\star(x_0, x) \Rightarrow A(x)$
PROUVONS QUE: PROUVONS QUE : il existe une propriété $I(x)$ telle que :

$$\forall x_0, x, x' \in \text{VALS} \ : \ \begin{cases} (1)\ Init(x_0) \Rightarrow I(x_0) \\ (2)\ I(x) \wedge \text{NEXT}(x, x') \Rightarrow I(x') \\ (3)\ I(x) \Rightarrow A(x) \end{cases}$$

PREUVE:Nous considérons la propriété suivante : $I(x) \ \widehat{=} \ \exists y \in \text{VALS} \cdot Init(y) \wedge \text{NEXT}^{\star}(y, x)$. $I(x)$ 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 S. □

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

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

□

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

☺ **Propriété 1.4**

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 1.8** invariant d'un système S

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

$$\forall x_0, x, x' \in \text{VALS} : \begin{cases} (1) \ \text{INIT}(x_0) \Rightarrow \text{I}(x_0) \\ (2) \ \forall i \in \{0, \ldots, n\} : \text{I}(x) \wedge x \ r_i \ x' \Rightarrow \text{I}(x') \end{cases}$$

Enfin, nous avons une propriété importante permettant de comprendre le ien entre la méthode de Floyd [15] et la méthode de Hoare [19].

☺ **Propriété 1.5**

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 : $\forall u.P(u) \Rightarrow Q \equiv (\exists u.P(u)) \Rightarrow Q$. □

L'invariant $I(x)$ exprime que x est une valeur accessible à partir d'une valeur initiale x_0

et l'expression $I(x)$ est une forme assertionnelle de cette propriété. On peut néanmoins définir explicitement cette relation et proposer un style relationnel pour ce principe d'induction [14, 12].

☺ **Propriété 1.6** (Principe relationnel d'induction d'un modèle relationnel)
Soit $(Th(s,c), X, \text{VALS}, Init(x), \{r_0, \ldots, r_n\})$ un modèle relationnel 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 $J(x_0, x)$, telle que :

$$\forall x_0, x, x' \in \text{VALS} \ : \ \begin{cases} (1)\ Init(x_0) \wedge x = x_0 \Rightarrow J(x_0, x) \\ (2)\ Init(x_0) \wedge J(x_0, x) \wedge \text{NEXT}(x, x') \Rightarrow J(x_0, x') \\ (3)\ Init(x_0) \wedge J(x_0, x) \Rightarrow A(x) \end{cases}$$

La propriété de sûreté $A(x)$ porte sur la valeur courante de X et dans le cas de la correction partielle, la postcondition est une relation entre l'état final et l'état initial. On pourrait donc mentionner cette occurence de x_0. 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` [28] et de comprendre comment fonctionnent les environnements comme `TLAPS` [30] ou encore `Rodin` [4]

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

1.4.1 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 1.10
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 : $X, Y, RESULT, CX, CY, CRESULT : int$

$CX := X$;
$CY := 0$;
$CRESULT := {\pi_1}^1(x)$;
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: Algorithme $ADDITION$ pour calculer la fonction primitive récursive $addition$

Exemple 1.11
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 1.12
L'$exponentiation$ est définie comme suit/
$\forall x, y \in \mathbb{N}$:
$$\begin{cases} exponentiation(x, 0) & = & \sigma(\zeta()) \\ exponentiation(x, y{+}1) & = & multiplication(x, exponentiation(x, y)) \end{cases},$$

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

precondition : $x, y \in \mathbb{N}$
postcondition : $result = multiplication(x, y)$
variables : $X, Y, RESULT, CX, CY, CRESULT : int$

$CX := X$;
$CY := 0$;
$cresult := \zeta()$;
while $CY < Y$ **do**
  Invariant : $0 \leq cy \wedge cy < y \wedge cx = x \wedge$
    $cresult = multiplication[cx, cy]$
  $CRESULT := addition[CX, CRESULT]$;
  $CY := CY{+}1$;
;
$RESULT := CRESUKT$;

Algorithme 2: Algorithme $MULTIPLICATION$ pour calculer la fonction primitive recursive function $multiplication$

precondition : $x, y \in \mathbb{N}$
postcondition : $result = exponentiation(x, y)$
variables : $X, Y, CX, CV, RESULRT, CRESULT : int$

$CX := X$;
$CY := 0$;
$CRESULT := \sigma(\zeta())$;
while $CY < Y$ **do**
  Invariant : $0 \leq cy \wedge cy < y \wedge cx = x \wedge$
    $cresult = exponentiation[cx, cy]$
  $CRESULT := MULTIPLICATION[CX, CRESULT]$;
  $CY := CY{+}1$;
;
$result := cresult$;

Algorithme 3: Algorithme $EXPONENTIATION$ pour calculer la fonction primitive récursive $exponentiation$

precondition : $x, y \in \mathbb{N}$
postcondition : $result = multiplication(x, y)$
variables : $V, Y, RESULT, CX, CY, CRESULT : int$

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

Algorithme 4: $MULTIPLICATION$ annotée

ensemble fini de valeurs et une variable cachée notée ℓ parcourt cet ensemble selon l'enchaînement.
La forme de présentation des algorithmes annotés met en œuvre la notion de *programmation par contrat* et conduit donc à une *vérification par contrat*. Pour cela, nous adoptons le langage des programmes annotés qui est utilisé dans les environnements comme GNAT[11], Frama-C citeframac, Spec#[6] ...et nous donnons une forme générale qui sera utilisée pour illustrée la mise en œuvre du principe d'induction pour les propriétés de sûreté : en particulier, la correction partielle et l'absence d'erreurs à l'exécution.
Un programme P *remplit* un contrat (pre,post) :

- P transforme une variable x à partir d'une valeur initiale x_0 et produisant une valeur finale x_f : $x_0 \xrightarrow{\mathrm{P}} x_f$
- x_0 satisfait pre : $\mathsf{pre}(x_0)$ and x_f satisfait post : $\mathsf{post}(x_0, x_f)$
- La relation à vérifier est celle de correction partielle : : $\mathsf{pre}(x_0) \wedge x_0 \xrightarrow{\mathrm{P}} x_f \Rightarrow \mathsf{post}(x_0, x_f)$

requires $pre(x_0)$
ensures $post(x_0, x_f)$
variables X
begin
$0 : P_0(x_0, x)$
$\mathtt{instruction}_0$
...
$i : P_i(x_0, x)$
...
$\mathtt{instruction}_{f-1}$
$f : P_f(x_0, x)$
end

- $pre(x_0) \wedge x = x_0 \Rightarrow P_0(x_0, x)$
- $pre(x_0) \wedge P_f(x_0, x) \Rightarrow post(x_0, x)$
- Les conditions de vérification pour toutes les annotations sont à définir et à vérifier.

Les conditions de vérification sont constituées des deux premiers points et nous allons définir et justifier ces conditions. Elles seront utilisées pour en faire une démarche outillée à l'aide de TLA et de Event-B/Rodin.

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 $xX = (PC, V)$ avec comme conditions $pc \in$ LOCATIONS et $v \in$ MEMORY.

$$x = (pc, v) \wedge pc \in \text{LOCATIONS} \wedge v \in \text{MEMORY} \tag{1.2}$$

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×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 1.9**
$\ell_1 \longrightarrow \ell_2 \stackrel{def}{=} pc = \ell_1 \wedge pc' = \ell_2$

Un exemple d'annotation d'un algorithme ou d'un programme est donné dans l'algorithme 1.4.1. 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 1.10** 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 $\text{A}(x)$ est une propriété de sûreté pour P, si

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

On sait que cette propriété implique qu'il existe une propriété d'état $\text{I}(x)$ telle que :

$$\forall x_0, x, x' \in \text{LOCATIONS} \times \text{MEMORY} : \begin{cases} (1)\ \text{INIT}(x_0) \Rightarrow \text{I}(x_0) \\ (2)\ \forall i \in \{0, \ldots, n\} : \text{I}(x) \wedge x\ r_i\ x' \Rightarrow \text{I}(x') \\ (3)\ \text{I}(x) \Rightarrow \text{A}(x) \end{cases}$$

La propriété $\text{I}(x)$ est un invariant (inductif) 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 $P_\ell(v)$:

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

Avant de donner une version plus proche de ce quoi est connu de cette méthode, nous allons utiliser la propriété suivante sur les formules logiques.

Propriété 2 *On suppose que les formules* $A_1, \ldots, A_n$ *sont exclusives et au moins une est vraie :*

$$\left(\begin{array}{l} \wedge\,(A_1 \vee \ldots \vee A_n) \\ \wedge\,(A_1 \implies B_1) \\ \wedge\ \ldots \\ \wedge\,(A_n \implies B_n) \end{array}\right) \equiv (A_1 \wedge B_1) \vee \ldots \vee (A_n \wedge B_n)$$

PREUVE:
On procède par induction sur le niombre n.
Cas 1 :

$$\left(\begin{array}{l} \wedge\, A_1 \\ \wedge\,(A_1 \implies B_1) \end{array}\right) \equiv (A_1 \wedge B_1)$$

Cas 2 : on suppose que cette transformation est correcte jusque n.

$$\left(\begin{array}{l} \wedge\,(A_1 \vee \ldots \vee A_n) \\ \wedge\,(A_1 \implies B_1) \\ \wedge\ \ldots \\ \wedge\,(A_n \implies B_n)\ \wedge\,(A_{n+1} \implies B_{n+1}) \end{array}\right)$$
$$\equiv$$
$$\left(\begin{array}{l} \wedge\,(A \equiv (A_1 \vee \ldots \vee A_n) \\ \wedge\,(A \vee A_{n+1}) \\ \wedge\,(A \implies \left(\begin{array}{l} \wedge\,(A_1 \implies B_1) \\ \wedge\ \ldots \\ \wedge\,(A_n \implies B_n) \end{array}\right) \\ \wedge\,(A_{n+1} \implies B_{n+1}) \end{array}\right)$$
$$\equiv$$
$$\left(\begin{array}{l} \wedge\,(A \equiv (A_1 \vee \ldots \vee A_n) \\ \wedge\,(A \wedge \left(\begin{array}{l} \wedge\,(A_1 \implies B_1) \\ \wedge\ \ldots \\ \wedge\,(A_n \implies B_n) \end{array}\right) \end{array}\right) \vee A_{n+1} \wedge B_{n+1})$$
$$\equiv$$
$$\left(\left(\begin{array}{l} \wedge\,(A_1 \vee \ldots \vee A_n) \\ \wedge\,(A_1 \implies B_1) \\ \wedge\ \ldots \\ \wedge\,(A_n \implies B_n) \end{array}\right)\right) \vee A_{n+1} \wedge B_{n+1})$$
$$\equiv$$
$$\left(\ (A_1 \wedge B_1) \vee \ldots \vee (A_n \wedge B_n)\ \right) \vee A_{n+1} \wedge B_{n+1})$$
$$\equiv$$
$$(A_1 \wedge B_1) \vee \ldots \vee A_{n+1} \wedge B_{n+1})$$

□

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

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

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

x comprend deux composantes et s'écrit sous la forme $x = (pc, v)$. De plus, pc a une valeur dans l'ensemble fini LOCATIONS qui s'écrit $\{\ell_0, \ldots, \ell_n\}$. On en déduit que $I(pc, v)$ peur s'écrire sous la forme $pc = \ell_0 \wedge P_{\ell_0}(v) \vee pc = \ell_n \wedge P_{\ell_n}(v)$. D'où la propriété suivante en appliquant la transformation sur les formule :

Propriété 3

$$I(pc, v) \equiv \left(\begin{array}{l} \wedge\ (pc = \ell_0 \vee \ldots \vee pc = \ell_n) \\ \wedge\ ((pc = \ell_0) \implies P_{\ell_0}(v)) \\ \wedge\ \ldots \\ \wedge\ ((pc = \ell_n) \implies P_{\ell_n}(v)) \end{array} \right)$$

ou encore

$$I(pc, v) \equiv \left(\begin{array}{l} \wedge\ (pc \in \{\ell_0, \ldots, \ell_n\} \\ \wedge\ ((pc = \ell_0) \implies P_{\ell_0}(v)) \\ \wedge\ \ldots \\ \wedge\ ((pc = \ell_n) \implies P_{\ell_n}(v)) \end{array} \right)$$

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 :

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

Le modèle relationnel pour le programme P annoté est donc défini comme suit :
$(Th(s, c), (PC, V), \text{LOCATIONS}\times\text{MEMORY}, Init(\ell, v), \{r_{\ell,\ell'} | \ell, \ell' \in \text{LOCATIONS} \wedge \ell \longrightarrow \ell'\})$. 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 1.7** Conditions initiales
Les deux propriétés suivantes sont équivalentes :
(1) $\forall x \in \text{VALS} : \text{INIT}(x) \Rightarrow \text{I}(x)$
(2) $\forall \ell \in \text{INPUTS}, v \in \text{MEMORY}.\textbf{pre}(\text{P})(v) \Rightarrow P_\ell(v)$

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 1.8** Pas d'induction
Les deux propriétés suivantes sont équivalentes :
(1) $\forall i \in \{0, \ldots, n\} : \mathrm{I}(x) \wedge x\ r_i\ x' \Rightarrow \mathrm{I}(x')$
(2) $\forall \ell, \ell' \in$ LOCATIONS $: \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$

PREUVE:

— $\forall \ell, \ell' \in$ LOCATIONS $: \mathrm{I}(x) \wedge x\ r_{\ell,\ell'}\ x' \Rightarrow \mathrm{I}(x')$
— appliquons la transformation $x = (\ell, v)$.
— $\forall \ell, \ell' \in$ LOCATIONS $: \mathrm{I}(\ell, v) \wedge (\ell, v)\ r_{\ell,\ell'}\ (\ell', v') \Rightarrow \mathrm{I}(\ell', v')$
— $\forall \ell, \ell' \in$ LOCATIONS $: \mathrm{I}(\ell, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \Rightarrow \mathrm{I}(\ell', v')$
— $\forall \ell, \ell' \in$ LOCATIONS :
$\exists \ell_1, v_1.(\ell_1 \in$ LOCATIONS $\wedge\ v_1 \in$ MEMORY $\wedge\ (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
$\wedge\ (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
$\Rightarrow$
$\exists \ell_2, v_2.(\ell_2 \in$ LOCATIONS $\wedge\ v_2 \in$ MEMORY $\wedge\ (\ell', v') = (\ell_2, v_2) \wedge P_{\ell_2}(v_2))$
— $\forall \ell, \ell', \ell_1 \in$ LOCATIONS$, v_1 \in$ MEMORY :
$(\ell_1 \in$ LOCATIONS $\wedge\ v_1 \in$ MEMORY $\wedge\ (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
$\wedge\ (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
$\Rightarrow$
$\exists \ell_2, v_2.(\ell_2 \in$ LOCATIONS $\wedge\ v_2 \in$ MEMORY $\wedge\ (\ell', v') = (\ell_2, v_2) \wedge P_{\ell_2}(v_2))$
— $\forall \ell, \ell', \ell_1 \in$ LOCATIONS$, v_1 \in$ MEMORY :
$(\ell_1 \in$ LOCATIONS $\wedge\ v_1 \in$ 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 $\ell \longrightarrow \ell'$.
— $\forall \ell, \ell' \in$ LOCATIONS$, v_1 \in$ MEMORY $: \ell \longrightarrow \ell' :$
$(P_\ell(v))\ \wedge\ cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)$
$\Rightarrow$
$(P_{\ell'}(v'))$
— $\forall \ell, \ell' \in$ LOCATIONS $: \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$

□

☺ **Propriété 1.9** Conclusion
Les deux propriétés suivantes sont équivalentes :
(1) $\forall x \in$ VALS.$\mathrm{I}(x) \Rightarrow \mathrm{A}(x)$
(2) $\forall \ell \in$ LOCATIONS.$P_\ell(v) \Rightarrow \mathrm{A}(\ell, v)$

PREUVE:

— $\mathrm{I}(x) \Rightarrow \mathrm{A}(x)$
appliquons la transformation $x = (\ell, v)$.

— $\mathrm{I}(\ell, v) \Rightarrow \mathrm{A}(\ell, v)$
$\exists \ell_1, v_1.(\ell_1 \in \text{LOCATIONS} \wedge v_1 \in \text{MEMORY} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
— $\Rightarrow$
$\mathrm{A}(\ell, v)$
— $\forall \ell_1 \in \text{LOCATIONS}, v_1 \in \text{MEMORY}$:
$(\ell_1 \in \text{LOCATIONS} \wedge v_1 \in \text{MEMORY} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
$\Rightarrow$
$\mathrm{A}(\ell, v)$
— $\forall \ell_1 \in \text{LOCATIONS}, v_1 \in \text{MEMORY}$:
$(\ell_1 \in \text{LOCATIONS} \wedge v_1 \in \text{MEMORY} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
$\Rightarrow$
$\mathrm{A}(\ell_1, v_1)$
— $\forall \ell_1 \in \text{LOCATIONS}, v_1 \in \text{MEMORY} : P_{\ell_1}(v_1)) \Rightarrow \mathrm{A}(\ell_1, v_1)$
— $\forall \ell \in \text{LOCATIONS}.P_\ell(v) \Rightarrow \mathrm{A}(\ell, v)$
□

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

☺ **Propriété 1.10**
Les conditions de vérification suivantes sont équivalentes :

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

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

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

☺ **Propriété 1.11** 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 $\{P_\ell(v) : \ell \in \text{LOCATIONS}\}$ pour ce programme. Si les conditions suivantes sont vérifiées :

$$\forall v, v' \in \text{MEMORY} : \begin{cases} (1)\ \forall \ell \in \text{INPUTS.PRE(P)}(v) \Rightarrow P_\ell(v) \\ (2)\ \forall \ell \in \text{LOCATIONS}.P_\ell(v) \Rightarrow \mathrm{A}(\ell, v) \\ (3)\ \forall \ell, \ell' \in \text{LOCATIONS} : \ell \longrightarrow \ell' \\ \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v') \end{cases},$$

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.

B

☼**Définition 1.11** Conditions 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 1.12** Floyd and Hoare

— $\forall v, v' \in$ MEMORY.$\forall \ell, \ell' \in$ 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 $\forall \ell, \ell' \in$ LOCATIONS.$\ell \longrightarrow \ell' \Rightarrow \forall v' \in$ MEMORY.$P_\ell(v) \wedge cond_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v \mapsto f_{\ell,\ell'}(v))$

— $\forall v, v' \in$ MEMORY.$\forall \ell, \ell' \in$ 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
$\forall \ell, \ell' \in$ LOCATIONS.$\ell \longrightarrow \ell' \Rightarrow \forall v' \in$ MEMORY. $(\exists v \in$ MEMORY.$P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)$
$P_{\ell'}(v')$

Ces deux équivalences montrent que l'axiome de Hoare[19] est différent de la condition de vérification choisie par Floyd[15]. Nous pouvons resumer les deux formes possibles de l'affectation suivante :

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

— $\forall v' \in$ MEMORY.$P_\ell(v) \Rightarrow P_{\ell'}(v \mapsto f_{\ell,\ell'}(v))$ correspond à l'axiomatique de Hoare.
— $\forall v' \in$ MEMORY. $(\exists v' \in$ 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.

$\ell_1 : P_{\ell_1}(v)$
WHILE $B(V)$ **DO**
  $\ell_2 : P_{\ell_2}(v)$
  ...
  $\ell_3 : P_{\ell_3}(v)$
END
$\ell_4 : P_{\ell_4}(v)$

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**
  $\ell_2 : P_{\ell_2}(v)$
  ...
  $\ell_3 : P_{\ell_3}(v)$
ELSE
  $m_2 : P_{\ell_2}(v)$
  ...
  $m_3 : P_{\ell_3}(v)$
FI
$\ell_4 : P_{\ell_4}(v)$

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 $\longrightarrow$.

1.5 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_0, v) est la postcondition de P pour v ; elle caractérise les valeurs finales de v.

Exemple 1.13

1. **pre**(P)(x, y, z)=$x, y, z \in \mathbb{N}$ et **post**(P)(x_0, y_0, z_0, x, y, z)=$z = x_0 \cdot y_0$
2. **pre**(Q)(x, y, z)=$x, y, z \in \mathbb{N}$ et **post**(Q)(x_0, y_0, z_0, x, y, z)=$z = x_0 + y_0$

Exemple 1.14 [13]
Nous reprenons un exemple simple de P. et R. Cousot pour illustrer les notions de préconditions et de postconditions 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 : $\underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}, x, y, r, q$
precondition : $\underline{x}, \underline{y}, \underline{r}, \underline{q} \in \mathbb{N}$
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 5: 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 : B

$$\forall \underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}. \textbf{pre}(\mathrm{P})(\underline{x}, \underline{y}, \underline{r}, \underline{q}) \wedge (\underline{x}, \underline{y}, \underline{r}, \underline{q}) \xrightarrow{\mathrm{P}} (\overline{x}, \overline{y}, \overline{r}, \overline{q}) \Rightarrow \textbf{post}(\mathrm{P})(\underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}) \tag{1.3}$$

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 [19] 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
precondition : $x, y \in \mathbb{N}$
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 6: Division euclidienne de deux nombres et spécification assertionnelle

---

L'algorithme 6 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 [26] 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 [26] propose un ensemble de règles de transformations permettant d'obtenir un programme conforme à la spécification w : [P , Q] ; la méthode B [9] peut aussi être utilisée pour mettre en œuvre ce calcul de raffinement constituant un calcul de développement.

1.5.1 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 $\{P_\ell(v) : \ell \in \text{LOCATIONS}\}$ pour ce programme.
- une propriété de sûreté définissant la correction partielle $\ell = output \Rightarrow$ **post**$(\text{P})(v)$ où $output$ est l'étiquette marquant la fin du programme P

☼**Définition 1.12**

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

☺ **Propriété 1.13**

Si les conditions suivantes sont vérifiées :

- $\forall \ell \in \text{INPUTS}.\forall v, v' \in \text{MEMORY}.$**pre**$(\text{P})(v) \Rightarrow P_\ell(v)$
- $\forall \ell \in \text{OUTPUTS}.\forall v, v' \in \text{MEMORY}.P_\ell(v) \Rightarrow$ **post**$(\text{P})(v)$
- $\forall \ell, \ell' \in \text{LOCATIONS} : \ell \longrightarrow \ell' : \forall v, v' \in \text{MEMORY}.$
  $(P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v'))$,

alors le programme P est partiellement correct par rapport à **pre**(P)(v) et **post**(P)(v).

Nous avons donné un point de vue assertionnelle et maintenant on peut donner la forme générale relationnelle.

Un programme P *remplit* un contrat (pre,post) :

- P transforme une variable v à partir d'une valeur initiale v_0 et produisant une valeur finale v_f : $v_0 \xrightarrow{\text{P}} v_f$
- v_0 satisfait pre : $\mathsf{pre}(v_0)$ and v_f satisfait post : $\mathsf{post}(v_0, v_f)$
- $\mathsf{pre}(v_0) \wedge v_0 \xrightarrow{\text{P}} v_f \Rightarrow \mathsf{post}(v_0, v_f)$

requires $pre(v_0)$
ensures $post(v_0, v_f)$
variables X

begin
$0 : P_0(v_0, v)$
$\texttt{instruction}_0$
...
$i : P_i(v_0, x)$
...
$\texttt{instruction}_{f-1}$
$f : P_f(v_0, v)$
end

Conditions de vérification pour la correction partielle

— $pre(v_0) \wedge v = v_0 \Rightarrow P_0(v_0, v)$
— $pre(v_0) \wedge P_f(v_0, v) \Rightarrow post(v_0, v)$
— Pour toute paire d'étiquettes ℓ, ℓ' telle que $\ell \longrightarrow \ell'$, on vérifie que, pour toutes valeurs $v, v' \in$ MEMORY

$$\left(\begin{array}{l} \left(\begin{array}{l} pre(v_0) \wedge P_\ell(v_0, v)) \\ \wedge cond_{\ell,\ell'}(v) \wedge x' = f_{\ell,\ell'}(v) \end{array} \right) \\ \Rightarrow P_{\ell'}(v_0, v') \end{array} \right),$$

1.5.2 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 1.15

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 1.13**

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 1.14**

Si les conditions suivantes sont vérifiées :

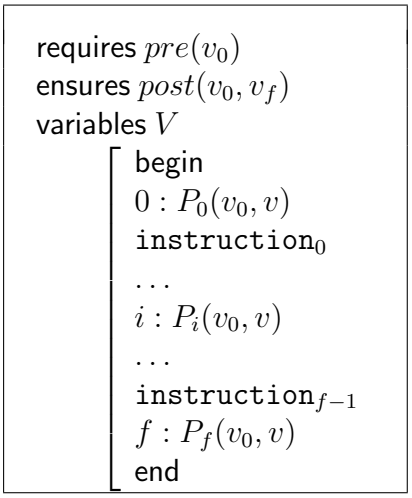
$$\begin{cases} (1)\ \forall \ell \in \text{INPUTS}. \forall v, v' \in \text{MEMORY}. \mathbf{pre}(\text{P})(v) \Rightarrow P_\ell(v) \\ (2)\ \forall m \in \text{LOCATIONS} - \{output\}, n \in \text{LOCATIONS}, \forall v, v' \in \text{MEMORY} : \\ \qquad m \longrightarrow n : P_m(v) \Rightarrow \mathbf{DOM}(m,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

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* [17] et sur l'utilisation de procédures permettant de `vérifier` que le séquent est valide ou non.

Un programme P *remplit* un contrat (pre,post) avec absences d'erreurs à l'exécution :

- P transforme une variable v à partir d'une valeur initiale v_0 et produisant une valeur finale v_f : $xv_0 \xrightarrow{\text{P}} v_f$
- v_0 satisfait pre : $\mathsf{pre}(v_0)$ and v_f satisfait post : $\mathsf{post}(v_0, v_f)$
- $\mathsf{pre}(v_0) \wedge x_0 \xrightarrow{\text{P}} v_f \Rightarrow \mathsf{post}(v_0, v_f)$
- $\mathbb{D}$ est le domaine RTE de V



Conditions de vérification pour la correction partielle et l'absence d'erreurs à l'exécution

— $pre(v_0) \wedge v = v_0 \Rightarrow P_0(v_0, v)$
— $pre(v_0) \wedge P_f(v_0, v) \Rightarrow post(v_0, v)$
— Pour toute paire d'étiquettes ℓ, ℓ'
telle que $\ell \longrightarrow \ell'$, on vérifie que, pour toutes valeurs $v, v' \in$ MEMORY
$$\left(\begin{array}{l} \left(\begin{array}{l} pre(v_0) \wedge P_\ell(v_0, v)) \\ \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \end{array} \right) \\ \Rightarrow P_{\ell'}(v_0, v') \end{array} \right),$$
— Pour toute paire d'étiquettes m, n
telle que $m \longrightarrow n$, on vérifie que, $\forall v, v' \in$ MEMORY : $pre(v_0) \wedge P_m(v_0, v) \Rightarrow$ **DOM**$(m, n)(v)$

— $pre(v_0) \wedge v = v_0 \Rightarrow P_0(v_0, v)$
— $pre(v_0) \wedge P_f(v_0, v) \Rightarrow post(v_0, v)$
— Pour toute paire d'étiquettes ℓ, ℓ'
telle que $\ell \longrightarrow \ell'$, on vérifie que, pour toutes valeurs $v, v' \in$ MEMORY
$$\left(\begin{array}{l} \left(\begin{array}{l} pre(v_0) \wedge P_\ell(v_0, v)) \\ \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \end{array} \right) \\ \Rightarrow P_{\ell'}(v_0, v') \end{array} \right),$$
— Pour toute paire d'étiquettes m, n tell e que $m \longrightarrow n$, on vérifie que, $\forall v, v' \in$ MEMORY : $pre(v_0) \wedge P_m(v_0, v) \Rightarrow$ **DOM**$(m, n)(v)$

Exemple 1.16 Exemple **DOM**(m,n)(x)
$DOM(\ell_0, \ell_1)(u) = u \in minint..maxint \wedge 5 \in minint..maxint \wedge u{+}5 \in minint..maxint$
où $\ell_0 : P_{\ell_0}(u) U := U{+}5; \ell_1 : P_{\ell_0}(u)$

1.6 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 [31] a proposé une méthode de preuves de machines de Turing, reprise plus tard par Floyd [15] et par Hoare [19]. P. et R. Cousot [13] 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 [27] 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.

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

[6] Mike Barnett, Rustan Leino, and Wolfram Schulte. The Spec# programming system : An overview. In *CASSIS 2004*, volume 3362. Springer-Verlag, 2004.

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

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

[9] Dominique Cansell and Dominique Méry. *The event-B Modelling Method : Concepts and Case Studies*, pages 33–140. Springer, 2007. See [8].

[10] ClearSy, www.atelierb.eu. *Atelier B, Version 4.4.2*, 2016.

[11] Ada Core. *GNAT*.

[12] 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.

[13] 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.

[14] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles*. PhD thesis, INPL, novembre 1985. Doctorat-ès-Sciences.

[15] 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.

[16] Frama-C. http ://www.frama-c.com.

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

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

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

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

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

[22] 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.

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

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

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

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

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

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

[29] 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.

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

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