Modélisation et vérification de systèmes informatique

8 septembre 2015

Dominique Méry

Université de Lorraine (Telecom Nancy)
LORIA, BP 70239,Campus Scientifique
54506 Vandœuvre-lès-Nancy Cedex, FRANCE
dominique.mery@loria.fr
http://www.loria.fr/∼mery
written on the 8 septembre 2015, please send bug reports to mery@loria.fr

Ce document est édité le 8 septembre 2015 au début du cours MVSI de l'année d'apprentissage 2A de Telecom Nancy et sera, sans doute, modifié et enrichi en fonction des cours et des séances d'exercices. Deux mots dans le titre caractérisent les objectifs :

- *modèles* : nous allons donner des éléments sur les techniques de modélisation formelle avec une vue sur les applications en génie logiciel, notamment en vérification et validation de logiciels, de programmes, de systèmes.
- *algorithmes* : nous allons considérer les questions de construction des algorithmes corrects et nous aborderons les questions de calculabilité, de décidabilité et d'indécidabilité.

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

- *abstraction* : une abstraction est une notion très importante quand on veut modéliser un système ; une abstraction désigne à la fois une action et l'objet de cette action. Abstraire un système, c'est en donner une vue détachée de la réalité mais conforme à la réalité du système. Cette notion d'abstraction permet de donner des vues simples mais fidèles des systèmes.
- *Raffinement* : un raffinement est une action inverse de l'abstraction et consiste à préciser le modèle en cours d'examen afin de lui donner des articifices identifiés sur le système en cours de modélisation.

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

Cette nouvelles édition ajoute la pratique d'un environnement, PAT [28], permettant d'analyser les systèmes informatiques dans le cadre de langages de programmation comme C# ou encore des formalismes spécifiquement développés pour permettre la construction d'outils de vérification.

Table des matières

1 Modélisation et vérification des programmes et des systèmes **5**
1.1 Modélisation d'un système 5
1.2 Propriété de sûreté dans un modèle relationnel 8
1.3 Conception d'une méthode de preuves de propriétés d'invariance et de sûreté pour un langage de programmation 10
1.3.1 Spécialisation des principes d'induction pour le cas des programmes . . 10
1.3.2 Propriétés de correction des programmes 17
1.4 Notations ensemblistes B 20
1.4.1 Prédicats logiques 21
1.4.2 Ensembles 21
1.4.3 Relations 22
1.4.4 Fonctions 22
1.5 Vue ensembliste 22
1.5.1 Propriétés de sûreté 23
1.5.2 Principes d'induction 24
1.6 Notes bibliographiques 26

Chapitre 1

Modélisation et vérification des programmes et des systèmes

Ce chapitre envisage la question de vérification des programmes et des systèmes ; cette vérification repose sur des données claires de ce que l'on veut vérifier et de ce que l'on souhaite effectivement mettre en évidence (correction partielle, correction totale, absence d'erreurs à l'exécution, bon typage, ...). Le besoin de justifier les programmes et les systèmes notamment répartis apparaît avec les premiers modèles de calcul comme les machines de Turing [30], repris par Floyd [12] pour les *flowcharts* ; Hoare [15] 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 [18, 17, 5, 19]. Un certain nombre de langages de programmation [24, 4, 14] intègrent des éléments permettant d'exprimer à la fois l'algorithmique mais aussi les *annotations* comme les pré-conditions, les post-conditions, les invariants de classe, ...
Au préalable, il faut définir un cadre sémantique pour le système à vérifier et être capable de modéliser un système. Nous utiliserons les notations ensemblistes de B [1, 2, 7] et de TLA$^+$ [21, 22], ainsi que les notations nécessaires pour utiliser l'environnement PAT [28], et nous donnons une explication de ce qui est en jeu dans le cadre de la modélisation.

1.1 Modélisation d'un système

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

requises pour la complétude sont exprimables dans le langage considéré.
Les propriétés d'un système S sont, en particulier, les propriétés de sûreté et les propriétés de fatalité. Les propriétés de sûreté sont, par exemple, la correction partielle d'un système S par rapport à ses spécifications, l'absence d'erreurs à l'exécution ; les propriétés de fatalité sont, par exemple, la terminaison d'un programme P par rapport à ses spécifications ou la correction totale de P par rapport à ses spécifications. Nous pourrions nous intéresser à d'autres propriétés de programmes comme ses performances mais cela impliquerait des modèles pour exprimer des propriétés dites non fonctionnelles.
Les propriétés sont exprimées dans un langage $\mathcal{L}$ dont les éléments sont combinés par des connecteurs logiques ou par des instanciation de variables ; la relation d'implication modulo l'équivalence définit une relation d'ordre partiel. Pour être plus précis, on peut définir une relation de satisfaction d'une propriété en utilisant la notation ensembliste :

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

Ces deux fonctions sont liées par la propriété suivante :
pour toute assertion x de $(\mathcal{L}, \Rightarrow)$, pour toute partie y de $(\mathcal{P}, \subseteq)$, $x \subseteq \alpha(y)$ si, et seulement si, $x \subseteq \alpha(y)$
Cela signifie que l'on peut utiliser une notation ensembliste mais qu'il faut ensuite utiliser une fonction de transfert des résultats du domaine concret vers le domaine abstrait. Dans le cas précédent, le domaine abstrait était celui des assertions et le domaine concret était celui des parties de l'ensemble des états.
Nous supposons qu'un système S est modélisé par un ensemble d'états Σ_{S}, noté Σ, et que $\Sigma \stackrel{def}{=}$ 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 sous une forme inductive de $s[\![\varphi(x)]\!]$. La définition inductive sera expliquée en détail dans les chapitres **??** et **??**.

Exemple 1.1

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

Nous utilisons des notations simplifiant la référence aux états ; ainsi, $s[\![x]\!]$ est la valeur de x en s et le nom de la variable x et sa valeur ne seront pas distinguées. $s'[\![x]\!]$ est la valeur de x en s' et sera notée x'. Ainsi, $s[\![x = 6]\!] \wedge s'[\![y = x{+}8]\!]$ se simplifiera en $x = 6 \wedge y' = x'{+}8$. La conséquence est que l'on pourra écrire la relation de transition comme une relation liant l'état des variables en s et l'état des variables en s'.
Avec TLA [21], Lamport introduit la notion d'action sous la fome d'une formule logique comprenant des occurrences primées et non-primées de variables. Une action A est une expression booléenne ayant des occurences libres des variables non-primées, primées et des symboles de constantes ; une action désigne une relation entre une paire d'états consécutifs, un état avant (exprimé l'aide des variables non-primés et des symboles de constantes) et un nouvel état (exprimé à l'aide des variables primées et des symboles de constantes).

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

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

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

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

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

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

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

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

La modélisation d'un système comprend la donnée des variables x, du prédicat caractériant les valeurs initiales des variables et une relation NEXT modélisant la relation entre les valeurs avant et les valeurs après. Les principes d'induction sont transcrits dans le cadre des modèles relationnels et nous introduisons la défintion de la sûreté dans un modèle relationnel.

Soit $(Th(s,c), x, \text{VALS}, \text{INIT}(x), \{r_0, \ldots, r_n\})$ 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 x, nous définissons les valeurs suivantes :
– x est la valeur courante de la variable x.
– x' est la valeur suivante de la variable x.
– x_0 ou $\underline{x}$ sont la valeur initiale de la variable x.
– $\overline{x}$ est la valeur finale de la variable x, quand cette notion a du sens.
Nous avons utilisé un style différent pour désigner la variable x et sa valeur x et la plupart du temps ces deux notations sont confondues. Dans la section suivante, nous allons définir les propriétés de sûreté pour les modèles relationnels de système.

1.2 Propriété de sûreté dans un modèle relationnel

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

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

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

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

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

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

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

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

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

PREUVE:

$\langle 1 \rangle 1$. SUPPOSONS QUE: il existe une propriété $I(x)$ telle que :

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

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

PREUVE:

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

$\langle 1 \rangle 2$. SUPPOSONS QUE: $\forall x, y \cdot x, y \in \text{VALS} \wedge Init(x) \wedge \text{NEXT}^{\cdot}(x, y) \Rightarrow A(y)$

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

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

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

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

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

□

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

☺ **Propriété 1.3**

Les deux énoncés suivants sont équivalents :

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

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

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

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

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

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

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

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

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

Enfin, nous avons une propriété importante permettant de comprendre le ien entre la méthode de Floyd [12] et la méthode de Hoare [15].

☺ **Propriété 1.4**
Les deux énoncés suivants sont équivalents :
– $\forall x, x' \in \text{VALS} : \text{I}(x) \wedge x\ r_i\ x' \Rightarrow \text{I}(x')$ (Hoare)
– $\forall x' \in \text{VALS} : (\exists x \in \text{VALS} : \text{I}(x) \wedge x\ r_i\ x') \Rightarrow \text{I}(x')$ (Floyd)

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

Nous appliquerons cette transformation plus tard et nous allons maintenant utiliser ce principe d'induction dans le cas d'un langage de programmation généraliste qui pourrait s'apparenter au langage C. L'objectif est de comprendre comment fonctionnent les outils de vérification dans le cas de langage comme Spec# avec le système `rise4fun` [27] et de comprendre comment fonctionnent les environnements comme `TLAPS` [29] ou encore `Rodin` [3]

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

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

Dans la cas de programmes ou d'algorithmes, la méthode de correction partielle peut être spécialisée en fonction du contexte très particulier des programmes. On considère un langage de programmation classique noté PROGRAMS et nous supposons que ce langage de programmation dispose de l'affectation, de la conditionnelle, de l'itération bornée, de l'itération non-bornée, de variables simples ou structurées comme les tableaux et de la définition de constantes.
On se donne un programme P de PROGRAMS ; ce programme comprend
– des variables notées globalement v,
– des constantes notées globalement c,
– des types associés aux variables notés globalement VALSet identifiés à un ensemble de valeurs possibles des variables,
– des instructions suivant un ordre défini par la syntaxe du langage de programmation.
Nous donnons trois exemples de programmes ou d'algorithmes que nous pouvons écrire et nous introduisons pour chacun de ces exemples la spécification des données et des résultats sous la forme de prédicates placés dans l'entête des algorithmes.

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

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

Exemple 1.3
La $multiplication$ est définie comme suit :

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

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

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

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

  $cresult := \sigma[cresult];$
  $cy := cy{+}1;$
;
$result := cresult;$

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

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

$cx := x;$
$cy := 0;$
$cresult := \zeta();$
while $cy < y$ **do**
  Invariant : $0 \leq cy \wedge cy < y \wedge cx = x \wedge$
  $cresult = multiplication[cx, cy]$

  $cresult := addition[cx, cresult];$
  $cy := cy{+}1;$
;
$result := cresult;$

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

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

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

$cx := x$;
$cy := 0$;
$cresult := \sigma(\zeta())$;
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: Iterative algorithm $EXPONENTIATION$ for computing the primitive recursive function $exponentiation$

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

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

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

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

☼**Définition 1.6**

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

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

Algorithme 4: $MULTIPLICATION$ annotée

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

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

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

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

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

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

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

PREUVE:Pour chaque étiquette ℓ, on définit $P_\ell(v) \stackrel{def}{=} \exists x. x = (\ell, v) \wedge \text{I}(x)$ et

$\mathrm{I}(x) \equiv \bigvee_{\ell \in \textsc{Locations}} \left(\bigvee_{v \in \textsc{Memory}} x = (\ell, v) \wedge P_\ell(v) \right)$ est évidemment vérifiée.$\square$

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

- $I(x) \stackrel{def}{=} \exists \ell, v.(\ell \in \textsc{Locations} \wedge v \in \textsc{Memory} \wedge x = (\ell, v) \wedge P_\ell(v))$
- $P_\ell(v) \stackrel{def}{=} \exists x.(x \in \textsc{Vals} \wedge x = (\ell, v) \wedge I(x))$

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

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

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

$$Init(\ell, v) \stackrel{def}{=} \ell \in \textsc{Inputs} \wedge \mathbf{pre}(\mathrm{P})(v).$$

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

- $\forall x \in \textsc{Vals} : \textsc{Init}(x) \Rightarrow \mathrm{I}(x)$
- $\forall \ell \in \textsc{Inputs}, v \in \textsc{Memory}.\mathbf{pre}(\mathrm{P})(v) \Rightarrow P_\ell(v)$

PREUVE:

- $\forall x \in \textsc{Vals} : \textsc{Init}(x) \Rightarrow \mathrm{I}(x)$
- appliquons la transformation $x = (\ell, v)$
- $\forall \ell, v \in \textsc{Locations} \times \textsc{Memory} : \textsc{Init}(\ell, v) \Rightarrow \mathrm{I}(\ell, v)$
- $\forall \ell, v \in \textsc{Locations} \times \textsc{Memory} : \ell \in \textsc{Inputs} \wedge \mathbf{pre}(\mathrm{P})(v) \Rightarrow \mathrm{I}(\ell, v)$
- $\forall \ell \in \textsc{Locations}, v \in \textsc{Memory} : \ell \in \textsc{Inputs} \wedge \mathbf{pre}(\mathrm{P})(v) \Rightarrow \mathrm{I}(\ell, v)$
- $\forall v \in \textsc{Memory}, \ell \in \textsc{Inputs} : \mathbf{pre}(\mathrm{P})(v) \Rightarrow \mathrm{I}(\ell, v)$
- $\forall v \in \textsc{Memory}, \ell \in \textsc{Inputs} : \mathbf{pre}(\mathrm{P})(v) \Rightarrow (\exists \ell', v'.(\ell' \in \textsc{Locations} \wedge v' \in \textsc{Memory} \wedge x = (\ell', v') \wedge P_{\ell'}(v')))$
- $\forall v \in \textsc{Memory}, \ell \in \textsc{Inputs} : \mathbf{pre}(\mathrm{P})(v) \Rightarrow (\exists \ell', v'.(\ell' \in \textsc{Locations} \wedge v' \in \textsc{Memory} \wedge (\ell, v) = (\ell', v') \wedge P_{\ell'}(v')))$
- $\forall v \in \textsc{Memory}, \forall \ell \in \textsc{Inputs}.\mathbf{pre}(\mathrm{P})(v) \Rightarrow P_\ell(v)$

$\square$

☺ **Propriété 1.6** Pas d'induction
Les deux propriétés suivantes sont équivalentes :

- $\forall i \in \{0, \ldots, n\} : \mathrm{I}(x) \wedge x \; r_i \; x' \Rightarrow \mathrm{I}(x')$
- $\forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$

PREUVE:

- $\forall \ell, \ell' \in \textsc{Locations} : \mathrm{I}(x) \wedge x \; r_{\ell,\ell'} \; x' \Rightarrow \mathrm{I}(x')$
- appliquons la transformation $x = (\ell, v)$.
- $\forall \ell, \ell' \in \textsc{Locations} : \mathrm{I}(\ell, v) \wedge (\ell, v) \; r_{\ell,\ell'} \; (\ell', v') \Rightarrow \mathrm{I}(\ell', v')$

– $\forall \ell, \ell' \in \textsc{Locations}$: $\mathbf{I}(\ell, v) \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \Rightarrow \mathbf{I}(\ell', v')$

– $\forall \ell, \ell' \in \textsc{Locations}$: $\begin{array}{l} \exists \ell_1, v_1.(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1)) \\ \wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell' \\ \Rightarrow \\ \exists \ell_2, v_2.(\ell_2 \in \textsc{Locations} \wedge v_2 \in \textsc{Memory} \wedge (\ell', v') = (\ell_2, v_2) \wedge P_{\ell_2}(v_2)) \end{array}$

– $\forall \ell, \ell', \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory}$:
$(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
$\wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
$\Rightarrow$
$\exists \ell_2, v_2.(\ell_2 \in \textsc{Locations} \wedge v_2 \in \textsc{Memory} \wedge (\ell', v') = (\ell_2, v_2) \wedge P_{\ell_2}(v_2))$

– $\forall \ell, \ell', \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory}$:
$(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
$\wedge (pc = \ell \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \wedge pc' = \ell'$
$\Rightarrow$
$(P_{\ell'}(v'))$

$pc = \ell \wedge \wedge pc' = \ell'$ **est remplacé par** $\ell \longrightarrow \ell'$.

– $\forall \ell, \ell' \in \textsc{Locations}, v_1 \in \textsc{Memory} : \ell \longrightarrow \ell'$:
$(P_\ell(v)) \ \wedge \ cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)$
$\Rightarrow$
$(P_{\ell'}(v'))$

– $\forall \ell, \ell' \in \textsc{Locations}$: $\ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$

□

☺ **Propriété 1.7** Conclusion

Les deux propriétés suivantes sont équivalentes :

– $\mathbf{I}(x) \Rightarrow \mathbf{A}(x)$
– $\forall \ell \in \textsc{Locations}.P_\ell(v) \Rightarrow \mathbf{A}(\ell, v)$

PREUVE:

– $\mathbf{I}(x) \Rightarrow \mathbf{A}(x)$

appliquons la transformation $x = (\ell, v)$.

– $\mathbf{I}(\ell, v) \Rightarrow \mathbf{A}(\ell, v)$
$\exists \ell_1, v_1.(\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1))$
– $\Rightarrow$
$\mathbf{A}(\ell, v)$

– $\forall \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory}$: $\begin{array}{l} (\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1)) \\ \Rightarrow \\ \mathbf{A}(\ell, v) \end{array}$

– $\forall \ell_1 \in \textsc{Locations}, v_1 \in \textsc{Memory}$: $\begin{array}{l} (\ell_1 \in \textsc{Locations} \wedge v_1 \in \textsc{Memory} \wedge (\ell, v) = (\ell_1, v_1) \wedge P_{\ell_1}(v_1)) \\ \Rightarrow \\ \mathbf{A}(\ell_1, v_1) \\ (P_{\ell_1}(v_1)) \end{array}$

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

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

□

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

☺ **Propriété 1.8**

Les conditions de vérification suivantes sont équivalentes :

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

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

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

☺ **Propriété 1.9** Méthode de correction de propriétés de sûreté
Soit $\text{A}(\ell, v)$ 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 :

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

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

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

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

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

☺ **Propriété 1.10** Floyd and Hoare
– $\forall v, v' \in \text{MEMORY}.\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$ est équivalent à $\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow \forall v' \in \text{MEMORY}.P_\ell(v) \wedge cond_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v \mapsto f_{\ell,\ell'}(v))$
– $\forall v, v' \in \text{MEMORY}.\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v')$ est équivalent à
$\forall \ell, \ell' \in \text{LOCATIONS}.\ell \longrightarrow \ell' \Rightarrow \forall v' \in \text{MEMORY}.\ (\exists v \in \text{MEMORY}.P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v)) \Rightarrow P_{\ell'}(v')$

Ces deux équivalences montrent que l'axiome de Hoare[15] est différent de la condition de vérification choisie par Floyd[12].
Nous pouvons resumer les deux formes possibles de l'affectation suivante :

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

Algorithme 5: Affectation

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

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

Les deux formes sont équivalentes et se trouvent dans la littérature. Nous allons maintenant proposer des techniques de preuve de correction pour certaines propriétés classiques.

On peut maintenant définir des conditions de vérification pour chaque structure de programme.

```
ℓ1 : P_ℓ1(v);
while B(v) do
    ℓ2 : P_ℓ2(v);
    ...;
    ℓ3 : P_ℓ3(v);
ℓ4 : P_ℓ4(v);
```

Algorithme 6: Structure d'itération

Pour la structure d'itération, les conditions de vérification sont les suivantes :

– $P_{\ell_1}(v) \wedge B(v) \Rightarrow P_{\ell_2}(v)$
– $P_{\ell_1}(v) \wedge \neg B(v) \Rightarrow P_{\ell_4}(v)$
– $P_{\ell_3}(v) \wedge B(v) \Rightarrow P_{\ell_2}(v)$
– $P_{\ell_3}(v) \wedge \neg B(v) \Rightarrow P_{\ell_4}(v)$

```
ℓ1 : P_ℓ1(v);
if B(v) then
    ℓ2 : P_ℓ2(v);
    ...;
    ℓ3 : P_ℓ3(v);
else
    m2 : P_ℓ2(v);
    ...;
    m3 : P_ℓ3(v);
;
ℓ4 : P_ℓ4(v);
```

Algorithme 7: Structure de conditionnelle

Pour la structure de conditionnelle, les conditions de vérification sont les suivantes :

– $P_{\ell_1}(v) \wedge B(v) \Rightarrow P_{\ell_2}(v)$
– $P_{\ell_3}(v) \Rightarrow P_{\ell_4}(v)$
– $P_{\ell_1}(v) \wedge \neg B(v) \Rightarrow P_{m_2}(v)$
– $P_{m_3}(v) \Rightarrow P_{\ell_4}(v)$

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

1.3.2 Propriétés de correction des programmes

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

Exemple 1.5

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

Exemple 1.6 [10]
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 : $\underline{x}, \underline{y}, \underline{r}, \underline{q}, \overline{x}, \overline{y}, \overline{r}, \overline{q}, x, y, r, q$
précondition : $\underline{x}, \underline{y}, \underline{r}, \underline{q} \in \mathbb{N}$
postcondition : $\overline{x}, \overline{y}, \overline{r}, \overline{q} \in \mathbb{N} \wedge \overline{x} = \underline{x} \wedge \overline{y} = \underline{y} \wedge \overline{x} = \overline{q} \cdot \underline{y} + \overline{r} \wedge \overline{r} < \underline{y}$

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

Algorithme 8: Division euclidienne de deux nombres et spécification relationnelle

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

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

Notre spécification de la correction partielle par rapport à la précondition et à la postcondition est relationnelle. On peut utiliser un style assertionnel qui s'intègre dans la démarche de la logique de Hoare [15] en introduisant les programmes assertés ou annotés (*asserted programs*). Par exemple, on écrira les précondition et postcondition comme suit :
– **pre**(P)(x, y, r, q) est définie par $x, y, r, q \in \mathbb{N}$.
– **post**(P)(x, y, r, q) est définie par $x, y, r, q \in \mathbb{N} \wedge x = q \cdot y + r \wedge r < y$
La spécification est notée {**pre**(P)(x, y, r, q)} **P** {**post**(P)(x, y, r, q)} et est appelée triplet de Hoare.

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

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

Algorithme 9: Division euclidienne de deux nombres et spécification assertionnelle

---

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

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

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

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

où $\mathrm{P}_{X,Y,Q,R}$ est un algorithme modifiant les variables X,Y,Q,R de manière à respecter la précondition et la postcondition. La notation : [,] est dûe à Morgan [25] 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 [25] propose un ensemble de règles de transformations permettant d'obtenir un programme conforme à la spécification w : $[P\ ,\ Q]$; la méthode B [7] peut aussi être utilisée pour mettre en œuvre ce calcul de raffinement constituant un calcul de développement.

Correction partielle d'un programme

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

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

☺ **Propriété 1.11**
Si les conditions suivantes sont vérifiées :
- $\forall \ell \in \textsc{Inputs}. \forall v, v' \in \textsc{Memory}. \mathbf{pre}(\mathrm{P})(v) \Rightarrow P_\ell(v)$
- $\forall \ell \in \textsc{Outputs}. \forall v, v' \in \textsc{Memory}. P_\ell(v) \Rightarrow \mathbf{post}(\mathrm{P})(v)$
- $\forall \ell, \ell' \in \textsc{Locations} : \ell \longrightarrow \ell' : \forall v, v' \in \textsc{Memory}.\ (P_\ell(v) \wedge cond_{\ell,\ell'}(v) \wedge v' = f_{\ell,\ell'}(v) \Rightarrow P_{\ell'}(v'))$,

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

Absence d'erreurs à l'exécution

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

Exemple 1.7

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

2. La transition correspond à une affectation de la forme $x := x{+}1$ ou $y := x{+}1$:

   $\mathbf{DOM}(x{+}1)(x,y) \stackrel{def}{=} \mathbf{DOM}(x)(x,y) \wedge x{+}2 \in int$

Parmi les cas d'erreurs, on pourra citer le débordement arithmétique, la référence à une adresse non définie, la division par zéro, ... Le prédicat $\mathbf{DOM}(\ell,\ell')(v)$ doit intégrer ces éléments pour chaque cas possible qui se ramène à une affectation ou un test en C par exemple.

L'absence d'erreurs à l'exécution vise à établir qu'un programme P ne va pas produire des erreurs durant son exécution par rapport à sa précondition et à sa postcondition. On ne prend pas en compte la terminaison et le programme peut naturellement boucler. Nous définissons donc les éléments suivants :

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

☼**Définition 1.10**

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

☺ **Propriété 1.12**

Si les conditions suivantes sont vérifiées :

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

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` [3]. Cette mécanisation est fondée sur la preuve d'énoncés appelés *séquents de Gentzen* [13] et sur l'utilisation de procédures permettant de `vérifier` que le séquent est valide ou non.

1.4 Notations ensemblistes B

Cette section présente les notations ensemblistes du langage B [1, 2, 7]. Ce langage est supporté par un outil disponible sous le nom de plate-forme `Rodin` [3] mais aussi est disponible à partir du site de la compagnie ClearSy `Atelier-B` [9]. Ces deux versions vous permettront de vérifier vos écrits et de démontrer des propriétés à partir d'une liste d'axiomes.

1.4.1 Prédicats logiques

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

1.4.2 Ensembles

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

1.4.3 Relations

Nom	Expression ASCII	Expression formatée	signification intuitive
ensemble des parties sur $S \times T$	`S<->T`	$S \leftrightarrow T$	ensemble des relations entre S et T
	dom(r)	$dom(r)$	domain of relation
	ran(r)	$ran(r)$	range of relation
	id(S)		$id(S)$ identity relation
	`S<|r`	$S \lhd r$	domain restriction
	`S<<|r`	$S \lhd r$	domain subtraction
	`r|>S`	$r \rhd S$	range restriction
	`r|>>S`	$r \rhd S$	range subtraction
	`r~`	r^{-1}	inverse of relation
	`r[S]`	$r[S]$	relational image
	`r1<+r2`	$r1 \lhd\!\!+ r2$	right overriding
	`r1><r2`	$r1 \otimes r2$	direct product `{x,(y,z) | x,y:r1 & x,z:r2}`
	`r1;r2`	$r1;r2$	relational composition `{x,y| x|->z:r1 & z|->y:r2}`
	`prj1(S,T)`		projection
	`prj2(S,T)`		projection

1.4.4 Fonctions

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

1.5 Vue ensembliste

Les propriétés de sûreté expriment que *rien de mauvais ne peut arriver [20]*. Par exemple, la valeur de la variable x est toujours comprise entre 0 et 567 ; la somme des valeurs courantes des variables x et y est égale à la valeur courante de la valeur z. On se donne un langage d'assertions $(\mathcal{P}(\Sigma), \subseteq)$ et une interprétation ensembliste de la relation de satisfaction. Ainsi, on écrira $s \in \{u | u \in \Sigma \wedge \varphi(u)\}$ pour exprimer que s est élément d'un ensemble d'états satisfaisant φ. Cela veut dire qu'on peut utiliser $\mathcal{P}(\Sigma)$ comme un langage pour exprimer des propriétés en tant qu'ensemble d'états ou propriétés d'états. Dans ce cas, les *assertions* sont les parties de Σ et l'expressivité du lanage est donc limitée à ces ensembles. Une question fondamentale est de savoir, si toutes les propriétés exprimables dans un langage $\mathcal{L}$ sont exprimables dans un autre langage $\mathcal{M}$. Nous définissons ce que nous entendons par *propriété de sûreté*.

☼**Définition 1.11** Structure de Kripke d'un système
Une structure de Kripke $\mathcal{MK}$ pour un système $\mathcal{S}$ est une structure $(\Sigma, \text{INIT}, \longrightarrow)$ où
– Σ est l'ensemble de tous les états.
– $\text{INIT} \subseteq \Sigma$ définit l'ensemble des états initiaux de S.
– $\longrightarrow$ est une relation binaire

1.5.1 Propriétés de sûreté

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

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

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

L'expression $\xrightarrow{\star}$ désigne la fermeture réflexive transitive de la relation $\longrightarrow$. La propriété de sûreté utilise une expression universelle pour quantifier les états. Pour démontrer une telle propriété, on peut soit vérifier la propriété pour chaque état possible de Σ_{S}, à condition que cet ensemble soit fini, ou bien trouver un principe d'induction. Pour ce qui est de la vérification pour chaque état, on utilise un algorithme de calcul des états accessibles à partir d'un état initial. Cette technique de calcul des états accessibles est souvent utilisée et est à la base des techniques de vérification automatique comme le *model checking* [23, 16, 8] Nous donnons maintenant un principe d'induction, pour prouver les propriétés de sûreté ; l'objectif est de montrer comment les techniques de preuves de propriétés de programmes sont construites et sur quels principes elles reposent.

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

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

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

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

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

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

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

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

PROUVONS QUE: Il existe une propriété INV telle que

$$\begin{cases} (1) \;\; Init_{\mathrm{S}} \subseteq INV \\ (2) \;\; INV \subseteq A \\ (3) \;\; \forall s, s' \in \Sigma_{\mathrm{S}} . s \in INV \wedge s \longrightarrow s' \Rightarrow s' \in INV \end{cases}$$

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

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

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

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

□

Cette propriété justifie la méthode de preuve par induction plus connue comme la méthode de Floyd/Hoare [12, 15], imaginée par Turing en 1949 [30]. La propriété énoncée donne une forme de l'induction qu'il faut ramener à des formes plus connues.
La propriété prop :induction définit un principe d'induction et nous donnons dans la section suivante des variations sur ce principe.

1.5.2 Principes d'induction

P. et R. Cousot [11] donne une synthèse complète sur les principes d'induction équivalents à ce principe d'induction :

☺ **Propriété 1.14**
Les propriétés suivantes sont équivalentes :

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

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

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

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

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

PREUVE:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

1.6 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 [30] a proposé une méthode de preuves de machines de Turing, reprise plus tard par Floyd [12] et par Hoare [15]. P. et R. Cousot [10] 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 [26] 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] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[3] 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.

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

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

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

[7] Dominique Cansell and Dominique Méry. *The event-B Modelling Method : Concepts and Case Studies*, pages 33–140. Springer, 2007. See [6].

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

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

[10] 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.

[11] 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.

[12] 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.

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

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

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

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

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

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

[19] 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.

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

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

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

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

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

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

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

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

[28] 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.

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

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