Remarques relatives à l'emploi des méthodes formelles (déductives) en sécurité des systèmes d'information

Éric Jaeger, DCSSI/SDS/LTI [1]
Secrétariat général de la défense nationale
51 boulevard de la Tour-Maubourg
75700 Paris SP 07, France

14 avril 2008

Table des matières

Chapitre 1

Préambule

1.1 Objet du document

Ce document rassemble des rappels, des remarques et des conseils concernant l'utilisation des méthodes formelles, plus particulièrement dans le domaine de la sécurité des systèmes d'information. Il s'adresse aux développeurs et aux évaluateurs appelés à mettre en œuvre les méthodes formelles dans le cadre du développement d'un produit de sécurité – notamment, mais pas exclusivement, pour la mise en oeuvre des composants formels des classes d'assurance des *CC* (*Critères Communs*, [2]).

Bien que ce document ait vocation à être générique, i.e. à considérer l'ensemble des problématiques associées à l'utilisation d'une méthode formelle quelconque, il n'aborde dans sa version actuelle que les méthodes formelles de développement proposant des langages permettant la spécification et l'implémentation, ainsi que des mécanismes de vérification basés sur la déduction – par opposition notamment à des approches par construction de modèles. Les différents points présentés pourront être illustrés par des exemples courts – et volontairement simplifiés – en *B* [3], *Coq* [4] ou encore *FoCal* [5] sans que cela ne remette en cause leur généralité.

La connaissance théorique ou pratique d'une méthode formelle est un pré-requis à la compréhension de ce document. Un bref rappel sur *B*, *Coq* et *FoCal* est donné en annexe A, et une synthèse sur quelques points à vérifier en annexe B. Un glossaire est également fourni en annexe C.

1.2 Avertissement

De par leur nature, les méthodes formelles permettent d'aborder avec rigueur la problématique de la correction des logiciels et des systèmes. L'utilisation des méthodes formelles est réputée garantir des niveaux d'assurance allant très au-delà de ce qui peut être obtenu par les approches plus classiques basées sur les vérifications informelles et le test, et est à ce titre préconisée par les standards relatifs à la sûreté de fonctionnement [6] ou à la sécurité des systèmes d'information [2].

À la lecture de ce document, il pourrait sembler que cette réputation ne soit pas totalement méritée. **Ce n'est pas le message de ce document**. Dans la pratique, les méthodes formelles proposent effectivement une approche optimale pour l'obtention d'un excellent niveau de confiance ; l'emploi de la moins efficace des méthodes formelles reste préférable à une approche informelle.

L'objectif de ce document est par contre d'aider à identifier clairement le niveau de confiance que l'on peut accorder à un développement prouvé, et de proposer quelques conseils permettant d'améliorer ce niveau de confiance.

Chapitre 2

Introduction

2.1 Méthodes formelles

Une méthode formelle au sens le plus large est une méthode d'ingénierie pour le développement de systèmes (de logiciels) basée sur des concepts logiques et mathématiques rigoureux et bien identifiés. Une méthode formelle définit généralement un ou plusieurs langages (pour la spécification, le développement, etc.) non ambigus et des principes de raisonnement explicites, justifiés et vérifiables.

Le spectre des méthodes formelles est large. On peut considérer qu'il va des systèmes de typage[1] totalement automatisés aux méthodes imposant de prouver (manuellement) qu'une implémentation proposée respecte les propriétés exigées par la spécification, en passant par l'analyse statique ou encore le *Model Checking*[2].

La mise en œuvre de méthodes formelles dans le cadre de la sécurité des systèmes d'information se justifie par la possibilité de garantir qu'un système respecte certaines propriétés attendues. Le choix d'une méthodologie formelle dans un contexte donné dépend des objectifs fixés. Une analyse d'adéquation doit donc être menée au préalable, en prenant par exemple en compte :
– la fidélité, i.e. la relation entre la modélisation de l'objet de l'analyse formelle et le système réel,
– l'expressivité, i.e. le type de propriétés pouvant être formellement décrites et analysées,
– la portée, i.e. les étapes du cycle de développement pouvant être couvertes,
– la simplicité,
– l'efficacité, i.e. la disponibilité d'outils offrant plus ou moins d'automatisation.
Dans ce document, nous nous intéressons essentiellement aux méthodes formelles offrant un ou plusieurs langages permettant la spécification, la conception et l'implémentation, ainsi que des méthodes de vérification basées sur des démarches de preuve par déduction. Ces méthodes sont généralement très expressives, mais au prix d'un faible niveau d'automatisation.

Dans les méthodes formelles considérées, le langage de spécification est généralement un langage logique, permettant de décrire des propositions construites à partir de prédicats, de quantificateurs et d'opérateurs logiques. Le langage d'implémentation, au contraire, est généralement un langage de programmation. L'essence des étapes de vérification est de prouver que deux descriptions d'un système, l'une logique, l'autre opérationnelle, sont cohérentes entre elles.

Exemple 2.1.1 (Division par 2 en *Coq*) *La fonction* div_2 *définie ci-dessous représente bien la division entière par* 2*, puisque* $\mathrm{div2}_{\mathrm{Spec}}$ *est démontrable :*

> **Fixpoint** $\mathrm{div2}(x:\mathbb{N}):\mathbb{N} := \mathbf{match}\ x\ \mathit{with}\ S(S(x')) \to S(\mathrm{div2}(x')) \mid \_ \to 0\ \mathbf{end}.$
> **Theorem** $\mathrm{div2}_{\mathrm{Spec}}:\forall\,(x:\mathbb{N}),\ n=2{*}\mathrm{div2}(n) \vee n=2{*}\mathrm{div2}(n)+1.$

[1] Systèmes qui peuvent être extrêmement sophistiqués, notamment en exploitant la notion de type dépendant.
[2] La notion de *Méthode Formelle* dans le cadre des *CC* (des composants *SPM*) est cependant plus restrictive.

2.2 Méthodologie formelle de développement

Une démarche standard de développement vise, en partant d'un cahier des charges informel, à la production d'un système en passant par plusieurs étapes intermédiaires, telles que la décomposition en modules, la conception, etc. Entre deux étapes, une démarche standard préconise également une ou plusieurs formes de vérification permettant de s'assurer que les produits de l'étape sont conformes à ce qui est attendu – par exemple sous la forme de documents justificatifs, de relectures croisées, ou de tests.

Une méthodologie formelle de développement respecte ce schéma général, en proposant des étapes et des opérations de vérification basées sur des formalismes logiques et mathématiques rigoureux rendus exploitables par l'utilisation de langages dont la sémantique est explicite, claire et non ambiguë – permettant notamment d'éviter les problèmes d'interprétation.

À partir du cahier des charges, la première étape consiste généralement à écrire une spécification formelle décrivant ce que le système doit faire et garantir. Cette spécification est idéalement déclarative, abstraite, non constructive et non effective – elle se limite à une description du *quoi*. Elle constitue déjà un objet susceptible d'analyse formelle, puisqu'il est par exemple possible d'en étudier, au moins partiellement, la cohérence.

La méthodologie formelle de développement peut ensuite permettre de décrire la conception voire l'implémentation – i.e. la description du *comment*. Ici encore la formalisation de la sémantique du langage de développement permet des analyses locales relatives notamment à la cohérence, mais aussi des analyses globales de respect des propriétés spécifiées.

L'utilisation d'une méthodologie formelle de développement doit reposer sur l'utilisation d'un ensemble d'outils constituant une implémentation de la méthode formelle. Bien qu'il soit envisageable de mener l'ensemble des vérifications à la main, dans la pratique seul l'emploi d'un outil formel qui effectue des validations de manière mécanique apporte un niveau de confiance satisfaisant.

2.3 Évaluation de la confiance

L'emploi d'une méthodologie formelle de développement permet d'envisager de produire un code dont il est prouvé qu'il est conforme à sa spécification. Une telle preuve est une contribution extrêmement importante à la confiance que l'on peut avoir dans le résultat du développement, notamment en raison de son caractère exhaustif.

Pourtant, malgré le caractère absolu d'une preuve formelle, on ne peut généralement pas en déduire que le système réel vérifiera forcément et inconditionnellement les propriétés attendues. Dans la suite de ce document, nous abordons en effet différentes problématiques qui peuvent se poser, en considérant les grandes étapes génériques d'une méthodologie formelle de développement.

Le chapitre 3 s'intéresse à la qualité intrinsèque de la spécification formelle, notant qu'elle peut être erronée ou incomplète – en raison d'un problème dans le cahier des charges ou lors de sa traduction en spécification. Il faut cependant noter que le simple effort de traduction d'un cahier des charges informel en une spécification formelle permet d'augmenter notablement le niveau de confiance dans le développement, en raison de la réflexion imposée et de la représentation utilisée qui permet notamment de lever les éventuelles ambiguïtés du cahier des charges. Par ailleurs, tout ne peut pas être formalisé, et c'est à la documentation d'associer aspects formels et informels.

Le chapitre 4 explicite comment une mauvaise compréhension de la méthodologie formelle de développement peut mener à des erreurs d'interprétation des résultats obtenus. Il met en évidence les conséquences possibles d'une mauvaise compréhension de la sémantique des langages utilisés, notamment pour la spécification, ou encore de la signification réelle des propriétés prouvées.

Le chapitre 5 donne quelques illustrations de spécifications non (ou difficilement) exprimables, en

raison de limitations inhérentes aux formalismes utilisés.

Le chapitre 6 complète le chapitre 5 en indiquant comment une mauvaise compréhension des outils implémentant une méthodologie formelle de développement peut mener à des erreurs d'interprétation.

Enfin, les chapitres 7 et 8 s'intéressent à la correction des outils implémentant une méthodologie formelle et à la validité des théories mathématiques et logiques sous-jacentes.

Dans le cadre du développement d'un produit de sécurité mettant en œuvre une méthodologie formelle de développement, avec l'objectif de fournir des garanties concernant le respect des propriétés de sécurité, ces différents points doivent être considérés. Cette analyse devra permettre d'identifier de bonnes pratiques, ou plus prosaïquement d'évaluer de manière fine le niveau de confiance réellement obtenu.

Chapitre 3

Formaliser suffisamment

Le cahier des charges décrit ce qui est attendu du système ; il peut être complété par des exigences relatives au processus de développement, notamment en précisant dans quelle mesure une méthodologie formelle de développement doit être mise en œuvre, et à quelles fins.

Si les méthodes formelles présentent beaucoup d'avantages – la formalisation des langages permettant de lever les ambiguïtés, l'explicitation de règles de raisonnement et de la méthodologie permettant d'analyser la cohérence ou la conformité d'une implémentation ainsi que la vérification indépendante *a posteriori* – leur mise en œuvre reste délicate et coûteuse. Il faut donc bien identifier ce qui mérite d'être formalisé, tout en justifiant les choix effectués. Le périmètre de la formalisation d'un système peut être réduit à un sous-système ou un sous-ensemble des fonctionnalités.

Une fois le périmètre de la formalisation décidé, par contre, il est préférable de s'astreindre à mettre en œuvre l'ensemble des mécanismes de contrôle et de vérification proposé par la méthodologie formelle de développement. *A contrario*, si ces mécanismes ne sont pas utilisés, la méthodologie de développement – bien que menée dans un environnement formel – se ramènera à une méthodologie standard, et le niveau d'assurance visé ne sera pas atteint.

3.1 Correction et suffisance

La première recommandation est de s'assurer que la spécification formelle décrit effectivement les propriétés qui sont attendues du système – tout au moins celles dont la formalisation a été jugée pertinente lors de la définition du périmètre de formalisation.

En effet, de manière évidente, si on omet dans la spécification formelle de détailler les propriétés attendues du système, aucune garantie ne peut être donnée quant à leur respect. Affaiblir une spécification se traduit par un accroissement, souvent incontrôlable, de l'ensemble des implémentations conformes – jusqu'à permettre des implémentations prouvées mais manifestement non satisfaisantes du point de vue de la sécurité ou même plus simplement des fonctionnalités.

Exemple 3.1.1 (Métro en B) *On spécifie les commandes d'accélération, de ralentissement, d'ouverture et de fermeture des portes d'un métro. Les propriétés recherchées doivent assurer que la vitesse est toujours positive, toujours inférieure à une valeur V_{Max} donnée, et que les portes ne sont jamais ouvertes lorsque le véhicule est en mouvement.*

La spécification formelle proposée décrit un état avec deux variables (v pour la vitesse et p pour les portes), quatre opérations, mais l'invariant (trop faible) se limite à la description des types des variables de l'état. Cependant, le développeur s'assure que la spécification des opérations garantit les propriétés attendues, en

bornant v et en interdisant l'ouverture des portes lorsque le métro est en mouvement :

MACHINE Metro
VARIABLES v, p
INVARIANT $v \in \mathbb{N} \wedge p \in \{\text{ouvert}, \text{ferme}\}$
OPERATIONS
accel $\triangleq$ **IF** $v < V_{\text{Max}}$ **THEN** $v :\in [v+1, V_{\text{Max}}]$
decel $\triangleq$ **IF** $v > 0$ **THEN** $v :\in [0, v-1]$
ouvre $\triangleq$ **IF** $v = 0$ **THEN** $p := \text{ouvert}$
ferme $\triangleq$ $p := \text{ferme}$

Pourtant, lorsque l'on renforce l'invariant en y intégrant l'ensemble des propriétés recherchées, les obligations de preuve complémentaires mettent en évidence que la spécification de l'opération accel *doit être corrigée pour interdire d'accélérer lorsque les portes sont ouvertes :*

MACHINE $\text{Metro}_{\text{Corr}}$
VARIABLES v, p
INVARIANT $v \in [0, V_{\text{Max}}] \wedge p \in \{\text{ouvert}, \text{ferme}\} \wedge\ v \neq 0 \Rightarrow p = \text{ferme}$
OPERATIONS
accel $\triangleq$ **IF** $v < V_{\text{Max}} \wedge p = \text{ferme}$ **THEN** $v :\in [v+1, V_{\text{Max}}]$
decel $\triangleq$ **IF** $v > 0$ **THEN** $v :\in [0, v-1]$
ouvre $\triangleq$ **IF** $v = 0$ **THEN** $p := \text{ouvert}$
ferme $\triangleq$ $p := \text{ferme}$

Le fait de formaliser l'ensemble des propriétés pertinentes permet de s'assurer, dans une certaine mesure, de la cohérence de la spécification. La problématique générale de la correction est cependant plus délicate, puisqu'il n'est pas réellement possible de fournir des critères d'analyse de la validité sémantique d'une spécification.

L'analyse de la suffisance ou de la correction de la spécification relève donc essentiellement d'une activité d'expertise humaine – qui pourra au mieux être assistée par des outils (par exemple d'animation de modèle).

Notons que certains concepts formels, notamment la complètude d'une spécification[1] [7], permettent de pousser plus avant l'analyse formelle d'une spécification, mais ne sont pas toujours supportés par la méthodologie de développement utilisée en raison des limitations sur l'expressivité du langage. De plus, il est rarement souhaitable d'éliminer toute liberté d'implémentation dès la spécification.

3.2 Totalité

Lors de la vérification de l'adéquation d'une spécification, il faut idéalement s'intéresser à l'identification des cas couverts. En effet, il peut arriver qu'une spécification ne soit que partielle, et n'impose aucune contrainte dans certaines situations.

La partialité d'une spécification semble *a priori* pouvoir se justifier dans certains contextes. Ainsi, une fonction calculant un racine carrée peut n'être spécifiée que pour les valeurs positives – ce qui peut signifier, selon les cas, soit que la fonction ne doit pas être appelée avec un paramètre négatif (notion de pré-condition), soit que le résultat d'un tel appel est sans importance et laissé au libre choix du développeur (spécification partielle).

Il est pourtant recommandé de s'assurer que la spécification est totale, i.e. ne laisse aucune zone d'ombre. En effet, une spécification partielle est rarement pleinement satisfaisante du point de vue de la sécurité, et dans certains cas peut se révéler catastrophique [8].

[1] Une spécification est complète si elle impose que deux implémentations conformes I_1 et I_2 sont forcément observationnellement identiques. Le terme monomorphicité est parfois utilisé.

Exemple 3.2.1 (Gestion de fichiers en B) *On spécifie un système de gestion de fichiers avec droits d'accès. Le type abstrait* USR *décrit l'ensemble des usagers (sources des requêtes),* FIL *l'ensemble de tous les fichiers possibles,* CNT *l'ensemble de tous les contenus possibles et enfin* RGT *les droits, ici r pour la lecture et w pour l'écriture. L'état du système est décrit par les variables* Fil *(l'ensemble des fichiers existants),* Cnt *(qui à chaque fichier associe un contenu),* Rgt *(qui pour une identité et un fichier existant associe des droits) et enfin* cpt *(le nombre de fichiers).*

La machine comprend des opérations permettant de créer, effacer, changer ou lire le contenu d'un fichier ou encore d'en changer les droits, et enfin de connaître le nombre de fichiers :

MACHINE filesystem
SETS USR; FIL; CNT; RGT $=\{r, w\}$
VARIABLES Fil, Cnt, Rgt, cpt
INVARIANT
  $\text{Fil} \subseteq \text{FIL} \wedge \text{Cnt} \in \text{Fil} \to \text{CNT} \wedge \text{Rgt} \subseteq (\text{USR} \times \text{Fil}) \times \text{RGT} \wedge \text{cpt} = \mathbf{card}(\text{Fil})$
INITIALISATION $\text{Fil} := \emptyset \parallel \text{Cnt} := \emptyset \parallel \text{Rgt} := \emptyset \parallel \text{cpt} := 0$
OPERATIONS
  $\text{new}(f, u, c) \triangleq$
    PRE $f \in \text{FIL} \wedge u \in \text{USR} \wedge c \in \text{Cnt}$ **THEN**
      IF $f \notin \text{Fil}$ **THEN**
        $\text{Fil} := \text{Fil} \cup \{f\} \parallel \text{Cnt} := \text{Cnt} \cup \{f \mapsto c\} \parallel \text{Rgt} := \text{Rgt} \cup (\{u \mapsto f\} \times \text{RGT}) \parallel \text{cpt} := \text{cpt} + 1$
  $\text{del}(f, u) \triangleq$
    PRE $f \in \text{FIL} \wedge u \in \text{USR}$ **THEN**
      IF $((u \mapsto f) \mapsto w) \in \text{Rgt}$ **THEN**
        $\text{Fil} := \text{Fil} - \{f\} \parallel \text{Cnt} := \{f\} \lhd \text{Cnt} \parallel \text{Rgt} := (\text{USR} \times \{f\}) \lhd \text{Rgt} \parallel \text{cpt} := \text{cpt} - 1$
  $\text{write}(f, u, c) \triangleq$
    PRE $f \in \text{FIL} \wedge u \in \text{USR} \wedge c \in \text{Cnt}$ **THEN**
      IF $((u \mapsto f) \mapsto w) \in \text{Rgt}$ **THEN** $\text{Cnt}(f) := c$
  $\text{out} \leftarrow \text{read}(f, u) \triangleq$
    PRE $f \in \text{Fil} \wedge u \in \text{USR}$ **THEN**
      IF $((u \mapsto f) \mapsto r) \in \text{Rgt}$ **THEN** $\text{out} := \text{Cnt}(f)$
  $\text{chrgt}(f, u_1, u_2, r) \triangleq$
    PRE $f \in \text{FIL} \wedge u_1 \in \text{USR} \wedge u_2 \in \text{USR} \wedge r \subseteq \text{RGT}$ **THEN**
      IF $f \in \text{Fil} \wedge ((u_1 \mapsto f) \mapsto w) \in \text{Rgt}$ **THEN** $\text{Rgt} := \text{Rgt} \lhd\!\!+ \{u_2 \mapsto f\} \times r$
  $\text{out} \leftarrow \text{count} \triangleq \text{out} := \text{cpt}$

Les pré-conditions se limitent à décrire le type des paramètres passés lors des appels aux opérations, ce qui correspond à des spécifications totales pour ces opérations. La seule exception est l'opération read *qui n'est définie que si le fichier passé en paramètre existe (puisque* $\text{Fil} \subseteq \text{FIL}$*).*

En l'occurence, cette pré-condition signifie que l'opération de lecture d'un fichier ne doit être appelée que si l'existence de ce fichier est garantie (i.e. a été vérifiée préalablement). Lorsque la pré-condition n'est pas vérifiée, le résultat de l'opération read *n'est pas spécifié.*

*Il ne faut pas sous-estimer les conséquences possibles d'une telle sous-spécification ; le non respect d'une pré-condition (***PRE***) correspond à une zone aveugle de la spécification, dans laquelle tout est possible – et ne doit pas être confondu avec le non respect d'une condition. En effet, considérons le code alternatif suivant pour l'opération* read *:*

  $\text{out} \leftarrow \text{read}(f, u) \triangleq$
    PRE $f \in \text{FIL} \wedge u \in \text{USR}$ **THEN**
      IF $f \in \text{Fil}$ **THEN**
        IF $((u \mapsto f) \mapsto r) \in \text{Rgt}$ **THEN** $\text{out} := \text{Cnt}(f)$

L'existence du fichier est vérifiée par l'utilisation d'un **IF***, la non existence du fichier se traduit par le passage dans la branche* **ELSE***. Lorsque la branche* **ELSE** *n'apparait pas dans la spécification, comme c'est le cas ici, elle est en fait implicite et correspond à un* **skip***, la substitution sans effet (qui ne modifie pas l'état du système). Le non respect d'une pré-condition, au contraire, permet de faire n'importe quoi.*

Ainsi, alors que l'opération read *semble spécifiée comme étant passive, son raffinement peut totalement modifier l'état de la machine lorsqu'il est appelé hors du domaine spécifié. Par exemple il est possible que*

le raffinement, lors de la lecture d'un fichier inexistant, crée un fichier fs *contenant une donnée secrète S du système et seulement accessible en lecture à un utilisateur* eni *:*

$$
\begin{array}{l}
\text{out} \leftarrow \text{read}(f, u) \triangleq \\
\quad \textbf{IF } f \in \text{Fil } \textbf{THEN} \\
\quad\quad \textbf{IF } ((u \mapsto f) \mapsto r) \in \text{Rgt } \textbf{THEN } \text{out} := \text{Cnt}(f) \\
\quad \textbf{ELSE} \\
\quad\quad \text{Fil} := \text{Fil} \cup \{\text{fs}\} \parallel \text{Cnt} := \text{Cnt} \cup \{\text{fs} \mapsto S\} \parallel \text{Rgt} := \text{Rgt} \cup \{(\text{eni} \mapsto \text{fs}) \mapsto r\}
\end{array}
$$

Notons que ce raffinement a aussi la possibilité de violer l'invariant : en effet, la création du fichier fs *ne s'accompagne pas ici d'une incrémentation de* cpt*, nous sommes donc en présence d'un fichier qui est virtuellement caché. On peut aussi remarquer que la valeur de sortie* out *n'a pas besoin d'être précisée.*

Exemple 3.2.2 (Tête de liste *Coq*) *La fonction* head*, qui renvoie le premier élément d'une liste d'entiers naturels, peut être décrite sous la forme d'une fonction partielle non définie pour la liste vide*[2] *:*

```
Require Import List.
Variable secret : ℕ.
Definition head(l : list ℕ)(p : l ≠ []) : {x : ℕ | exists l' : list ℕ, l = x :: l'}.
  destruct l as [ | h l']; intros H.
    exists secret; destruct H; apply refl_equal.
    exists h; exists l'; apply refl_equal.
Defined.
```

Alternativement, head *peut aussi être décrite comme une fonction totale, mais dont la spécification est partielle, i.e. elle ne décrit pas le résultat retourné lorsque la liste passée en paramètre est vide :*

```
Definition head(l : list ℕ) : {x : ℕ | l ≠ [] → exists l' : list ℕ, l = x :: l'}.
  destruct l as [ | h l'].
    exists secret; intros H; destruct H; apply refl_equal.
    exists h; intros H; exists l'; apply refl_equal.
Defined.
```

Dans les deux cas, l'extraction du code[3] *de* head *en* OCaml *donne :*

```
let rec head = function [] → secret | h :: _ → h
```

Dans le style fonctionnel propre à Coq*, il n'y a pas de notion d'état qui pourrait être modifié par un appel de fonction hors spécification. Cependant, il reste possible au développeur d'implémenter les comportements de son choix, y compris des comportements indésirables visant par exemple à révéler des informations confidentielles (ici la valeur* secret*).*

Notons que la notion de violation de pré-condition est dénuée de sens dans le cadre formel. En effet, par définition, l'appel à une fonction ou une opération ne peut se faire que si la pré-condition est vérifiée.

La notion de violation de pré-condition est par contre tout à fait réaliste dans le monde réel [8]. Elle peut tout simplement résulter d'une erreur de développement entre les interfaces des modules formels et des modules standards, d'une panne (accidentelle ou non) ou de toute autre forme de violation du modèle associé à la formalisation. Pour illustrer ce principe, considérons un système informatique qui gère un ensemble de données ; dans la pratique, ces données pourront être concrétisées par des fichiers accessibles depuis le système d'exploitation, sans contrôle de l'application développée formellement.

[2] Dans le système *Coq* toute fonction est totale ; les fonctions partielles sont représentées par l'utilisation d'un type dépendant (ici une preuve que la liste en paramètre n'est pas vide) qui correspond à une forme de pré-condition.

[3] Nous utilisons ici une forme particulière de programmation autorisée par *Coq*, en utilisant le mécanisme d'extraction de programmes fonctionnels à partir des preuves constructives, de manière automatique.

3.3 Typage

Les méthodes formelles proposent, le plus souvent, une forme de typage. Une bonne pratique de formalisation consiste à exploiter au mieux ce typage pour limiter les erreurs de spécification – de la même façon que le typage permet de limiter les erreurs de programmation.

Exemple 3.3.1 (Champs d'un paquet *IP* en *B*) *La spécification d'une opération permettant de construire un paquet* IP *peut être faite de la façon suivante :*

$$\begin{array}{l}\text{out} \leftarrow \text{build}(a_s, a_d, p_s, p_d, \ldots) \triangleq \\ \quad \textbf{PRE}\ a_s \in [0, 2^{32}-1] \wedge a_d \in [0, 2^{32}-1] \wedge p_s \in [0, 2^{16}-1] \wedge p_d \in [0, 2^{16}-1] \ldots \textbf{THEN} \\ \quad\quad \text{out} := (((a_s \mapsto a_d) \mapsto p_s) \mapsto p_d) \ldots\end{array}$$

Dans l'absolu, une adresse ou un numéro de port sont effectivement des valeurs entières et bornées. Pourtant, que signifie l'addition d'une adresse avec la valeur 2*, ou encore la multiplication d'une adresse par un numéro de port ? Il est préférable de travailler avec des ensembles abstraits*[4] *:*

$$\begin{array}{l}\textbf{SETS}\ \text{ADD}; \text{PRT} \\ \ldots \\ \text{out} \leftarrow \text{build}(a_s, a_d, p_s, p_d, \ldots) \triangleq \\ \quad \textbf{PRE}\ a_s \in \text{ADD} \wedge a_d \in \text{ADD} \wedge p_s \in \text{PRT} \wedge p_d \in \text{PRT} \ldots \textbf{THEN} \\ \quad\quad \text{out} := (((a_s \mapsto a_d) \mapsto p_s) \mapsto p_d) \ldots\end{array}$$

Avec cette formalisation, certaines erreurs grossières seront évitées, notamment l'instanciation d'un champ adresse avec un numéro de port.

3.4 Pratiques de développement

Malgré les garanties données par l'utilisation des méthodes formelles, les développeurs doivent continuer à appliquer les pratiques standards du développement : l'existence de garanties sous la forme de preuves formelles ne justifie pas la disparition de la documentation, le manque de commentaires, l'absence de règles de nommage ou encore une modularisation inadaptée.

Plus particulièrement, la problématique du choix entre une approche offensive et une approche défensive de la programmation doit être considérée avec attention.

Les méthodes formelles peuvent en effet permettre de justifier l'élimination d'un certain nombre de vérifications – lorsqu'il est possible de démontrer qu'elles sont inutiles (redondantes) par construction. Le style de programmation peut donc évoluer d'une approche défensive vers une approche offensive ; la validité des valeurs manipulées n'est plus testée mais supposée (sous la forme de pré-conditions).

Pourtant, les illustrations données dans la section 3.2 montrent ce qu'une telle approche peut avoir de dangereux. En effet, peut-on toujours garantir le respect des pré-conditions, i.e. de la spécification d'emploi, notamment aux frontières du développement formel ?

De manière plus générale, il est également pertinent de s'interroger sur le respect du modèle d'exécution formel dans la pratique – par exemple, le compilateur ou le microprocesseur peuvent être bogués. Plus fondamentalement, le modèle d'exécution sous-jacent peut être purement théorique et irréalisable dans la pratique. Souvent par exemple le fait que l'on développe pour un système ayant une mémoire finie n'est pas modélisé, et des algorithmes corrects mais irréalisables dans la pratique car provoquant immanquablement un débordement peuvent être validés.

Enfin, même dans les cas les plus favorables, un système peut rester sensible à certaines formes de perturbations (erreurs, pannes ou injections de fautes) qui sont, par définition, ignorées par le

[4] Ensembles qui pourront être ensuite raffinés par des intervalles d'entiers.

modèle formel si le développeur n'y a pas volontairement et explicitement intégré des considérations dysfonctionnelles.

Il semble donc utile, conformément au principe de défense en profondeur, de maintenir dans un code formel des éléments de vérification même si ceux-ci sont redondants et inutiles. De manière plus générale, la question posée ici peut remonter jusqu'au niveau de la spécification : celle-ci ne devrait pas toujours exclure *a priori* des états non sûrs – au risque de ne formaliser qu'une situation hypothétique dans laquelle les propriétés de sécurité sont maintenues... mais pas établies. Les analyses de la validité des raisonnements par clôture, de la pertinence du typage (cf. la sous-section 4.1.3), voire la prise en compte des aspects dysfonctionnels, devraient donc être menées et documentées.

3.5 Un dernier rappel

Pour en terminer avec cette section qui insiste sur la nécessité de formaliser suffisamment, il semble utile de rappeler qu'une spécification peut accepter bien plus d'implémentations (de modèle) qu'on ne le pense.

Exemple 3.5.1 (Description abstraite de $\mathbb{N}$ en *Coq*) *On tente de définir* $\mathbb{N}$ *comme une structure abstraite munie d'une constante* 0*, d'une fonction* S *et d'une relation* eq *(l'égalité) :*

Variable $\mathbb{N}:\mathbf{Set}$.
Variable $0:\mathbb{N}$.
Variable $S:\mathbb{N}\to\mathbb{N}$.
Variable $\mathrm{eq}:\mathbb{N}\to\mathbb{N}\to\mathbf{Prop}$.

Afin que cette structure abstraite corresponde bien à la description des nombres naturels on décrit également les propriétés attendues :

Variable $\mathrm{eq\_congr}:\forall\,(P:\mathbb{N}\to Prop)(n\,m:\mathbb{N}),\,P\,n\to\mathrm{eq}\,n\,m\to P\,m$.
Variable $0\_not\_S:\forall\,(n:\mathbb{N}),\,\neg\,\mathrm{eq}\,0\,(S\,n)$.
Variable $\mathrm{S\_inj}:\forall\,(n\,m:\mathbb{N}),\,\mathrm{eq}\,(S\,n)\,(S\,m)\to\mathrm{eq}\,n\,m$.

Il est facile de se convaincre que ces propriétés sont correctes. Il est plus délicat de se convaincre qu'elles sont suffisantes, i.e. qu'elles décrivent seulement les nombres naturels (ou une structure isomorphe). Et effectivement, la description de l'égalité par exemple est ici insuffisante puisqu'elle accepte le modèle dans lequel eq *est la relation vide (i.e.* $\forall\,(n\,m:\mathbb{N}),\,\neg\,\mathrm{eq}\,n\,m$*). Pour interdire ce modèle, il faut compléter notre spécification comme suit :*

Variable $\mathrm{eq\_refl}:\forall\,(n:\mathbb{N}),\,\mathrm{eq}\,n\,n$.

Cette spécification reste cependant insuffisante vis à vis de notre objectif de décrire $\mathbb{N}$ *et seulement* $\mathbb{N}$*. En effet, la spécification décrit maintenant les structures contenant les entiers naturels, mais pas seulement, puisqu'il n'y a pas d'exigence de surjectivité. Pour être plus clair, rien n'interdit ici que* $\mathbb{N}$ *contienne autre chose que les nombres naturels (obtenus par l'utilisation de* 0 *et de* S*). Il est donc nécessaire d'ajouter la propriété suivante pour exclure d'autres formes de valeurs :*

Variable $0\_\mathrm{S\_surj}:\forall\,(n:\mathbb{N}),\,\mathrm{eq}\,n\,0\vee\exists\,p:\mathbb{N},\,\mathrm{eq}\,n\,(S\,p)$.

Ce n'est toujours pas suffisant. En effet, quelques structures exotiques qui ne sont pas $\mathbb{N}$ *restent acceptables avec cette spécification. On pourra par exemple considérer la structure définie par* $\mathbb{N}\cup\{\omega\}$*, où* ω *est une valeur supplémentaire pour laquelle* $S\,\omega=\omega$*, ou encore* $\mathbb{N}\uplus\mathbb{Z}$*, etc.*

De manière générale, cela signifie qu'il reste important de se convaincre, sans aller jusqu'à imposer la complétude, que la spécification est suffisamment détaillée pour ne permettre que des implémentations acceptables.

Chapitre 4

Comprendre la formalisation

Dans ce chapitre, nous nous intéressons aux problèmes qui pourraient résulter d'une mauvaise compréhension par l'utilisateur des principes sous-jacents associés à une méthode formelle.

4.1 Maîtrise de la méthode formelle

Les méthodes formelles reposent sur des concepts et des théories qui, sans être obligatoirement complexes, ne sont pas nécessairement maîtrisés ou mêmes connus par les utilisateurs. L'existence d'outils automatisant les processus et offrant des interfaces homme-machine peut en effet masquer les aspects les plus abstraits. Une compréhension partielle de la théorie d'une méthode formelle, ou du fonctionnement des outils formels associés, peut cependant mener à des erreurs d'interprétation ayant des conséquences potentielles graves.

4.1.1 Notion d'invariant

Il est fréquent dans une méthode formelle d'utiliser la notion d'invariant, censée décrire une propriété vraie à tout instant de la vie d'un système ou de l'exécution d'un logiciel – tout au moins sous certaines hypothèses d'environnement. Pourtant, la définition exacte d'un invariant est généralement subtilement différente, puisqu'il correspond à une propriété qui n'est vraie qu'à des instants pertinents du point de vue de la méthode formelle employée.

De fait, les états transitoires ne sont généralement pas considérés, ni le temps d'exécution d'un programme : le B ne garantit que la préservation d'un invariant après la réalisation totale d'une opération (et sous l'hypothèse que l'invariant était vrai avant l'appel de l'opération), *Coq* travaille modulo β (i.e. sans prendre en compte le calcul, par exemple $1+2=3$), etc.

Exemple 4.1.1 (Spécification d'un sas en B) *On spécifie les commandes d'ouverture et de fermeture des deux portes d'un sas de sous-marin en cherchant à garantir qu'à tout instant, au plus une porte est ouverte. L'état du système est décrit par deux variables, p_e pour la porte externe et p_i pour la porte interne :*

MACHINE Sas
VARIABLES p_e, p_i
INVARIANT $p_e, p_i \in \{\text{ouvert}, \text{ferme}\} \wedge \neg(p_i = \text{ouvert} \wedge p_e = \text{ouvert})$
OPERATIONS
$\text{ouvre}_e \triangleq$ **IF** $p_i = \text{ferme}$ **THEN** $p_e := \text{ouvert}$
$\text{ferme}_e \triangleq p_e := \text{ferme}$
$\text{ouvre}_i \triangleq$ **IF** $p_e = \text{ferme}$ **THEN** $p_i := \text{ouvert}$
$\text{ferme}_i \triangleq p_i := \text{ferme}$

Cette spécification est cohérente, au sens où elle assure le respect de l'invariant. Pourtant, elle ne permet pas d'assurer la propriété recherchée, en raison de la sémantique associée à la notion d'invariant. En effet, l'invariant est seulement une propriété qui doit être vraie à l'initialisation du système (omise ici), et maintenue par l'exécution (totale) d'une opération. En particulier l'invariant peut être faux pendant les états transitoires. Ainsi, le raffinement suivant est tout à fait valide (i.e. peut être prouvé conforme) :

REFINEMENT Sas_{Ref} **REFINES** Sas
VARIABLES p_e, p_i
INVARIANT $p_e, p_i \in \{\text{ouvert}, \text{ferme}\} \wedge \neg(p_i = \text{ouvert} \wedge p_e = \text{ouvert})$
OPERATIONS
$\text{ouvre}_e \triangleq$ **IF** $p_i = \text{ferme} \wedge \text{attack}$ **THEN** $p_i := \text{ouvert}; p_e := \text{ouvert}; \text{wait}; p_i := \text{ferme}$
$\text{ferme}_e \triangleq p_e := \text{ferme}$
$\text{ouvre}_i \triangleq$ **IF** $p_e = \text{ferme}$ **THEN** $p_i := \text{ouvert}$
$\text{ferme}_i \triangleq p_i := \text{ferme}$

où wait *est une commande passive (sans effet sur l'état) mais dont le temps d'exécution est non négligeable, et* attack *une condition arbitrairement compliquée sur laquelle l'attaquant peut agir à sa guise. Il est donc possible d'implémenter la spécification* Sas *sous la forme d'un système qui ouvre les deux portes simultanément pendant plusieurs secondes, tout en masquant ce comportement pendant des tests.*

Notons que si un invariant toujours vrai est recherché, y compris pendant les éventuels états transitoires, certaines méthodes formelles permettent d'intégrer des sémantiques de type interruption[1] (cf. notamment [9], ou [10] pour un problème dual).

4.1.2 Hypothèses implicites

Un autre problème qui peut se poser est celui de la méconnaissance de la sémantique de certaines constructions formelles, ou encore des hypothèses générales mais implicites. Cette méconnaissance peut se traduire par des spécifications qui, bien qu'en apparence cohérentes, deviennent incohérentes lorsque combinées avec les hypothèses implicites.

Comme souvent, l'incohérence d'une spécification n'est pas évidente à détecter, et si elle n'est pas activement recherchée elle peut même se révéler en pratique confortable puisqu'elle permet de prouver tout résultat attendu en utilisant des raisonnements par l'absurde.

L'implémentation constitue une méthode très efficace pour s'assurer de l'absence d'incohérences, puisqu'il est impossible d'implémenter une spécification incohérente[2]. L'implémentation (formelle) n'est cependant pas toujours un objectif dans les projets de développement[3].

Exemple 4.1.2 (Ensembles abstraits en *B*) *En* B *un ensemble abstrait (défini dans la clause* **SETS** *d'une machine) est toujours supposé non vide et fini.*

Contredire cette hypothèse implicite, par exemple en associant à une déclaration **SETS** *S une contrainte* **PROPERTIES** *$S = \emptyset$, est donc immédiatement incohérent, sans qu'aucune alerte ne soit générée. En fait un outil tel que l'*Atelier B *pourra même tirer bénéfice de la contradiction pour décharger automatiquement l'ensemble des obligations de preuve. Si le développement se poursuit jusqu'à l'implémentation, le problème sera mis en évidence par l'impossibilité d'exhiber un ensemble S qui soit à la fois vide et non vide.*

Pour mémoire, le B-Book *[3] indique qu'un tel ensemble abstrait est concrétisé pendant la preuve par une hypothèse de la forme $S \in \mathbb{P}_1(\textbf{INT})$ où* **INT** *est le type des entiers machine. L'implémentation dans* B4Free

[1] Une interruption est un événement pouvant se produire à tout instant, y compris pendant l'exécution d'un programme, et peut être attachée à une propriété qui devra donc être toujours vérifiée. Cependant une telle sémantique devra nécessairement proposer une granularité temporelle ou l'atomicité de l'exécution de certaines instructions.

[2] L'implémentation constitue un modèle, au sens de la logique, des contraintes exprimées. À titre d'illustration, une valeur v peut être spécifiée par $v = 0 \wedge v = 1$, ce qui ne pose pas de problème (logique) tant qu'elle reste abstraite. Il est cependant évident qu'il est impossible d'exhiber un témoin, i.e. une valeur concrète qui vérifie la spécification.

[3] Pour le développement de produits de sécurité, les *CC* n'imposent par exemple à certains niveaux d'assurance qu'une modélisation formelle de la politique de sécurité sans preuve de conformité de l'implémentation.

et Click'n'Prove *génère au contraire une hypothèse de la forme* $S \in \mathbb{F}_1(\mathbb{Z})$*, ce qui est subtilement différent, puisque si dans les deux cas* S *est fini et non vide, dans le second cas les valeurs contenues dans* S *ne sont pas obligatoirement bornées.*

Exemple 4.1.3 (Entiers relatifs en *Coq*) *Le système* Coq *permet de définir des types inductifs, par exemple les entiers naturels,* $\textbf{Inductive}\,\mathbb{N} : \textbf{Set} := 0 : \mathbb{N} \mid S : \mathbb{N} \to \mathbb{N}$.

Il peut sembler évident de définir un entier relatif comme étant un entier naturel marqué d'un signe, et d'ajouter un axiome d'égalité pour indiquer que $+0$ *et* -0 *sont deux représentations de la même valeur :*

$\textbf{Inductive}\,\mathbb{Z}' : \textbf{Set} := \text{plus} : \mathbb{N} \to \mathbb{Z}' \mid \text{minus} : \mathbb{N} \to \mathbb{Z}'$
$\textbf{Axiom}\,\text{zero\_unsigned} : \text{plus}(0) = \text{minus}(0).$

Malheureusement, l'axiome zero_unsigned *est incohérent, et il devient possible avec lui de prouver n'importe quel résultat :*

Theorem inconsistent : **False**.
Proof.
  generalize zero_unsigned; **intros** H; **inversion** H.
Qed.

La sémantique d'une définition inductive dans Coq *a de nombreuses propriétés implicites*[4] *: bonne fondation (tout terme a une profondeur d'induction finie), surjectivité des constructeurs (tout terme est produit à partir des constructeurs) et injectivité des constructeurs (distinguabilité de deux termes qui ne sont pas structurellement identiques). C'est ce dernier point qui rend incohérent l'axiome* zero_unsigned *dans notre exemple. En effet,* plus *et* minus *étant des constructeurs différents, on peut toujours prouver en* Coq *que* $\text{plus}(n)$ *est (structurellement) différent de* $\text{minus}(m)$ *quels que soient* m *et* n.

L'approche correcte consiste donc soit à définir avec plus de précautions un type inductif représentant les entiers relatifs, soit à utiliser une équivalence représentant l'égalité des valeurs mais ne se ramenant pas à l'égalité de Coq*, soit enfin à adopter une approche par types abstraits de la forme suivante :*

$\textbf{Variable}\,\mathbb{Z} : \textbf{Set}.$
$\textbf{Variable}\,\Pi_+ : \mathbb{N} \to \mathbb{Z}.$
$\textbf{Variable}\,\Pi_- : \mathbb{N} \to \mathbb{Z}.$
$\textbf{Axiom}\,\text{zero\_unsigned} : \Pi_+(0) = \Pi_-(0).$
...

4.1.3 Hypothèses cachées : le typage

L'utilisation du typage est recommandée dans la section 3.3 comme un moyen d'éviter certaines erreurs élémentaires de spécification et de programmation.

Il est cependant nécessaire de bien analyser tous les tenants et aboutissants associés au typage afin de s'assurer qu'il ne masque pas certains problèmes potentiels. En effet, le typage est une contrainte qui apparaît dans une spécification ou un code source, mais le plus souvent disparaît du programme exécuté[5]. Il s'agit donc d'une contrainte logique qui n'a pas toujours une réalité opérationnelle – et en particulier peut être volontairement ignorée par un attaquant.

Ainsi, une fonction de chiffrement peut avoir une spécification de la forme $\mathit{cipher} : K \to M \to M$ où K est le type abstrait des clés et M le type abstrait des messages. Cette spécification permet d'éviter des erreurs de développement simples, telles que l'inversion accidentelle des deux paramètres. Dans la pratique, pourtant, les données de ces deux types sont encodées sous la forme de flux binaires, et ne sont pas distinguables si des mesures spécifiques n'ont pas été prises. Il devient donc pertinent

[4] En fait, on peut voir une définition inductive dans *Coq* comme relevant à la fois de la spécification et de l'implémentation ; le modèle (au sens logique) associé est celui d'une algèbre de termes.

[5] Le typage est généralement statique, vérifié à la compilation ; pour certains langages (*Java* par exemple) la vérification du typage n'est pas toujours totalement réalisable statiquement et le code est automatiquement instrumenté à la compilation pour permettre des vérifications complémentaires pendant l'exécution.

de se demander si un attaquant exploitant l'application peut violer le modèle et faire passer un message pour une clé ou vice-versa.

Un exemple bien connu, décrit dans [11], décrit une attaque contre certaines implémentations de l'API cryptographique *PKCS#11*, synthétisée ci-dessous.

Exemple 4.1.4 (Attaque par type-punning) *Une autorité centrale fournit des ressources cryptographiques matérielles à ses usagers. Chaque ressource est individualisée et peut effectuer des opérations cryptographiques à la demande de son usager, mais doit préserver la confidentialité de ses clés. Pour permettre des échanges de clés entre ressources, une fonctionnalité d'export de clés noircies (wrappées) est néanmoins proposée. Les opérations cryptographiques offertes par la ressource sont donc les suivantes :*
– *Chiffrement d'un message M par la clé $n^\circ \#K$:* $C \leftarrow \mathrm{cipher}(M, \#K)$.
– *Déchiffrement d'un chiffré C par la clé $n^\circ \#K$:* $M \leftarrow \mathrm{uncipher}(C, \#K)$.
– *Export de la clé $n^\circ \#K$ wrappée par la clé $n^\circ \#W$:* $D \leftarrow \mathrm{export}(\#K, \#W)$
– *Import de la clé wrappée D avec la clé de wrapping $n^\circ \#W$:* $\mathrm{import}(D, \#W)$
Dans une modélisation formelle de cette API dans laquelle les chiffrés et les clés wrappées sont de types différents, il est possible de prouver que les clés de la ressource ne pourront pas être récupérées en clair par l'usager. Malheureusement, les implémentations des chiffrés et des clés wrappées sont indistinguables, et les clés utilisées par la ressource ne sont pas labellisées avec leur rôle. Il est donc possible pour un usager de récupérer en clair une clé en faisant la séquence d'appels (mal typés) suivants :

$$D \leftarrow \mathrm{export}(\#K, \#W); K \leftarrow \mathrm{uncipher}(D, \#W)$$

La preuve de sécurité n'est valable que sous l'hypothèse de respect du typage.

Cela démontre combien il est important d'identifier les hypothèses implicites qui sont associées à l'emploi des types – qu'il s'agisse de détecter les conséquences possibles des violations de typage, ou d'utiliser des mécanismes permettant de préserver de manière intègre les informations de typage dans les implémentations, par exemple en utilisant des labels de sécurité pour prévenir de telles attaques.

D'autres problématiques similaires peuvent être liées à l'utilisation, par exemple, de types énumérés dont la représentation binaire admet des valeurs supplémentaires. Le comportement réel d'une fonction spécifiée comme dépendant d'un paramètre d'un type énuméré vis à vis d'une valeur inattendue (et inenvisageable dans le modèle formel) est en général non spécifié.

4.2 Compréhension de la logique

Au-delà des interprétations erronées pouvant résulter d'une compréhension partielle de la méthode formelle utilisée, certains problèmes plus subtils peuvent se poser, liés de manière plus fondamentale à une compréhension partielle de la logique sous-jacente.

4.2.1 Vacuité

De manière générale, un ensemble ou un type peut être vide, et supposer l'existence d'un élément lui appartenant est donc potentiellement une source d'incohérence.

Exemple 4.2.1 (Type vide en *Coq*) *On définit de manière inductive le type des listes bicolores de naturels (des listes de naturels marqués en rouge ou en bleu) par :*

$$\textbf{Inductive}\ \mathrm{blst} : \textbf{Set} := \mathrm{red} : \mathrm{blst} \to \mathbb{N} \to \mathrm{blst} \mid \mathrm{blue} : \mathrm{blst} \to \mathbb{N} \to \mathrm{blst}$$

Cette définition inductive, bien que valide, n'a cependant pas la signification attendue. En effet, il y manque un constructeur atomique, et une blst *ne peut être obtenue qu'à partir d'une autre* blst *; dès lors le type défini ici (le plus petit point fixe) est vide.*

Supposer l'existence d'un élément de blst *dans le développement est incohérent :*

Variable bl : blst.
Theorem inconsistent : **False**.
Proof.
  induction bl **as** [_ H_I | _ H_I]; **apply** H_I.
Qed.

De manière plus subtile, en l'absence d'une telle hypothèse, la théorie est cohérente – rien n'interdisant de définir un type vide. Il reste néanmoins possible de prouver tout théorème de la forme $\forall\,(b:\text{blst}),\ P$*, et en particulier :*

Theorem blst_empty : $\forall\,(b:\text{blst}),$ **False**.
Proof.
  intros b; **induction** b **as** [_ H_I | _ H_I]; **apply** H_I.
Qed.

Cette situation est beaucoup plus délicate à détecter, puisque le développeur n'a aucune raison de s'étonner de sa capacité à prouver certaines propriétés P *attendues sur l'idée qu'il se fait des* blst*.*

Dès lors, il est prudent lors de la définition d'un type T *que le développeur s'assure de la bonne définition en fournissant une preuve de la forme* **Theorem** $T_{\text{notempty}} : \exists\, t:T,$ **True**.

Exemple 4.2.2 (Type vide en *FoCal*) *Selon le même principe que dans l'exemple précédent, il est possible de définir en* FoCal *une espèce vide, puis une espèce paramétrée par une valeur de cette espèce vide, qui est donc paradoxale :*

type $\text{empty}_t = \text{Empty}$ **in** $\text{empty}_t \to \text{empty}_t$; ;;
species empty_s **inherits** basic_object =
  rep = empty_t;
  theorem is_empty : **all** x **in self**, **basics#False**
    proof : {* ... *};
end
species $\text{absurd}_s(s$ **is** empty_s, e **in** $s)$ **inherits** basic_object =
end

Une problématique similaire peut se poser lors de l'introduction, dans un développement formel, de variables contraintes par un ensemble de propriétés insatisfiable. Une telle incohérence peut apparaître dans une spécification et rester masquée jusqu'à une tentative d'implémentation qui mettra en évidence l'impossibilité d'exhiber un témoin – si une telle tentative d'implémentation est faite, ce qui rappelons le n'est pas toujours le cas.

Exemple 4.2.3 (Variable non instanciable en *B*) *La spécification suivante est (manifestement) incohérente :*

MACHINE absurd_var
VARIABLES v
INVARIANT $v \in \mathbb{N} \wedge v = 0 \wedge\ v = 1$
ASSERTION $0 = 1$

*Dans la pratique cependant, cette machine passe le type-checking et est prouvée automatiquement 'à 100%'. Dans l'*Atelier B*, en effet, la preuve d'existence d'une instanciation valide est différée jusqu'à l'implémentation (sans le signaler) ; si une telle implémentation n'est pas prévue, il convient d'être extrêmement prudent – d'autant que l'incohérence peut être nettement moins évidente que dans l'exemple donné ici*[6]*.*

Ce principe s'applique également pour des constantes ou des paramètres, des variables locales introduites par exemple par une substitution **ANY***, etc. :*

MACHINE absurd_cst
CONSTANTS f
PROPERTIES $f \in \mathbb{N} \to \mathbb{N} \wedge \forall\, x, y,\ x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge x < y \Rightarrow f(x) > f(y)$
ASSERTION $0 = 1$

[6] L'insatisfiabilité d'un ensemble de contraintes, dans le cas général, est un problème indécidable.

Notons que les problèmes présentés dans cette sous-section sont des instanciations de propriétés dont la forme la plus générale est $P \Rightarrow Q$, qui est vrai dès lors que P est faux. Un système dont la spécification est de la forme $P \Rightarrow Q$ peut être prouvé conforme si P est faux ; cela correspond à une réponse acceptable en logique, mais elle est généralement symptomatique d'un problème de spécification car dénuée de sens pratique.

4.2.2 Vivacité

Les exemples précédents basés sur la notion de type ou d'ensemble vide ne correspondent pas à une spécification incohérente, mais plus simplement à une spécification dont la sémantique naturelle n'est pas celle qui est attendue.

Ce type de problématique peut également se poser dans la spécification d'un système supposé vivace (susceptible d'évolution), mais qui dans la pratique peut ne pas l'être. En effet, la spécification peut combiner un certain nombre de contraintes qui, si elles ne sont pas globalement insatisfiables, n'admettent qu'une unique solution – et représentent donc par exemple un état figé.

Exemple 4.2.4 (Problème d'*Adam* et *Eve* dans le *B-Book*) *La* Méthode B *est illustrée dans [3] par le développement complet d'une base de données permettant de gérer des individus. Le développement va de la spécification à l'implémentation, et est prouvable de bout en bout.*

Pour autant, la spécification proposée, si elle n'est manifestement pas contradictoire (puisqu'elle est implémentée), impose des contraintes inadéquates. En effet, l'invariant impose que tout individu dans la base de données ait un père et une mère, alors que l'initialisation de la base de données se fait avec l'ensemble vide. Dès lors, le système est dans un état bloqué : il est impossible d'introduire un individu dans la base de données, puisqu'il est impossible de lui associer un père et une mère [12].

L'implémentation propose donc un codage des opérations qui semble naturel, mais les opérations ne peuvent pas être appelées puisqu'il est impossible de satisfaire à leur pré-condition. En pratique, une implémentation triviale dans laquelle chaque opération ne fait rien serait tout aussi (prouvablement) valide.

On pourra aussi noter, à titre de seconde illustration, l'exemple 3.2.1 de la gestion de fichiers, dans lequel il est possible de mettre le système dans un état bloqué en retirant l'ensemble des droits à tous les utilisateurs pour chaque fichier.

4.2.3 Spécifications quasi-contradictoires

Au-delà de la problématique de l'incohérence ou de la vivacité, il peut arriver que certaines spécifications soient quasi contradictoires.

Une telle spécification comporte des propriétés de la forme $P \Rightarrow Q$ et $P \Rightarrow \neg Q$, ce qui signifie qu'elle devient incohérente dès lors que P est vrai. Ce type de description est révélateur d'un problème de spécification, et ne devrait pas être toléré.

Chapitre 5

Maîtriser l'expressivité

Les méthodologies formelles de développement, en proposant des langages logiques, permettent d'exprimer de nombreuses propriétés. Cependant, dans la pratique, quelle que soit la méthode formelle utilisée certaines propriétés ne seront pas exprimables.

De telles limitations ne sont pas évidentes *a priori*, et peuvent n'être détectées que pendant le développement. Il est parfois possible de contourner le problème en modifiant la modélisation – parfois au prix d'une forte complexification de la représentation des propriétés ou du système – mais dans d'autre cas la limitation doit tout simplement être acceptée.

Dans ce chapitre, nous illustrons quelques exemples de propriétés non (ou difficilement) exprimables, mais aussi les problématiques associées à une mauvaise perception de ces limitations.

5.1 Moyens de contournement

Une limitation de l'expressivité affectant la formalisation d'un développement, une fois identifiée, doit être documentée. Une démarche de contournement, par exemple en complétant le développement formel par des preuves papiers et des vérifications menées avec d'autres outils, doit être proposée.

Exemple 5.1.1 (Invariant d'espèce en *FoCal*) FoCal *permet de décrire une hiérarchie d'espèces qui sont des types abstraits associant des méthodes (programmes) et des propriétés (propositions logiques). On peut par exemple spécifier formellement la structure* $\mathbb{Z}/p\mathbb{Z}$*, où* p *est un paramètre de l'espèce, et décrire des méthodes correspondant à la constante* zero*, la constante* one*, l'addition* add*, etc.*

Une implémentation valable est d'utiliser comme représentation interne de l'espèce le type des entiers machines (pour p *suffisamment petit), et de coder une classe d'équivalence par une valeur entre* 0 *et* $p-1$*. Le codage de l'addition, par exemple, sera de la forme* $\mathrm{add}(x, y) = (x+y)\%p$ *où* $+$ *et* $\%$ *sont les opérations d'addition et de modulo sur les entiers machines.*

Cette implémentation respecte un invariant d'espèce, puisque tout habitant de l'espèce, représenté par un entier machine, est en fait entre 0 *et* $p-1$*.* FoCal *ne permet cependant pas de prouver le respect d'un tel invariant d'espèce ; l'expression d'une telle propriété nécessiterait une forme de quantification sur toutes les méthodes*[1]*. La preuve de préservation d'un tel invariant, si elle est demandée, nécessite donc une argumentation (une preuve papier) en complément des preuves vérifiées par l'outil. Pour minimiser le recours à la preuve sur papier, il est également possible d'ajouter manuellement pour chaque méthode de cette espèce la propriété correspondant à la préservation de cet invariant.*

[1] Notons qu'en *B*, c'est précisément la sémantique de la clause **INVARIANT**. Le générateur d'obligations de preuves produit les propositions à démontrer, en particulier une preuve de préservation pour chaque opération (i.e. par méthode en *FoCal*).

5.2 Propriétés de sécurité non exprimables

5.2.1 Confidentialité

Un bon exemple de spécification difficile ou impossible à exprimer dans de nombreuses méthodes formelles, telles que considérées ici, est celui de la confidentialité d'une information.

Traditionnellement, la confidentialité d'une information est traitée formellement au travers de la définition d'une politique de contrôle d'accès et du développement d'un moniteur dont on démontre qu'il est conforme à sa spécification, i.e. qu'il ne répond à une requête d'accès à une information que si la source de la requête a le droit et le besoin d'en connaître (cf. notamment les discussions dans [13, 14]). Si cette approche apporte certes des éléments de confiance, on peut noter qu'elle n'aborde que rarement la question de l'origine des paramètres utilisés par le moniteur pour prendre sa décision : Comment savoir qui fait une requête ? Comment connaître le niveau de sécurité associé à une information ? Par ailleurs, même sous nombre d'hypothèses favorables relatives notamment au non contournement du moniteur, elle ne semble pas forcément suffisante si on se pose la question de la confidentialité sous l'angle de la théorie de l'information – i.e. des canaux cachés [15, 16, 17] ou encore de la non interférence [18, 19].

Exemple 5.2.1 (Gestionnaire de login en B) *On spécifie le contrôle d'accès d'un système mono-session, basé sur la connaissance d'un login et d'un mot de passe associé. Le type abstrait* USR *décrit l'ensemble de toutes les identités possibles, et* PWD *l'ensemble de tous les mots de passe possibles. La constante* root *décrit l'identité de l'administrateur et* nouser *est une identité spéciale indiquant que personne n'est connecté au système. L'état du système est décrit par les variables* Acc *(qui décrit l'ensemble des identités pour lesquelles un compte existe),* log *(qui donne l'identité actuellement connectée,* nouser *signifiant que personne n'est connecté) et* Pwd *(qui à chaque compte associe un mot de passe).*

La machine comprend des opérations permettant de se connecter ou se déconnecter, de créer ou d'éliminer des comptes, non détaillées ici. Nous nous intéressons ici à l'opération list *qui permet à un utilisateur de connaître l'ensemble des comptes existants*[2] *:*

MACHINE login
SETS USR; PWD
CONSTANTS root, nouser
PROPERTIES root ∈ USR ∧ nouser ∈ USR \ {root}
VARIABLES Acc, log, Pwd
INVARIANT
  Acc ⊆ USR ∧ root ∈ Acc ∧ nouser ∉ Acc ∧ log ∈ Acc ∪ {nouser} ∧ Pwd ∈ Acc → PWD
INITIALISATION
  Acc := {root} ‖ log := nouser ‖ Pwd :∈ {root} → PWD
OPERATIONS
  ...
  out ← list ≙
    IF log ∈ Acc **THEN**
      ANY s **WHERE** $s \in$ **seq**(USR) ∧ **ran**(s) = Acc ∧ **size**(s) = **card**(Acc)
      THEN out := s
    ELSE
      out := ∅
  ...

Les mots de passe sont des données confidentielles, mais cette confidentialité n'est pas spécifiée ici. On peut considérer qu'elle est implicite en l'absence d'opération renvoyant un mot de passe. On peut aussi noter que seule l'opération login *serait spécifiée comme dépendant des mots de passe.*

[2] Le B ne permettant pas de raffiner les entrées et sorties des opérations, la spécification proposée ici doit donc directement décrire les entrées et sorties sous forme de structure implémentables, en l'occurrence les ensembles sont représentés par des listes.

Pourtant, nous pouvons raffiner la spécification de la façon suivante :

REFINEMENT $\text{login}_{\text{Ref}}$ **REFINES** login
SETS USR; PWD
CONSTANTS root, nouser, guess
PROPERTIES $\text{root} \in \text{USR} \land \text{nouser} \in \text{USR} \setminus \{\text{root}\} \land \text{guess} \in \text{PWD}$
VARIABLES Acc, log, Pwd
INVARIANT
$\text{Acc} \subseteq \text{USR} \land \text{root} \in \text{Acc} \land \text{nouser} \notin \text{Acc} \land \text{log} \in \text{Acc} \cup \{\text{nouser}\} \land \text{Pwd} \in \text{Acc} \rightarrow \text{PWD}$
INITIALISATION
$\text{Acc} := \{\text{root}\} \parallel \text{log} := \text{nouser} \parallel \text{Pwd} :\in \{\text{root}\} \rightarrow \text{PWD}$
OPERATIONS
...
$\text{out} \leftarrow \text{list} \triangleq$
IF $\text{log} \in \text{Acc}$ **THEN**
ANY s **WHERE** $s \in \mathbf{seq}(\text{USR}) \land \mathbf{ran}(s) = \text{Acc} \land \mathbf{size}(s) = \mathbf{card}(\text{Acc})$
THEN IF $\text{Pwd}(\text{root}) \leq \text{guess}$ **THEN** $\text{out} := \mathbf{sort}(s)$ **ELSE** $\text{out} := \mathbf{rev}(\mathbf{sort}(s))$
ELSE
$\text{out} := \emptyset$
...

Dans ce raffinement, la liste des comptes retournée par l'opération list *est triée dans l'ordre croissant si le mot de passe de l'administrateur est inférieur (selon un ordre quelconque) à une valeur* guess*, et décroissant sinon. Ici* guess *est une valeur introduite par un développeur malicieux pendant le raffinement, qui peut être modifiée par effet de bord ; l'attaquant pourra donc en quelques essais retrouver* Pwd(root).

Fondamentalement, le problème est qu'il n'est pas vraiment possible de spécifier la propriété recherchée, i.e. la confidentialité de Pwd. *Plusieurs approches permettant de spécifier, plus ou moins directement, une telle propriété sont envisageables, parmi lesquelles :*
– *Pouvoir spécifier que l'apparition de* Pwd *dans le code de* list *n'est pas légale.*
– *Pouvoir spécifier que le résultat de* list *est constant même en substituant à* Pwd *une valeur* Pwd' *quelconque – ce qui nécessite de manipuler des substitutions et propositions de manière explicite.*
– *Pouvoir vérifier que la spécification de* list *est complète pour éliminer le canal caché.*
Certaines de ces approches peuvent nécessiter l'emploi d'autres outils que ceux fournis par le B.

Bien entendu, on peut, dès la spécification, imposer que la liste soit triée ; cependant, le problème mis en évidence ici n'est pas tant l'impossibilité de faire une spécification complète[3] *que celle de spécifier formellement que la spécification doit être complète – ce qui correspondrait à une spécification quantifiant sur tous les raffinements possibles.*

Il reste possible en B *de mieux contrôler les dépendances en spécifiant les opérations à l'aide de fonctions (constantes) abstraites :*

CONSTANTS ..., fct
PROPERTIES $\ldots \land \text{fct} \in \mathbb{P}(\text{USR}) \rightarrow \mathbf{seq}(\text{USR})$
OPERATIONS
$\text{out} \leftarrow \text{list} \triangleq$ **IF** $\text{log} \in \text{Acc}$ **THEN** $\text{out} := \text{fct}(\text{Acc})$ **ELSE** $\text{out} := \emptyset$

Cette spécification modifiée permet de garantir que la valeur retournéee par list *ne dépend pas des variables de l'état autres que* Acc[4]*. Pour autant, il reste possible d'utiliser la variable* Pwd *pour affecter le comportement de* list *:*

$\text{out} \leftarrow \text{read} \triangleq$
$\text{out} := \text{encode}(\text{Pwd}(\text{root}))$;
IF $\text{Pwd}(\text{root}) < \text{guess}$ **THEN** $\text{wait}(10)$ **ELSE** $\text{wait}(20)$;
IF $\text{log} \in \text{Acc}$ **THEN** $\text{out} := \text{fct}(\text{Acc})$ **ELSE** $\text{out} := \emptyset$

[3]Notons de plus que la complétude n'est pas toujours stable par raffinement. Si on travaille avec une méthode formelle permettant de raffiner les types en entrée et en sortie d'une opération, la spécification de *list* consistant à renvoyer l'ensemble *Acc* est complète. Pourtant, son raffinement implémentant la valeur de retour sous la forme d'une liste ne l'est pas, puisqu'à un ensemble donné correspondent généralement plusieurs listes.
[4]Il est cependant difficile d'introduire le même type de restrictions sur les constantes elles-mêmes.

Dans ce raffinement, le développeur malicieux joue à la fois sur un état transitoire du système (état éventuellement observable) et sur un canal temporel.

Il faut donc souvent pour ce type de problématique envisager des méthodes complémentaires, par exemple permettant une forme plus évoluée de typage, ou de l'analyse de dépendance[5].

5.2.2 Propriétés négatives

De manière générale, les méthodologies formelles de développement considérées dans ce document ne permettent pas, ou seulement difficilement, d'exprimer des propriétés 'négatives' telles que la non dépendance (ci-dessus, pour la confidentialité) ou encore la non inversibilité d'une fonction (par exemple en cryptographie).

Exemple 5.2.2 (Spécification d'une fonction de hachage) *Une fonction de hachage est une fonction de $\mathbb{W}$, l'ensemble des mots binaires finis, dans $\mathbb{W}(n)$, l'ensemble des mots binaires de longueur n. Les propriétés attendues d'une fonction de hachage H sont*[6] *:*
- *Résistance à la première pré-image, connaissant h il est difficile de trouver w tel que $H(w)=h$.*
- *Résistance à la seconde pré-image, connaissant w il est difficile de trouver $w' \neq w$ tel que $H(w)=H(w')$.*
- *Résistance aux collisions, il est difficile de trouver $w \neq w'$ tels que $H(w)=H(w')$.*

Ce sont ces propriétés qui justifient l'emploi des fonctions de hachage en tant que primitives cryptographiques dans différentes applications de sécurité.

Dans un système d'exploitation Unix*, les hachés des mot de passe sont stockés dans un fichier accessible en lecture à tous les utilisateurs. L'identification de l'utilisateur U repose sur sa capacité à fournir le mot de passe P_U, qui sera haché et comparé à la valeur stockée H_U. La propriété de sécurité fondamentale recherchée est non pas la confidentialité du mot de passe P_U mais plus généralement l'impossibilité pour un utilisateur U' d'usurper l'identité de U en trouvant un mot de passe P' tel que $H(P') = H_U$ – i.e. la résistance à la première pré-image.*

Pour la signature électronique, on joint à un message M le signé du condensat $[H]_K$[7] ; pour cette application la propriété de résistance à la seconde pré-image est nécessaire à la sécurité, afin de garantir qu'un attaquant ne disposant pas de la clé K (et donc ne pouvant contrefaire $[H]_K$) ne pourra pas fabriquer un message M'. Si c'est l'attaquant qui est à l'origine du message M[8]*, c'est la résistance aux collisions qui est recherchée.*

Ces exemples illustrent que les propriétés des fonctions de hachage sont utilisées pour justifier la sécurité du système – et que dans un développement formel il serait souhaitable que cette justification se traduise par une preuve construite à partir d'une caractérisation formelle d'une fonction de hachage.

Mais cette caractérisation formelle et générique semble pour le moins difficile à définir. Si nous prenons l'exemple de la résistance aux collisions, une caractérisation triviale serait :

$$\forall (x, y : \mathbb{W}),\ x \neq y \Rightarrow H(x) \neq H(y)$$

Cette spécification affirme que H est injective, mais elle est manifestement incohérente, pour une simple question de cardinalité. Il est possible d'éliminer l'incohérence en étant moins précis sur le type de H, i.e. par exemple $H : \mathbb{W} \to \mathbb{W}$ voire $H : \mathbb{W} \to \mathbb{H}$ où $\mathbb{H}$ est un type abstrait – avec dans ce dernier cas un typage artificiel qui peut masquer certaines vulnérabilités (cf. l'exemple 4.1.4). Mais une telle spécification n'aborde pas réellement la problématique sous-jacente, qui nécessite a priori *une modélisation de la connaissance des utilisateurs et des attaquants.*

Le plus souvent, ces sortes de propriétés sont représentées de manière indirecte (et implicite) par une absence d'opération, de méthode ou de fonction correspondante dans la spécification du système,

[5] La question de la dépendance est elle-même complexe. Par exemple, le résultat de la projection $\Pi_1(x, y) = x$ semble ne pas dépendre de y. Pourtant, dans un système avec appels par valeurs (*Call by Value*), la durée d'évaluation de $\Pi_1(x, y)$ dépend de celle de y, et dans le cas extrême si y comporte une boucle infinie ne termine pas.

[6] Ces propriétés sont présentées par difficulté décroissante.

[7] On signe un condensat, ou haché, plutôt que le message pour de simples questions de performances.

[8] Par exemple pour un contrat entre deux parties, le rédacteur du contrat peut chercher à obtenir en fait deux versions M et M' ayant le même haché afin que la signature de M soit aussi une signature valide de M'.

le raisonnement se faisant par clôture. On utilisera par exemple en *Coq* l'absence d'un constructeur dans un type inductif ou en *B* l'absence d'une opération dans une machine.

Exemple 5.2.3 (Clôture des connaissances en *Coq*) *On modélise les propriétés d'une fonction de hachage de la façon suivante :*

Variable User : **Set**.
Variable Info : **Set**.
Variable Hash : Info → Info.
Axiom Hash_No_Coll : ∀ (i_1, i_2 : Info), $i_1 \neq i_2 \rightarrow \text{Hash}(i_1) \neq \text{Hash}(i_2)$.
Inductive knows : User → Info → **Prop** :=
...
| hash : ∀ (u : User)(i : Info), knows $u\ i \rightarrow$ knows u Hash(i)
...

Le prédicat knows *est un prédicat inductif, avec les caractéristiques habituelles de ce type de constructions, notamment la surjectivité des constructeurs (cf. l'exemple 4.1.3). C'est donc l'absence d'un constructeur de la forme* unhash : ∀ (u : User)(i : Info), knows u Hash(i) → knows $u\ i$ *qui indique (implicitement) que* Hash *est non inversible.*

On retrouve ce type d'approche dans l'analyse de protocoles de sécurité ou d'API. Un tel système est représenté par un ensemble d'échanges ou d'interfaces, avec l'hypothèse implicite que la description est complète (il n'y a pas d'autres messages ou mots de commande que ceux décrits). La préservation d'un invariant, par exemple, est alors clairement vérifiée sous hypothèse de complétude de la spécification. Plus encore, la sécurité d'un protocole ou d'une API est souvent étudiée vis à vis d'un attaquant, qui est également représenté par un ensemble de capacités. Ici encore, toute preuve de sécurité correspondante ne sera valable que sous l'hypothèse constituée par ce modèle d'attaquant[9].

[9] Tout attaque sur un système prouvé nécessite une remise en cause du modèle du système ou du modèle de l'attaquant, ce qui constitue en pratique une bonne heuristique de recherche de vulnérabilités sur un système prouvé.

Chapitre 6

Maîtriser l'outil formel

Nous nous intéressons dans ce chapitre aux outils formels, et aux problématiques qui peuvent se poser lors de leur utilisation en raison d'une méconnaissance par l'utilisateur des choix relatifs à l'implémentation qui ont été fait, et qui tout en étant *a priori* valides, peuvent avoir des conséquences sur la validité des développements.

6.1 À propos de l'*Atelier B* et de *B4Free*

Un point déjà exposé précédemment, notamment dans l'exemple 4.2.3, est que la *Méthode B* vise normalement à permettre des développements complets allant de la spécification à l'implémentation.

À ce titre, certaines obligations de preuves sont différées – notamment celles relatives à l'existence d'instances pour les paramètres, constantes et variables satisfaisant les contraintes exprimées. Cette approche se justifie dans la mesure où l'implémentation constitue une preuve constructive de l'existence de telles valeurs, en exhibant un témoin ; il est donc inutile d'imposer trop tôt une preuve.

L'*Atelier B* optimise également la recherche de preuve en exploitant la notion de *delta-lemmes*. De nombreuses définitions de base en *B* sont des définitions conditionnelles (par exemple la division n'est définie que si le diviseur n'est pas 0). Afin d'éviter une complexification essentiellement administrative des preuves, les conditions de validité des définitions sont supposées vérifiées dans les preuves, et sont reportées sous la forme de *delta-lemmes* qui devront être prouvés ultérieurement.

Dans les deux cas, les vérifications différées ne sont pas identifiées explicitement par l'outil, ni même comptabilisées dans les métriques indiquant le taux d'avancement. Il convient donc de considérer avec beaucoup de précautions les indications de couverture de preuve, puisqu'un développement 'prouvé à 100%' ne l'est généralement pas.

6.2 À propos de *Coq*

Le système *Coq* dispose d'un mécanisme considéré comme robuste et fiable pour la vérification de preuves. Il faut noter que cette vérification n'est menée qu'à la conclusion de la preuve, à l'utilisation de la commande **Qed**[1]. Il est possible, dans certains cas extrêmes, d'introduire une preuve invalide qui apparemment permettra de résoudre un but mais sera rejetée lors de cette étape de vérification.

Par ailleurs, le mot clé **Admitted** permet de décharger instantanément une obligation de preuve.

[1] Ou, alternativement, **Save** ou **Defined**.

Un développement *Coq a priori* totalement prouvé doit donc néanmoins faire l'objet de quelques vérifications complémentaires.

6.3 À propos de *FoCal*

Dans la version de l'outil *FoCal* actuellement disponible (*v0.4.0*), plusieurs simplifications importantes sont faites, qui doivent être identifiées. En particulier, la terminaison des fonctions récursives n'est pas prouvée, les preuves associées exigées par *Coq*[2] étant déchargées par l'utilisation d'un axiome[3].

Par ailleurs, il est toujours possible dans *FoCal* de décharger une obligation de preuve en utilisant le mot clé **assumed**.

Une code *FoCal* dont la preuve est certifiée par *Coq* reste donc potentiellement incorrect, et des vérifications complémentaires restent nécessaires.

[2] *Coq* est utilisé comme outil de vérification des preuves fournies dans le source *FoCal* ou générées par *Zenon*.
[3] En l'occurrence l'axiome *Idontwanttoproveit* qui est paradoxal.

Chapitre 7

Vérifier les outils

Nous appelons 'outils formels' les implémentations des méthodes formelles sous la forme d'ateliers de développement, de type-checkers, de compilateurs, de prouveurs, etc. Nous nous intéressons dans ce chapitre à la validité de ces outils, i.e. à la fidélité de la représentation qu'ils donnent de la méthode formelle. En effet, l'implémentation peut être biaisée, en raison de simplifications (jugées nécessaires par les développeurs pour des questions d'automatisation, de performance) ou d'erreurs lors du développement des outils.

Les erreurs d'implémentation peuvent se situer à de nombreux endroits dans un outil formel : type-checker, compilateur, générateur d'obligations de preuve, prouveur. De telles erreurs peuvent se traduire par une perte de la complétude (pour un prouveur cela signifie que certains résultats prouvables dans la théorie ne le sont pas avec l'outil) ou de la correction (pour un prouveur cela signifie qu'on peut prouver des résultats non prouvables dans la théorie, avec un risque d'incohérence).

Un outil incomplet peut potentiellement poser des problèmes au développeur, qui n'arrivera pas à prouver les résultats attendus. Mais l'utilisation d'un outil incorrect peut avoir des conséquences beaucoup plus graves, en particulier si des problématiques de sécurité sont posées. Par exemple, la connaissance par un développeur malicieux d'un paradoxe accepté par un prouveur peut lui permettre de prouver ce qu'il veut, i.e. piéger un produit tout en obtenant une certification formelle.

Il faut donc se poser la question de la confiance que l'on peut avoir dans les outils utilisés pour un développement. Idéalement, un outil formel pourrait lui-même faire l'objet d'un développement formel, mais cela ne va pas sans poser un problème d'œuf et de poule. Tout au plus peut-on utiliser un outil O_A, implémentant une méthode M_A et reputé de confiance, pour implémenter un outil O_B prouvé correct vis-à-vis de la méthode M_B.

Ici encore, remettre en cause la confiance dans les outils formels peut sembler exagérément pessimiste, d'autant que la vérification de la correction des outils est une tâche lourde et difficile. Pourtant, quelques erreurs dans des outils éprouvés ont été identifiées[1]. Il reste donc toujours intéressant, au minimum, de vérifier si des travaux de validation ou de développements formels des outils formels ont été menés (cf. par exemple [20, 21, 22, 23] pour le *B*, ou encore [24]).

7.1 À propos de *B*

Exemple 7.1.1 (Prouveur de l'*Atelier B*) *L'*Atelier B *est une implémentation industrielle de la* Méthode B *qui a été utilisée avec succès dans de nombreux projets industriels et académiques, depuis plusieurs années ;* B4Free *en est une version gratuite.*

[1] Mais ne sont pas toujours documentées ou connues de la communauté des utilisateurs.

*Le prouveur de l'*Atelier B *n'implémente pas exactement la logique du* B-Book *mais une version simplifiée. Les simplifications sont réalisées à des fins d'optimisation,* a priori *sans conséquences graves (i.e. sans enrichissement de la logique). Ainsi, la sémantique des ensembles en compréhension définie dans [3] par :*

$$E \in \{x \mid x \in S \wedge P\} \Leftrightarrow E \in S \wedge [x := E]P \text{ si } x \backslash S$$

est (apparemment) simplifiée dans le prouveur pour être remplacée par :

$$E \in \{x \mid P'\} \Leftrightarrow [x := E]P'.$$

Cette simplification est faite sous hypothèse de type-checking, qui impose que tout ensemble en compréhension soit de la forme $\{x \mid x \in S \wedge P\}$ *avec* $x \backslash S$*. Sous cette hypothèse ces deux sémantiques sont effectivement équivalentes.*

*Malheureusement, par défaut, le type-checking n'est effectué dans l'*Atelier B *ou dans* B4Free *que lors de la vérification d'une machine, et pas dans le prouveur*[2]*.*

Cela signifie que si les termes issus du code source d'un développement sont bien type-checkés, il reste possible d'introduire en cours de preuve des termes supplémentaires qui ne le sont pas. Pour cela, il suffit par exemple de faire une coupure, i.e. l'introduction d'un lemme prouvable.

En combinant la simplification de la sémantique des ensembles en compréhension et l'absence de type-checking pendant la preuve, il devient possible d'introduire le paradoxe de Russel *dans la logique, et donc de prouver n'importe quel résultat, de la façon suivante :*

ah$(\{x \mid x \notin x\} \in \{x \mid x \notin x\} \Rightarrow \{x \mid x \notin x\} \notin \{x \mid x \notin x\})$
rn
ah$(\{x \mid x \notin x\} \notin \{x \mid x \notin x\} \Rightarrow \{x \mid x \notin x\} \in \{x \mid x \notin x\})$
rn
ax$(\{x \mid x \notin x\})$
dc$(xx \in xx)$

*Cette preuve consiste simplement à ajouter à l'environnement (***ah***) les hypothèses paradoxales* $\mathcal{R} \in \mathcal{R} \Rightarrow \mathcal{R} \notin \mathcal{R}$ *et* $\mathcal{R} \notin \mathcal{R} \Rightarrow \mathcal{R} \in \mathcal{R}$*, avec* $\mathcal{R}$ *l'ensemble non type-checkable* $\{x \mid x \notin x\}$*. Ces hypothèses doivent bien entendu être prouvées, ce qui est ici immédiat (***rn***). Le paradoxe peut ensuite être mis en évidence au profit du prouveur automatique en faisant une preuve par cas (***dc***) sur* $\mathcal{R} \in \mathcal{R} \vee \mathcal{R} \notin \mathcal{R}$ *pour décharger instantanément n'importe quelle obligation de preuve.*

Il est donc important de vérifier que la vérification du type des termes introduits en cours de preuve est activée, i.e. que le fichier de configuration du prouveur intègre l'option suivante[3] *:*

ATB∗PR∗Enable_TC_Command : **True**

7.2 À propos de *Coq*

Aucune erreur d'implémentation n'est connue dans l'outil *Coq*[4], qui est réputé de confiance.

7.3 À propos de *FoCal*

L'outil *FoCal* est récent et n'a pas fait l'objet d'une analyse poussée permettant de mettre en évidence d'éventuelles erreurs d'implémentation. Il comporte néanmoins actuellement des simplifications permettant de mettre en jeu des paradoxes.

[2] Le type-checking du *B* a la particularité de ne pas être stable pendant la preuve. En partant d'un ensemble de termes type-checkés et en faisant une preuve valide, il est possible de passer par des termes intermédiaires qui ne sont pas type-checkables. Faire un type-check systématique dans le prouveur pourrait poser nuire à la complétude.

[3] Configuration fournie par la société éditrice de l'*Atelier B* en réponse aux questions posées par la *DCSSI* à propos de l'existence de ce paradoxe dans l'installation par défaut.

[4] Pour être précis, dans le mécanisme de typage utilisé pour valider les preuves.

Chapitre 8

Valider la théorie

Nous nous intéressons ici à la problématique de la validité de la méthode formelle vue comme un objet théorique constitué d'une logique, d'un langage de programmation et de règles de vérification.

8.1 Cohérence de la logique

La problématique de la cohérence de la théorie est relative à une forme de correction de la syntaxe et de la sémantique d'une logique. L'un des problèmes possibles est l'existence d'un paradoxe qui permet de prouver et de réfuter une proposition donnée. Si un tel paradoxe existe, il conduit à une forme d'effondrement, puisqu'il permet de prouver toute proposition.

Cette question devrait être abordée lors de l'évaluation, mais elle est difficile, l'étude de la cohérence d'une logique non triviale ne pouvant être réalisée que dans une méta-logique (cf. les résultats de *Gödel*). De nombreux travaux académiques existent à ce sujet (cf. par exemple [25, 26]).

La solution pratique, dans le cadre du développement d'un produit de sécurité, consiste à se reposer sur les travaux académiques qui ont pu être réalisés à ce sujet, ou tout au moins à admettre la cohérence de la logique d'une méthode formelle dès lors que celle-ci est reconnue par la communauté scientifique et réputée de confiance. Cette approche n'est cependant possible que pour des méthodes éprouvées ; l'utilisation d'une méthode formelle moins connue ou moins ancienne nécessite de se poser la question de la cohérence.

Exemple 8.1.1 (Logique du *B-Book*) *Le* B-Book *décrit une logique des prédicats du premier ordre, avec des constructions ensemblistes, en détaillant une syntaxe et des règles d'inférence.*

Un mécanisme complémentaire, dit type-checking*, est aussi présenté. Sa vocation est de d'interdire la manipulation de termes logiques pourtant autorisés par la syntaxe, tels que* $\mathcal{R} = \{x \mid x \notin x\}$*. En effet, le simple fait de pouvoir écrire ce terme et de le manipuler avec les règles d'inférence permet de prouver* $\mathcal{R} \in \mathcal{R} \Leftrightarrow \mathcal{R} \notin \mathcal{R}$*, le paradoxe de* Russel.

Muni du type-checking, la logique du B-Book *est réputée cohérente. Bien qu'aucune étude académique sur le sujet n'existe (à notre connaissance), aucun paradoxe n'a été mis en évidence par la communauté.*

8.2 Correction des résultats logiques

Au-delà de la cohérence de l'association de la syntaxe et des règles fondamentales de déduction, la théorie logique associée à la méthode formelle peut présenter d'autres problèmes.

Par exemple, un certain nombre de résultats peuvent être présentés comme des théorèmes découlant

de ces règles primitives – l'ensemble de ces théorèmes formant une bibliothèque de méthodes de preuve. Ces résultats peuvent être utiles pour proposer des règles de raisonnement plus efficaces et être promus au statut de principes fondamentaux utilisés par exemple pour mener des preuves. Il convient de vérifier la preuve de ces résultats, qui ne doivent en aucun cas être admis.

En effet, l'introduction d'un principe de raisonnement présenté comme un théorème de la logique, mais en réalité non prouvable, se traduit par l'enrichissement de la logique avec de nouvelles règles de déduction, au risque de la rendre incohérente.

Exemple 8.2.1 (Résultats improuvables du *B-Book*) *Le* B-Book*, après l'introduction de la syntaxe et des règles de déduction, propose de nombreux théorèmes (des propositions réputées prouvables) plus intéressants à l'emploi. Trois de ces théorèmes se révèlent cependant impossibles à démontrer [22] :*

$$E_1 \mapsto F_1 = E_2 \mapsto F_2 \Rightarrow E_1 = E_2$$
$$E_1 \mapsto F_1 = E_2 \mapsto F_2 \Rightarrow F_1 = F_2$$
$$S_1 \subseteq S_2 \wedge T_1 \subseteq T_2 \Rightarrow S_1 \times T_1 \subseteq S_2 \times T_2$$

En effet, les règles de déduction proposées sont insuffisantes pour décrire certaines propriétés logiques attendues des produits cartésiens ; en particulier, rien n'impose dans [3] que les produits cartésiens ne contiennent que des paires (surjectivité), ou que deux paires ne peuvent être égales que si leurs constituants le sont (injectivité).

Admettre ces résultats ne semble cependant pas remettre en cause la cohérence de la logique (ils ne semblent pas non plus réfutables dans la logique de B*), mais ici encore aucune étude académique sur le sujet n'existe.*

8.3 Cohérence des sémantiques

Une méthode formelle permet de manipuler, selon des règles mathématiques précises et explicites, des entités abstraites qui représentent parfois certains objets ayant une réalité physique. De fait, la sémantique donnée à une entité abstraite doit être cohérente avec la sémantique de l'objet concret qui lui est associé. Ce problème peut se poser lors d'un développement, lorsque des choix de modélisation de la réalité sont faits, mais aussi de manière plus fondamentale dans la théorie d'une méthode formelle.

Par exemple, certaines méthodes formelles proposent un langage de programmation. Les programmes écrits dans ce langage sont des entités abstraites susceptibles d'analyses (par exemple pour vérifier la conformité à une spécification) et ont donc une sémantique logique. Mais ce sont également des objets concrets qui ont vocation à s'exécuter sur un ordinateur, et ont une sémantique opérationnelle. La cohérence de ces deux sémantiques devrait également être vérifiée.

Exemple 8.3.1 (Boucle WHILE du *B-Book*) *Le* B-Book *propose le* GSL *(*Generalized Substitution Language*) comme langage de spécification et de programmation ; un sous-ensemble de ce langage permet l'écriture de programmes impératifs.*

La sémantique logique d'une substitution écrite en GSL *est une sémantique par transformation de prédicats : une substitution peut être appliquée à un prédicat pour engendrer un nouveau prédicat. Par exemple, si I est un prédicat dépendant de l'état d'un système, et décrivant un invariant (supposé) de ce système, et S est une substitution pouvant modifier l'état du système, la préservation de l'invariant I par exécution de S se traduit pas $I \Rightarrow [S]I$ où $[S]I$ est le prédicat I transformé par l'application de S.*

La sémantique opérationnelle d'une substitution est moins formalisée dans [3] ; nous considérons donc ici que la sémantique opérationnelle d'une substitution écrite en B0*, i.e. dans le sous-ensemble implémentable du* GSL*, est donnée par sa traduction vers le langage* C*.*

L'une des substitutions identifiées dans le B-Book *est la substitution* **WHILE** *dont la sémantique logique*

est donnée par :

$$\left[\begin{array}{l}\textbf{WHILE}\ P\\ \textbf{DO}\ S\\ \textbf{INVARIANT}\ I\\ \textbf{VARIANT}\ V\\ \textbf{END}\end{array}\right] R \Leftrightarrow \begin{array}{lll} & I\\ \wedge & \forall\, x \cdot (I \wedge P \Rightarrow [S]I)\\ \wedge & \forall\, x \cdot (I \Rightarrow V \in \mathbb{N})\\ \wedge & \forall\, x \cdot (I \wedge P \Rightarrow [n := V][S](V < n))\\ \wedge & \forall\, x \cdot (I \wedge \neg P \Rightarrow R)\end{array}$$

où R est un prédicat quelconque, P la condition de boucle, I l'invariant de boucle et V le variant de boucle (une valeur naturelle qui doit décroître à chaque itération, garantissant la terminaison).

La sémantique opérationnelle en B *est trivialement :*

$$\left[\!\left[\begin{array}{l}\textbf{WHILE}\ P\\ \textbf{DO}\ S\\ \textbf{INVARIANT}\ I\\ \textbf{VARIANT}\ V\\ \textbf{END}\end{array}\right]\!\right] = \textbf{while}\ [\![P]\!]\{[\![S]\!]\}$$

où $[\![]\!]$ est l'opérateur d'interprétation permettant de traduire une substitution opérationnelle en programme, et un prédicat en un booléen.

Le variant V et l'invariant I ont donc un rôle logique – garantir la terminaison de la boucle et expliciter un invariant facilitant la preuve des propriétés – mais aucun rôle opérationnel puisqu'ils sont purement et simplement omis. Ils n'ont effectivement aucun intérêt dans un langage impératif classique.

Un exemple d'utilisation de la boucle **WHILE** *est donné dans [3]. Sous l'hypothèse de non vacuité de S, un sous-ensemble de $\mathbb{N}$, cette substitution permet de calculer le minimum m de S – i.e. qu'elle réalise le prédicat $m = \mathbf{min}(S)$:*

$$\left[\begin{array}{ll} & \textbf{WHILE}\ m \notin S\\ & \quad\textbf{DO}\ m := m+1\\ m := 0; & \quad\textbf{INVARIANT}\ m \in [0, \mathbf{min}(S)]\\ & \quad\textbf{VARIANT}\ \mathbf{min}(S) - m\\ & \textbf{END}\end{array}\right] (m = \mathbf{min}(S)) \Leftrightarrow$$

$$\left(\begin{array}{lll} & 0 \in [0, \mathbf{min}(S)]\\ \wedge & \forall\, m,\ m \in [0, \mathbf{min}(S)] \wedge m \notin S \Rightarrow [m := m+1](m \in [0, \mathbf{min}(S)])\\ \wedge & \forall\, m,\ m \in [0, \mathbf{min}(S)] \Rightarrow \mathbf{min}(S) - m \in \mathbb{N}\\ \wedge & \forall\, m,\ m \in [0, \mathbf{min}(S)] \wedge m \notin S \Rightarrow (\mathbf{min}(S) - m - 1) < (\mathbf{min}(S) - m)\\ \wedge & \forall\, m,\ m \in [0, \mathbf{min}(S)] \wedge m \in S \Rightarrow m = \mathbf{min}(S)\end{array}\right)$$

Si nous modifions légèrement la substitution en remplaçant l'invariant $m \in [0, \mathbf{min}(S)]$ par $m \in \mathbb{N}$, certes plus faible, mais valide, nous ne modifions pas la sémantique opérationnelle – le programme reste identique et permet toujours, de manière évidente, d'extraire le minimum d'un ensemble.

Sur le plan logique, cette modification ne devrait avoir pour conséquence, au pire, que d'empêcher de prouver que le prédicat $m = \mathbf{min}(S)$ est réalisé. Pourtant, en pratique, cette modification a une conséquence beaucoup plus importante, puisqu'elle permet de réfuter la réalisation de ce prédicat, i.e. d'affirmer que le programme ne permet pas toujours d'extraire le minimum !

En effet, les deux propositions suivantes sont réfutables :

$$\forall\, m,\ m \in \mathbb{N} \Rightarrow \mathbf{min}(S) - m \in \mathbb{N}$$
$$\forall\, m,\ m \in \mathbb{N} \wedge m \in S \Rightarrow m = \mathbf{min}(S)$$

On déduit de cet exemple qu'il y a, dans certains cas, une incohérence entre la sémantique logique et la sémantique opérationnelle d'une substitution en B[1].

[1] Cette incohérence peut ici être considérée comme non fondamentale, ne permettant apparemment pas d'exploitation malicieuse. De plus elle peut aisément être éliminée par une correction de la sémantique logique de la substitution **WHILE** [12].

Exemple 8.3.2 (Compilation de *FoCal*) *Dans le cas de* FoCal*, les sémantiques du langage de programmation (du type* ML*) sont données indirectement par le processus de compilation, d'une part vers* Coq*, d'autre part vers* OCaml*. Cette compilation est configurable au travers des commandes* **caml_link** *et* **coq_link**, *comme dans l'exemple suivant :*

type int = **caml_link** int ; **coq_link** $\mathbb{Z}$; ;

Cet exemple illustre les incohérences possibles entre la sémantique logique utilisée par Coq *(ici la modélisation par* $\mathbb{Z}$*, non bornée) et la sémantique opérationnelle projetée dans* OCaml *(ici la modélisation par* int*, bornée). Cette incohérence induit le risque que certains dysfonctionnements (par exemple ici un débordement) ne seront pas détectés lors de la preuve.*

Exemple 8.3.3 (Types inductifs en *FoCal*, *Coq* et *OCaml*) *Dans les systèmes* FoCal, Coq *et* OCaml *il est possible de définir des types inductifs. Les différentes sémantiques associées à une définition ne sont pourtant pas toujours cohérentes. Considérons par exemple les définitions suivantes dans ces trois langages :*

type strange_lst = Cons **in** $\mathbb{N} \rightarrow$ strange_lst $\rightarrow$ strange_lst; ;
Inductive strange_lst : **Set** = cons : $\mathbb{N} \rightarrow$ strange_lst $\rightarrow$ strange_lst.
type strange_lst = Cons **of** ($\mathbb{N} *$ strange_lst); ;

Ces définitions sont réputées équivalentes – la première définition en FoCal *étant par exemple compilée vers les deux autres en* Coq *et* OCaml*. Pourtant, comme illustré précédemment, la définition en* Coq *correspond à un type vide, mais la définition en* OCaml *admet la valeur suivante :*

let rec sl1 = Cons(1, sl1); ;

Dans la pratique, cette incohérence ne se traduit pas automatiquement par un paradoxe. En effet, la valeur OCaml *exhibée ici ne peut être introduite dans* Coq*. Par contre, elle justifie de prendre des précautions sur les développements.*

Par exemple, il est possible de développer en Coq *de nombreuses fonctions opérant sur les* strange_lst *(obtention du premier élément, tri), et de prouver toute propriété désirable, sans avoir jamais besoin d'exhiber une valeur de ce type. Ce développement, via le mécanisme d'extraction, peut aboutir à la fourniture d'une bibliothèque de fonctions en* Ocaml *formellement prouvée. Pourtant, tout emploi opérationnel de cette bibliothèque pourra se révéler catastrophique.*

Bibliographie

[1] DCSSI : Direction centrale de la sécurité des systèmes d'information (www.ssi.gouv.fr).

[2] ISO/IEC 15408 : Common criteria for information technology security evaluation (www.commoncriteriaportal.org).

[3] Abrial, J.R. : The B-Book - Assigning Programs to Meanings. Cambridge University Press (August 1996)

[4] The Coq development team : The Coq proof assistant reference manual. LogiCal Project. (2004)

[5] The FoCal development team : The FoCal project (focal.inria.fr).

[6] IEC 61508 : Functional safety of electrical, electronic, programmable electronic safety-related systems (www.iec.ch/zone/fsafety).

[7] Schönegge, A. : Proof obligations for monomorphicity

[8] Clark, J.A., Stepney, S., Chivers, H. : Breaking the model : Finalisation and a taxonomy of security attacks

[9] Andronick, J., Chetali, B., Paulin-Mohring, C. : Formal verification of security properties of smart card embedded source code. In Fitzgerald, J., Hayes, I.J., Tarlecki, A., eds. : FM. Volume 3582 of Lecture Notes in Computer Science., Springer (2005) 302–317

[10] Boulmé, S., Potet, M.L. : Interpreting invariant composition in the B Method using the Spec# ownership relation : A way to explain and relax B restrictions. [27] 4–18

[11] Clulow, J. : On the security of PKCS#11. In Walter, C.D., Çetin Kaya Koç, Paar, C., eds. : CHES. Volume 2779 of Lecture Notes in Computer Science., Springer (2003) 411–425

[12] Mussat, L. Conversation privée (2005)

[13] Haddad, A. : Meca : A tool for access control models. [27] 281–284

[14] Benaïssa, N., Cansell, D., Méry, D. : Integration of security policy into system modeling. [27] 232–247

[15] Lampson, B.W. : A note on the confinement problem. Commun. ACM **16**(10) (1973) 613–615

[16] He, J., Gligor, V.D. : Information-flow analysis for covert-channel identification in multilevel secure operating systems. In : CSFW. (1990) 139–149

[17] Trostle, J.T. : Multiple trojan horse systems and covert channel analysis. In : CSFW. (1991) 22–33

[18] Goguen, J., Meseguer, J. : Security policies and security models. In : IEEE Symposium on Security and Privacy, IEEE Press (1992) 11–20

[19] Abadi, M., Banerjee, A., Heintze, N., Riecke, J.G. : A core calculus of dependency. In ACM, ed. : POPL '99. Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX, New York, NY, USA, ACM Press (1999) 147–160

[20] Cirstea, H., Kirchner, C. : Using rewriting and strategies for describing the B predicate prover. In Kirchner, C., Kirchner, H., eds. : CADE-15 : Workshop on Strategies in automated deduction. Volume 1421 of Lecture Notes in Computer Science., Lindau, Germany, Springer (1998) 25–36

[21] Berkani, K., Dubois, C., Faivre, A., Falampin, J. : Validation des règles de base de l'Atelier B. Technique et Science Informatiques **23**(7) (2004) 855–878

[22] Jaeger, É., Dubois, C. : Why would you trust B ? In Dershowitz, N., Voronkov, A., eds. : LPAR. Volume 4790 of Lecture Notes in Computer Science., Springer (2007) 288–302

[23] Colin, S., Petit, D., Rocheteau, J., Marcano, R., Mariano, G., Poirriez, V. : BRILLANT : An open source and XML-based platform for rigourous software development. In : SEFM (Software Engineering and Formal Methods), Koblenz, Germany, AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press (september 2005) selectivity : 40/120.

[24] Ridge, T., Margetson, J. : A mechanically verified, sound and complete theorem prover for first order logic. In Hurd, J., Melham, T.F., eds. : TPHOLs. Volume 3603 of Lecture Notes in Computer Science., Springer (2005) 294–309

[25] M.J.C. Gordon : Mechanizing programming logics in higher-order logic. In G.M. Birtwistle, P.A. Subrahmanyam, eds. : Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification), Banff, Canada, Springer-Verlag, Berlin (1988) 387–439

[26] Aydemir, B., Charguéraud, A., Pierce, B.C., Weirich, S. : Engineering aspects of formal metatheory (April 2007) Manuscript.

[27] Julliand, J., Kouchnarenko, O., eds. : B 2007 : Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings. In Julliand, J., Kouchnarenko, O., eds. : B. Volume 4355 of Lecture Notes in Computer Science., Springer (2006)

[28] Bertot, Y., Castéran, P. : Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag (2004)

[29] Boulmé, S. : Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel. Thèse de doctorat, Université Paris 6 (2000)

[30] Prevosto, V. : Conception et Implantation du langage FoC pour le développement de logiciels certifiés. Thèse de doctorat, Université Paris 6 (September 2003)

[31] Boulmé, S., Hardin, T., Rioboo, R. : Polymorphic data types, objects, modules and functors : is it too much ? Research report, LIP6 (2000) Available on http ://www.lip6.fr/reports/lip6.2000.014.html.

Annexe A

Rappels sur *B*, *Coq* et *FoCal*

A.1 Rappels sur *B*

La *Méthode B* décrite dans le *B-Book* [3] et repose sur :
– Une logique des prédicats du premier ordre, complétée par des notions ensemblistes.
– Un langage de substitutions généralisées (le *GSL*) pour décrire ou implémenter des programmes.
– Une méthodologie pour aller d'une spécification formelle à une implémentation prouvée conforme.
Un développement B est structuré en machines. Une machine B est composée d'un état (représenté par des variables), d'opérations (representées par des substitutions) permettant de lire ou modifier l'état, et d'un invariant (une propriété logique) qui doit être préservé par les opérations.

La logique permet d'exprimer les propriétés d'une machine et de conduire les preuves exigées. Le langage associé à cette logique comporte de manière native de nombreuses notions ensemblistes.

Exemple A.1.1 (Quelques définitions ensemblistes natives)

$\mathbb{N}$	*Ensemble des nombres naturels*
$\mathbb{Z}$	*Ensemble des nombres relatifs*
INT	*Ensemble des entiers machines (signés sur 32 bits)*
$x \mapsto y$	*Couple de première projection x et de seconde projection y*
$D \times I$	*Produit cartésien de D par I*
$D \rightarrow I$	*Ensemble des fonctions totales de domaine D et d'image I*
$\mathbb{P}(S)$	*Ensemble des sous-ensembles de S*
$\mathbb{P}_1(S)$	*Ensemble des sous-ensembles non vides de S*
$\mathbb{F}(S)$	*Ensemble des sous-ensembles finis de S*
$\mathbb{F}_1(S)$	*Ensemble des sous-ensembles finis non vides de S*
$D \leftrightarrow I$	*Ensemble des relations de domaine D et d'image I*
$D \lhd R$	*Relation R limitée aux couples dont la première projection est dans D*
$D \lhd\!\!\!\cdot\ R$	*Relation R sans les couples dont la première projection est dans D*
$R \rhd I$	*Relation R limitée aux couples dont la seconde projection est dans I*
$R \rhd\!\!\!\cdot\ I$	*Relation R sans les couples dont la seconde projection est dans I*
$R_1 <\!\!+ R_2$	*Relation R_1 surchargée par R_2, i.e.* $(\mathbf{dom}(R_2) \lhd\!\!\!\cdot\ R_1) \cup R_2$

Le *GSL* quant à lui permet de décrire des substitutions qui peuvent être abstraites, déclaratives et non déterministes (lorsqu'il s'agit de spécifier une opération) ou au contraire concrètes, impératives et déterministes (losqu'il s'agit d'implémenter une opération).

Exemple A.1.2 (Substitution non déterministe) *$x :\in S$ donne à x une valeur de S.*

Exemple A.1.3 (Substitution parallèle) *$S_1 \parallel S_2$ est la composition parallèle des substitutions S_1 et S_2. Par exemple $x := y \parallel y := x$ swappe les valeurs des variables x et y.*

Exemple A.1.4 (Racine carrée d'une valeur entière) **ANY** x **WHERE** P *donne à* x *une valeur satisfaisant la propriété* P*, sans indiquer comment trouver cette valeur. Par exemple la racine carrée est décrite par* **ANY** x **WHERE** $x*x \leq n < (x+1)*(x+1)$ **THEN** $\sqrt{(n)} := x$ **END**.

En ce qui concerne la méthodologie, le B définit la relation de raffinement entre machines. Une machine concrète M_C raffine une machine abstraite M_A si tout comportement légal (respectant les pré-conditions) et observable de M_C est un comportement possible de M_A. Le résultat fondamental est qu'une implémentation est correcte vis à vis de sa spécification si elle raffine cette spécification[1]. La définition du raffinement reposant sur une notion d'observabilité, elle est indépendante de la représentation interne de l'état d'un système.

Exemple A.1.5 (Maximier) *Le* Maximier *est spécifié par un état* $S \subseteq \mathbb{N}$*, une opération* store *qui permet d'ajouter une valeur dans l'état, et une opération* get *qui retourne le maximum des valeurs stockées :*

MACHINE $\text{Maximier}_{\text{Spec}}$
VARIABLES S
INVARIANT $S \subseteq \mathbb{N}$
INITIALISATION $S := \emptyset$
OPERATIONS
  $\text{store}(n) \triangleq$ **PRE** $n \in \mathbb{N}$ **THEN** $S := S \cup \{n\}$ **END**
  $\text{out} \leftarrow \text{get} \triangleq$ **PRE** $S \neq \emptyset$ **THEN** $m := \mathbf{max}(S)$ **END**
END

Le raffinement de Maximier *modifie la représentation interne, puisque l'ensemble* S *est remplacé par une unique valeur* s *:*

REFINEMENT $\text{Maximier}_{\text{Ref}}$ **REFINES** $\text{Maximier}_{\text{Spec}}$
VARIABLES s
INVARIANT $s = \mathbf{max}(S \cup \{0\})$
INITIALISATION $s := 0$
OPERATIONS
  $\text{store}(n) \triangleq$ **IF** $s < n$ **THEN** $s := n$ **END**
  $\text{out} \leftarrow \text{get} \triangleq$ **BEGIN** $m := s$ **END**
END

Ce raffinement, malgré le changement de représentation interne, est tout à fait valide, puisque aucune séquence d'appels aux opérations proposées ne permet de distinguer $\text{Maximier}_{\text{Ref}}$ *de* $\text{Maximier}_{\text{Spec}}$.

Le raffinement a également la propriété d'être transitif, ce qui permet d'aller de la spécification à l'implémentation en un nombre arbitraire d'étapes. À chacune de ces étapes, la méthodologie décrit les obligations de preuves qui doivent être déchargées pour s'assurer de la validité du raffinement. Le dernier raffinement, appelé implémentation, n'utilise qu'un sous-ensemble du *GSL*, le langage *B0*, qui ne comporte que des substitutions implémentables sous la forme d'instructions d'un langage impératif standard (tel que le *C* ou l'*ADA*).

A.2 Rappels sur *Coq*

Le système *Coq* [4, 28] est un assistant de preuve basée sur la théorie des types. Il propose un environnement avec une logique d'ordre supérieur pour la construction et la vérification de preuves, ainsi que le développement et l'analyse de programmes fonctionnels codés dans un langage interne de type *ML*, avec *pattern-matching*.

En *Coq*, tout terme a un type, et tout type est un terme. La hiérarchie des types distingue les termes (par exemple 0), les types simples (par exemple $\mathbb{N}$), la sorte **Set** qui correspond au type des

[1] Si l'implémentation raffine la spécification, cela signifie effectivement que tout comportement de l'implémentation est un comportement acceptable du point de vue de la spécification.

types simples, la sorte **Prop** qui correspond au type des propositions logiques, et la sorte **Type** qui est le type de **Set** et **Prop**.

Coq offre très peu de constructions natives, essentiellement le produit dépendant $\forall(x:A), B$ dont la version non dépendante (notée $A \to B$) représente le type des fonctions totales de A dans B mais aussi l'implication logique, et les définitions inductives[2]. Ces deux notions sont suffisantes pour définir toutes les autres notions utiles.

Il est possible de définir des types simples, des types polymorphes (paramétrés par un type) ou des types dépendants (paramétrés par une valeur). Les types dépendants permettent notamment de représenter les prédicats.

Exemple A.2.1 (Type simple) *Le type inductif* $\mathbb{N}$ *est défini par :*

$$\textbf{Inductive}\ \mathbb{N}:\textbf{Set} := 0:\mathbb{N} \mid S:\mathbb{N}\to\mathbb{N}.$$

0 *est un constructeur qui définit une constante dans* $\mathbb{N}$*, et* S *est un constructeur qui définit une fonction permettant de fabriquer une (nouvelle) valeur de* $\mathbb{N}$ *à partir d'une valeur de* $\mathbb{N}$*. Le type* $\mathbb{N}$ *correspond au plus petit point fixe pour ces constructeurs, i.e. aux termes de la forme* $S^n(0)$ *pour toute valeur finie de* n*.*

Exemple A.2.2 (Type polymorphe) *Le type option est défini par :*

$$\textbf{Inductive}\ \text{option}\ (A:\textbf{Set}):\textbf{Set} := \text{Some}:A\to\text{option}\ A \mid \text{None}:\text{option}\ A.$$

Ce type est polymorphe, paramétré par le type A*.* None A *est une constante de type* option A*, et* Some A *une fonction qui pour une valeur de type* A *retourne une valeur de type* option A*.*

Exemple A.2.3 (Type dépendant) *Le type des listes de longueur* n *d'éléments de* $\mathbb{Z}$ *est défini par :*

$$\textbf{Inductive}\ \text{dlist}:\mathbb{N}\to\textbf{Set} := \text{dnil}:\text{list}\ 0 \mid \text{dcons}:\forall\,(n:\mathbb{N}),\ \mathbb{Z}\to\text{list}\ n\to\text{list}\ (S\ n)$$

Ce type est dépendant, paramétré par la valeur naturelle n*.* dnil *est une constante de type* dlist 0 *i.e. une liste de longueur nulle, et* dcons *une fonction qui pour une valeur de* $\mathbb{Z}$ *et une liste de longueur* n *retourne une liste de longueur* $(S\ n)$*.*

Exemple A.2.4 (Prédicat de parité) *Le prédicat* even *définit les nombres naturels pairs :*

$$\textbf{Inductive}\ \text{even}:\mathbb{N}\to\textbf{Prop} := \text{even}_Z:\text{even}\ 0 \mid \text{even}_D:\forall(n:\mathbb{N}),\ \text{even}\ n\to\text{even}\ (S(S\ n))$$

Mais even *peut aussi être vu comme un type dépendant, i.e. une famille de types logiques indexée par les nombres naturels. Pour toute valeur* $n:\mathbb{N}$*,* (even n) *décrit un type contenant ou non des termes.*

Par exemple (even 0) *est un type contenant la constante* even_Z*. L'interprétation logique est que* even_Z *est une preuve de la proposition* (even 0)*. De la même façon,* (even_D 0 even_Z) *est une preuve de la proposition* (even 2)*. Réciproquement,* (even 1) *est un type vide (on peut prouver qu'il est impossible de construire un terme de ce type). L'inteprétation logique est que la proposition* (even 1) *est fausse.*

A.3 Rappels sur *FoCal*

Le système *FoCal* [5, 29, 30] est un environnement de développement de programmes certifiés. Il permet l'écriture de programmes dans un langage fonctionnel, mais aussi de propriétés logiques et de preuves, dans un même fichier source. Le compilateur *FoCal* produit des programmes *OCaml* et des preuves pour le système *Coq*. L'outil *Zenon* permet également de produire de manière automatique ou assistée les preuves nécessaires avant de les faire vérifier par *Coq*.

FoCal permet un développement orienté objet [31], avec des mécanismes d'héritage et de paramétrisation, par la définition d'espèces associant des méthodes (des constantes ou des fonctions)

[2]Les principes d'induction associés à un type inductif sont automatiquement générés à la définition.

et des propriétés. Une méthode peut être déclarée (seule sa signature, son type, est fournie) ou définie (l'implémentation est donnée) ; de même une propriété logique peut être déclarée (seule la propriété est exprimée) ou définie (la preuve est donnée). Dans tous les cas, en dehors de l'espèce, seule son interface peut être utilisée (une espèce définit un type abstrait, il n'est possible de manipuler ses valeurs qu'au travers des méthodes fournies).

Une espèce peut hériter de une ou plusieurs espèces (i.e. qu'elle hérite de l'ensemble des méthodes et propriétés de ces espèces). Un espèce peut être paramétrée par d'autres espèces ou par des valeurs.

Une espèce peut aussi contenir une représentation, qui correspond à l'implémentation de l'espèce sous la forme d'un type de données (d'une certaine manière, la représentation correspond à la définition de l'espèce ; tant qu'elle n'est pas fournie l'espèce est seulement déclarée). Lorsque la représentation est définie, toute les méthodes implémentées, et toutes les propriétés prouvées, l'espèce est dite finalisée, et peut être transformée en collection. Une espèce ne peut hériter d'une collection, i.e. qu'une collection fige définitivement la signature.

Exemple A.3.1 (Hiérarchie d'espèces) *L'espèce* superset *définit une famille de types munis d'une relation* equal.

```
species superset inherits basic_object =
  sig equal in self → self → bool;
  property equal_refl : all s in self, !equal(s, s);
  ...
  let diff(x, y) = basics#not_b(!equal(x, y));
  theorem equal_sym : all s1 s2 in self, !equal(s1, s2) → !equal(s2, s1)
  proof : ...
  ...
end
```

Le relation equal *est déclarée, mais pas définie. Elle devra être implémentée avant finalisation de l'espèce. Elle est décrite par des propriétés (telles que* equal_refl*) qui devront également être prouvées avant finalisation. À partir des déclarations il est cependant possible d'implémenter d'autres méthodes (comme par exemple* diff*) ou de prouver des théorèmes (comme par exemple* equal_sym*).*

Une nouvelle espèce peut-être définie, paramétrée par un type super *(une instance de l'espèce* superset*), pour décrire le type des sous-ensembles de* super *:*

```
species decsubset(super is superset) inherits basic_object =
  sig member in super → self → bool;
  sig empty in self;
  property empty_Spec : all v in super, not !member(v, !empty);
  ...
end
```

Cette espèce déclare une méthode member *(la fonction d'appartenance) et une constante* empty *(le sous-ensemble vide).* empty$_{\text{Spec}}$ *est une propriété caractérisant* empty *et* member. decsubset *est paramétrée par* superset*, mais n'en hérite pas. Ainsi,* decsubset *ne propose pas par exemple de relation* equal.

Annexe B

Synthèse

B.1 Recommandations génériques

B.1.1 À propos de la méthode

- Quelle est la logique utilisée, et quelle est la confiance dans la cohérence de cette logique ?
- Des théorèmes viennent-ils enrichir la logique, sont-ils validés ?
- La méthode formelle décrit-elle des hypothèses laissées implicites dans les développements ?
- Quel est le langage de spécification utilisé, son expressivité, sa sémantique ?
- Quel est le langage de développement utilisé, sa sémantique ?
- Quel est le niveau de confiance réel obtenu à différentes étapes de l'utilisation de la méthode lors d'un développement, des preuves ont-elles été différées ou des résultats admis ?

B.1.2 À propos de l'outil

- L'outil utilisé est-il fidèle à la méthode formelle ou y a-t-il des modifications, des enrichissements ou des simplifications ?
- L'outil utilisé est-il validé ?
- Quel est précisément le niveau de confiance réel obtenu à différentes étapes de l'utilisation de l'outil ? Y a-t-il des preuves différées ? Y a-t-il des résultats admis ?
- Y a-t-il une méthodologie de mise en œuvre ?

B.1.3 À propos du développement

- La documentation décrit-elle les objectifs, le périmètre et les limites de la formalisation ?
- Les pratiques de développpement standard sont-elles appliquées ?
- La spécification formelle décrit-elle l'ensemble des propriétés pertinentes ?
- Les mécanismes offerts par la méthode et les outils sont-ils mis en œuvre de manière efficace ?
- La spécification est-elle totale, couvrant tous les cas d'utilisation ?
- La spécification est-elle complète, ou les degrés de liberté laissés sont-ils identifiés, justifiés ?
- La spécification est-elle cohérente ?
- La spécification repose-t-elle sur des hypothèses implicites liées à l'outil où à la méthode formelle, ces hypothèses sont-elles documentées ?
- La formalisation se limite-elle à une spécification, propose-t-elle une conception ou une implémentation ?
- Les objets manipulés existent-ils ? Les ensembles ou types manipulés sont-ils non vides ?

B.2 Recommandations pour *B*

- Les ensembles définis dans la clause **SETS** sont toujours implicitement finis et non vides, ces hypothèses ne doivent jamais être contredites.
- L'existence d'instances valides pour les variables, les constantes, les paramètres ou les variables locales (apparaissant par exemple dans une substitution **ANY**) est supposée, il est nécessaire de vérifier qu'une preuve d'existence est fournie.
- La delta-lemmes doivent être prouvés (*Atelier B*).
- La vérification de typage dans le prouveur doit être activée, ou la preuve doit être vérifiée afin de contrôler qu'elle ne met pas en jeu un paradoxe (*Atelier B*, *B4Free*).

B.3 Recommandations pour *Coq*

- Attention aux propriétés des constructions inductives (injectivité, surjectivité, bonne fondation).
- Vérifier la non vacuité des types (notamment inductifs).
- Éviter ou justifier l'introduction d'hypothèses (**Axiom**, **Hypothesis**, **Variable**, **Parameters**).

B.4 Recommandations pour *FoCal*

- Tant qu'il n'y a pas d'implémentation (de collection) la spécification peut être incohérente, et l'existence d'une instance valide doit être vérifiée.
- La vérification par *Coq* doit être effectuée.
- L'utilisation de la clause **Assumed** dans les preuves doit être documentée et justifiée.
- La terminaison des fonctions récursives doit être vérifiée manuellement.
- La validité des règles de traduction vers *Coq* et *OCaml* doit être documentée et vérifiée.

Annexe C

Glossaire

Abstraction Dans une méthode formelle permettant une forme de raffinement, l'abstraction est ce dont dérive la concrétisation. L'abstraction d'un code peut être sa spécification, ou son type abstrait (approche objet).

Cohérence (Logique) Une logique est dite cohérente si, à partir des règles de déduction proposées, il y a des propositions qui ne sont pas prouvables. Une logique incohérente est une logique dans laquelle toutes les propositions sont prouvables (y compris les propositions qui sont 'fausses'). Une telle incohérence est mise en évidence par l'exhibition d'un paradoxe.

Cohérence (Spécification) Une spécification est dite cohérente si elle n'est pas contradictoire. La cohérence d'une spécification est généralement indécidable, mais est assurée s'il est possible d'exhiber une instanciation ou une implémentation. En effet, une spécification incohérente ne peut être implémentée : par exemple, la spécification $x=0 \wedge x=1$ est incohérente, il est impossible d'instancier la constante x avec une valeur respectant sa spécification.

Complétude (Spécification) Une spécification est dite complète si elle est suffisamment précise pour assurer que deux implémentations de cette spécification sont observationnellement équivalentes. Une spécification est incomplète si elle laisse de réels degrés de liberté pendant le processus de raffinement.

Complétude (Prouveur) Un prouveur est dit complet s'il permet de prouver tout résultat prouvable dans la logique qu'il implémente.

Correction (Prouveur) Un prouveur est dit correct s'il ne permet de prouver que des résultats prouvables dans la logique qu'il implémente.

Décidabilité Une proposition est dite décidable si il existe une procédure calculatoire, dont le temps d'exécution peut être borné *a priori* en fonction de ses paramètres, qui permet de vérifier si la proposition est vraie (prouvable) ou fausse (réfutable). Par exemple, la primalité d'un entier n est décidable puisqu'il suffit de vérifier pour chaque entier $2 \leq i < n$ que i ne divise pas n. La prouvabilité d'une proposition logique est généralement indécidable.

Implémentation Une implémentation est une réalisation concrète (une valeur, un programme, un système) d'une spécification (une proposition logique).

Interprétation Pour une représentation concrète donnée (une variable, un programme, un système), l'interprétation de cette représentation concrète donne l'objet abstrait (théorique) associé. Par exemple, l'interpétation d'une liste d'entiers machines peut être un ensemble de nombre naturels. L'interprétation est contextuelle et souvent implicite dans un développement formel.

Modèle (Logique) Pour une proposition, on parle de modèle pour décrire une instanciation qui associe à chaque terme (constante) apparaissant dans la proposition une valeur permettant de rendre vraie cette proposition . Par exemple, $[S = \mathbb{N}, \otimes = +]$ est un modèle de la proposition $\forall\, x, y : S,\ x \otimes y = y \otimes x$. L'existence d'un modèle garantit la satisfiabilité d'une proposition – ou, en d'autres termes, l'existence d'une implémentation garantit la cohérence d'une spécification.

Pré-condition Une pré-condition est une condition devant être vérifiée préalablement à l'utilisation d'une fonction, d'un programme ou d'un système par l'appelant. Dans une spécification, une pré-condition limite la portée de la propriété exprimée, ce qui se traduit le plus souvent par une spécification partielle (non totale).

Preuve papier Une preuve papier est une preuve manuelle non vérifiée mécaniquement.

Preuve vérifiée Une preuve vérifiée est une preuve qui a été vérifiée mécaniquement par un prouveur (réputé) correct et complet.

Preuve automatique Une preuve automatique est une preuve générée par un prouveur, sans intervention humaine. Une telle preuve peut devoir être vérifiée, en l'absence de garanties que le prouveur utilisé est correct.

Preuve assistée Une preuve assistée est une preuve écrite par un humain mais vérifiée au fur et à mesure de son élaboration par un prouveur.

Prédicat Un prédicat est une propriété logique pouvant dépendre de une ou plusieurs variables. Un prédicat peut être prouvable ('vrai'), réfutable ('faux', sa négation est prouvable) ou indécidable.

Programmation défensive Discipline de programmation dans laquelle la validité des valeurs manipulées (par exemple le respect des pré-conditions par les paramètres passés en entrée) est systématiquement vérifiée. La programmation défensive vise à améliorer la robustesse d'un système, notamment en limitant la propagation des fautes, au prix d'une baisse des performances.

Programmation offensive Discipline de programmation dans laquelle la validité des valeurs manipulées n'est pas vérifiée mais supposée car résultant de la construction du système.

Proposition Une proposition est un prédicat ne dépendant pas de variables ; par abus de langage, le terme proposition est parfois utilisé à la place du terme prédicat.

Prouveur On désigne dans ce document par prouveur tout outil qui permet soit de générer automatiquement des preuves, soit de vérifier leur validité.

Raffinement Le raffinement est un processus qui permet d'aller, en une ou plusieurs étapes, d'une spécification à une implémentation, ou de manière plus générale de représentations abstraites (non déterministes, non exécutables, non implémentables) à des représentations concrètes.

Sémantique (Langage) La sémantique d'un langage permet de donner du sens aux termes (définis par la syntaxe) de ce langage. Pour un langage de programmation, la sémantique pourra par exemple être opérationnelle et décrire le comportement d'un programme. Pour un langage logique, la sémantique pourra être décrite à travers un ensemble de règles de déduction permettant de distinguer les propositions 'vraies' (prouvables) et les propositions 'fausses' (réfutables).

Spécification Dans ce document, une spécification correspond toujours à une spécification formelle, un ensemble de propositions logiques décrivant les contraintes devant être respectées par un système.

Syntaxe (Langage) La syntaxe d'un langage (de programmation, de spécification) décrit l'ensemble des termes qui peuvent être écrits.

Témoin Un témoin est une valeur (une implémentation) qui vérifie un prédicat, ou par extension une valeur appartenant à un type donné.

Totalité Une fonction est dite totale lorsqu'elle associe un résultat à chaque valeur de son domaine, et partielle dans le cas contraire (par exemple, la racine carrée est généralement définie comme une fonction partielle sur $\mathbb{Z}$). Par extension une spécification est dite totale lorsqu'elle couvre l'ensemble des cas d'utilisation possibles du système décrit, partielle lorsque certains cas d'emploi ne sont pas spécifiés. Une spécification peut être totale sans être complète : une spécification de la division entière laissant le choix entre une division par excès (arrondi au supérieur) ou par défaut (arrondi à l'inférieur) est totale sans être complète.