Secrétariat général de la défense et de la sécurite nationale
Agence nationale de la sécurité des systèmes d'information

Université Pierre et Marie Curie
Laboratoire d'informatique de Paris 6

# Résumé de mémoire de thèse

# Étude de l'apport des méthodes formelles déductives pour les développements de sécurité

par Éric Jaeger

Thèse de l'École de Paris soutenue le 8 mars 2010

au Lip6, 104 boulevard Kennedy – 75015 Paris, France

pour l'obtention du titre de docteur en informatique

Mémoire et développements disponibles sur `http://www.ssi.gouv.fr`

# 1 Introduction

## 1.1 Méthodes formelles et niveaux d'assurance

Le développement de logiciels, d'équipements ou de systèmes présente de nombreux défis, et en pratique des problèmes résiduels sont malheureusement très fréquents dans les versions déployées. De tels défauts sont cependant inacceptables pour les systèmes critiques.

De nombreuses approches ont été proposées pour assurer une meilleure maîtrise des systèmes critiques, insistant par exemple sur la méthodologie de développement, la documentation, le test, les revues. Les *approches formelles* se distinguent par leur capacité à donner des garanties mathématiques quant à l'absence de certains défauts, et à ce titre leur utilisation est encouragée ou exigée par certains standards relatifs à la certification de sécurité (Critères Communs [CC]) ou de sûreté (Iec 61508 [IEC]).

Nous nous intéressons principalement dans la suite aux *méthodes formelles déductives* telles que B, FoCaLize, Coq, dans lesquelles il est possible de spécifier un système dans un langage formel, de l'implémenter, puis de prouver la conformité de l'implémentation vis à vis de la spécification.

## 1.2 Méthodes formelles déductives et sécurité

Plus particulièrement, c'est l'utilisation des méthodes formelles déductives dans le domaine de la *sécurité* qui est considérée dans le mémoire – *i.e.* lorsque le système doit résister à des actions malveillantes d'agents intelligents. En effet, dans une telle démarche il est légitime de s'interroger sur la portée et le niveau exact de la confiance résultant d'un développement formel ; une *sécurité prouvée* correspond-elle à une sécurité réelle et inconditionnelle ?

Nous identifions dans la suite différents problèmes potentiels, pour lesquels nous proposons des recommandations pour les développeurs mais aussi pour les évaluateurs indépendants susceptibles d'intervenir dans le cadre d'une certification de sécurité d'un produit développé tout ou en partie formellement.

Deux points sont tout particulièrement développés après cette revue générale. Tout d'abord, nous étudions le concept de *raffinement* au travers d'une formalisation générique et intuitive, pour comprendre précisemment les phénomènes tels que le *paradoxe du raffinement*. Nous abordons par ailleurs la question de la validation de la théorie et des outils associés à une méthode formelle au travers d'un plongement profond de la logique de la méthode B en Coq ; ce plongement est également utilisé pour prouver de nouveaux résultats pour la logique du B.

## 1.3 Avertissement préalable

Il nous paraît important d'insister sur le fait qu'il ne fait pour nous aucun doute que l'utilisation des méthodes formelles en général, et des méthodes formelles déductives en particulier, apporte d'indéniables avantages. Elles ont effectivement le potentiel d'éliminer totalement certaines classes d'erreur – une perspective inenvisageable avec d'autres formes d'approches.

Cependant, il reste pertinent d'identifier clairement la portée exacte des garanties apportées, et le niveau de confiance associé. C'est particulièrement important dans le domaine de la sécurité, lorsque la menace est représentée par des agents malveillants et intelligents – par exemple un développeur malicieux – capables de s'adapter notamment à la mise en œuvre des méthodes formelles.

# 2 Méthodes formelles déductives

Plusieurs approches sont proposées pour améliorer la qualité des logiciels, sur la base de bonnes pratiques, d'un référentiel documentaire ou de tests poussés par exemple. Les approches formelles ont le même objectif mais se distinguent par l'emploi de démarches mathématiques visant à garantir l'absence de certaines catégories d'erreurs. Elles regroupent notamment le typage, l'interprétation abstraite ou encore le *Model Checking*, et peuvent être utilisées lors des phases de spécification, de conception, d'implémentation ou encore de tests [WLBF09]. Dans la suite de ce document, nous nous intéressons plus spécifiquement aux *méthodes formelles déductives*, c'est à dire aux méthodes qui permettent de raisonner par déduction sur les spécifications et les implémentations ; ce sont les plus utilisées dans le domaine de la certification de sécurité, probablement car elles offrent le meilleur niveau d'expressivité. Nous décrivons succinctement trois de ces méthodes, B, Coq et FoCaLize.

## 2.1 La méthode B

La méthode B, developpée principalement par *J-R. Abrial*, est une méthode formelle dont l'usage est répandu aussi bien dans l'industrie que dans le monde académique, notamment pour des projets en rapport avec la sûreté ou la sécurité [BDM98, Bie96, Jaf07, SL00]. Plusieurs environnements de développement B existent, industriels ou académiques.

La méthode B permet la dérivation des programmes corrects par construction à partir d'une spécification. Elle définit une logique du premier ordre avec des notions ensemblistes, un langage de spécification et d'implémentation – le Gsl – et une démarche de développement basée sur le concept de *raffinement*. La logique permet de décrire des préconditions, des invariants, et de faire des preuves. Le Gsl définit des instructions abstraites, déclaratives et non deterministes (*i.e.* des spécifications) et d'autres concrètes, impératives et déterministes (*i.e.* des programmes), par exemple :

**Exemple 2.1 (Spécification de la racine carrée)**

$$\textbf{ANY } x \textbf{ WHERE } x^2 \leq n < (x+1)^2 \textbf{ THEN } s := x$$

*La substitution* **ANY** *est un opérateur (non exécutable) qui trouve une valeur vérifiant une propriété ; ici on ne sait pas comment calculer* $s$*, mais on sait ce que doit être* $s$*.*

Les développements B sont constitués de *machines*, des modules qui associent un état (sous la forme de variables), un invariant sur l'état et des opérations (des substitutions du Gsl) pour lire ou modifier l'état. Partant d'une machine abstraite, qui doit être une pure spécification, des machines de plus en plus concrètes sont décrites jusqu'à obtenir une implémentation qui peut être traduite en code exécutable. La conformité de l'implémentation par rapport à la spécification initiale se fait en prouvant que chaque machine de cette suite est un raffinement de la précédente.

Prouver le raffinement consiste à garantir que tout comportement observable de la machine concrète est un comportement possible de la machine abstraite. Ceci est indépendant de la représentation de l'état interne de la machine, comme illustré par l'exemple classique du *Maximier* :

**Exemple 2.2 (Maximier)**

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

**MACHINE** $M_C$ **REFINES** $M_A$
**VARIABLES** $s$
**INVARIANT** $s = \mathbf{max}(S \cup \{0\})$
**INITIALISATION** $s := 0$
**OPERATIONS**
$store(n) \triangleq$ **IF** $s < n$ **THEN** $s := n$
$m \leftarrow get \triangleq m := s$

L'état de la machine est décrit par la clause **VARIABLES** et contraint par la clause **INVARIANT** ; pour $M_A$ il s'agit d'un sous-ensemble $S$ de nombres naturels, alors que pour $M_C$ c'est un nombre naturel $s$. $M_C$ donne également un *invariant de collage* qui intuitivement indique que si les deux machines sont utilisées en parallèle, alors à tout instant $s = \mathbf{max}(S)$. Ces deux représentations de l'état sont très différentes mais $M_C$ raffine bien $M_A$ : tout comportement de $M_C$ est un comportement possible de $M_A$. La formalisation du raffinement repose sur une sémantique des constructions du GSL sous la forme de transformateurs de prédicats.

## 2.2 L'assistant de preuves COQ

COQ [Coq, BC04] est un assistant de preuves basé sur une logique d'ordre supérieur, permettant la construction et la vérification de preuves, ainsi que le développement et l'analyse de programmes fonctionnels écrits dans un langage à la ML. COQ est basé sur le *Calcul des Constructions Inductives* [CP88, Wer94] et permet de définir des constructions inductives calculatoires (dans la sorte **Set**) :

**Exemple 2.3 (Définition inductive des naturels à la *Peano*)**

**Inductive** $\mathbb{N} : \mathbf{Set} \triangleq 0 : \mathbb{N} \mid S : \mathbb{N} \to \mathbb{N}$

*$\mathbb{N}$ est ici défini comme le plus petit ensemble de termes stable par application des constructeurs $0$ et $S$ ; étant bien fondé, l'induction sur $\mathbb{N}$ est possible (le principe d'induction structurelle associé est automatiquement dérivé par* COQ*).*

COQ permet aussi de définir inductivement des constructions logiques (dans la sorte **Prop**) :

**Exemple 2.4 (Définition inductive du prédicat de parité)**

**Inductive** $\mathrm{even} : \mathbb{N} \to \mathbf{Prop} \triangleq ev_0 : \mathrm{even}\ 0 \mid ev_2 : \forall (n : \mathbb{N}),\ \mathrm{even}\ n \to \mathrm{even}\ S(S\ n)$

La logique de COQ est constructive (ou intuitionniste), en particulier le *Tiers Exclu* n'est pas prouvable. Elle permet de bénéficier de l'isomorphisme de *Curry-Howard*, c'est à dire par exemple d'extraire des programmes à partir de preuves :

**Exemple 2.5 (Strong specification style)**

```
Require Import Arith.

Theorem div2:forall (n:nat), {m:nat | n=m+m \/ n=S(m+m)}.
Proof.
 induction n as [| n Hind].
  exists 0; left; apply refl_equal.
  destruct Hind as [m Hind].
   destruct (eq_nat_dec n (m+m)) as [Heq | Hdiff].
    exists m; rewrite Heq; right; apply refl_equal.
    assert (Heq:n=S(m+m)).
     destruct Hind as [Hind | Hind].
      destruct (Hdiff Hind).
      apply Hind.
    exists (S m); rewrite Heq; left; simpl;
     rewrite <- plus_n_Sm; apply refl_equal.
Defined.
```

*div2 prouve que pour toute valeur* $n : \mathbb{N}$ *il existe une valeur* $m$ *qui est sa moitié. Mais en* Coq *div2 est aussi un programme de paramètre* $n$ *retournant la valeur* $m$*, qui peut être obtenu avec la commande* **Extraction** *div2 :*

```
let rec div2 = function
  | 0 -> 0
  | S n0 ->
      let hind = div2 n0 in
      (match eq_nat_dec n0 (plus hind hind) with
         | Left -> hind
         | Right -> S hind)
```

## 2.3 L'environnement FoCaLize

FoCaLize [Foc] est une méthode formelle permettant de spécifier et de progresser de manière incrémentale vers une implémentation, tout en prouvant la conformité. Elle propose une forme de développement orienté objet, dont la brique fondamentale est *l'espèce*, regroupant des déclarations (*i.e.* des types), des définitions, des propriétés et des preuves, ainsi qu'une représentation (le type des données manipulées) ; les espèces peuvent aussi être composées par héritage et paramétrisation.

Une fois toutes les déclarations définies et les propriétés prouvées, une espèce est complète et peut être transformée en *collection*, un type abstrait dont la représentation et les définitions sont masquées, ce qui impose que ses valeurs ne peuvent être manipulées qu'à travers l'interface fournie (garantissant la préservation des propriétés annoncées).

Dans l'exemple suivant l'espèce $\Omega$ décrit toute collection pour laquelle une relation d'équivalence $=$ sur ses éléments est définie :

**Exemple 2.6 (Superset)**

**species** $\Omega \triangleq$
  **signature** $(=) : \mathbf{Self} \to \mathbf{Self} \to \mathbb{B}$
  **property** $=_{\text{refl}} : \forall\, s : \mathbf{Self},\ s = s$
  **property** $=_{\text{symm}} : \forall\, s_1, s_2 : \mathbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_1$
  **property** $=_{\text{tran}} : \forall\, s_1, s_2, s_2 : \mathbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_3 \Rightarrow s_1 = s_3$

Nous pouvons ensuite définir une espèce $\wp$ spécifiant les sous-ensembles de valeurs de son paramètre $\omega$ implémentant $\Omega$ :

**Exemple 2.7 (Powerset)**

**species** $\wp(\omega$ **is** $\Omega) \triangleq$
  **signature** $\emptyset$ : **Self**
  **signature** $(\in)$ : $\omega \to$ **Self** $\to \mathbb{B}$
  **signature** $(+)$ : $\omega \to$ **Self** $\to$ **Self**
  **property** $\in_\emptyset$: $\forall\, s{:}\omega,\ \neg s \in \emptyset$
  **property** $\in_+$: $\forall\, s_1, s_2{:}\omega, \forall\, S{:}$**Self**, $s_1 \in (S + s_2) \Leftrightarrow s_1 \in S \vee s_1 = s_2$

Une collection implémentant $\wp(\omega)$ pourra par exemple utiliser une représentation sous la forme de listes de valeurs de $\omega$, ou des fonctions dans $\omega \to \mathbb{B}$. Cette dernière représentation ne permet pas d'implémenter l'inclusion si $\omega$ est infini. Il peut donc être pertinent de poursuivre la spécification en décrivant les sous-ensembles extensionnellement comparables $\mathcal{F}$, *i.e.* les sous-ensembles pour lesquels l'inclusion peut être implémentée. A noter que $\mathcal{F}$ hérite de $\wp$ mais aussi de $\Omega$, puisqu'une égalité est également définie entre sous-ensembles extensionnellement comparables :

**Exemple 2.8 (Sous-ensembles extensionnels)**

**species** $\mathcal{F}(\omega\ is\ \Omega) \triangleq$
  **inherit** $\Omega, \wp(\omega)$
  **signature** $(\subseteq)$ : **Self** $\to$ **Self** $\to \mathbb{B}$
  **property** $\subseteq_{\text{spec}}$ : $\forall\, S_1, S_2{:}$**Self**, $S_1 \subseteq S_2 \Leftrightarrow (\forall\, s{:}\omega,\ s \in S_1 \Rightarrow s \in S_2)$
  **property** $=_{\text{ext}}$ : $\forall\, S_1, S_2{:}$**Self**, $S_1 = S_2 \Leftrightarrow S_1 \subseteq S_2 \wedge S_2 \subseteq S_1$

# 3 Méthodes formelles et sécurité

Nous adoptons la vision du développement selon un *cycle en V*, qui identifie différentes étapes telles que la spécification, la conception, l'implémentation. Dans cette approche la correction est tout d'abord un problème de traçabilité entre deux étapes, et une méthode formelle déductive propose une analyse mathématique de cette traçabilité, i.e. d'une *similarité* suffisante entre les différentes *descriptions*. Nous analysons dans cette section quels sont les pièges ou problèmes potentiels qui peuvent cependant subsister pour les développements de sécurité.

## 3.1 Rappels préliminaires

### 3.1.1 Spécification formelle

Au moins deux descriptions formelles d'un système sont généralement considérées pour un développement, la *spécification* et l'*implémentation*. La spécification est une description qui est idéalement déclarative, abstraite, haut niveau et non déterministe (le *quoi*) ; par contraste l'implémentation est impérative, concrète, bas niveau et déterministe (le *comment*). La distinction entre une approche déclarative et une approche impérative est illustrée par l'exemple suivant :

**Exemple 3.1 (Spécification de la racine carrée en** B**)**

**ANY** $x$ **WHERE** $x^2 \leq n < (x+1)^2$ **THEN** $s := x$

*Cette spécification est déterministe (pour toute valeur de* $n \in \mathbb{N}$ *il y a exactement une valeur possible pour* $s$*), mais ce n'est pas un programme ; il n'y a aucune information sur la façon dont cette valeur est calculée. Cette spécification peut par contre être vue comme un oracle permettant de vérifier si le résultat donné par une implémentation est correct.*

On peut noter que le simple fait d'écrire une spécification formelle constitue déjà une amélioration par rapport aux approches classiques ; en effet, le langage formel permet d'éliminer les ambiguités ou certaines catégories d'expressions dénuées de sens.

### 3.1.2 Raffinement

La notion de *raffinement* est souvent rencontrée dans les approches formelles pour décrire le processus de développement sous-jacent (cf. par exemple [GFL05] pour une synthèse). Dans le cadre des méthodes formelles déductives, cette notion est plus spécifiquement liée à la formalisation de la notion de traçabilité qui permet de garantir à terme la conformité, par exemple au travers d'une sémantique de transformateurs de prédicats comme en Z, en B ou en Event-B.

Le terme raffinement est utilisé ici de manière plus générale pour décrire le processus de développement formel allant de la spécification à l'implémentation ; il inclut notamment le choix des structures de données, des algorithmes, etc. tout en garantissant la conformité.

### 3.1.3 Logique

Un langage formel permet le plus souvent de décrire des spécifications incohérentes, i.e. contradictoires (par exemple exiger qu'un variable soit égale à la fois à 0 et à 1). Il y a trois observations importantes à propos des spécifications incohérentes :

- la cohérence d'une spécification n'est pas décidable ;
- une spécification incohérente ne peut pas être implémentée ;
- supposer l'existence d'une implémentation permet de prouver n'importe quel résultat.

En raison du premier point, les outils formels ne cherchent pas à détecter automatiquement les incohérences, même dans les cas triviaux – en notant qu'un objectif d'implémentation garantira de toute façon la détection, tôt ou tard, de toute incohérence. Le troisième point résulte du fait qu'il est toujours possible de prouver $\mathbf{False} \Rightarrow P$.

## 3.2 Spécifier les systèmes sécurisés

### 3.2.1 Spécifications incohérentes

Une spécification incohérente peut être désastreuse, puisque la contradiction peut longtemps rester indétectée alors qu'elle permet de décharger n'importe quelle obligation de preuve. Certes, une spécification incohérente n'est pas implémentable, mais il faut noter que par exemple dans le cas des Critères Communs v2.3, au niveau EAL-5, seule une spécification formelle de la politique de sécurité est demandée. La possibilité pour un développeur malicieux d'exploiter une incohérence jamais détectée est alors réelle.

De telles incohérences sont très aisées à introduire, et peuvent être difficiles à détecter par un relecteur, comme dans les exemples suivants :

**Exemple 3.2 (Propriété insatisfiable en B)**

**MACHINE** absurd_cst
**CONSTANTS** $f$
**PROPERTIES** $f \in \mathbb{N} \rightarrow \mathbb{N} \wedge \forall\, x, y,\ x < y \Rightarrow f(x) > f(y)$

**Exemple 3.3 (Violation des propriétés des types inductifs en Coq)**

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

Il convient également de se défier des types vides, dont l'abus par un développeur malicieux peut avoir des conséquences similaires :

**Exemple 3.4 (Type vide en Coq)** *Nous définissons de manière maladroite les listes bicolores, en oubliant par erreur le constructeur atomique pour les listes vides :*

$$\textbf{Inductive}\ \text{blst} : \textbf{Set} \quad \triangleq \quad \text{red} : \text{blst} \to \mathbb{N} \to \text{blst} \quad \mid \quad \text{blue} : \text{blst} \to \mathbb{N} \to \text{blst}.$$

*Dès lors le type* blst *est en fait vide, et supposer l'existence d'une valeur dans ce type est incohérent. Cela signifie aussi que toute propriété de la forme* $\forall\, (b : \text{blst}),\ P$ *est prouvable.*

La détection d'incohérences ou de conditions insatisfiables peut être facilitée par l'utilisation d'outils tels que les *Model Checkers* [CGL96, YML99], les animateurs de modèles [PL07] ou les génerateurs de tests [CD08, JL07].

### 3.2.2 Quelques malentendus

Nous considérons maintenant la problématique des spécifications insuffisantes, plus délicate à traiter puisqu'elle relève d'une différence entre l'intention de la spécification et sa signification formelle.

L'insuffisance peut par exemple résulter d'une mauvaise compréhension des principes sous-jacents à une méthode formelle : toute méthode définit (le plus souvent implicitement) les aspects observables et non observables d'un système, et ne pas comprendre cette partition peut avoir des conséquences graves.

La notion de raffinement vise en effet à assurer qu'une spécification et une implémentation sont *suffisamment* similaires ; mais cette similarité ne doit pas être trop forte, au risque de se réduire à l'égalité entre la spécification et l'implémentation, i.e. d'interdire de progresser pendant le développement. Il est typique par exemple de ne pas prendre en compte les états transitoires, les calculs, la mémoire ou le temps utilisé, mais cela peut être exploité par un développeur malicieux, comme dans l'exemple suivant :

**Exemple 3.5 (États transitoires non observables)** *La spécification suivante décrit un sas dont la propriété de sûreté est qu'à tout instant au plus une porte est ouverte :*

**MACHINE** Airlock
**VARIABLES** $\text{door}_1, \text{door}_2$
**INVARIANT** $\text{door}_1, \text{door}_2 \in \{\text{open}, \text{locked}\} \wedge \neg(\text{door}_1 = \text{open} \wedge \text{door}_2 = \text{open})$
**INITIALISATION** $\text{door}_1 := \text{locked} \parallel \text{door}_2 := \text{locked}$
**OPERATIONS**
$\text{open}_1 \triangleq$ **IF** $\text{door}_2 = \text{locked}$ **THEN** $\text{door}_1 := \text{open}$
$\text{close}_1 \triangleq \text{door}_1 := \text{locked}$
$\text{open}_2 \triangleq$ **IF** $\text{door}_1 = \text{locked}$ **THEN** $\text{door}_2 := \text{open}$
$\text{close}_2 \triangleq \text{door}_2 := \text{locked}$

*L'invariant est parfois vu comme décrivant une propriété toujours vraie, mais ce n'est*

*pas tout à fait correct. Ainsi, le raffinement suivant est formellement valide :*

**MACHINE** Trapped_Airlock
**REFINES** Airlock
**CONSTANTS** code := 147
**VARIABLES** $\text{door}_1, \text{door}_2, \text{state}$
**INVARIANT** $\text{door}_1, \text{door}_2 \in \{\text{open}, \text{locked}\} \wedge \neg(\text{door}_1 = \text{open} \wedge \text{door}_2 = \text{open})$
**INITIALISATION** $\text{door}_1 := \textit{locked} \parallel \text{door}_2 := \textit{locked} \parallel \text{state} := 0$
**OPERATIONS**
$\text{open}_1 \triangleq$ **IF** $\text{door}_2 = \text{locked}$ **THEN** $\text{door}_1 := \text{open}$;
**IF** $\text{state} = \text{code}$ **THEN** $\text{door}_2 := \text{open}$; wait; $\text{door}_2 := \text{locked}$
$\text{close}_1 \triangleq \text{door}_1 := \text{locked}$; $\textit{state} := (\textit{state} * 2)$ **mod** 256
$\text{open}_2 \triangleq$ **IF** $\text{door}_1 = \text{locked}$ **THEN** $\text{door}_2 := \text{open}$
$\text{close}_2 \triangleq \text{door}_2 := \text{locked}$; $\textit{state} := (\textit{state} * 2 + 1)$ **mod** 256

wait *est une opération passive mais lente ; il est donc possible d'ouvrir simultanément les deux portes pendant plusieurs secondes, la preuve de préservation de l'invariant garantissant simplement que* avant *et* après *l'exécution d'une opération, au plus une porte est ouverte. À noter ici que nous avons fait l'effort de masquer ce comportement dangereux en utilisant une condition particulière peu susceptible d'être déclenchée pendant des tests.*

### 3.2.3 Spécifications partielles

Il est également pertinent de s'interroger sur la *totalité* d'une spécification : le comportement du système est-il décrit dans toutes les circonstances ? Il est en effet fréquent dans une spécification de ne décrire qu'un sous-ensemble des situations envisageables, mais le degré de liberté laissé au développeur est alors facilement sous-estimé :

**Exemple 3.6 (Spécifications partielles)** *Nous spécifions en* Coq *la fonction* head *qui retourne le premier élément d'une liste de deux façon différentes :*

$$\text{head}_1(l : \textbf{list}\ \mathbb{N})(p : l \neq [\,]) : \{x : \mathbb{N} \mid \exists\, l' : \text{list}\ \mathbb{N},\ l = x :: l'\}$$

$$\text{head}_2(l : \textbf{list}\ \mathbb{N}) : \{x : \mathbb{N} \mid l \neq [\,] \rightarrow \exists\, l' : \text{list}\ \mathbb{N},\ l = x :: l'\}$$

*Dans les deux cas la fonction doit retourner la tête de de la liste passée en paramètre si celle-ci n'est pas vide. Il est possible de raffiner ces deux spécifications pour aboutir à une même implémentation où le développeur renvoie une valeur secrète pour une liste vide :*

$$\textbf{let}\ \text{head}_* = \textbf{function}\ [\,] \rightarrow \text{secret} \mid h :: \_ \rightarrow h$$

**Exemple 3.7 (Pré-condition en** B**)** *Les opérations dans une machine* B *sont associées à des pré-conditions qui correspondent à la définition du domaine de spécification. En dehors de ce domaine, l'opération n'est pas spécifiée – et en pratique, un développeur malicieux peut dans de telles situations implanter des comportements violant l'invariant.*

Une spécification partielle ne peut pas garantir la sécurité, et il est important dans de telles situations de favoriser un style offensif (utilisant des gardes) plutôt que défensif (utilisant des pré-conditions) ; dans un langage impératif, de telles gardes correspondent à un **IF** (ou à un **match** en OCaml).

## 3.3 Propriétés non exprimables

De manière générale, certaines propriétés attendues en sécurité peuvent être très difficiles à exprimer dans le cadre d'un développement basé sur une méthode formelle déductive. La confidentialité, par exemple, peut être décrite à travers un moniteur de contrôle d'accès filtrant des requêtes, mais cela ne constitue qu'une vision partielle de la propriété recherchée.

Un développeur malicieux peut en effet faire fuir de l'information en utilisant des techniques de type *canaux cachés* (en exploitant le non déterminisme, la non observabilité formelle des états transitoires ou le temps de calcul par exemple). La prévention de ce type de piégeages sera plus facile à garantir par des techniques d'analyse de dépendances plutôt que par les seules méthodes formelles déductives.

Par ailleurs, la cryptographie, outil essentiel de la sécurité, repose également sur de nombreuses propriétés qui peuvent être très difficiles à capturer dans une spécification. On pourra par exemple considérer l'exemple des fonctions de hachage, dont une propriété essentielle est la non inversibilité : spécifier maladroitement cette propriété aboutira rapidement à une incohérence. Des méthodes existent pour décrire les propriétés de sécurité les plus subtiles, mais elles sont moins triviales qu'on ne pourrait le penser.

## 3.4 Un colosse aux pieds d'argile

Les conséquences d'une incohérence dans la théorie de la méthode formelle ou dans l'outil qui l'implémente sont au moins aussi graves que celles d'une incohérence dans la spécification. Elles permettent en effet à un développeur malicieux de fournir un programme piégé tout en obtenant un certificat de conformité.

### 3.4.1 Cohérence de la logique

La question de la cohérence de la logique est difficile à traiter. Au mieux la vérification repose sur des travaux longs et difficiles, mais elle a pu être abordée notamment dans différents travaux académiques (cf. [Bar99, Cha98, BFM99, Bur00, BBM98] par exemple pour Coq et la méthode B). Nous présentons une revue détaillée du B dans la section 5, à travers un plongement profond en Coq.

### 3.4.2 Validité des outils

Même pour une théorie cohérente, il reste possible d'avoir des erreurs dans l'implémentation de l'outil formel. Un prouveur par exemple peut être incorrect, au sens où il permet de prouver des résultats qui ne sont pas prouvables dans la théorie, ce qui peut être exploité par un développeur malicieux.

Cela justifie le développement d'outils formels prouvés, ou tout au moins bien maîtrisés. Cette problématique est abordée notamment par [RM05, CK98, Cha98, BDFF04] ; nous proposons également un développement de ce genre dans la section 5, en étendant le plongement profond de la logique de B en Coq avec un prouveur prouvé.

### 3.4.3 Sauts de foi

Les méthodes formelles déductives permettent, à l'issue d'un développement, de fournir un programme prouvé conforme. On peut cependant s'interroger sur cette dernière étape reposant sur une traduction de confiance, et plus précisemment sur la cohérence entre la sémantique formelle et le fonctionnement réel du programme sur un système.

En Coq par exemple, il est possible de décrire des structures de données et des fonctions dans un langage à la Ml, puis de les extraire sous la forme d'un programme OCaml qui peut être compilé et exécuté. Nous avons identifié une différence notable dans l'interprétation des types inductifs :

**Exemple 3.8 (Sémantiques incohérentes pour les types inductifs)** *Comme expliqué au paragraphe 3.2.1, la définition suivante représente un type vide en* Coq *:*

$$\textbf{Inductive}\ E : \textbf{Set} := \mathrm{nxt} : E \to E$$

*Il est donc possible de prouver toute propriété de la forme* $\forall\,(e:E),\ P$*.*

*La traduction automatique en* OCaml *produit le code* `type E = Nxt of E.` *Il est intéressant de noter que ce type n'est pas vide, puisqu'il contient notamment* `let rec e= Nxt e.` *De manière plus générale,* OCaml *admet des valeurs qui ne sont pas strictement inductives ; par exemple* `let rec omega=S omega` *est considéré comme un nombre naturel. En conséquence, il est possible d'utiliser des programmes certifiés sur des valeurs inattendues, avec des conséquences imprévisibles.*

D'autres illustrations de ce type de problèmes peuvent être données, par exemple en B. Elles semblent difficiles à traiter de manière générique.

### 3.5 Contredire le modèle

Supposons maintenant que nous disposons d'un système prouvé, en ayant pris garde d'éviter les problémes précédemment identifiés. Est-ce à dire que le système est parfaitement et inconditionnellement sûr ?

La sécurité repose de manière évidente sur le respect des hypothèses qui ont été utilisées pendant le développement. Celle-ci peuvent être explicitées dans le modèle (sous la forme d'axiomes) ou être *implicites*, liées à la méthode formelle ou aux choix de modélisation. Un exemple très significatif est donné par le typage, qui dans une spécification formelle décrit implicitement l'ensemble des actions possibles et impossibles, alors que le plus souvent il disparaît à la compilation. Contredire le typage à l'exécution est donc envisageable et peut mener à une attaque. À titre d'illustration, nous pouvons considérer une attaque sur l'API cryptographique Pkcs#11 décrite dans [Clu03], dans laquelle une clé de chiffrement est obtenue en la faisant passer pour un message.

## 4 Étude du concept de raffinement

Le raffinement est un concept explicite dans plusieurs méthodes formelles (B, Z par exemple) ; c'est aussi ici le concept implicite qui représente un développement formel dans un cycle en V. Une mauvaise compréhension de ce concept peut mener à des problèmes de sécurité – résultant par exemple de ce qu'on appelle généralement le *paradoxe du raffinement*.

Nous proposons ici une vision simple, génerique et intuitive de ce concept, afin de mettre en évidence certains faits et de raisonner sur ses applications. Le raffinement de la méthode B est utilisé comme modèle principal, mais nous prenons également en compte des formes de raffinements pouvant être réalisées en Coq ou en FoCaLize.

### 4.1 Description informelle

La raffinement définit une relation de *similarité* entre différentes *descriptions* d'un système, par exemple une spécification abstraite, de haut niveau, déclarative et non déterministe,

et une implémentation concrète, de bas niveau, impérative et déterministe. La similarité garantit que les propriétés de la spécification sont préservées par l'implémentation – tout au moins pour une certaine classe de propriétés "pertinentes".

Un raffinement doit aussi avoir les caractéristiques suivantes :

- *transitivité* (si $D_C$ raffine $D_I$ et $D_I$ raffine $D_A$, alors $D_C$ raffine $D_A$), ce qui permet d'avoir un nombre arbitraire d'étapes dans un développement formel ;
- *monotonie* (si $D_C^1$ raffine $D_A^1$ et $D_C^2$ raffine $D_A^2$ alors la composition de $D_C^1$ et $D_C^2$ raffine la composition de $D_A^1$ et $D_A^2$), ce qui permet de faire des développements formels modulaires.

Ces quelques observations permettent d'ores et déjà de proposer des orientations quant à la définition d'un raffinement générique. Par exemple, la transitivité justifie l'utilisation d'un langage commun pour toutes les descriptions successives d'un système. Le fait de ne pas vouloir prendre en compte les algorithmes, et même de pouvoir comparer une spécification déclarative et une implémentation impérative, justifie une approche "boîte noire", purement extensionnelle (le système est perçu uniquement par ses interfaces, i.e. ses entrées et ses sorties).

Il est aussi intéressant de s'interroger sur les constituants typiques d'une étape de raffinement ; en effet, nous pouvons identifier plusieurs activités sous-jacentes.

Le *raffinement de données* consiste à définir ou modifier la représentation des données – par exemple utiliser des listes pour implanter des ensembles, ou passer d'une représentation à la *Peano* à une représentation binaire pour les entier naturels. Le *raffinement par choix* réduit le non déterminisme. Le *raffinement par complétion* est un concept proche qui consiste à étendre le domaine de définition (par exemple en décrivant le comportement de la fonction head, renvoyant la tête d'une liste, pour la liste vide). Le *raffinement par décomposition* introduit un nouveau niveau de détail dans la description ; cela peut correspondre à de nouvelles transitions et/ou états intermédiaires dans un automate, ou encore à décomposer un problème en sous-problèmes plus simples dans une approche modulaire. Le *raffinement opératif* associe à une description déclarative (totale et déterministe) une version impérative, i.e. un algorithme.

Cette liste de constituants n'est sans doute pas exhaustive, mais elle suffit pour discuter de nombreux aspects intéressants.

## 4.2 Quelques formes simplifiées de raffinement

Nous formalisons dans un premier temps des formes simplifiées de raffinements pour mettre en évidence des phénomènes intéressants – dans un paradigme fonctionnel pur (ce qui signifie par exemple que les états internes, tels que l'état d'une machine en B, sont transformés en paramètres explicites) et en utilisant des relations comme descriptions du système. Plus précisément, $S \leftrightarrow T$ est l'ensemble de relations entre $S$ et $T$, et les implémentations sont des relations fonctionnelles. Ici $\mathbf{dom}(R)$ designe le domaine de la relation $R$, $\mathbf{ran}(R)$ son image, $\equiv$ l'égalité extensionnelle, $S \lhd R$ la restriction sur le domaine $S$ de la relation $R$, $R \rhd S$ la restriction sur l'image $S$ de la relation $R$, et $R\{S\}$ la projection de $S$ par la relation $R$.

### 4.2.1 Raffinement de données

**Définition 4.1 (Raffinement de données)** *$f_A : D_A \to R_A$ raffine $f_C : D_C \to R_C$ modulo les interprétations $I_D : D_A \leftrightarrow D_C$ pour le domaine et $I_R : R_A \leftrightarrow R_C$ pour l'image ssi :*

$$f_A \approx f_C[I_D, I_R] \triangleq \forall (x_A : D_A)(x_C : D_C),\ (x_A, x_C) \in I_D \Rightarrow (f_A(x_A), f_C(x_C)) \in I_R$$

Intuitivement, si $x_A$ et $x_C$ représentent la même valeur, alors nous imposons que $f_A(x_A)$ et $f_C(x_C)$ représentent la même valeur.

Cette définition est une équivalence monotone, *i.e.* qu'elle vérifie les propriétés souhaitables pour un raffinement. Sa principale caractéristique est qu'elle met en évidence des paramètres du raffinement, les interprétations – qui représentent notamment le concept d'*invariant de collage* dans un raffinement en B – dont nous pouvons étudier l'influence.

Par exemple, il est clair que si l'interprétation $I_D$ est vide, le raffinement est trivial : n'importe quoi raffine n'importe quoi. En fait, avoir une interprétation $I_D$ partielle correspond à ne prendre en compte qu'une partie de la spécification $f_A$, *i.e.* à une forme de pré-condition. Mais nous ne pouvons éviter ce genre de situation, car nous n'avons pu identifier aucune contrainte pertinente sur $(I_D, I_R)$ : toutes nos tentatives de restreindre les interprétations acceptables interdisent certaines pratiques de développement jugées utiles.

Réciproquement, nous verrons comment certaines des attaques exposées auparavant peuvent être caractérisées par l'étude des interprétations associées.

### 4.2.2 Raffinement par choix

La seconde forme de raffinement simplifié considérée est le raffinement par choix – i.e. la réduction du non déterminisme – et par complétion.

Plusieurs formes différentes ont été envisagées, la difficulté étant de capturer simultanément les notions de pré-conditions et de gardes, toutes deux présentes dans les développements formels. Cela mène à décrire un système par une une relation $R$ associée à une précondition $P$ sur le domaine :

**Définition 4.2 (Raffinement par choix)** *Pour deux pré-conditions $p_A, p_C \subseteq D$ et deux relations $r_A, r_C \in D \leftrightarrow R$, $(p_C, r_C)$ raffine $(p_A, r_A)$ ssi :*

$$(p_A, r_A) \mid\!\succcurlyeq (p_C, r_C) \quad \triangleq \quad p_A \subseteq p_C \wedge p_A \lhd r_C \subseteq r_A$$

Cette définition est, sous certaines conditions, un ordre monotone :

**Proposition 4.1 (Monotonie du raffinement par choix)**

$$(p_A, r_A \rhd p'_A) \mid\!\succcurlyeq (p_C, r_C) \Rightarrow (p'_A, r'_A) \mid\!\succcurlyeq (p'_C, r'_C) \Rightarrow (p_A, r'_A \circ r_A) \mid\!\succcurlyeq (p_C, r'_C \circ r_C)$$

La condition sur la monotonie est issue de la preuve faite en Coq, mais a une signification intuitive très intéressante. Elle indique notamment qu'il est délicat de composer des sous-systèmes dont le domaine de définition est partiel.

## 4.3 Raffinement générique

### 4.3.1 Définition

Nous combinons les deux définitions précédentes pour définir un raffinement qui permet simultanément le raffinement de données, le raffinement par choix, le raffinement de complétion, la raffinement opératoire ; le raffinement par décomposition est également rendu possible par l'exploitation des résultats de monotonie donnés plus loin.

**Définition 4.3 (Raffinement générique)** *Pour une spécification formée d'une précondition $p_A \subseteq D_A$ et d'une relation $r_A \in D_A \leftrightarrow R_A$, et une implémentation formée d'une pré-condition $p_C \subseteq D_C$ et d'une relation $r_C \in D_C \leftrightarrow R_C$, $(p_C, r_C)$ raffine $(p_A, r_A)$ modulo les interprétations $I_D : D_A \leftrightarrow D_C$ et $I_R : R_A \leftrightarrow R_C$ ssi:*

$$(p_A, r_A) \rightsquigarrow (p_C, r_C)[I_D, I_R] \quad \triangleq \quad I_D\{p_A\} \subseteq p_C \ \wedge \ p_A \lhd (r_C \circ I_D) \subseteq I_R \circ r_A$$

Nous avons pu expérimenter cette définition avec différents exemples typiques :

- le *Maximier* (une illustration classique de la méthode B décrite dans le B-Book) avec des interprétations associant à une liste de valeurs naturelles son maximum ;
- le raffinement *partiel* de l'addition sur une représentation de *Peano* (non bornée) par un algorithme sur des représentations (finies) sur 4 bits ;
- La représentation des *miracles*, *i.e.* des descriptions qui raffinent tout et ne sont raffinées que par des miracles (correspondants à une spécification insatisfiable).

### 4.3.2 Propriétés

**Proposition 4.2 (Transitivité du raffinement générique)**

$$(p_A, r_A) \rightsquigarrow (p_I, r_I)[I_D, I_R] \Rightarrow (p_I, r_I) \rightsquigarrow (p_C, r_C)[I'_D, I'_R] \Rightarrow (p_A, r_A) \rightsquigarrow (p_C, r_C)[I'_D \circ I_D, I'_R \circ I_R]$$

*Un résultat de monotonie est également donné dans le paragraphe 4.4.1.*

Comme attendu, le raffinement de données et le raffinement par choix sont des cas particuliers de raffinement générique :

**Proposition 4.3 (Relations entre les raffinements)**

$$f_A \approx f_C[I_D, I_R] \Rightarrow p_A \subseteq \mathbf{dom}(I_D) \Rightarrow \mathbf{ran}(I_D) \subseteq p_C \Rightarrow (p_A, f_A) \rightsquigarrow (p_C, f_C)[I_D, I_R]$$

$$(p_A, r_A) \mid\!\succcurlyeq (p_C, r_C) \Rightarrow (p_A, r_A) \rightsquigarrow (p_C, r_C)[\mathrm{id}, \mathrm{id}]$$

## 4.4 Exploitation du raffinement générique

### 4.4.1 Le compositeur distrait

Un chef d'équipe doit faire développer une implémentation conforme à une spécification $(p_A, r_A)$ (avec $p_A \subseteq D_A$ and $r_A \in D_A \leftrightarrow R_A$), modulo les interprétations $I_D : D_A \leftrightarrow D_C$ et $I_R : R_A \leftrightarrow R_C$. Il décide de décomposer le problème, choisit un support abstrait intermédiaire $M_A$, et rédige deux nouvelles spécifications $(p_A, r_A^1)$ avec $r_A^1 : D_A \leftrightarrow M_A$ et $(p_A^2, r_A^2)$ avec $p_A^2 \subseteq M_A$ et $r_A^2 : M_A \leftrightarrow R_A$ telles que $r_A = r_A^2 \circ r_A^1$.

Il demande à deux équipes indépendantes d'implémenter ces spécifications. Il fournit également $I_D$ à la première équipe et $I_R$ à la seconde mais oublie de fournir une interprétation intermédiaire commune $I_M$, et pire encore il n'impose même pas un type de données concret commun. Après quelques temps, il reçoit donc :

- Un raffinement $(p_C^1, r_C^1)$, avec $p_C^1 \subseteq D_C$ et $r_C^1 : D_C \leftrightarrow M_C^1$ ;
- Une preuve de conformité $(p_A, r_A^1) \rightsquigarrow (p_C^1, r_C^1)[I_D, I_M^1]$ avec $I_M^1 : M_A \leftrightarrow M_C^1$ ;
- Un raffinement $(p_C^2, r_C^2)$, avec $p_C^2 \subseteq M_C^2$ et $r_C^2 : M_C^2 \leftrightarrow R_C$
- Une preuve de conformité $(p_A^2, r_A^2) \rightsquigarrow (p_C^1, r_C^1)[I_M^2, I_R]$ avec $I_M^2 : M_A \leftrightarrow M_C^2$.

Bien entendu, il découvre que $M_C^1 \neq M_C^2$ :ces raffinements ne peuvent pas être composés. Il cherche donc à spécifier un programme de *collage* $G$ tel que $r_C^2 \circ G \circ r_C^1$ soit bien typé et raffine la spécification initiale ; le résultat de monotonie donne cette spécification :

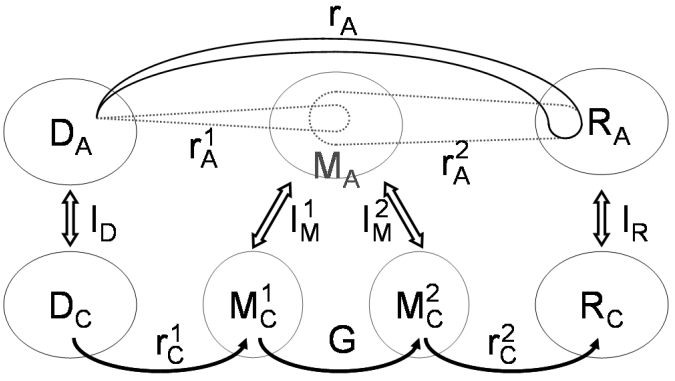
**Proposition 4.4 (Monotonie étendue du raffinement générique)**

$$(p_A, r_A^1 \rhd \mathbf{dom}(p_A^2 \lhd I_M^2)) \rightsquigarrow (p_C^1, r_C^1)[I_D, I_M^1] \quad \Rightarrow$$

$$(p_A^2, r_A^2) \rightsquigarrow (p_C^2, r_C^2)[I_M^2, I_R] \quad \Rightarrow$$

$$G \circ I_M^1 \subseteq I_M^2 \quad \Rightarrow$$

$$(p_A, r_A^2 \circ r_A^1) \rightsquigarrow (p_C, r_C^2 \circ G \circ r_C^1)[I_D, I_R]$$




Il est intéressant de noter que la condition sur $G$ impose que ce soit un raffinement de l'identité modulo les interprétations $I_M^1$ and $I_M^2$. Effectivement, notre chef d'équipe, considérant calmement la situation, pouvait réecrire $r_A = r_A^2 \circ id \circ r_A^1$. $G$ est alors effectivement un composant intermédiaire dans une vision standard du résultat de monotonie.

Cela mène à remarquer qu'il y a de nombreux raffinements de l'identité, représentant par exemple des traducteurs entre représentations, ou encore des mécanismes correcteurs d'erreurs lorsque les interprétations modélisent l'apparition d'erreurs.

### 4.4.2 Le paradoxe du raffinement

Le paradoxe du raffinement est le nom donné à certains phénomènes *a priori* surprenants que l'on peut observer par exemple en B :

**Exemple 4.1 (Paradoxe du raffinement en** B**)** *La spécification décrit une machine très simple avec une opération* getbool *retournant un booléen :*

**MACHINE** bool_choice
**OPERATIONS** $b \leftarrow \text{getbool} \triangleq b := \top \,[]\, b := \bot$

*Il est courant de faire l'erreur de penser que cette machine n'a que deux raffinements, celui pour lequel* getbool *renvoie toujours* $\top$*, et celui pour lequel* getbool *renvoie toujours* $\bot$*. En pratique d'autres raffinements sont possibles, par exemple :*

**MACHINE** covert_channel
**REFINES** bool_choice
**VARIABLES** secret
**INVARIANT** $\text{secret} \in \mathbb{N}$
**OPERATIONS** $b \leftarrow \text{getbool} \triangleq b := (\text{secret}\%2 = 0); \text{secret} := \text{secret}/2$

*Le "paradoxe" est qu'on peut être surpris par l'introduction d'une dépendance entre la sortie de* getbool *et une valeur* secret *qui n'apparaissait même pas dans la spécification. Pour*

*autant, le paradoxe n'est qu'apparent ; la spécification n'interdit pas une telle dépendance, et en fait le langage du* B *ne permet pas de décrire de telles contraintes.*

En utilisant le raffinement générique, cependant, nous pouvons noter que ce raffinement malicieux est associé à une interprétation très particulière. En effet, moralement, l'état de la machine *bool_choice* est **Unit** (type contenant une seule valeur), et est raffiné par $\mathbb{N}$. Non seulement cette interpétation n'est pas fonctionnelle, mais en pratique elle introduit une nouvelle dimension, c'est à dire une variable cachée. La signature de l'attaque, ici, est clairement donnée par la combinaison d'une spécification non déterministe et une interprétation non fonctionnelle. Cela nous permet aussi de donner une illustration du paradoxe du raffinement dans d'autres méthodes où il n'existe normalement pas :

**Exemple 4.2 (Paradoxe du raffinement en** FoCaLize**)**

**species** $\Omega \triangleq$
  **signature** $(=) : \mathbf{Self} \to \mathbf{Self} \to \mathbb{B}$
  **property** $=_{\text{refl}} : \forall\, s : \mathbf{Self},\ s = s$
  **property** $=_{\text{symm}} : \forall\, s_1, s_2 : \mathbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_1$
  **property** $=_{\text{tran}} : \forall\, s_1, s_2, s_2 : \mathbf{Self},\ s_1 = s_2 \Rightarrow s_2 = s_3 \Rightarrow s_1 = s_3$
**end**

**species** mybool $\triangleq$
  **inherit** $\Omega$;
  **signature** true, false : **Self**;
  **property** surjective : $\forall\, b$ : **Self**, $b = \text{true} \vee b = \text{false}$;
  **property** injective : $\text{true} \neq \text{false}$;
  **signature** bool_function : **Self** $\to \mathbb{B}$;
**end**

*L'implémentation triviale est d'utiliser* $\mathbb{B}$ *comme support de l'espèce, et l'une des* 4 *fonctions booléennes pour* bool_function*.*

*Mais il est aussi possible d'avoir des implémentations malicieuses, par exemple en utilisant* int *comme représentation, les valeurs paires encodant* true *et les valeurs impaires encodant* false*, et en implémentant* bool_fun *de manière inattendue, par exemple par la fonction retournant* $\top$ *lorsque son paramètre est un carré,* $\bot$ *sinon.*

## 4.5 Quelques remarques

Le raffinement, tel que décrit ici, est un concept fondamental dans les méthodes formelles déductives, qui permet de capturer les pratiques de développement tout en garantissant la conformité. La définition formelle d'un raffinement se doit d'être aveugle à certains aspects des descriptions des systèmes, et il est possible à un développeur malicieux d'abuser de cet aveuglement pour introduire des comportements inattendus.

Plusieurs approches sont alors possibles pour interdire ce genre d'attaques. Plutôt que de contraindre la spécification, ou de modifier la définition du raffinement – deux approches peu pertinentes ici en raison des impacts pratiques – notre intention est de proposer une vision différente, plus intuitive. La mise en évidence des interprétations comme paramètres du raffinement apporte de nombreux éclaircissements.

# 5 Vérification d'une Méthode Formelle

L'utilisation des méthodes formelles apporte un meilleur niveau d'assurance, mais quel est exactement ce gain lorsque la théorie ou les outils ne sont pas formellement vérifiés eux-mêmes ? En sécurité en particulier, un paradoxe dans la théorie ou le prouveur peut être exploité par un développeur malicieux pour obtenir un certificat tout en implémentant des comportements indésirables.

Nous nous intéressons ici à la problématique de la validation de la théorie et des outils pour la méthode B, qui est fréquemment utilisée dans l'industrie pour les développements de sécurité (cf. par exemple [Jaf07, SL00, Bie96, HHGB07]). Pour cette validation, nous avons développé BiCoq, un plongement profond de la logique du B en Coq, permettant de vérifier les résultats connus, de développer un prouveur prouvé, mais aussi de démontrer de nouveaux résultats.

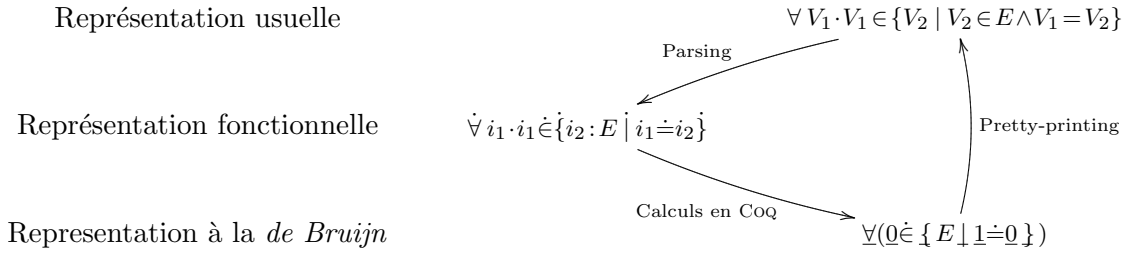
Il faut noter que l'effort associé à ce développement est relativement important. Les différentes versions de BiCoq représentent plusieurs dizaines de milliers de lignes. En effet, nous avons aussi cherché à définir des représentations efficaces, optimiser les méthodes de preuves, etc. Ces éléments, plus techniques, ne sont pas donnés dans ce résumé.

## 5.1 Plonger la logique de B

### 5.1.1 Syntaxe

La première étape du plongement consiste à représenter par un type Coq les termes syntaxiquement corrects de la logique du B-Book. Notre plongement, bien que se voulant précis pour limiter les déviations, introduit plusieurs adaptations dans la représentation de cette syntaxe. Ces adaptations ont été jugées acceptables et nécessaires, et sont clairement identifiées. La plus importante correspond à l'utilisation d'une représentation basée sur des indices de *de Bruijn*, pour traiter correctement les problèmes de noms de variables. Bien que très visible, cette adaptation n'est en pratique qu'une solution technique sans impact sémantique réel ; au-delà de la simplicité qu'elle apporte dans l'implémentation, elle a aussi permis d'identifier quelques erreurs mineures dans les définitions standard du B qui peuvent mener par exemple à des problèmes de captures involontaires.

Pour conserver une relative ergonomie dans le développement, nous avons également introduit dans le plongement des fonctions permettant de passer facilement d'une représentation naturelle des termes du B à la représentation de *de Bruijn*. Elles définissent un sucre syntaxique permettant d'utiliser ce que nous appelons une *représentation fonctionnelle* :




D'autres modifications ont été apportées, par exemple la fusion dans une sorte syntaxique unique des expressions et ensembles B, ou encore une prise en compte dans la syntaxe des règles de bonne formation (appelées *type-checking* dans le B-Book). Par exemple les ensembles en compréhension $\{V|P\}$ doivent en fait être de la forme $\{V \mid V \in S \wedge P\}$ avec $V$ une variable n'apparaissant pas libre dans $S$, pour éviter le paradoxe de *Russell*.

Dernière modification importante, nous n'avons pas représenté dans les sortes syntaxiques les constructions de la forme $[V := E]T$, qui correspondent à l'application de l'opération d'affectation. Pour diverses raisons, nous pensons effectivement que cette approche introduit une confusion entre deux concepts très différents, la sémantique de l'affectation comme transformateur de prédicat, et une opération de substitution qui relève du méta-langage de la logique. Nous distinguons dans notre développement ces deux constructions, la seconde étant encodé sous la forme d'une fonction Coq opérant sur les termes B.

### 5.1.2 Règles d'inférence

L'étape suivante consiste à représenter les règles d'inférence de la logique du B. Pour cela, nous représentons chaque règle sous la forme de un ou plusieurs constructeurs d'un prédicat inductif Coq. Ce prédicat permet en fait de distinguer les séquents du B qui sont prouvables:

- $\Gamma \dot{\vdash} P$ signifie qu'il existe une preuve B de $P$ sous les hypothèses $\Gamma$ ;
- $\Gamma \dot{\vdash} \dot{\neg} P$ signifie qu'il existe une preuve B de $\dot{\neg} P$ sous les hypothèses $\Gamma$ ;
- $\neg(\Gamma \dot{\vdash} P)$ signifie qu'il n'y a aucune preuve B de $P$ sous les hypothèses $\Gamma$.

Grâce à la représentation fonctionnelle qui émule la représentation naturelle du B, la traduction des règles d'inférence est un exercice syntaxique simple – et aisément justifiable par exemple vis à vis d'un relecteur sceptique.

Quelques adaptations ont cependant été nécessaires. L'une d'elle concerne la règle d'inférence indiquant que l'ensemble **BIG** est infini ; cette règle repose sur une définition complexe basée sur un point fixe qu'il est difficile de plonger. Elle a été remplacée par une règle montrant que l'ensemle **BIG** dans notre plongement contient au moins une infinité dénombrable de valeurs, en utilisant les naturels de Coq.

D'autres adaptations ont été nécessaires, cette fois pour corriger des *insuffisances* de la logique de B, identifiées pendant la validation, comme indiqué dans la paragraphe suivant.

### 5.1.3 Validation des résultats standards

À partir du plongement de la syntaxe et des règle d'inférence, il est possible de vérifier les théorèmes du B-Book. Cela a été fait pour l'ensemble des résultats pour le calcul propositionnel, la plupart des résultats pour le calcul des prédicats, et quelques résultats pour les constructions ensemblistes – en mettant en évidence quelques problèmes.

La notion d'union de deux ensembles par exemple, telle que définie en B, ne peut pas être plongée de manière satisfaisante. En effet, elle dépend de paramètres implicites, et nous n'avons pas trouvé de solution générique pour représenter cette construction de manière fidèle. Il faut également noter que la définition du B-Book permet des captures accidentelles de variables, ce qui n'est clairement pas souhaitable. Des observations similaires peuvent être faites à propos d'autres constructions ensemblistes, ce qui justifie le fait que peu de résultats ont finalement pu être vérifiés.

D'autres petits problèmes ont été observés. Par exemple les quantifications multiples, souvent utilisées, ne sont pas toujours conformes à la syntaxe ou aux règles de bonne formation, certains symboles sont utilisés indifféremment pour représenter des constructions syntaxiques distinctes, etc.

L'observation la plus intéressante est que *plusieurs théorèmes du* B-Book *ne sont en fait pas prouvables avec les règles d'inérence proposées* – les règles relatives au produit

cartésien et aux paires étant insuffisantes. Il est clair que seul un plongement profond permet de mettre en évidence ce défaut, apparemment non connu de la communauté B ; il faut également noter que ces théorèmes sont vérifiables avec les prouveurs B, ce qui indique qu'ils sont admis comme des axiomes.

## 5.2 Un prouveur prouvé

Le plongement profond, qui a permis la validation de la théorie, a également été utilisé pour développer un prouveur prouvé.

En pratique, nous avons intégré dans BiCoq des fonctions écrites dans le langage Ml de Coq représentant l'application de règles d'inférence ou de théorèmes. Ces fonctions prennent en paramètre un séquent syntaxique du B (représentant le but), et produisent une liste de séquents (les sous-buts) ; elles sont prouvées correctes, et en fournissant au moins une fonction par règle d'inférence nous obtenons un prouveur correct et complet.

Bien entendu, il n'est pas directement utilisable en l'état, ne fournissant que peu d'automatisation et aucune interface homme-machine ; il peut cependant être couplé avec d'autres outils existants, par exemple la plate-forme Brillant [CPR+05], ou agir comme vérificateur de preuves fournies par d'autres prouveurs, sur la base d'un langage commun (qui reste à définir).

## 5.3 De nouveaux résultats pour le B

Au-delà de la validation de la théorie et des outils, un plongement comme BiCoq donne aussi l'opportunité de démontrer formellement de nouveaux résultats complexes ; nous illustrons cette démarche par la preuve de nouveaux principes de congruence pour la logique du B. La logique du B définit en effet une unique règle de congruence :

**Définition 5.1 (Utilisation de l'égalité en B)**

$$\frac{\Gamma \vdash E_1 = E_2 \quad \Gamma \vdash [V := E_1]P}{\Gamma \vdash [V := E_2]P}$$

Bref, si $E_1 = E_2$, toute occurrence de $E_1$ peut être remplacée par une occurrence de $E_2$.

Cette règle d'inférence a deux limitations importantes : elle ne concerne que les expressions, et non les prédicats équivalents, et d'autre part elle ne permet pas le remplacement de sous-termes dont les variables libres sont liées par le contexte.

Nous introduisons donc dans un premier temps une opération de substitution permettant de remplacer une variable propositionelle – une nouvelle construction ajoutée à la syntaxe de la logique de B – par un prédicat, avant de prouver les résultats suivants :

**Proposition 5.1 (Remplacement de prédicats équivalents)**

$$\Gamma \dot{\vdash} P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a \backslash P_1 \rangle Q \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a \backslash P_2 \rangle Q$$

$$\Gamma \dot{\vdash} P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a \backslash P_1 \rangle Q \Leftrightarrow \langle a \backslash P_2 \rangle Q$$

$$\Gamma \dot{\vdash} P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \dot{\vdash} \langle a \backslash P_1 \rangle E \doteq \langle a \backslash P_2 \rangle E$$

Ces résultats, bien qu'intéressants (en pouvant considérablement simplifier certaines preuves), souffrent encore d'une limitation importante : ils ne permettent pas de justifier le remplacement d'un sous-terme dont certaines variables libres sont capturées par le contexte.

Nous définissons donc d'autres opérations, des substitutions permettant la capture de variables libres ; ces opérations sont nommées greffe. Pour ces opérations, nous montrons que sous certaines conditions, il est par exemple effectivement possible de remplacer une expression dont certaines variables sont capturées par une autre expression égale :

**Proposition 5.2 (Remplacement de sous-termes liés)**

$$\Gamma \vdash E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \perp_E E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \vdash [i \lhd E_1]E \doteq [i \lhd E_2]E$$

$$\Gamma \vdash E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \perp_P E_1 \doteq E_2 \quad \Rightarrow \quad \Gamma \vdash [i \lhd E_1]P \Leftrightarrow [i \lhd E_2]P$$

$$\Gamma \vdash P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \perp_E P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \vdash \langle n \lhd P_1 \rangle E \doteq \langle n \lhd P_2 \rangle E$$

$$\Gamma \vdash P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \perp_P P_1 \Leftrightarrow P_2 \quad \Rightarrow \quad \Gamma \vdash \langle n \lhd P_1 \rangle P \Leftrightarrow \langle n \lhd P_2 \rangle P$$

$\Gamma \perp_F T$ est la condition d'orthogonalité ; intuitivement, elle exige que toute variable apparaissant simultanément libre dans $\Gamma$ et $T$ ne soit pas capturée dans $F$.

L'intérêt de ces résultats est par exemple de justifier le dépliage sous des lieurs de définitions conditionnelles telle que la division.

# 6 Conclusion et perspectives

## 6.1 Identifier et éviter les pièges

Ce mémoire démarrait par une question concernant la portée et le niveau de la confiance résultant de l'utilisation des méthodes formelles déductives pour le développement de systèmes sécurisés.

Cette question, motivée par quelques expériences personnelles mettant en évidence la difficulté de spécifier certaines propriétés de sécurité, nous a mené à faire une analyse détaillée du processus de développement formel. Et, en adoptant une vision certes très critique, il y a effectivement de nombreux problèmes qui peuvent découler de spécifications incohérentes ou simplement insuffisantes, ou encore d'usages inattendus – quelle que soit la méthode formelle utilisée.

Cependant, au-delà d'illustrations de conséquences possibles de spécifications inappropriées, du paradoxe du raffinement ou encore d'hypothèses trop optimistes et non contrôlées, nous avons aussi tenté de comprendre les causes fondamentales, de proposer des recommandations à l'attention des développeurs et des évaluateurs, ou encore de débattre de solutions techniques envisageables.

Nous avons mis en évidence pourquoi certaines pratiques habituelles en sûreté de fonctionnement doivent être reconsidérées dans le cadre du développement d'applications de sécurité. Par exemple, les préconditions dans une spécification ne sont pas adaptées pour garantir la sécurité, et il est préférable d'utiliser des gardes. Nous avons justifié en quoi l'utilisation de certains outils standards, par exemple l'animation de modèles ou la génération de tests, permettent d'assister l'analyse de la qualité d'une spécification.

Nous avons aussi illustré comment des spécifications inappropriées peuvent aussi résulter d'une mauvaise compréhension de certains aspects de la théorie sous-jacente. Pour mieux comprendre certains phénomènes subtils, nous avons proposé une vision intuitive et générique du raffinement qui permet de mettre en évidence comment un développeur malicieux peut par exemple introduire des dépendances cachées dans un développement prouvé, que ce soit en B, en Coq ou en FoCaLize – et en fait dans toute méthode formelle permettant une forme de raffinement.

Il faut noter que cette formalisation du concept de raffinement ne vise pas à proposer une approche alternative et spécialisée pour la sécurité. En effet, en prenant en compte la perspective d'une utilisation industrielle des méthodes formelles, il ne nous paraît par opportun d'introduire des concepts relatifs à l'observabilité ou à l'aléa (concept très différent du non déterminisme). Bien que de telles démarches aient une justification académique indéniable, l'accroissement de la complexité aurait un coût en termes de formation et d'utilisation potentiellement inacceptable dans l'industrie, sans même parler de la disponibilité d'outils associés.

Notre approche, au contraire, est de favoriser la compréhension de la méthode utilisée et de mettre en évidence les gains pouvant résulter de l'utilisation, en parallèle, d'autres approches (souvent automatisées). Par exemple, un calcul de dépendances permet de traiter les problèmes relatifs à la confidentialié et aux canaux cachés bien plus efficacement que des contraintes artificielles sur la forme des spécifications ou une modification de la théorie du raffinement. Cela justifie pleinement selon nous la mise en place d'environnements de développement (IDE) intégrant différentes approches, un des objectifs de FOCALIZE.

## 6.2 Valider la théorie et les outils

Ayant noté les conséquences potentielles de paradoxes dans la théorie ou les outils formels, nous avons également analysé dans ce mémoire la problématique de la validation d'une méthode formelle, en l'illustrant par le plongement de la logique du B dans l'assistant de preuve COQ.

Une telle validation n'est pas un problème simple, et son coût est loin d'être négligeable. Cependant, nous avons montré qu'une telle démarche est réalisable mais aussi nécessaire, en identifiant plusieurs omissions ou défauts dans la théorie de la logique du B. La validation d'une théorie formelle nous paraît relever pleinement de la communauté académique : elle doit être faite une et une seule fois, pour le bénéfice de toute la communauté formelle, y compris les utilisateurs industriels.

Cette validation doit en pratique être mécanisée, mais plusieurs outils parfaitement adaptés à cette tâche existent, et au-delà de la confiance apportée leur utilisation apporte d'autres avantages. Dans notre cas par exemple, nous avons bénéficié de l'utilisation de COQ non seulement pour la validation de la logique B, mais aussi pour explorer différentes formes de représentations des langages, pour développer des outils formels certifiés, et enfin pour prouver de nouveaux résultats complexes. Avec le soutien d'un tel assistant de preuve, nous avons été capables d'aborder en toute confiance des questions complexes – par exemple la substitution de sous-termes liés – les détails administratifs étant pris en charge par l'environnement formel. De manière plus générale de tels outils nous semblent pertinents pour traiter efficacement de théories mathématiques.

Enfin, concernant les outils permettant la mise en œuvre des méthodes formelles, nous avons noté que la correction et la complétude d'un prouveur par exemple ne sont pas les seuls objectifs à prendre en compte. En effet, il est aussi justifié d'attendre de ces outils une certaine simplicité de mise en œuvre, une bonne ergonomie et un niveau d'automatisation élevé – ici encore pour favoriser l'utilisation des méthodes formelles dans l'industrie. Cela justifie selon nous pleinement la coexistence, et en fait la coopération, entre différents outils et approches.

## 6.3 Quelques perspectives

Ce mémoire propose quelques activités complémentaires qui n'ont pas pu être menées faute de temps. Nous proposons ici un rapide résumé des perspectives identifiées.

L'intégration efficace de plusieurs approches dans un environnement formel pour permettre des développements de systèmes sécurisés, par exemple, constituerait une amélioration importante. Il s'agit d'une tendance affirmée par exemple au niveau de la communauté des utilisateurs de B – quoiqu'en pratique elle semble justifíee par des considérations d'efficacité et d'ergonomie plutôt que de sécurité – et reste l'un des grands objectifs du projet FoCaLize. Si nous ne sommes pas convaincus que l'intégration sous la forme d'une théorie unifée (pour le raffinement par exemple) serait exploitable dans un contexte industriel, la capacité de combiner plusieurs analyses orthogonales, telles que des preuves de conformité et des analyses de dépendances, devrait permettre d'améliorer notablement la sécurité intrinsèque à un coût minimal.

Au-delà des techniques existantes, il nous semble également que certaines approches novatrices pourraient être proposées pour l'intégration dans de tels environnements formels. Ainsi, le calcul de dépendance est une analyse menée au niveau implémentation, et permet par exemple à un évaluateur indépendant d'identifier des flux d'information inattendus. Il serait cependant intéressant de pouvoir spécifier les dépendances autorisées ou interdites à un niveau plus abstrait et de garantir la transformation de ces contraintes pendant le raffinement. Cela constituerait un outil pertinent en sûreté (analyse de propagation des fautes) comme en sécurité (mise en évidence de l'exploitation de canaux cachés).

Par ailleurs, des outils d'assistance à la rédaction et la validation de spécifications restent selon nous à développer. De tels outils permettraient de porter un regard critique sur certaines propriétés, en générant des obligations de preuves complémentaires (pour détecter des spécifications trivialement correctes), en produisant des contre-exemples (mettant en évidence des incohérences ou des insuffisances), etc. Il devrait aussi être possible de transformer, ou tout au moins de fournir une assistance pour la transformation de la spécification formelle, en totalité ou en partie, en gardes intégrées directement dans l'implémentation.

Nous avons également débattu du problème de la validation de la théorie et des outils associés à une méthode formelle. Dans le cadre de BiCoq de nombreuses activités restent à mener, par exemple valider effectivement la cohérence de la logique du B ou étudier le rôle précis des mécanismes de type-checking décrits dans le B-Book. L'intégration de BiCoq avec d'autres environnements B semble aussi souhaitable, notamment comme nouvelle base robuste pour l'intégration de plongements superficiels existants, ou encore comme un certificateur de preuves fournies par d'autres prouveurs B. Le développement de nouveaux résultats pour le B, à l'image des propriétés de congruence, constitue aussi une perspective intéressante permettant par exemple de proposer des tactiques de preuves plus efficaces, et à terme d'améliorer l'automatisation des prouveurs pour le B.

Enfin, bien que cela ne soit pas abordé dans ce résumé, nos travaux nous ont aussi mené à développer et comparer différentes formes de représentations à la *de Bruijn* pour les termes, ainsi que des opérations associées. Différentes améliorations ont été proposées, qui semblent avoir des applications dans le domaine de la mécanisation des langages et de la logique. Par exemple, nous avons présenté un $\lambda$-calcul simplement typé pour lequel il n'est pas nécessaire de gérer un contexte de typage, dont les propriétés restent à étudier. Par ailleurs, nous pensons que le développement d'une librairie pour la mécanisation des langages qui combinerait différentes représentations tout en fournissant les outils permettant le passage de l'une à l'autre (par des homomorphismes) fournirait une environnement intéressant permettant pour tout problème de choisir la représentation la plus appropriée.

### 6.4 Intérêt des méthodes formelles

La présentation et l'analyse des méthodes formelles proposée dans ce mémoire est naturellement biaisée, considérant des scénarios "pire cas" et des exemple bien choisis visant à illustrer nos propos. Cela nous mène à faire de nombreuses remarques sur les pièges auxquels tout développement formel d'un système sécurisé est confronté.

Comme nous l'avons indiqué, cependant, nous considérons que les méthodes formelles en général, et les méthodes formelles déductives en particulier, constituent des outils puissants qui permettent d'atteindre de très hauts niveaux d'assurance – notamment en permettant le développement de systèmes certifiés sans erreurs d'implémentation.

Selon nous, les méthodes formelles fournissent donc une "*silver bullet*" pour l'ingénierie des systèmes, et une utilisation appropriée permet de détecter et d'éradiquer les défauts dans les protocoles, les non conformités dans les programmes, ou d'autres formes de problèmes tels que les débordements de tampons mémoire, les accès hors bornes dans les tableaux, ou les exceptions dues à des opérations invalides. Un autre apport des méthodes formelles déductives, souvent ignoré, est de permettre d'obtenir une authentique compréhension du système développé : prouver un programme, au-delà des aspects de certification, reste un moyen remarquable de reconsidérer et justifier sa conception.

De notre point de vue, les méthodes formelles restent insuffisamment exploitées dans l'industrie. Elles ont en effet la réputation d'être complexes et coûteuses à mettre en œuvre, et pour cette raison leur utilisation reste cantonnée au développement de systèmes critiques, avec comme principal objectif le processus de certification – c'est à dire que les méthodes formelles semblent perçues comme un moyen de convaincre les évaluateurs indépendants et les autorités de certification, et non comme un outil pour améliorer la robustesse des systèmes.

Le fait est que la complexité réelle d'un développement formel n'est pas tant liée à la méthode formelle elle-même – dès lors bien sûr qu'une formation appropriée a été donnée – mais bien au système développé lui-même. Il est clair par exemple que la difficulté d'un développement logiciel est trop souvent sous-estimée ; les méthodes formelles sont alors simplement coupables de mettre en évidence cette complexité cachée en exigeant la fourniture de justifications, et en interdisant toute forme d'omission.

Quant au coût de mise en oeuvre, différentes études montrent que l'usage approprié des méthodes formelles permet en fait de réduire de manière significative le coût global, prenant en compte non seulement les coûts de développement mais aussi les coûts de vérification et de maintenance – tout au moins pour les systèmes critiques.

Nous avons décidé, pour le travail décrit dans le mémoire, d'utiliser systématiquement des méthodes formelles pour mécaniser nos analyses et certifier nos développements. Cela illustre clairement selon nous l'intérêt et la faisabilité d'une démarche formelle. Nous espérons que cette pratique donnera un peu plus de poids à certaines de nos analyses et recommandations.

## References

[ABF+05] B.E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The POPLmark challenge. In Hurd and Melham [HM05], pages 50–65.

[ABHR99] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In ACM, editor, *POPL '99. Proceedings of the 26th*

*ACM SIGPLAN-SIGACT on Principles of programming languages, January 20–22, 1999, San Antonio, TX*, pages 147–160, New York, NY, USA, 1999. ACM Press.

[ABM01] Alessandro Avellone, Marco Benini, and Ugo Moscato. How to avoid the formal verification of a theorem prover. *Logic Journal of the IGPL*, 9(1), 2001.

[Abr96] J. R. Abrial. *The B-Book - Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[ACCL91] M. Abadi, L. Cardelli, P-L. Curien, and J-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

[ACM05] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in Event-B. In Treharne et al. [TKHS05], pages 222–241.

[ACPM05] June Andronick, Boutheina Chetali, and Christine Paulin-Mohring. Formal verification of security properties of smart card embedded source code. In Fitzgerald et al. [FHT05], pages 302–317.

[acs96] *13th Annual Computer Security Applications Conference (ACSAC 1996), 9-13 December 1996, San Diego, CA, USA*. IEEE Computer Society, 1996.

[AHN08] Klaus Aehlig, Florian Haftmann, and Tobias Nipkow. A compiled implementation of normalization by evaluation. In Mohamed et al. [MMT08], pages 39–54.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[AP02] A. Azurat and I.S.W.B. Prasetya. A survey on embedding programming logics in a theorem prover. Technical Report UU-CS-2002-007, Institute of Information and Computing Sciences, Utrecht University, 2002.

[Bac81] R-J. Back. On correct refinement of programs. *J. Comput. Syst. Sci.*, 23(1):49–68, 1981.

[Bac88] R-J. Back. A calculus of refinements for program derivations. *Acta Inf.*, 25(6):593–624, 1988.

[Bar99] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, November 1999.

[BAW98] Ralph-Johan J. Back, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[BBBB08] Egon Börger, Michael J. Butler, Jonathan P. Bowen, and Paul Boca, editors. *Abstract State Machines, B and Z, First International Conference, ABZ 2008, London, UK, September 16-18, 2008. Proceedings*, volume 5238 of *Lecture Notes in Computer Science*. Springer, 2008.

[BBHR02] Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson, editors. *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, France, January 23-25, 2002, Proceedings*, volume 2272 of *Lecture Notes in Computer Science*. Springer, 2002.

[BBM98] Patrick Behm, Lilian Burdy, and Jean-Marc Meynadier. Well defined B. In Bert [Ber98], pages 29–45.

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.

[BC06] Mike Bond and Jolyon Clulow. Integrity of intention (a theory of types for security APIs). *Information Security Technical Report*, 11(2):93 – 99, 2006.

[BCM07] Nazim Benaïssa, Dominique Cansell, and Dominique Méry. Integration of security policy into system modeling. In Julliand and Kouchnarenko [JK06], pages 232–247.

[BDD07] Richard Bonichon, David Delahaye, and Damien Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In Dershowitz and Voronkov [DV07], pages 151–165.

[BDFF04] K. Berkani, C. Dubois, A. Faivre, and J. Falampin. Validation des règles de base de l'Atelier B. *Technique et Science Informatiques*, 23(7):855–878, 2004.

[BDGK00] Jonathan P. Bowen, Steve Dunne, Andy Galloway, and Steve King, editors. *ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29 - September 2, 2000, Proceedings*, volume 1878 of *Lecture Notes in Computer Science*. Springer, 2000.

[BDL01] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *J. Cryptology*, 14(2):101–119, 2001.

[BDM98] P. Behm, P. Desforges, and J. M. Meynadier. MÉTÉOR : An industrial success in formal development. In Bert [Ber98], page 26.

[Ber98] D. Bert, editor. *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, volume 1393 of *Lecture Notes in Computer Science*. Springer, 1998.

[BF02] J.P. Bodeveix and M. Filali. Type synthesis in B and the translation of B to PVS. In Bert et al. [BBHR02], pages 350–369.

[BFM99] J.-P. Bodeveix, M. Filali, and C. Muñoz. A formalization of the B-Method in Coq and PVS. In *Electronic Proceedings of the B-User Group Meeting at the World Congress on Formal Methods FM 99*, pages 33–49, 1999.

[BGG+92] R. J. Boulton, A. Gordon, M. J. C. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with embedding hardware description languages in HOL. In Stavridou et al. [SMB92], pages 129–156.

[Bie96] P. Bieber. Formal techniques for an ITSEC-E4 secure gateway. In *ACSAC* [acs96], pages 236–246.

[Boe10] Mathieu Boespflug. Conversion by evaluation. In Carro and Peña [CP10], pages 58–72.

[Bou07] Sylvain Boulmé. Intuitionistic refinement calculus. In Rocca [Roc07], pages 54–69.

[BP99] R. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[BP00] Richard Banach and Michael Poppleton. Retrenchment, refinement, and simulation. In Bowen et al. [BDGK00], pages 304–323.

[BP07] Sylvain Boulmé and Marie-Laure Potet. Interpreting invariant composition in the b method using the Spec# ownership relation: A way to explain and relax B restrictions. In Julliand and Kouchnarenko [JK06], pages 4–18.

[Bro96] Manfred Broy, editor. *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*, 1996.

[Bur00] Lilian Burdy. *Traitement des expressions dépourvues de sens de la théorie des ensembles – Application à la méthode B.* Thèse de doctorat, Conservatoire National des Arts et Métiers, may 2000.

[BvW00] Ralph-Johan Back and Joakim von Wright. Encoding, decoding and data refinement. *Formal Asp. Comput.*, 12(5):313–349, 2000.

[CC] ISO/IEC 15408: Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/.

[CD08] M. Carlier and C. Dubois. Functional testing in the FoCaL environment. In B. Berckert and R. Hahnle, editors, *Test And Proof (TAP'2008)*, volume 4966, pages 84–98. LNCS, 2008.

[CD09] Ana Cavalcanti and Dennis Dams, editors. *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*. Springer, 2009.

[CGL96] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking. In Broy [Bro96], pages 305–349.

[CH01] Bill Councill and George T. Heineman. Summary. pages 741–752, 2001.

[Cha98] P. Chartier. Formalisation of B in Isabelle/HOL. In Bert [Ber98], pages 66–82.

[CHL96] P-L. Curien, T. Hardin, and J-J. Lévy. Confluence properties of weak and strong calculi of explicit substitutions. *Journal of the ACM*, 43(2):362–397, March 1996.

[CK98] H. Cirstea and C. Kirchner. Using rewriting and strategies for describing the B predicate prover. In Kirchner and Kirchner [KK98], pages 25–36.

[Clu03] Jolyon Clulow. On the security of PKCS#11. In Walter et al. [WcKKP03], pages 411–425.

[CM06a] Dominique Cansell and Dominique Méry. Formal and incremental construction of distributed algorithms: On the distributed reference counting algorithm. *Theor. Comput. Sci.*, 364(3):318–337, 2006.

[CM06b] Judicaël Courant and Jean-François Monin. Defending the bank with a proof assistant. In *WITS 2006*, Vienna, March 2006. In WITS proceedings.

[CM09] Samuel Colin and Georges Mariano. Coq, l'alpha et l'omega de la preuve pour B ? 14 pages + annexe de deux pages, 2009.

[Coq] The Coq proof assistant. http://coq.inria.fr.

[CP88] Thierry Coquand and Christine Paulin. Inductively defined types. In Martin-Löf and Mints [MLM90], pages 50–66.

[CP95] Jeffrey A. Clark and Dhiraj K. Pradhan. Fault injection. *Computer*, 28(6):47–56, 1995.

[CP10] Manuel Carro and Ricardo Peña, editors. *Practical Aspects of Declarative Languages, 12th International Symposium, PADL 2010, Madrid, Spain, January 18-19, 2010. Proceedings*, volume 5937 of *Lecture Notes in Computer Science*. Springer, 2010.

[CPR+05] S. Colin, D. Petit, J. Rocheteau, R. Marcano, G. Mariano, and V. Poirriez. BRILLANT : An open source and XML-based platform for rigourous software development. In *SEFM (Software Engineering and Formal Methods)*, Koblenz, Germany, september 2005. AGKI (Artificial Intelligence Research Koblenz), IEEE Computer Society Press. selectivity : 40/120.

[CPW06] A. Charguéraud, B. C. Pierce, and S. Weirich. Proof engineering: Practical techniques for mechanized metatheory, September 2006. Submitted for publication.

[CSC] John A. Clark, Susan Stepney, and Howard Chivers. Breaking the model: Finalisation and a taxonomy of security attacks.

[dB72] N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, pages 381–392, 1972.

[Del00] D. Delahaye. A tactic language for the system Coq. In Parigot and Voronkov [PV00], pages 85–95.

[DG07] Jim Davies and Jeremy Gibbons, editors. *Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2-5, 2007, Proceedings*, volume 4591 of *Lecture Notes in Computer Science*. Springer, 2007.

[DHK00] G. Dowek, T. Hardin, and C. Kirchner. Higher order unification via explicit substitutions. *Inf. Comput.*, 157(1-2):183–235, 2000.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DL07] Z. Dargaye and X. Leroy. Mechanized verification of CPS transformations. In Dershowitz and Voronkov [DV07], pages 211–225.

[DV07] Nachum Dershowitz and Andrei Voronkov, editors. *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, volume 4790 of *Lecture Notes in Computer Science*. Springer, 2007.

[DY81] Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, Stanford, CA, USA, 1981.

[ED07] Didier Essamé and Daniel Dollé. B in large-scale projects: The canarsie line cbtc experience. In Julliand and Kouchnarenko [JK06], pages 252–254.

[FHT05] John Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, editors. *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings*, volume 3582 of *Lecture Notes in Computer Science*. Springer, 2005.

[Flo67] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[Foc] The FoCaLize project. http://focalize.inria.fr/.

[GFL05] Frédéric Gervais, Marc Frappier, and Régine Laleau. Vous avez dit raffinement? Technical Report CEDRIC-829, CNAM, march 2005.

[GHS] Integrity real-time operating system. http://www.ghs.com/products/rtos/integrity.html.

[GM92] J.A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE Press, 1992.

[GMW09] Herman Geuvers, James McKinna, and Freek Wiedijk. Pure type systems without explicit contexts. Submitted at TLCA'09, 2009.

[Gor93] A. D. Gordon. A mechanisation of name-carrying syntax up to alpha-conversion. In Joyce and Seger [JS94], pages 413–425.

[Had07] Amal Haddad. Meca: A tool for access control models. In Julliand and Kouchnarenko [JK06], pages 281–284.

[HAF01] M. Randall Holmes and J. Alves-Foss. The Watson theorem prover. *J. Autom. Reasoning*, 26(4):357–408, 2001.

[Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 07(5):11–19, 1990.

[HHGB07] Sarah Hoffmann, Germain Haugou, Sophie Gabriele, and Lilian Burdy. The B-Method for the construction of microkernel-based systems. In Julliand and Kouchnarenko [JK06], pages 257–259.

[HM05] J. Hurd and T. F. Melham, editors. *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*. Springer, 2005.

[Hoa92] C. A. R. Hoare. Programs are predicates. In *FGCS*, pages 211–218, 1992.

[HR84] C. A. R. Hoare and A. W. Roscoe. Programs as executable predicates. In *FGCS*, pages 220–228, 1984.

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

[Jae05] Éric Jaeger. De C à B, l'analyse de code par les méthodes formelles. Master's thesis, Université Paris 7, September 2005.

[Jaf07] Eddie Jaffuel. Using B machines for model-based testing of smartcard software. In Julliand and Kouchnarenko [JK06], page 2.

[JK06] Jacques Julliand and Olga Kouchnarenko, editors. *B 2007: Formal Specification and Development in B, 7th International Conference of B Users, Besançon, France, January 17-19, 2007, Proceedings*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2006.

[JL07] Eddie Jaffuel and Bruno Legeard. LEIRIOS test generator: Automated test generation from B models. In Julliand and Kouchnarenko [JK06], pages 277–280.

[JLH+09] Steffen Jost, Hans-Wolfgang Loidl, Kevin Hammond, Norman Scaife, and Martin Hofmann. "carbon credits" for resource-bounded computations using amortised analysis. In Cavalcanti and Dams [CD09], pages 354–369.

[Jos88] Mark B. Josephs. A state-based approach to communicating processes. *Distributed Computing*, 3(1):9–18, 1988.

[JS94] J. J. Joyce and C-J. H. Seger, editors. *Higher Order Logic Theorem Proving and its Applications, 6th International Workshop, HUG '93, Vancouver, BC, Canada, August 11-13, 1993, Proceedings*, volume 780 of *Lecture Notes in Computer Science*. Springer, 1994.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Matthews and Anderson [MA09], pages 207–220.

[KK98] Claude Kirchner and Hélène Kirchner, editors. *Automated Deduction - CADE-15, 15th International Conference on Automated Deduction, Lindau, Germany, July 5-10, 1998, Proceedings*, volume 1421 of *Lecture Notes in Computer Science*. Springer, 1998.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16(10):613–615, 1973.

[Lam02] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *CoRR*, abs/0902.2137, 2009.

[Lin05] Sam Lindley. *Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages*. PhD thesis, University of Edinburgh, College of Science and Engineering, School of Informatics, 2005.

[MA09] Jeanna Neefe Matthews and Thomas E. Anderson, editors. *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*. ACM, 2009.

[Mag03a] Nicolas Magaud. *Changements de Représentation des Données dans le Calcul des Constructions*. PhD thesis, Université de Nice Sophia-Antipolis, October 2003.

[Mag03b] Nicolas Magaud. Changing Data Representation within the Coq System. In *TPHOLs'2003*, volume 2758. LNCS, Springer-Verlag, 2003.

[M.J88] M.J.C. Gordon. Mechanizing programming logics in higher-order logic. In G.M. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automatic Theorem Proving (Proceedings of the Workshop on Hardware Verification)*, pages 387–439, Banff, Canada, 1988. Springer-Verlag, Berlin.

[MLM90] Per Martin-Löf and Grigori Mints, editors. *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*. Springer, 1990.

[MM04] Annabelle McIver and Carrol Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems*. Monographs in Computer Science. Springer Verlag, 2004.

[MMM09] Annabelle McIver, Larissa Meinicke, and Carroll Morgan. Security, probability and nearly fair coins in the cryptographers' café. In Cavalcanti and Dams [CD09], pages 41–71.

[MMT08] Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.

[Mor90] C. Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Muñ99] C. Muñoz. PBS: Support for the B-method in PVS, 1999.

[Mus05] L. Mussat, 2005. Private Communication.

[NV07] M. Norrish and R. Vestergaard. Proof pearl: de Bruijn terms really do work. In Schneider and Brandt [SB07], pages 207–222.

[PK99] Laurence Pierre and Thomas Kropf, editors. *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*. Springer, 1999.

[PL07] Daniel Plagge and Michael Leuschel. Validating Z specifications using the ProBAnimator and model checker. In Davies and Gibbons [DG07], pages 480–500.

[PV00] M. Parigot and A. Voronkov, editors. *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*. Springer, 2000.

[RCMP04] Jérôme Rocheteau, Samuel Colin, Georges Mariano, and Vincent Poirriez. Évaluation de l'extensibilité de PhoX: B/PhoX un assistant de preuves pour B. In Valérie M. Morain, editor, *Journées Francophones des Langages Applicatifs (JFLA 2004)*, pages 37–54. INRIA, 2004.

[Req08] Antoine Requet. Bart: A tool for automatic refinement. In Börger et al. [BBBB08], page 345.

[RM05] Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first order logic. In Hurd and Melham [HM05], pages 294–309.

[Roc07] Simona Ronchi Della Rocca, editor. *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*. Springer, 2007.

[SB07] K. Schneider and J. Brandt, editors. *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*, volume 4732 of *Lecture Notes in Computer Science*. Springer, 2007.

[Sch] Arno Schönegge. Proof obligations for monomorphicity.

[Sch95] Arno Schönegge. Would you ever risk a non-monomorphic specification?, 1995.

[SL00] D. Sabatier and P. Lartigue. The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. *Formal Methods in System Design*, 17(3):245–272, 2000.

[SMB92] V. Stavridou, T. F. Melham, and R. T. Boute, editors. *Theorem Provers in Circuit Design, Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience, Nijmegen, The Netherlands, 22-24 June 1992, Proceedings*, volume A-10 of *IFIP Transactions*. North-Holland, 1992.

[SV07] Marko Samer and Helmut Veith. On the notion of vacuous truth. In Dershowitz and Voronkov [DV07], pages 2–14.

[TCS] DoD 5200.28-STD: Trusted computer system evaluation criteria. http://csrc.nist.gov/publications/history/dod85.pdf.

[TKHS05] Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors. *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*. Springer, 2005.

[WcKKP03] Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors. *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*. Springer, 2003.

[Wer94] Benjamin Werner. *Une théorie des constructions inductives*. Thèse de doctorat, Université Paris 7, 1994.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John S. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4), 2009.

[WN04] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2004)*, volume 3223 of *LNCS*, pages 305–320. Springer, 2004.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Pierre and Kropf [PK99], pages 54–66.