Pirates et sécurité informatique : la logique à la rescousse !

Jean Goubault-Larrecq

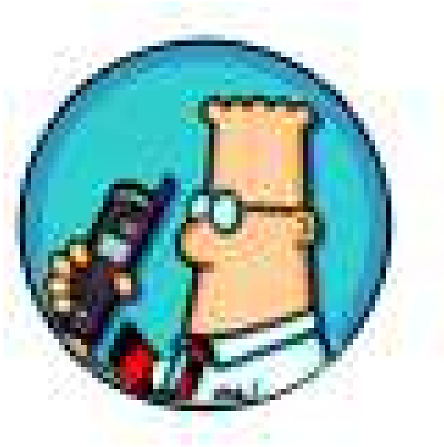
LSV/CNRS UMR 8643, ENS Cachan

Plan

1. Comment envoyer des messages secrets ?
2. Une étude de quelques attaques.
3. Un modèle logique pour la vérification de protocoles cryptographiques
4. Une méthode de vérification (logique et automates)

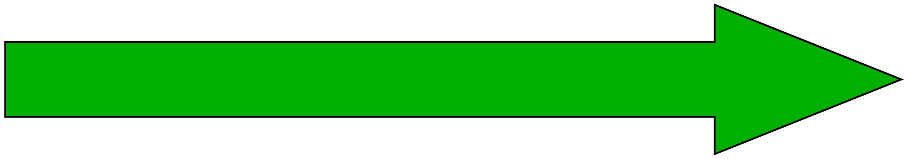
Alice veut envoyer un message à Bob...

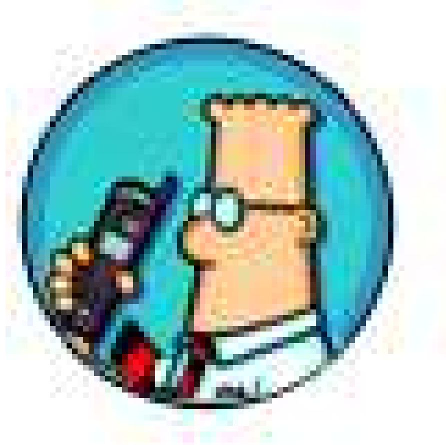
solution : 2 ; 3 ; 1 ; 1



Alice veut envoyer un message à Bob...

solution : 2 ; 3 ; 1 ; 1

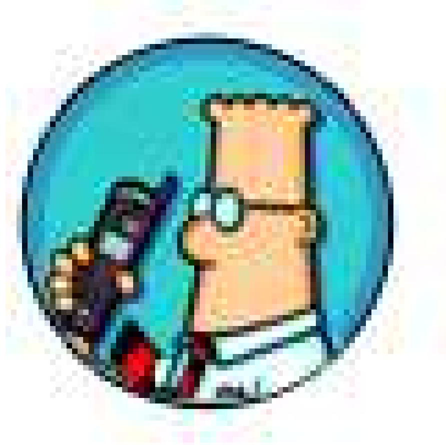




Mais Charlie peut espionner le message : pas de secret.

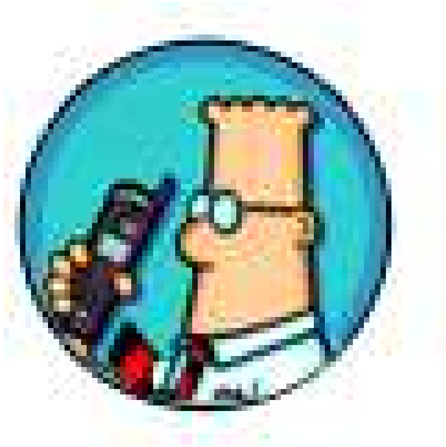
La solution standard : le chiffrement

vroxwlrq = 5> 6> 4> 4



La solution standard : le chiffrement

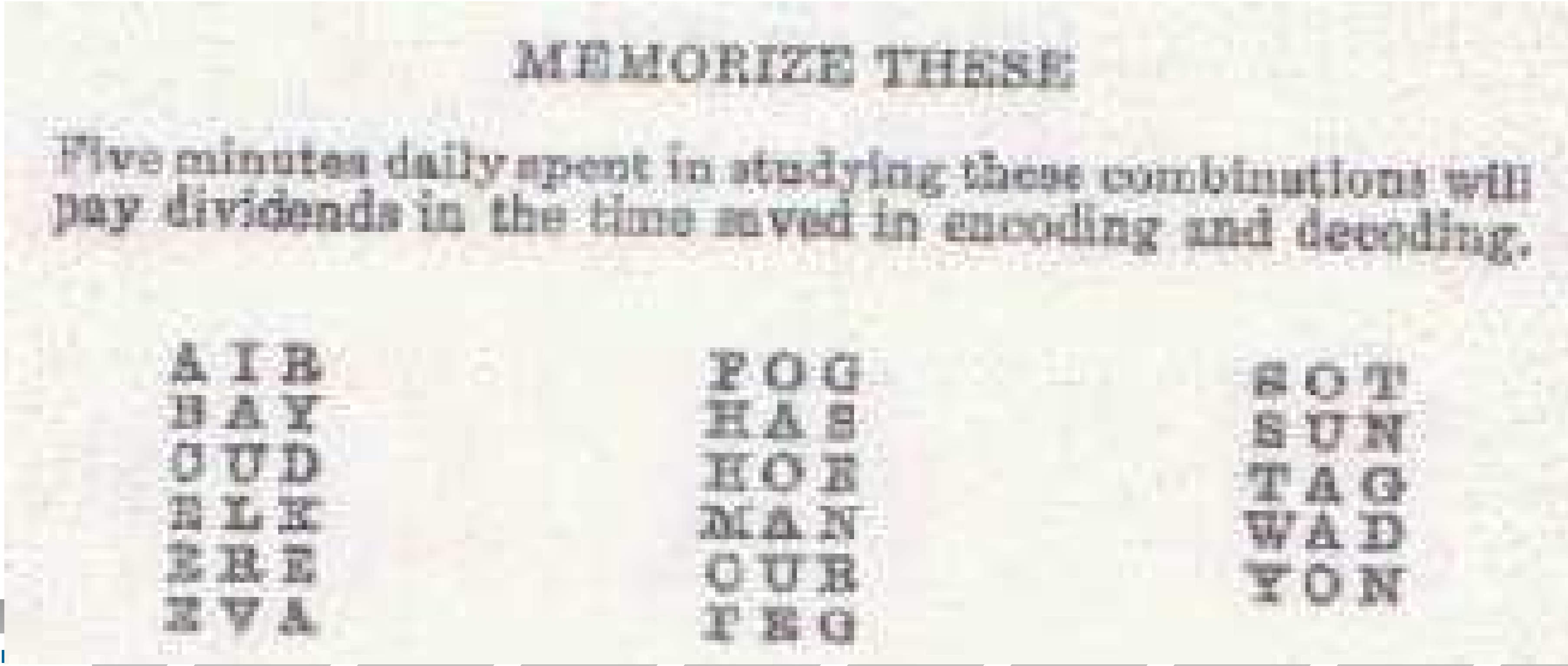
vroxwlrq = 5> 6> 4> 4



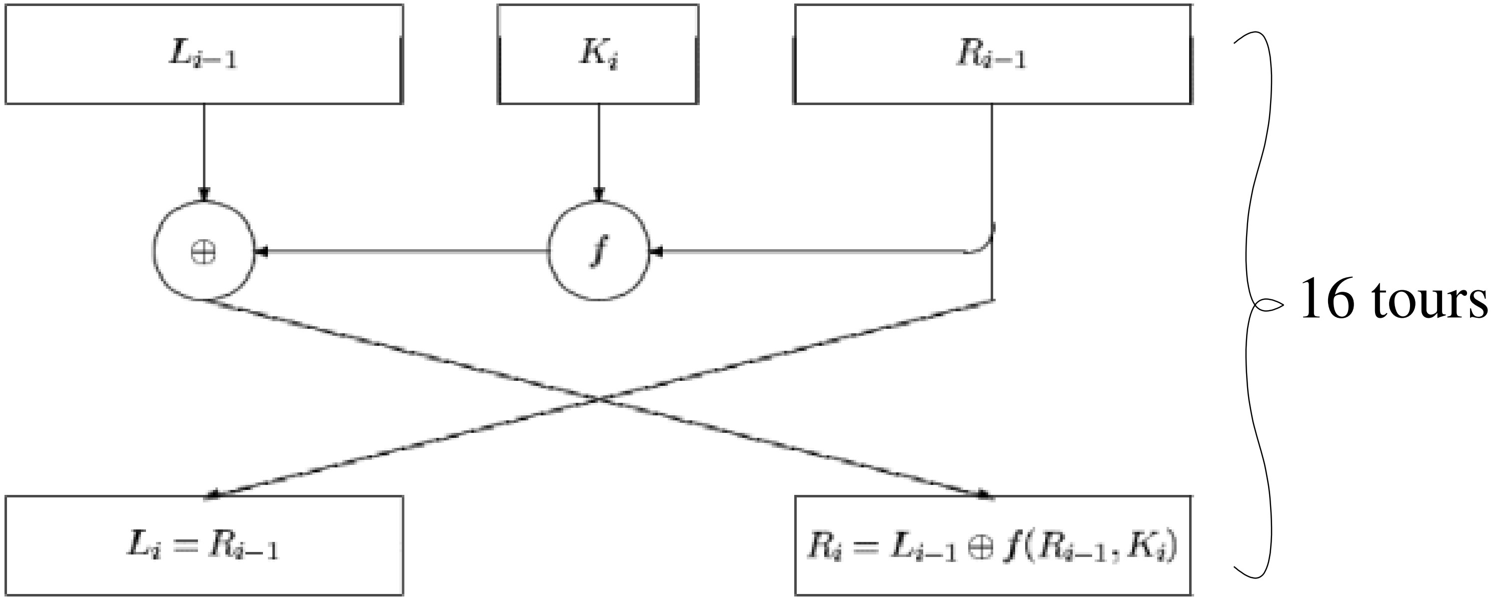
Mais l'algorithme de chiffrement est trop simple...

Un meilleur système de chiffrement

Envoyer $k$ième lettre $\oplus$ $k$ième lettre d'un one-time pad, $1 \leq k \leq$ longueur :
($\oplus$ $=$ou exclusif, ou bien addition mod 26, etc.)



Et il y en a d'autres, le DES par exemple...



Un petit point de vocabulaire

En Français, on dit :

– chiffrer, pas “crypter” (to encrypt)

– déchiffrer, pas “décrypter” (to decrypt)

Un petit point de vocabulaire

En Français, on dit :

– chiffrer, pas "crypter" (to encrypt)

– déchiffrer, pas "décrypter" (to decrypt)

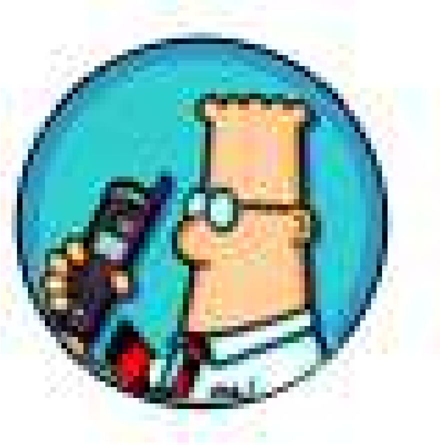
– décrypter... (to decrypt, to crack)

Laboratoire Spécification et Vérification

Avec un chiffrement amélioré...

sw2z7o61 ; JB EM >= 4

= $\{$solution : 2 ; 3 ; 1 ; 1$\}_K$

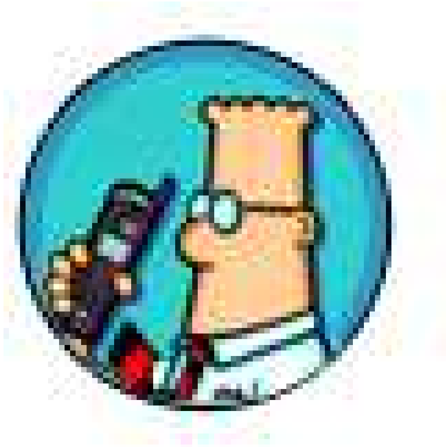


Laboratoire Spécification et Vérification

... Charlie peut encore nuire !

sw2z7o61 ; IB DM @= 4

$= \{\text{solution} : 1 ; 2 ; 3 ; 1\}_K$



L'intégrité du message est compromise.

Une solution : éviter le chiffrement par blocs

Par ex., CBC : Découper le messages en blocs (de 64 bits pour DES par ex.) :

B_1	B_2	...	B_n

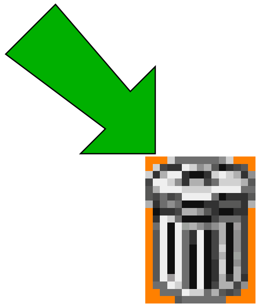
Envoyer :

IV_0	$\{IV_0 \oplus B_1\}_K = IV_1$	$\{IV_1 \oplus B_2\}_K = IV_2$	...	$\{IV_{n-1} \oplus B_n\}_K = IV_n$

Empêche de tronquer ou remplacer des bouts.

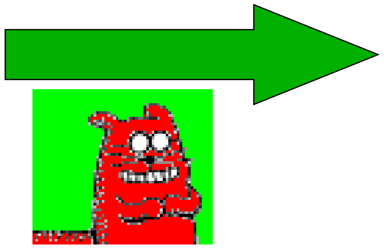
Intégrité : inclure des sommes de contrôle (hash).

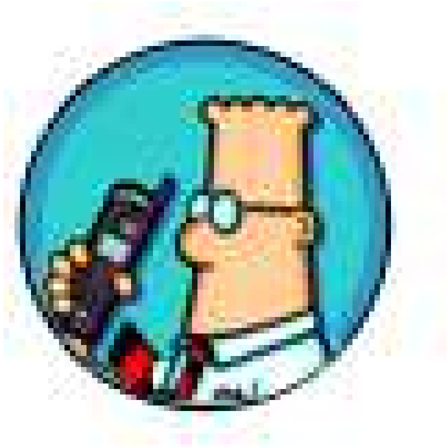
On n'arrête pas Charlie !



jm ss :o1t =hs

= $\{$je ne sais pas$\}_K$



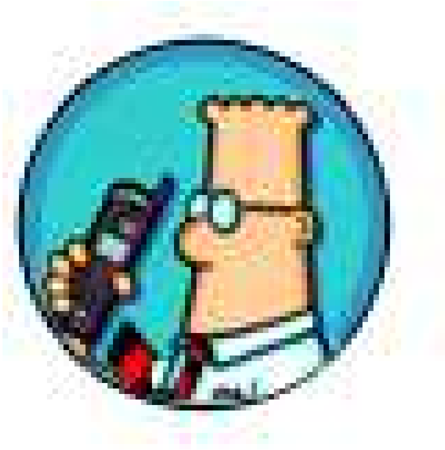


Charlie peut rejouer un vieux message, même sans connaître la clé K.

Alice inclut un temps dans son message...

sw2z7o61 ; JB EM >= 4 - C9@5=

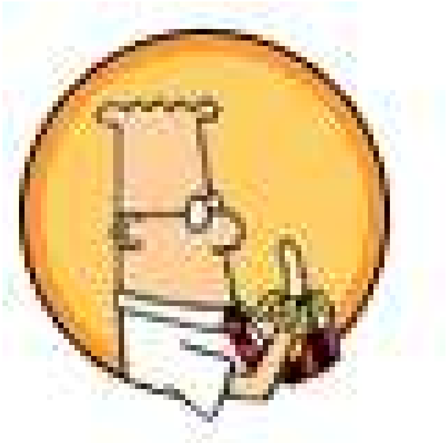
= $\{$solution : 2 ; 3 ; 1 ; 1 - 09 :12$\}_K$



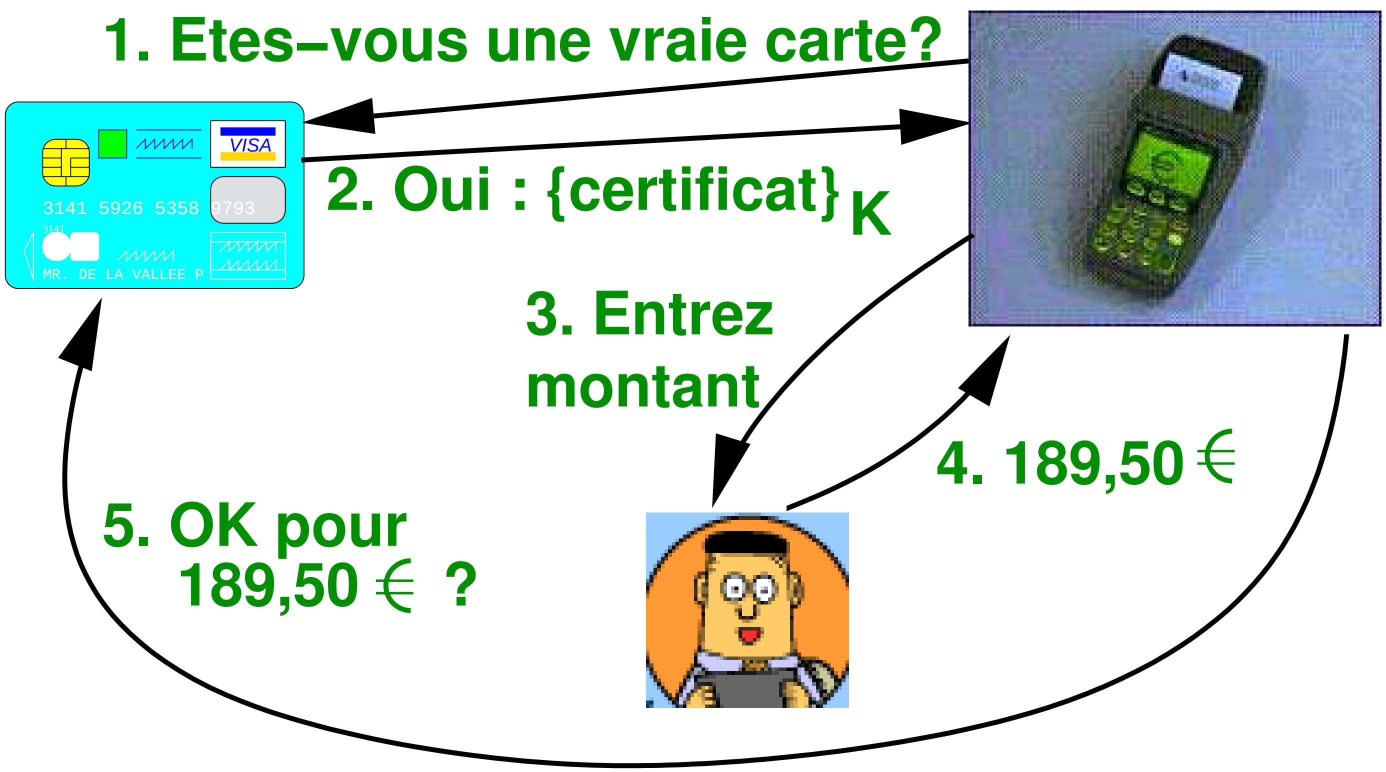
...Charlie ne peut plus rejouer de vieux messages.

sw2z7o61 ; IB DM @= 4 - C8@4@

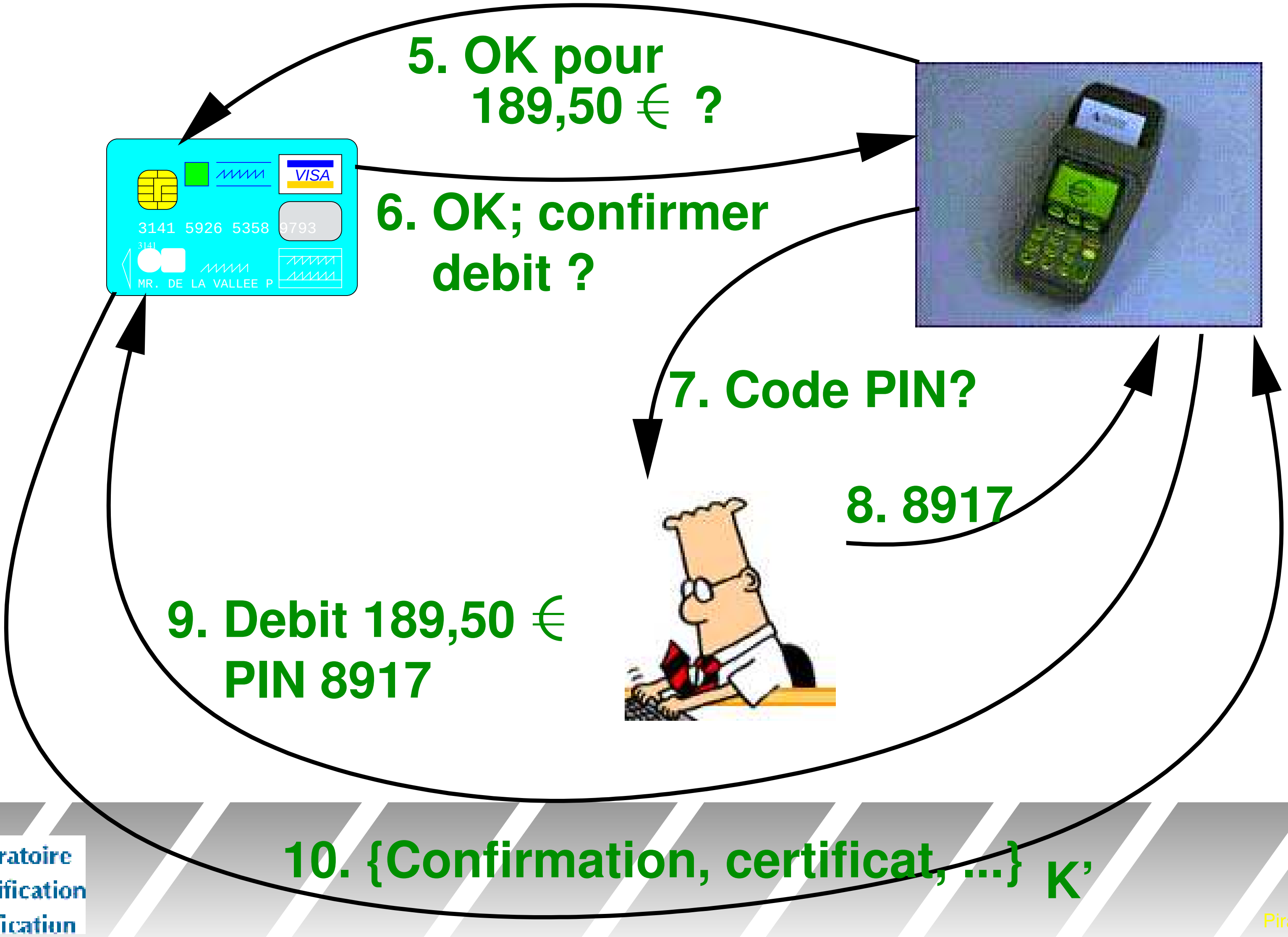
= $\{$solution : 1 ; 2 ; 3 ; 1 - 08 :05$\}_K$



Un exemple concret : les cartes à puce



Les cartes à puce, suite



En résumé

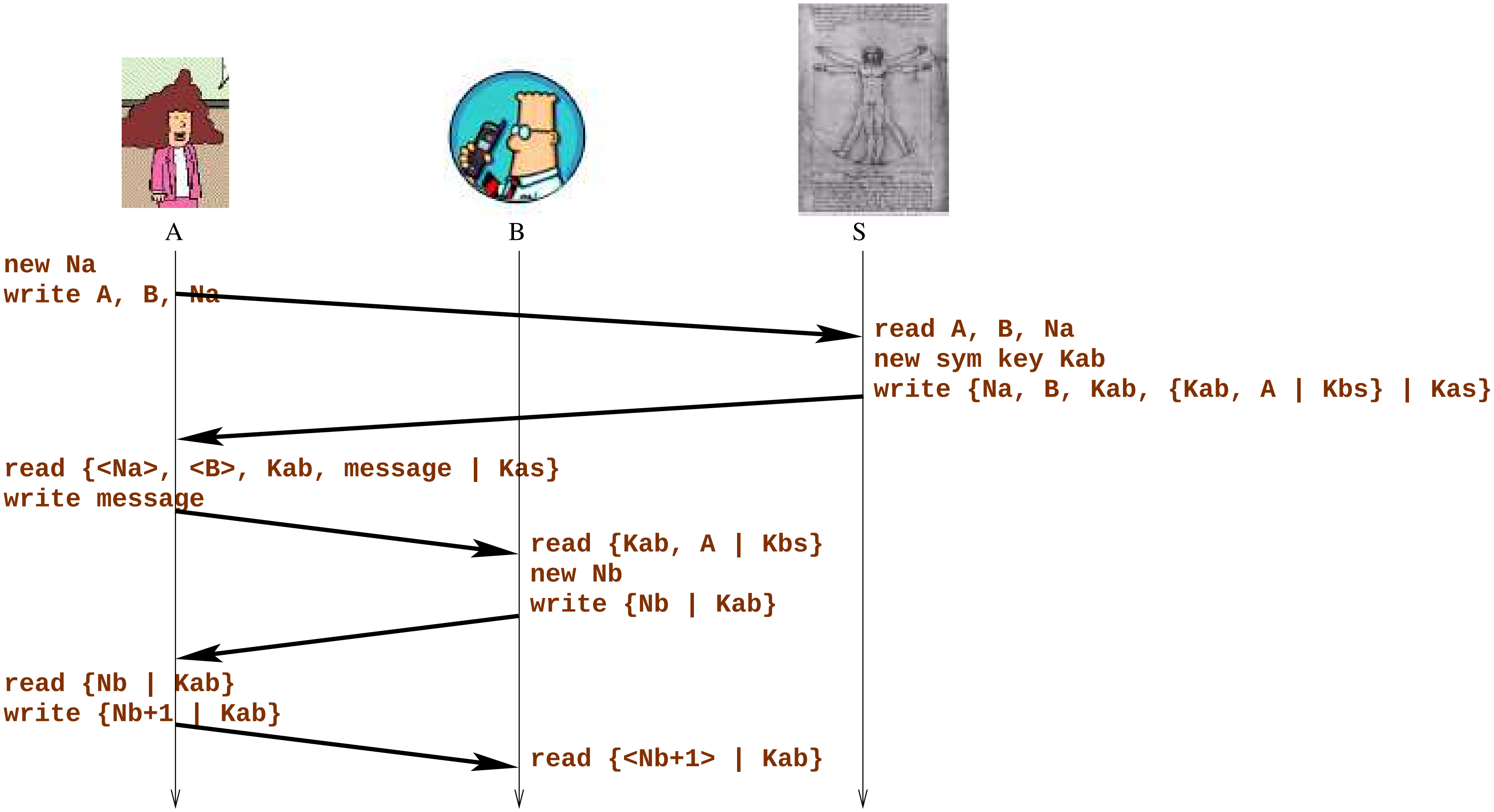
Alice a dû utiliser :

– un algorithme de chiffrement fort secret ;
– des sommes de contrôle intégrité ;
– des estampilles temporelles fraîcheur ;
– plus quelques autres précautions de codage (CBC, types explicites, ...) ;
– plus divers authenticité, unicité, ...

Toutes ces précautions assurent-elles que Charlie n'arrivera pas à ses fins ?

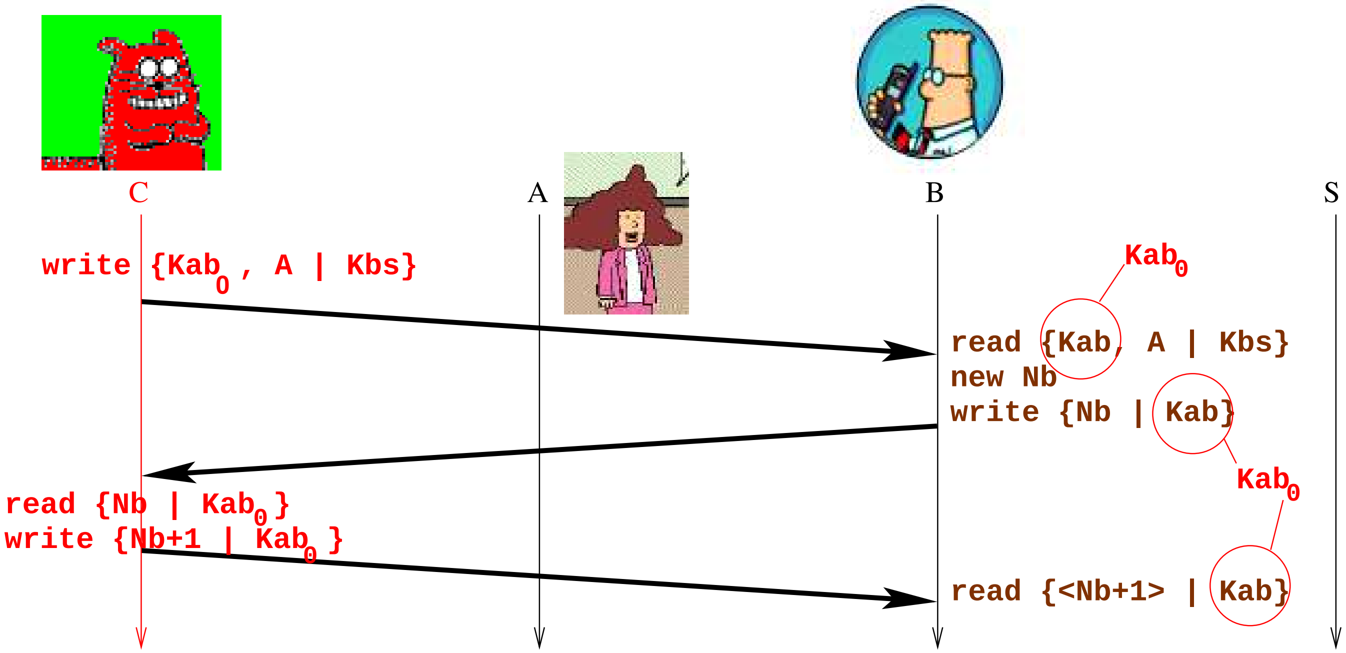
2. Une étude de quelques attaques

Étudions un exemple [NeedhamSchroeder78]



Laboratoire Spécification et Vérification

Une attaque (secret, authentication)



Miracle : la cryptographie à clé publique

A fabrique K_a, K_a^{-1}, publie K_a, garde K_a^{-1}.
Pour chiffrer M, B utilise la clé publique K_a :

$$M' = \{M\}_{K_a}.$$

A déchiffre avec sa clé privée : $$M = \{M'\}_{K_a^{-1}}.$$

Laboratoire Spécification et Vérification

Miracle : la cryptographie à clé publique

A fabrique K_a, K_a^{-1}, publie K_a, garde K_a^{-1}.
Pour chiffrer M, B utilise la clé publique K_a :

$$M' = \{M\}_{K_a}.$$

A déchiffre avec sa clé privée : $M = \{M'\}_{K_a^{-1}}$.

Arg : M' n'est pas forcément authentique (K_a est publique...) $\Longrightarrow$ B signe avec sa clé privée K_b^{-1}.

Miracle : la cryptographie à clé publique

A fabrique K_a, K_a^{-1}, publie K_a, garde K_a^{-1}.
Pour chiffrer M, B utilise la clé publique K_a :

$$M' = \{M\}_{K_a}.$$

A déchiffre avec sa clé privée : $M = \{M'\}_{K_a^{-1}}$.

Arg : M' n'est pas forcément authentique (K_a est publique...) $\Longrightarrow$ B signe avec sa clé privée K_b^{-1}.

Secret+authenticité : B envoie $\{M, \underbrace{\{h(M)\}_{K_b^{-1}}}_{M'}\}_{K_a}$

A vérifie ensuite $h(M) = \{M'\}_{K_b}$.

Laboratoire Spécification et Vérification

L'algorithme RSA [RivestShamirAdleman78]

Génération de clés : tirer deux grands nombres premiers $p \neq q$. $N = pq$, $\phi(N) = (p-1)(q-1)$. Tirer $d \in [2, \phi(N)[$ premier avec $\phi(N)$. Calculer e tel que $de = 1 \mod \phi(N)$ (Bezout).
Clé publique : (N, e) privée : (N, d).

Chiffrement : $\{M\}_{(N,e)} = M^e \mod N$.

Déchiffrement : $\{M'\}_{(N,d)} = M'^d \mod N$.

(Exercice : ces fonctions sont inverses.)

Laboratoire Spécification et Vérification

La cryptographie à clé publique, c'est bien mais...

– C'est lourd et lent (RSA est typiquement 1000 fois plus lent que DES).

Laboratoire Spécification et Vérification

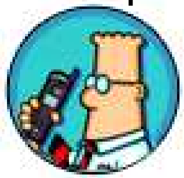
La cryptographie à clé publique, c'est bien mais...

– C'est lourd et lent (RSA est typiquement 1000 fois plus lent que DES).
– Ça ne change rien au fait que les protocoles cryptographiques contiennent des bugs purement logiques !

Un autre exemple classique [NeedhamSchroeder78] [aussi !]

Clé publique de B;
B déchiffre avec sa clé privée

A



B

```
new Na
write {Na, A | Kb}
```

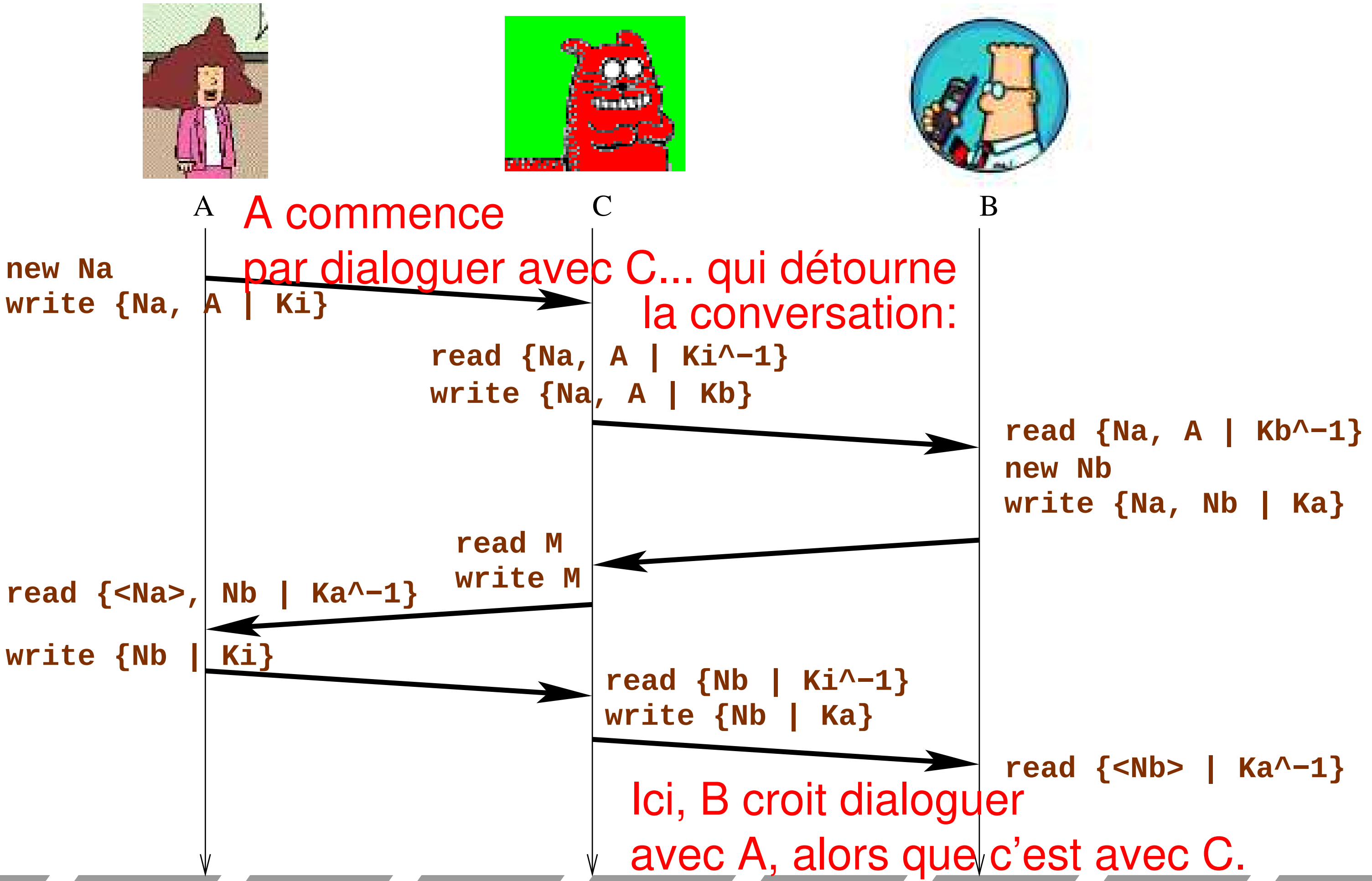
```
read {Na, A | Kb^-1}
new Nb
write {Na, Nb | Ka}
```

Clé publique de A;
A déchiffre avec sa clé privée.

```
read {<Na>, Nb | Ka^-1}
write {Nb | Kb}
```

```
read {<Nb> | Kb^-1}
```

Caramba, encore raté !

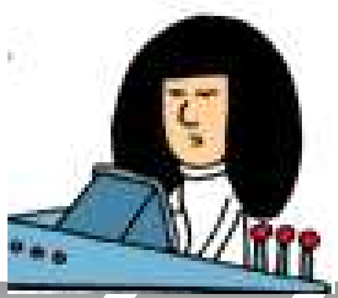


Quelle proportion de protocoles publiés sont faux ?

– Les 2 protocoles de [NeedhamSchroeder78] : faux.
– [OtwayRees87] : OK si exécution séquentielle, faux si un principal peut jouer le rôle de A et de B dans deux sessions différentes en même temps.
– [DenningSacco81], Yahalom [BAN89], Wide Mouthed Frog [BAN89], CCITT X.509 [BAN89], Andrew Secure RPC [Satyanarayanan89], SPLICE [YamaguchiOkayamaMiyahara90], [Snekkenes92], station-to-station protocol

[DiffieVanOorschotWiener92], KSL [KehneSchönwälderLandendörfer92], [NeumanStubblebine93], [WooLam94] : faux (à des degrés divers).

Il y a donc besoin de méthodes systématiques (automatiques ?) de vérification de protocoles cryptographiques.



Vérification exhaustive

Un moyen direct : énumérer toutes les traces d'exécution possibles du protocole, et tester à chaque état les propriétés P (secret, etc.)

Vérification exhaustive

Un moyen direct : énumérer toutes les traces d'exécution possibles du protocole, et tester à chaque état les propriétés P (secret, etc.)

NB Les propriétés P à tester sont d'inaccessibilité (d'un état où P échoue).

Vérification exhaustive

Un moyen direct : énumérer toutes les traces d'exécution possibles du protocole, et tester à chaque état les propriétés P (secret, etc.)

NB Les propriétés P à tester sont d'inaccessibilité (d'un état où P échoue).

Thm [EvenGoldreich83,DurginLincolnMitchell Scedrov99] (Sous quelques hypothèses) Ces propriétés P sont indécidables.

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à états finis (automate fini).

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à états finis (automate fini).

1. messages de tailles non bornées

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à états finis (automate fini).

1. messages de tailles non bornées
2. nonces en nombre arbitraire

Laboratoire Spécification et Vérification

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à états finis (automate fini).

1. messages — de tailles non bornées
2. nonces — en nombre arbitraire
3. canaux — capacité illimitée de l'intrus

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à états finis (automate fini).

1. messages de tailles non bornées
2. nonces en nombre arbitraire
3. canaux capacité illimitée de l'intrus
4. instances principaux en nombre quelconque.

Laboratoire Spécification et Vérification

Pourquoi la vérification est-elle difficile ?

On est très loin d'un système à états finis (automate fini).

1. messages de tailles non bornées
2. nonces en nombre arbitraire
3. canaux capacité illimitée de l'intrus
4. instances principaux en nombre quelconque.

Pour une présentation unifiée, on écrira des programmes Prolog purs = ens. de clauses de Horn.

3. Un modèle logique pour les protocoles cryptographiques

Qu'est-ce qu'une modélisation ?

On cherche à décrire formellement ce que fait un protocole.

On ne cherche pas (encore) à vérifier automatiquement des propriétés sur ces protocoles (cf. partie 4...)

Les clauses de Horn

```
on_route(rome).

on_route(Place) :-
       move(Place,Method,NewPlace),
       on_route(NewPlace).

move(home,taxi,halifax).
move(halifax,train,gatwick).
move(gatwick,plane,rome).

?on_route(home,X).     [false :- on_route(home,X).]
```

Sources d'infini — 1. Les messages

Espace des messages

$$\begin{array}{rcll} M & ::= & D & \text{Données de base} \\ & \mid & K & \text{Clés brutes} \\ & \mid & (M, \ldots, M) & \text{n-uplets} \\ & \mid & \{M|K\} & \text{Chiffrements} \end{array}$$

Taille des messages non bornée.

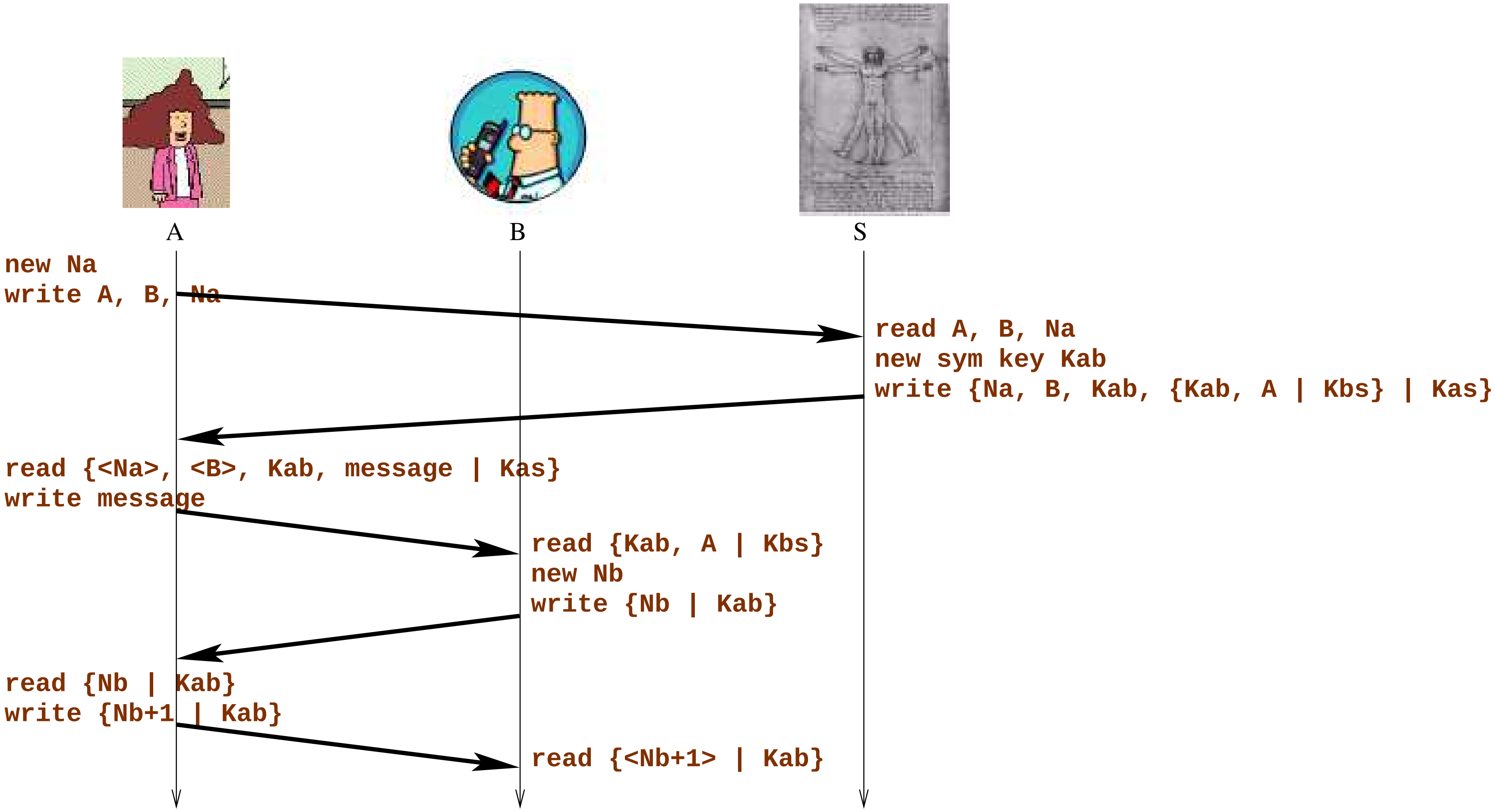
Codage de systèmes de transitions sous forme de clauses de Horn

États :
$(n+1)$-uplets $s =$(pc, valeur x_1, ..., valeur x_n).
Transitions :

$$s \xrightarrow{\text{condition}} s'$$

```
trans(S, S') :- condition.
```

Retour sur Needham-Schroeder à clés privées



Sources d'infini — 1. Les messages

État de A : $s_A = (pc, N_a, A, B, K_{as}, K, M)$.
Transitions :

```
transA((1,_,A,B,Kas,K,M), (2,Na,A,B,Kas,K,M)) :-
      new(Na), write((A, B, Na)).
transA((2,Na,A,B,Kas,_,_), (3,Na,A,B,Kas,Kab,M)) :-
      read({(Na, B, Kab, M) | Kas}).
transA((3,Na,A,B,Kas,Kab,M), (4,Na,A,B,Kas,Kab,M)) :-
      write (M).
     ...
```

Mise en parallèle

Le prédicat `transA` décrit bien un principal A.
Mais on veut décrire A || B || S :

```
global((EtatA,EtatB,EtatS), (EtatA',EtatB,EtatS)) :-
            transA (EtatA, EtatA').
global((EtatA,EtatB,EtatS), (EtatA,EtatB',EtatS)) :-
            transB (EtatB, EtatB').
global((EtatA,EtatB,EtatS), (EtatA,EtatB,EtatS')) :-
            transS (EtatS, EtatS').
```

Laboratoire Spécification et Vérification

Les exécutions globales

```
  /* Cloture reflexive-transitive: */
exec((EtatA,EtatB,EtatC), (EtatA,EtatB,EtatC)).
exec((EtatA,EtatB,EtatC), (EtatA',EtatB',EtatC')) :-
    global ((EtatA,EtatB,EtatC),
            (EtatA1,EtatB1,EtatC1)),
    exec ((EtatA1,EtatB1,EtatC1),
          (EtatA',EtatB',EtatC')).
```

Accessibilité

Demandons à Prolog si on peut atteindre
– un état global où la propriété dangereuse P est réalisée (ex : non-secret de K_{ab})
– à partir d'un état vérifiant des hypothèses initiales H (secret de K_{as}, ...)

```
? H ((EtatA0,EtatB0,EtatS0)),
  exec((EtatA0,EtatB0,EtatS0), (EtatA,EtatB,EtatS)),
  P((EtatA,EtatB,EtatS)).
```

(Reste à définir `new`, `write`, `read`...)

Sources d'infini — 2. Les nonces

Les nonces : `new(Na)` est censé créer une nouvelle donnée de base et la mettre dans `Na`.

Sources d'infini — 2. Les nonces

Les nonces : `new(Na)` est censé créer une nouvelle donnée de base et la mettre dans `Na`.
On fait une approximation : si deux nonces sont créés par le même processus au même pc, on les identifie.
$\rightarrow$ remplacer la variable `Na` par une cste `na_A_2`.

Laboratoire Spécification et Vérification

Sources d'infini — 2. Les nonces

Les nonces : `new(Na)` est censé créer une nouvelle donnée de base et la mettre dans `Na`. On fait une approximation : si deux nonces sont créés par le même processus au même pc, on les identifie.
$\rightarrow$ remplacer la variable `Na` par une cste na_A_2.
$\rightarrow$ correct pour la vérification de propriétés d'accessibilité.

(Autres approches : logique linéaire [Lincoln].)

Skolémiser les nonces, en Prolog

```
transA((1,_,A,B,Kas,K,M), (2,Na,A,B,Kas,K,M)) :-
      new(Na), write((A, B, Na)).
     ...
```

$$\longrightarrow$$

```
transA((1,_,A,B,Kas,K,M), (2,na_A_2,A,B,Kas,K,M)) :-
      write((A, B, na_A_2)).
     ...
```

Laboratoire Spécification et Vérification

Sources d'infini — 3. Les canaux

Les canaux de communication contiennent une quantité non bornée de messages en attente, de tailles non bornées.

Le problème est classique (automates à files, etc.), mais aucune solution classique ne s'applique.

Sources d'infini — 3. Les canaux

Les canaux de communication contiennent une quantité non bornée de messages en attente, de tailles non bornées.

Le problème est classique (automates à files, etc.), mais aucune solution classique ne s'applique.

Plus compliqué : l'intrus détourne les canaux, et :

– intercepte les messages,

– fabrique de nouveaux messages,

– chiffre, déchiffre, jette, duplique, ...

Le modèle de [DolevYao83]

On modélise le cas le pire pour la sécurité, où :

- tout message émis est envoyé directement à l'intrus ;
- tout message reçu est fabriqué directement par l'intrus ;
- l'intrus peut chiffrer, déchiffrer, etc., mais pas décrypter. (En particulier, il peut fabriquer des messages de taille arbitraire.)

En particulier, ceci subsume le comportement normal des canaux...

Laboratoire Spécification et Vérification

Capacité déductive des intrus

$$\frac{}{E, M \mapsto M}\,(Ax)$$

$$\frac{E \mapsto M \quad E \mapsto K}{E \mapsto \{M \mid K\}}\,(CryI) \qquad \frac{E \mapsto \{M \mid K\} \quad E \mapsto K^{-1}}{E \mapsto M}\,(CryE)$$

$$\frac{E \mapsto M_1 \ \ldots \ E \mapsto M_n}{E \mapsto (M_1, \ldots, M_n)}\,(TupI) \qquad \frac{E \mapsto (M_1, \ldots, M_n)}{E \mapsto M_i}\,(TupE_i), 1 \leq i \leq n$$

La modélisation en clauses de Horn

- Écrire des clauses pour `ded(E,M)` ($\equiv$ $E \mapsto M$) ;
- `write(M)`=ajouter `M` à `E` ;
- `read(M)`=imposer `ded(E,M)`.

```
transA((1,_,A,B,Kas,K,M), E,
       (2,na_A_2,A,B,Kas,K,M), cons((A, B, na_A_2), E))
transA((2,Na,A,B,Kas,_,_), E,
       (3,Na,A,B,Kas,Kab,M), E) :-
      ded(E,{(Na, B, Kab, M) | Kas}).
transA((3,Na,A,B,Kas,Kab,M), E,
       (4,Na,A,B,Kas,Kab,M), cons(M,E)).
     ...
```

Malheureusement,

Un intérêt de Prolog ici était de pouvoir simuler un protocole.

Malheureusement,

Un intérêt de Prolog ici était de pouvoir simuler un protocole.
Avec notre modélisation de l'intrus, ça ne fonctionne plus :

Prolog boucle

... même sans boucle dans le protocole ...

Malheureusement,

Un intérêt de Prolog ici était de pouvoir simuler un protocole.
Avec notre modélisation de l'intrus, ça ne fonctionne plus :

Prolog boucle

... même sans boucle dans le protocole ...

(La stratégie de recherche de preuve Prolog recherche des déductions par $\mapsto$ arbitrairement profondes.)

Sources d'infini — 4. Les instances

A, B, S sont des rôles (des spécifications), qui peuvent avoir un nombre arbitraire d'instances, les principaux.

Important :

- Les serveurs S opèrent en parallélisme non borné en pratique ;
- Certaines attaques n'apparaissent qu'à partir du moment où suffisamment de principaux sont en parallèle (Ex : Otway-Rees).

Parallélisme non borné et complicité

Une solution élégante [DebbabiMejriTawbiYahmadi97,JGL00] : approximer les comportements des principaux S en multi-session parallèle par de nouvelles règles pour $\mid\!\!\rightarrow$.

Par ex., si S fait `read M; ...; write M'`, on ajoute :

$$\frac{E \mid\!\!\rightarrow \mathtt{M}}{E \mid\!\!\rightarrow \mathtt{M}'}$$

$\rightarrow$ considère S comme un complice de l'intrus.

Exemple de complicité (Otway-Rees)

$$\begin{array}{l}\texttt{read}(M, A, B,\\ \quad \{N_a, \langle M\rangle, \langle A\rangle, \langle B\rangle \mid K_{as}\},\\ \quad \{N_b, \langle M\rangle, \langle A\rangle, \langle B\rangle \mid K_{bs}\})\\ \texttt{new\_key} K_{ab};\\ \texttt{write}(M, \{N_a, K_{ab} \mid K_{as}\},\\ \quad \{N_b, K_{ab} \mid K_{bs}\});\end{array} \sim \frac{\begin{array}{l}E \mapsto (M, A, B,\\ \quad \{N_a, M, A, B \mid K_{as}\},\\ \quad \{N_b, M, A, B \mid K_{bs}\})\end{array}}{\begin{array}{l}E \mapsto (M, \{N_a, k\_S\_2 \mid K_{as}\},\\ \quad \{N_b, k\_S\_2 \mid K_{bs}\})\end{array}}$$

4. Comment vérifier un protocole cryptographique ?

Rappel

$$\begin{array}{rl} & \text{Notre protocole } \pi \\ + & \text{les hypothèses initiales } H \\ + & \text{la propriété } P \text{ dangereuse à surveiller} \\ \hline & \text{(gros) ensemble } S \text{ de clauses de Horn.} \end{array}$$

Fait P est fausse de π sous l'hypothèse H (sécurité)

$\iff S$ non-contradictoire.

(enfin, au moins $\Longleftarrow$... c'est l'essentiel.)

Comment tester si un ensemble de clauses est contradictoire ?

Beaucoup de méthodes, c'est le thème de la démonstration automatique :

- Chercher un contre-exemple par énumération exhaustive [DavisPutnam60] ;
- Méthode de tableaux [Beth55,Hintikka55,Smullyan68], élimination de modèles [Loveland69], méthode des connexions [Bibel81,Andrews81], ...
- Résolution (et variantes) [Robinson65].

La résolution (pour les clauses de Horn)

Une seule règle (pensez clôture transitive) :

$$\frac{\begin{array}{c} P(t) :- Q_1(t_1), ..., Q_i(t_i), ..., Q_m(t_m). \\ Q_i(u) :- R_1(u_1), ..., R_n(u_n). \end{array}}{P(t\sigma) :- Q_1(t_1\sigma), ..., R_1(u_1\sigma), ..., R_n(u_n\sigma), ..., Q_m(t_m\sigma).} \ (Res)$$

avec $\sigma = mgu(t_i, u)$.

La résolution (pour les clauses de Horn)

Une seule règle (pensez clôture transitive) :

$$\frac{\begin{array}{l} P(t) :- Q_1(t_1), \ldots, Q_i(t_i), \ldots, Q_m(t_m). \\ Q_i(u) :- R_1(u_1), \ldots, R_n(u_n). \end{array}}{P(t\sigma) :- Q_1(t_1\sigma), \ldots, R_1(u_1\sigma), \ldots, R_n(u_n\sigma), \ldots, Q_m(t_m\sigma).} \ (Res)$$

avec $\sigma = mgu(t_i, u)$.

Thm [Correction, complétude] Un ensemble S de clauses de Horn est contradictoire $\Longleftrightarrow$ on peut déduire `false.` de S par la seule règle (Res).

Un exemple

$$
\begin{aligned}
\texttt{pair(0).} & & & (1)\\
\texttt{pair(s(s(X)))} & \texttt{ :- } & \texttt{pair(X).} & (2)\\
\texttt{false} & \texttt{ :- } & \texttt{pair(s}^{10}\texttt{(0)).} & (3)
\end{aligned}
$$

Un exemple

$$\texttt{pair(0).} \qquad (1)$$

$$\texttt{pair(s(s(X)))} \quad \texttt{:-} \quad \texttt{pair(X).} \qquad (2)$$

$$\texttt{false} \quad \texttt{:-} \quad \texttt{pair(s}^{10}\texttt{(0)).} \qquad (3)$$

$$(1), (2) \Longrightarrow \quad \texttt{pair(s}^{2}\texttt{(0)).} \qquad (4)$$

Un exemple

$$\begin{array}{rrcll}
& \texttt{pair(0).} & & & (1)\\
& \texttt{pair(s(s(X)))} & \texttt{:-} & \texttt{pair(X).} & (2)\\
& \texttt{false} & \texttt{:-} & \texttt{pair(s}^{10}\texttt{(0)).} & (3)\\
\hline
(1),(2) \Longrightarrow & \texttt{pair(s}^{2}\texttt{(0)).} & & & (4)\\
(2),(2) \Longrightarrow & \texttt{pair(s}^{4}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} & (5)
\end{array}$$

Un exemple

$$
\begin{array}{rrcll}
 & \texttt{pair(0).} & & & (1) \\
 & \texttt{pair(s(s(X)))} & \texttt{:-} & \texttt{pair(X).} & (2) \\
 & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{10}\texttt{(0)).} & (3) \\
\hline
(1),(2) \Longrightarrow & \texttt{pair(s}^{2}\texttt{(0)).} & & & (4) \\
(2),(2) \Longrightarrow & \texttt{pair(s}^{4}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} & (5) \\
(5),(5) \Longrightarrow & \texttt{pair(s}^{8}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} & (6)
\end{array}
$$

Laboratoire Spécification et Vérification

Un exemple

$$
\begin{array}{rrcll}
 & \texttt{pair(0).} & & & (1) \\
 & \texttt{pair(s(s(X)))} & \texttt{:-} & \texttt{pair(X).} & (2) \\
 & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{10}\texttt{(0)).} & (3) \\
\hline
(1),(2) \Longrightarrow & \texttt{pair(s}^{2}\texttt{(0)).} & & & (4) \\
(2),(2) \Longrightarrow & \texttt{pair(s}^{4}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} & (5) \\
(5),(5) \Longrightarrow & \texttt{pair(s}^{8}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} & (6) \\
(4),(6) \Longrightarrow & \texttt{pair(s}^{10}\texttt{(0)).} & & & (7)
\end{array}
$$

Un exemple

$$
\begin{array}{rrcll}
 & \texttt{pair(0).} & & & (1) \\
 & \texttt{pair(s(s(X)))} & \texttt{:-} & \texttt{pair(X).} & (2) \\
 & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{10}\texttt{(0)).} & (3) \\
\hline
(1),(2) \Longrightarrow & \texttt{pair(s}^{2}\texttt{(0)).} & & & (4) \\
(2),(2) \Longrightarrow & \texttt{pair(s}^{4}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} & (5) \\
(5),(5) \Longrightarrow & \texttt{pair(s}^{8}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} & (6) \\
(4),(6) \Longrightarrow & \texttt{pair(s}^{10}\texttt{(0)).} & & & (7) \\
(7),(3) \Longrightarrow & & & \texttt{false.} &
\end{array}
$$

Un exemple, modifié

$$\begin{array}{rcl}
\texttt{pair(0).} & & \\
\texttt{pair(s(s(X)))} & \texttt{:-} & \texttt{pair(X).} \\
\texttt{false} & \texttt{:-} & \texttt{pair(s}^{11}\texttt{(0)).} \\
\hline
 & \ldots & \\
\texttt{pair(s}^{2n}\texttt{(X))} & \texttt{:-} & \texttt{pair(X).} \\
\texttt{pair(s}^{2n}\texttt{(0)).} & & \\
\texttt{false} & \texttt{:-} & \texttt{pair(s}^{2n+1}\texttt{(0)).}\ (0 \leq n \leq 5) \\
 & \ldots &
\end{array}$$

Que s'est-il passé ?

La résolution boucle $\rightarrow$ nombre infini de clauses.

`false.` jamais engendré $\Longrightarrow S$ n'est pas contradictoire… mais on ne le saura jamais !

Que s'est-il passé ?

La résolution boucle $\rightarrow$ nombre infini de clauses.

`false.` jamais engendré $\Longrightarrow S$ n'est pas contradictoire... mais on ne le saura jamais !

Thm Le problème "un ensemble de clauses de Horn est-il contradictoire ?" est indécidable.
Preuve On peut même coder une machine de Turing universelle par un programme de la forme [DevienneLebègueRoutierWürtz94] :

P(t_0).   P(t_2) :-P(t_1).   false :-P(t_3).

Laboratoire Spécification et Vérification

Que peut-on faire de mieux ?

Pour limiter l'espace de recherche de la résolution :

– Raffinements (complets) de la résolution : unit-résolution, input-résolution, résolution linéaire, hyperrésolution, résolution sémantique, résolution ordonnée, cg-résolution, lock-résolution, ...
  Note : Prolog = S(election) L(inear) D(efinite)-résolution $\sim$ input-résolution.
– Stratégies d'effacement : des tautologies, des clauses pures, des clauses subsumées, ...

Laboratoire Spécification et Vérification

La résolution ordonnée

Pareil qu'avant :

$$\frac{\begin{array}{l} P(t) :- Q_1(t_1), ..., Q_i(t_i), ..., Q_m(t_m). \\ Q_i(u) :- R_1(u_1), ..., R_n(u_n). \end{array}}{P(t\sigma) :- Q_1(t_1\sigma), ..., R_1(u_1\sigma), ..., R_n(u_n\sigma), ..., Q_m(t_m\sigma).} \,(Res)$$

... mais uniquement si $Q_i(t_i)$, $Q_i(u)$ maximaux dans leurs clauses (pour un ordre stable $>$).

Thm [Correction, complétude] S contradictoire $\Longleftrightarrow$ `false.` déductible.

L'exemple modifié, par résolution ordonnée

$$
\begin{array}{lrcll}
 & \texttt{pair(0).} & & & (1)\\
 & \texttt{pair(s(s(X)))} & \texttt{:-} & \texttt{pair(X).} & (2)\\
 & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{11}\texttt{(0)).} & (3)\\
\hline
(2),(3) \Longrightarrow & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{9}\texttt{(0)).} & (4)\\
(4),(3) \Longrightarrow & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{7}\texttt{(0)).} & (5)\\
(5),(3) \Longrightarrow & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{5}\texttt{(0)).} & (6)\\
(6),(3) \Longrightarrow & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{3}\texttt{(0)).} & (7)\\
(7),(3) \Longrightarrow & \texttt{false} & \texttt{:-} & \texttt{pair(s}^{1}\texttt{(0)).} & (8)
\end{array}
$$

Stop. $\Longrightarrow$ non-contradiction.

Et alors ?

La résolution ordonnée termine sur ce type d'exemples.

Peut-on identifier une forme de clauses pour laquelle la résolution ordonnée terminerait toujours ?

On va voir qu'il y a une réponse positive liée à la théorie des automates.

Sous-classes décidables de la logique du premier ordre

Observation Si $\mathcal{C}$ est une classe d'ensembles de clauses S telle que :

l'ensemble des clauses déductibles de tout S fini $\in \mathcal{C}$ par résolution ordonnée est fini

alors $\mathcal{C}$ est une classe décidable.

Ex : $\mathcal{C} = \{$clauses $P($gros terme$)$:- $P_i($petits$).\}$.

Quelques résultats non triviaux

Corollaire la classe monadique [Ackermann54] est décidable. $(F ::= P(x)|F \wedge F|\neg F|\forall x \cdot F|\exists x \cdot F$.)

Preuve Mettre en forme clausale et utiliser l'observation précédente.

(Nécessite deux ou trois autres astuces...)

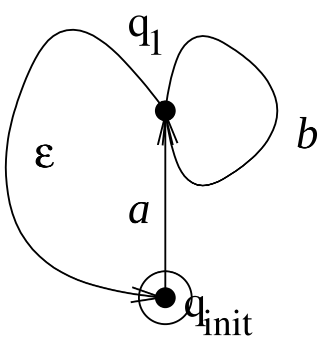
Corollaire [BachmairGanzingerWaldmann92] La satisfaction de contraintes ensemblistes est décidable.

(Ensembles $E ::= X|E \cap E|\complement E|f(E, ..., E)|f_i^{-1}(E)$,

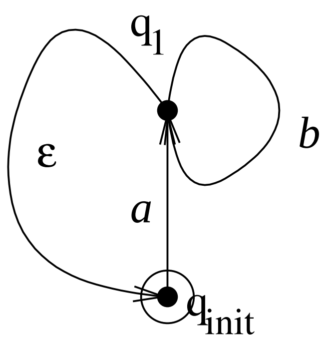
Contraintes = conjonctions finies de $E \subseteq E$.)

Laboratoire Spécification et Vérification

Automates



Automates et clauses de Horn

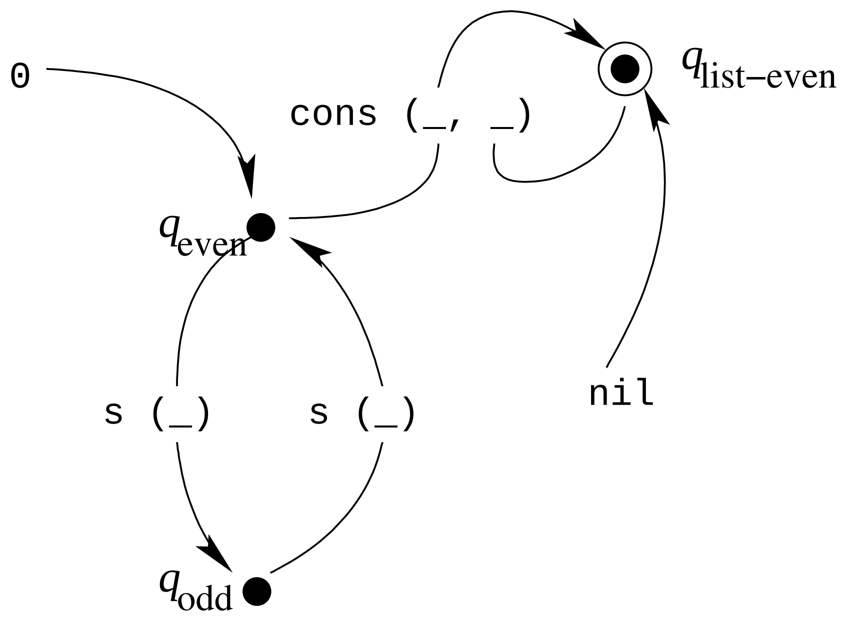



```
init(•).
q1(a(X)) :- init(X).
q1(b(X)) :- q1(X).
init(X) :- q1(X).
```

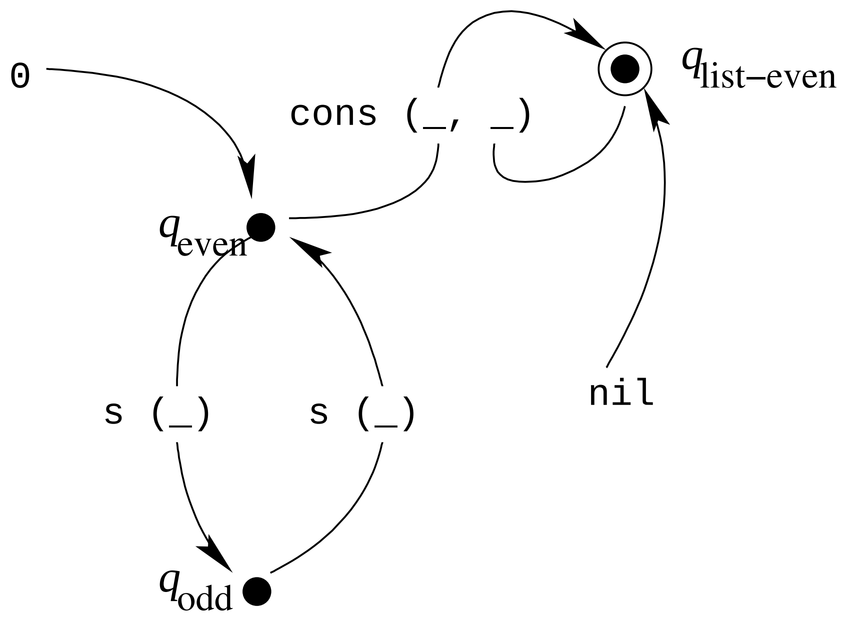
(mot $a_1 a_2 \ldots a_n$ = $a_n(\ldots(a_2(a_1(\bullet)))\ldots)$)

Laboratoire Spécification et Vérification

Automates d'arbres

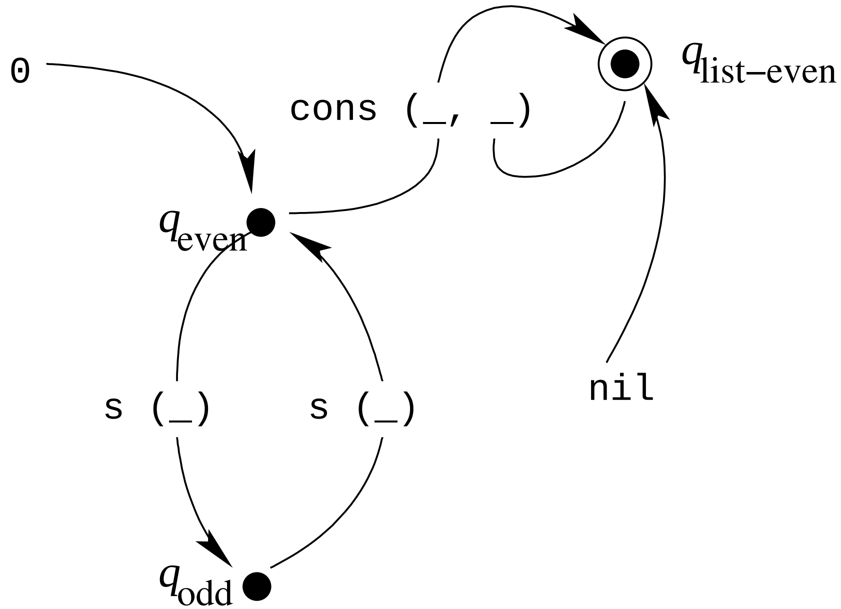


Automates d'arbres et clauses de Horn




```
even(0).
odd(s(X)) :- even(X).
even(s(X)) :- odd(X).
list-even(cons(X,Y)) :-
    even(X), list-even(Y).
list-even(nil).
```

Automates d'arbres et clauses de Horn




```
even(0).
odd(s(X)) :- even(X).
even(s(X)) :- odd(X).
list-even(cons(X,Y)) :-
    even(X), list-even(Y).
list-even(nil).
```

Non-vacuité $\Longleftrightarrow$ Contradiction

(de $q_{list\text{-}even}$) (avec `false :- list-even(X).`)

Décidable !

La forme logique des automates

Un automate d'arbres est un ensemble de clauses de la forme

$$P(f(X_1, \ldots, X_n)) :- P_1(X_1), \ldots, P_n(X_n)$$

La forme logique des automates

Un automate d'arbres est un ensemble de clauses de la forme

$$P(f(X_1, \ldots, X_n)) :- P_1(X_1), \ldots, P_n(X_n)$$

ϵ-transitions : $P(X) :- Q(X)$

Laboratoire Spécification et Vérification

La forme logique des automates

Un automate d'arbres est un ensemble de clauses de la forme

$$P(f(X_1, \ldots, X_n)) :- P_1(X_1), \ldots, P_n(X_n)$$

ϵ-transitions : $P(X) :- Q(X)$

$\wedge$-transitions (alternants) : $P(X) :- Q_1(X), Q_2(X)$

Laboratoire Spécification et Vérification

La forme logique des automates

Un automate d'arbres est un ensemble de clauses de la forme

$$P(f(X_1,\ldots,X_n)) :- P_1(X_1),\ldots,P_n(X_n)$$

ϵ-transitions : $P(X) :- Q(X)$

$\wedge$-transitions (alternants) : $P(X) :- Q_1(X), Q_2(X)$

pushdown : $P(X_i) :- Q(f(X_1,\ldots,X_n))$

Laboratoire Spécification et Vérification

La forme logique des automates

Un automate d'arbres est un ensemble de clauses de la forme

$$P(f(X_1, \ldots, X_n)) :- P_1(X_1), \ldots, P_n(X_n)$$

ϵ-transitions :	$P(X) :- Q(X)$
$\wedge$-transitions (alternants) :	$P(X) :- Q_1(X), Q_2(X)$
pushdown :	$P(X_i) :- Q(f(X_1, \ldots, X_n))$
égalités entre frères :	poser $X_i \equiv X_j$

Laboratoire Spécification et Vérification

La forme logique des automates

Un automate d'arbres est un ensemble de clauses de la forme

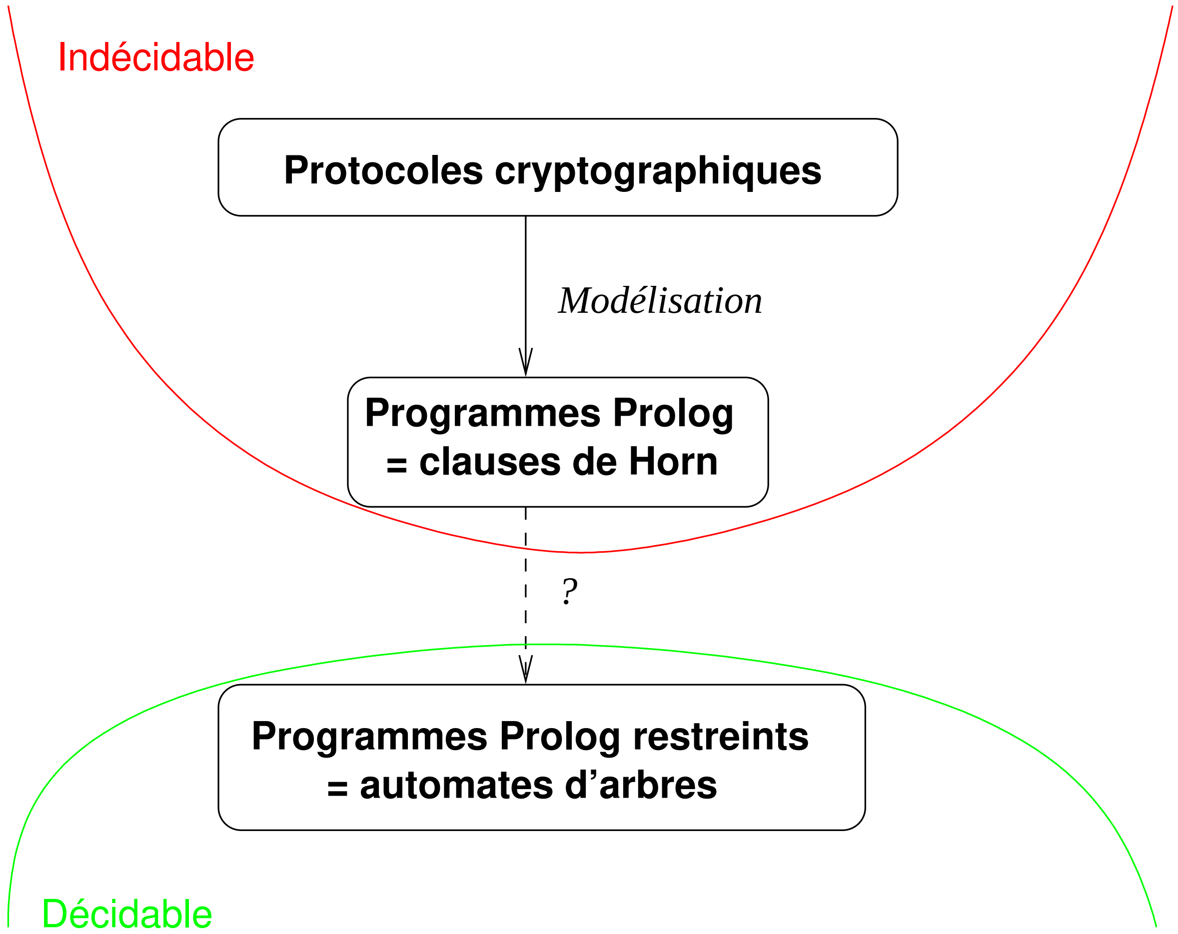
$$P(f(X_1, \ldots, X_n)) :- P_1(X_1), \ldots, P_n(X_n)$$

ϵ-transitions :	$P(X) :- Q(X)$
$\wedge$-transitions (alternants) :	$P(X) :- Q_1(X), Q_2(X)$
pushdown :	$P(X_i) :- Q(f(X_1, \ldots, X_n))$
égalités entre frères :	poser $X_i \equiv X_j$

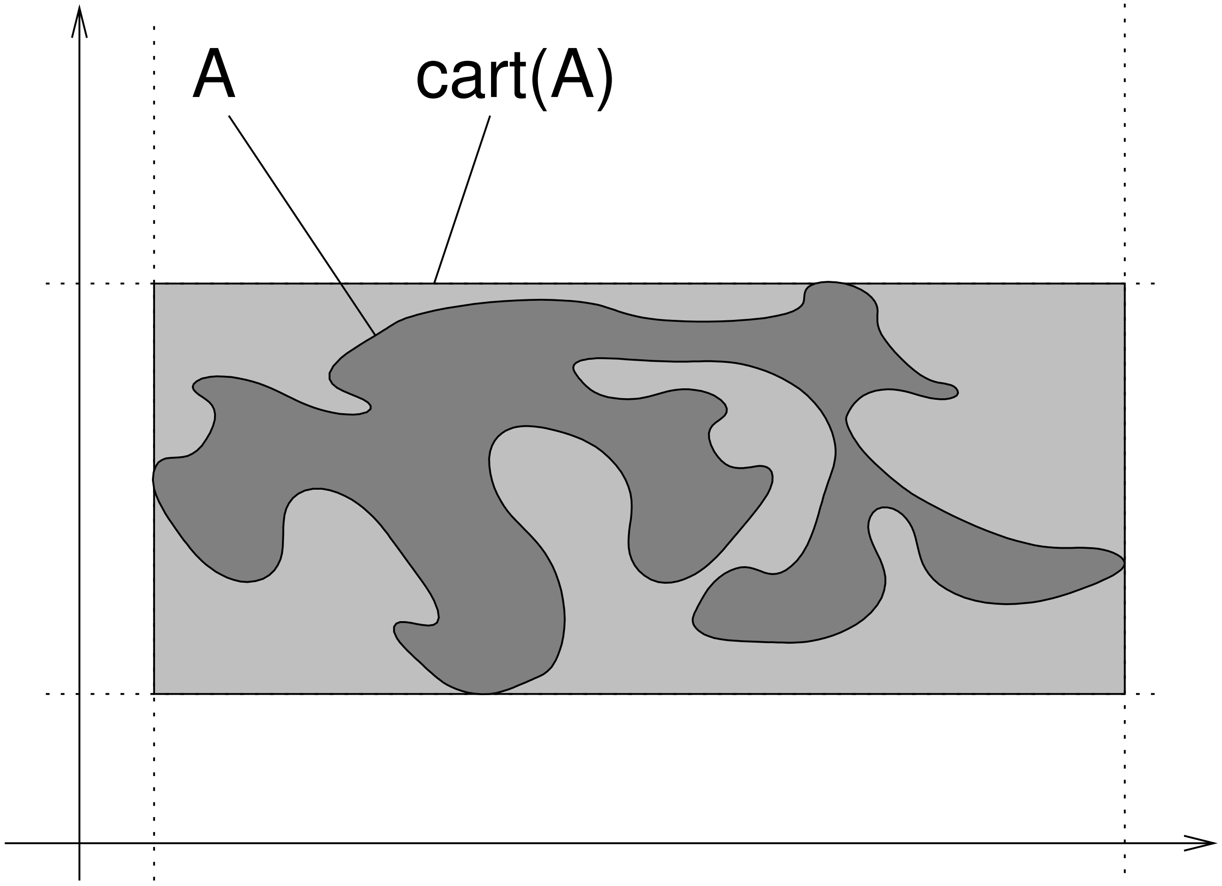
Thm La non-vacuité d'automates alternants avec contraintes d'égalité entre frères est décidable.

Preuve Par résolution ordonnée.

C'est très bien tout ça, mais où en sommes-nous ?



L'idée de l'approximation cartésienne



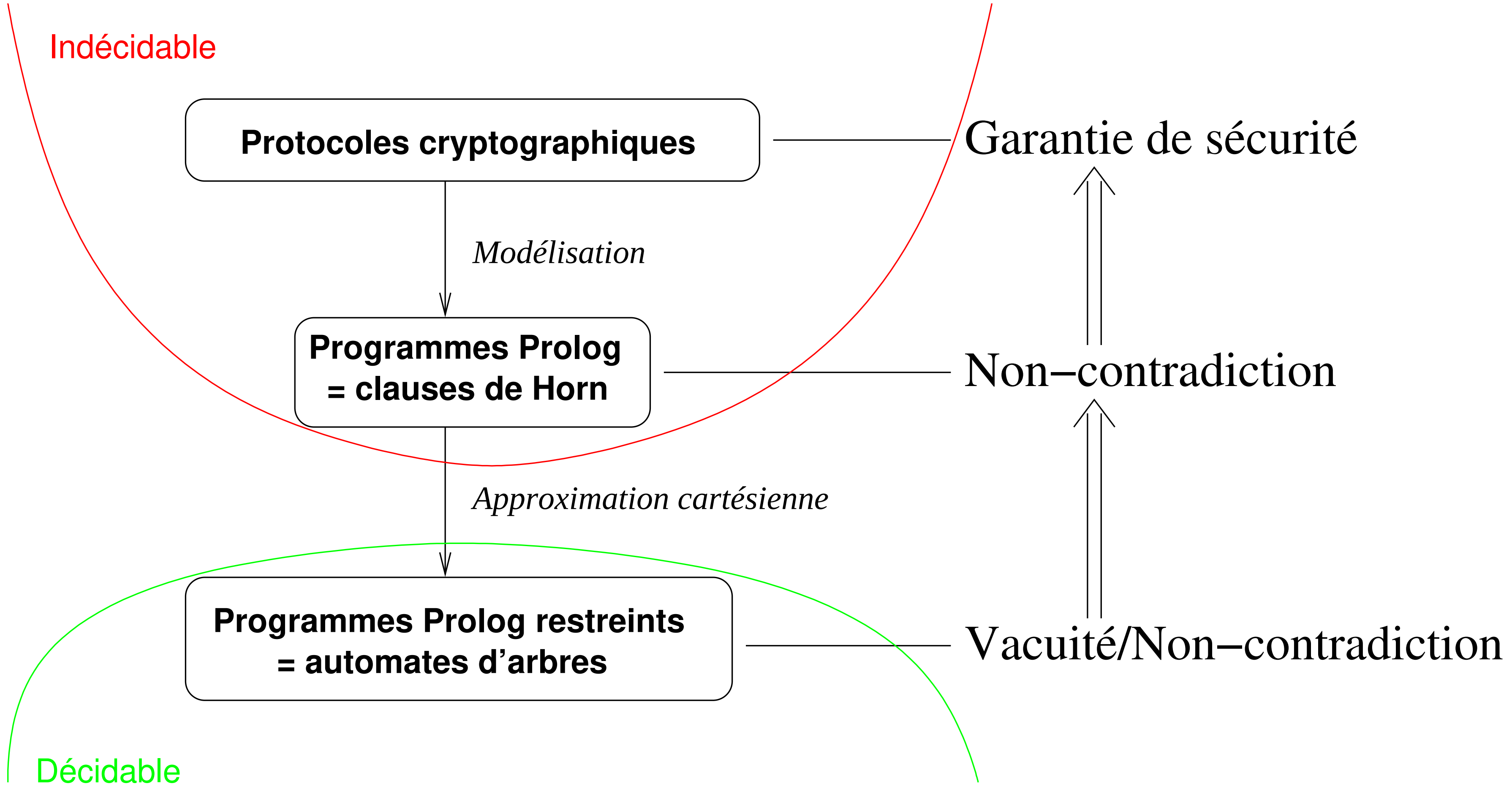
L'approximation cartésienne de clauses de Horn [CharatonikPodelski98]

$$P(t[X_1, \ldots, X_n]) :- Q(u)$$
$$\downarrow$$
$$\left\{ \begin{array}{rcl} P(t[X_1, \ldots, X_n]) & \texttt{:-} & \texttt{typeX1}(X_1), \ldots, \texttt{typeXn}(X_n) \\ \texttt{typeX1}(X_1) & \texttt{:-} & Q(u) \\ & \ldots & \\ \texttt{typeXn}(X_n) & \texttt{:-} & Q(u) \end{array} \right.$$

... est un automate d'arbres.(modulo quelques détails)

Laboratoire Spécification et Vérification

Et voilà !

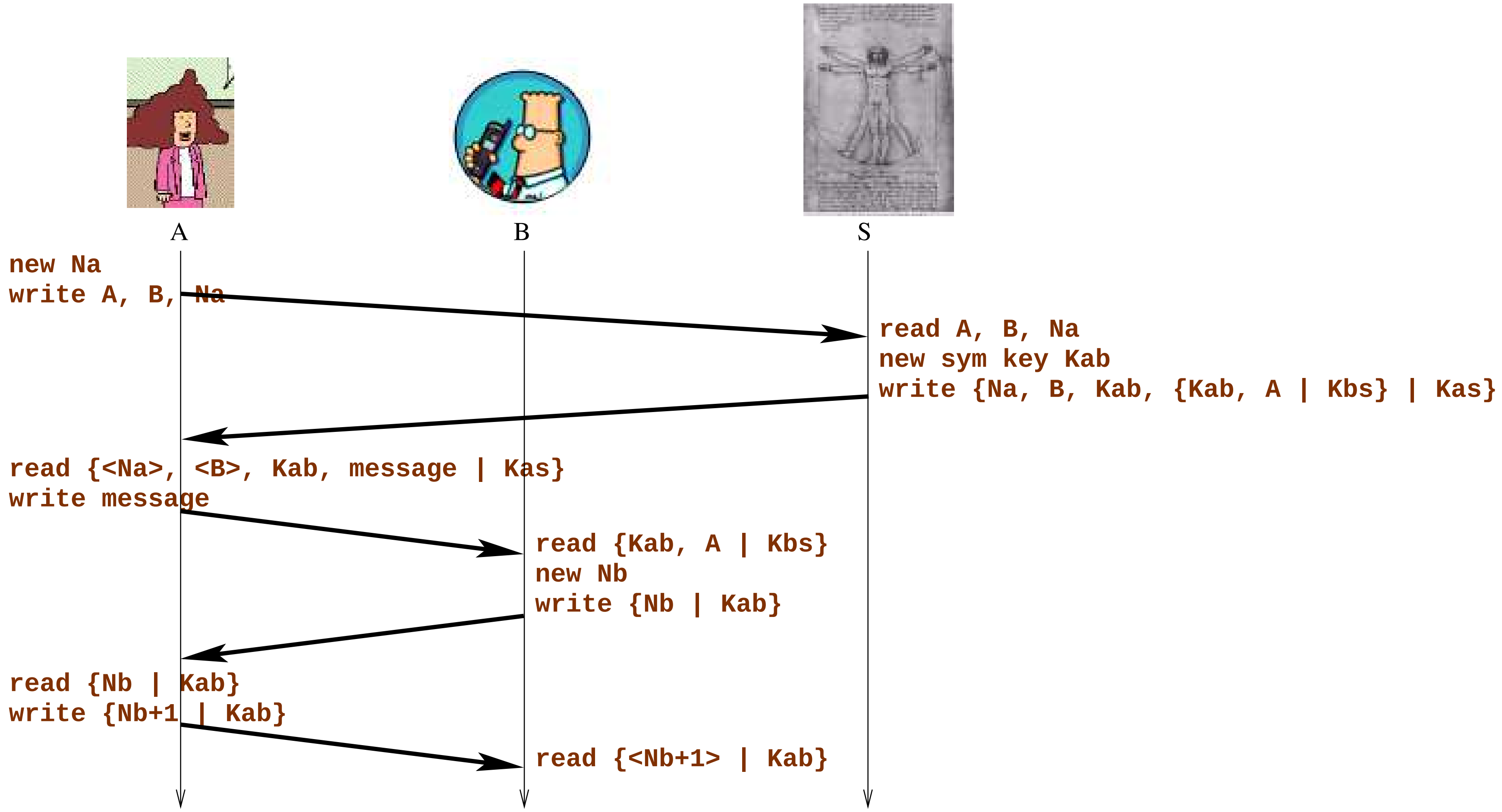

Indécidable
Protocoles cryptographiques
Garantie de sécurité
Modélisation
Programmes Prolog
= clauses de Horn
Non−contradiction
Approximation cartésienne
Programmes Prolog restreints
= automates d'arbres
Vacuité/Non−contradiction
Décidable

Améliorations possibles

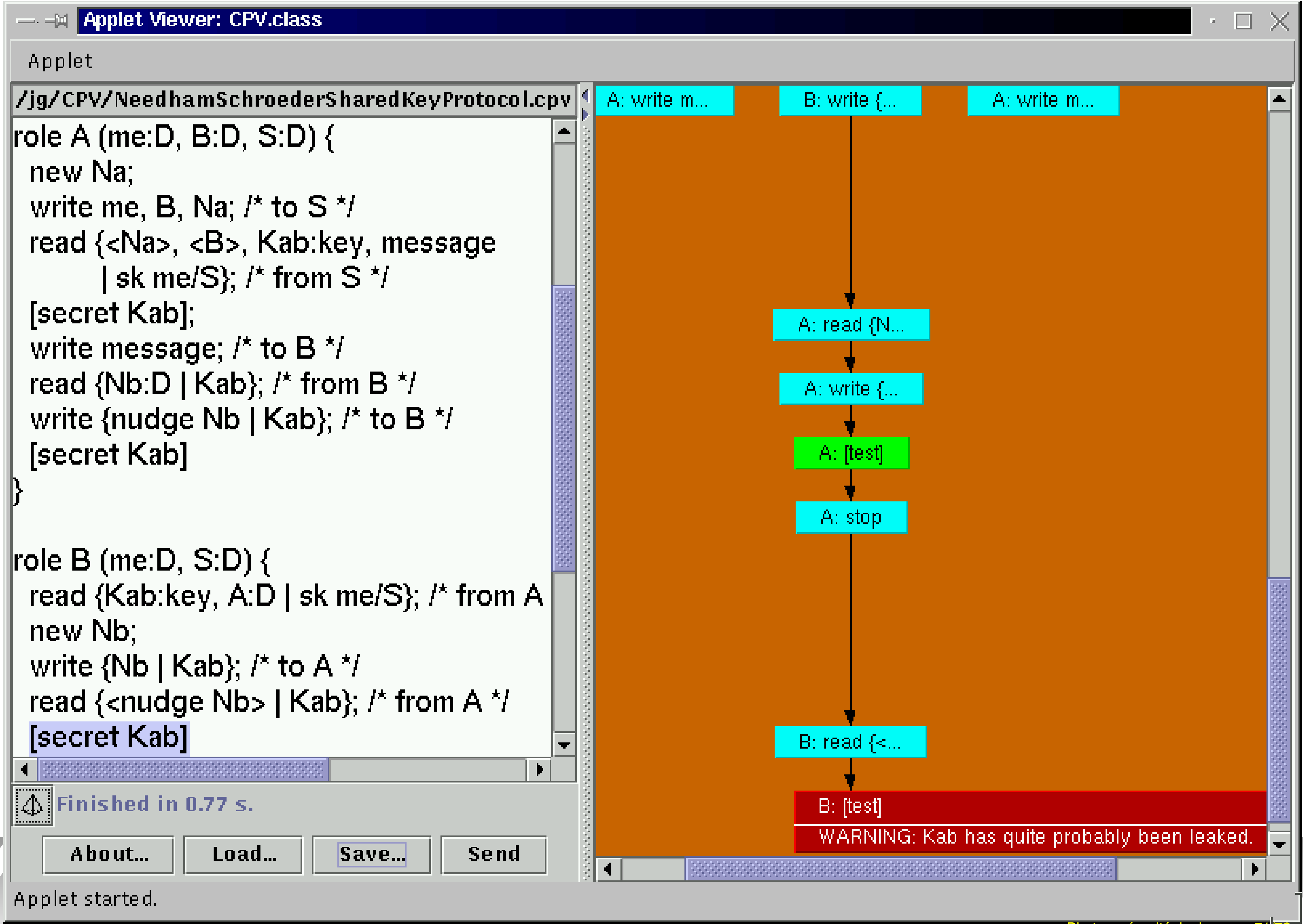
- Automates plus fins (à une mémoire [ComonCortierMitchell01] $\Longrightarrow$ sous-classe décidable large de protocoles crypto) ;
- Ne traiter par automates que les parties état initiaux et déductions de l'intrus (pas de perte d'information ici), et "exécuter symboliquement" le programme (interprétation abstraite [CousotCousot77], cf. [Monniaux99,JGL00]).

Laboratoire Spécification et Vérification

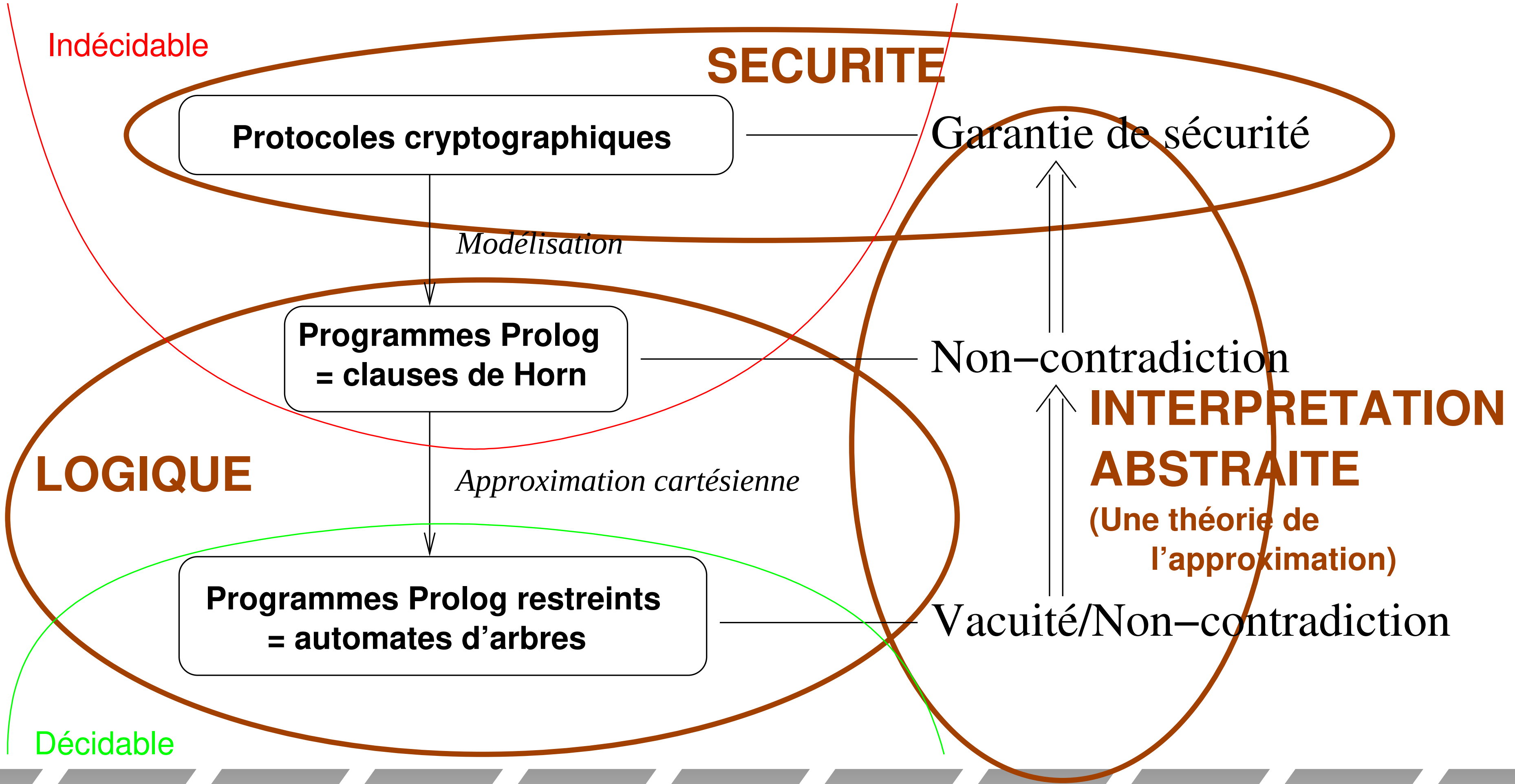
Testons tout ça [NeedhamSchroeder78]



Laboratoire Spécification et Vérification


Applet Viewer: CPV.class
Applet
/jg/CPV/NeedhamSchroederSharedKeyProtocol.cpv
role A (me:D, B:D, S:D) {
new Na;
write me, B, Na; /* to S */
read {<Na>, <B>, Kab:key, message
| sk me/S}; /* from S */
[secret Kab];
write message; /* to B */
read {Nb:D | Kab}; /* from B */
write {nudge Nb | Kab}; /* to B */
[secret Kab]
}
role B (me:D, S:D) {
read {Kab:key, A:D | sk me/S}; /* from A
new Nb;
write {Nb | Kab}; /* to A */
read {<nudge Nb> | Kab}; /* from A */
[secret Kab]
Finished in 0.77 s.
About...
Load...
Save...
Send
A: write m...
B: write {...
A: write m...
A: read {N...
A: write {...
A: [test]
A: stop
B: read {<...
B: [test]
WARNING: Kab has quite probably been leaked.
Applet started.

Le paysage final



Laboratoire Spécification et Vérification