Développement de systèmes sécurisés avec l'atelier FoCaLiZe

M. Jaume[1] & R. Rioboo[2]

1: SPI - LIP6 - UPMC,
4 Place Jussieu, 75252 Paris Cedex 05, France
Mathieu.Jaume@lip6.fr
2: ENSIIE, CPR-CEDRIC-CNAM,
1 Square de la Résistance, 91025 Evry Cedex, France
Renaud.Rioboo@ensiie.fr

Nous présentons dans cet article un ensemble de définitions réutilisables, et formellement vérifiées avec l'atelier FoCaLiZe, des concepts utilisés pour modéliser des systèmes sécurisés par une politique de sécurité. Ces différents concepts sont introduits très progressivement par le biais de composants logiciels abstraits connectés les uns aux autres par le jeu des mécanismes d'héritage et de paramétrisation de FoCaLiZe. Tout au long de ce développement, la définition d'un système de contrôle d'accès sécurisé illustre ces concepts généraux à l'aide d'instances concrètes.

1. Introduction

Les systèmes critiques sont soumis à évaluation de conformité vis-à-vis des normes de développement applicables dans le domaine considéré. Ces normes requièrent souvent l'application de méthodes formelles pour garantir des propriétés de sûreté et de sécurité essentielles à l'utilisation de ces systèmes. Nous proposons ici un cadre permettant la spécification et le développement de systèmes sécurisés avec l'atelier FoCaLiZe [Foc10] permettant de faciliter, et parfois d'automatiser, la mise en œuvre des méthodes formelles tout au long du cycle de vie de développement d'un système. Afin de garantir des propriétés de sécurité sur un système, il est souvent nécessaire de spécifier une politique de sécurité permettant d'indiquer quels sont les objectifs à atteindre (confidentialité, intégrité, confinement, etc.) ainsi qu'un mécanisme opérationnel permettant de contraindre le comportement du système de manière à ce qu'il respecte la politique. Il existe plusieurs manières de spécifier une politique de sécurité, parmi lesquelles on distingue deux principales approches :

- l'approche par propriétés (qui consiste à spécifier les propriétés de sécurité que doivent respecter les états du système pour être considérés comme sûrs),
- l'approche par règles (qui consiste à spécifier les conditions de déclenchement d'une action, c'est-à-dire les propriétés que doit satisfaire un état du système pour qu'une action soit autorisée lorsque le système se trouve dans cet état).

Etant donnée une notion commune d'état, ces deux approches, formalisées dans [Jau10], ne sont pas toujours équivalentes : tandis qu'une politique exprimée "par propriétés" peut toujours être exprimée en adoptant l'approche par règles, il existe des politiques exprimées "par règles" qui ne peuvent être spécifées en adoptant l'approche par propriétés (c'est le cas lorsqu'il existe un état du système accessible de plusieurs manières différentes, certaines étant autorisées, et d'autres non). Toutefois, dans ce cas, il est toujours possible d'enrichir la notion d'état du système (en ajoutant une

notion de passé du système) pour se ramener à l'approche par propriétés. Nous adoptons donc ici l'approche par propriétés, qui permet de distinguer clairement la spécification de propriétés de sécurité du mécanisme opérationnel permettant de garantir ces propriétés. Cette approche est classique dans le développement de logiciels.

Le programme chargé de mettre en application une politique de sécurité (le moniteur de référence) est souvent considéré comme l'une des clés de voûte de la sécurité d'un système. Sa conception et son développement doivent donc être menés de manière à garantir sa fiabilité et sa sûreté. En effet, toute faille au sein de ce programme pourrait entraîner des violations de la politique de sécurité. Nous proposons donc ici une formalisation, et une implantation avec l'environnement de développement FoCaLiZe [Foc10], de la notion de politique de sécurité "par propriétés", ainsi qu'un mécanisme opérationnel permettant de contraintre les exécutions d'un système de transition afin que les états accessibles de ce système satisfassent la politique de sécurité. FoCaLiZe est un environnement de développement fournissant un langage fonctionnel incorporant des aspects orientés objets, permettant d'écrire des spécifications, des programmes et les preuves de correction des programmes vis-à-vis des spécifications. C'est donc un environnement particulièrement adapté au développement de systèmes "sûrs", qui permet d'accroître la confiance dans le code produit. De plus, les traits objets du langage fourni par FoCaLiZe permettent l'obtention d'un code générique, pouvant être instancié (par raffinements successifs de spécifications) pour différentes politiques et différents systèmes. Nous présentons ici deux exemples d'utilisation de ce code dans le domaine du contrôle d'accès, mais l'architecture proposée permet d'envisager bien d'autres domaines.

Ce développement est le prolongement de travaux antérieurs visant à définir un cadre générique pour la spécification, l'analyse et la mise en œuvre de politiques de sécurité. Le cadre proposé permet de généraliser les approches présentées dans [JM07, JM08] (contrôle d'accès), [JTM10, JTM11] (contrôle de flots d'information), et [BCJK11] (utilisation de systèmes de réécriture pour la définition de politiques). L'objectif à plus long terme de ce travail est de fournir une bibliothèque certifiée de politiques de sécurité facilement réutilisable, et de concevoir au sein du cadre proposé des mécanismes de comparaison et de composition. Il existe encore relativement peu d'implantations génériques de systèmes sécurisés basés sur la preuve formelle. L'approche que nous présentons ici est similaire à celle développée dans [SP07], où les auteurs proposent, à partir d'une définition abstraite d'une notion simple de politique de sécurité, d'utiliser les mécanismes de raffinement de la méthode B pour développer un moniteur de référence détectant les violations de politiques de sécurité d'un réseau de type TCP/IP. Toutefois, notre développement utilise des mécanismes différents, ceux de l'atelier FoCaLiZe, et repose sur une notion abstraite de politique qui permet l'expression d'un plus grand nombre de politiques concrètes.

2. L'environnement de développement FoCaLiZe

L'atelier FoCaLiZe [Foc10, Rio09], et la méthodologie sous-jacente, ont été conçus pour répondre au besoin d'accroissement de la confiance dans le développement de systèmes critiques. FoCaLiZe est utilisé dans différents domaines et en particulier dans le domaine de la sécurité. FoCaLiZe a par exemple été utilisé avec succès pour formaliser la politique de sécurité au sol d'un aéroport [DÉDG06] ou encore pour filtrer les requêtes d'une base de données afin d'appliquer une politique de contrôle d'accès [BM07]. L'atelier FoCaLiZe fournit un environnement de développement intégré (IDE) logiquement fondé, avec une sémantique claire et qui permet d'obtenir des implantations efficaces. FoCaLiZe offre un langage muni de traits objets (héritage multiple, liaison tardive, redéfinition, ...) permettant non seulement de structurer un développement de manière à le rendre facilement réutilisable, mais aussi d'obtenir un logiciel par raffinements successifs en passant progressivement de la spécification à l'implantation. FoCaLiZe permet d'écrire au sein d'un même cadre de travail des spécifications, des programmes et des preuves. Cet atelier permet aux programmeurs d'écrire les preuves formelles d'adéquation entre le code et la spécification, ces preuves sont faites par le

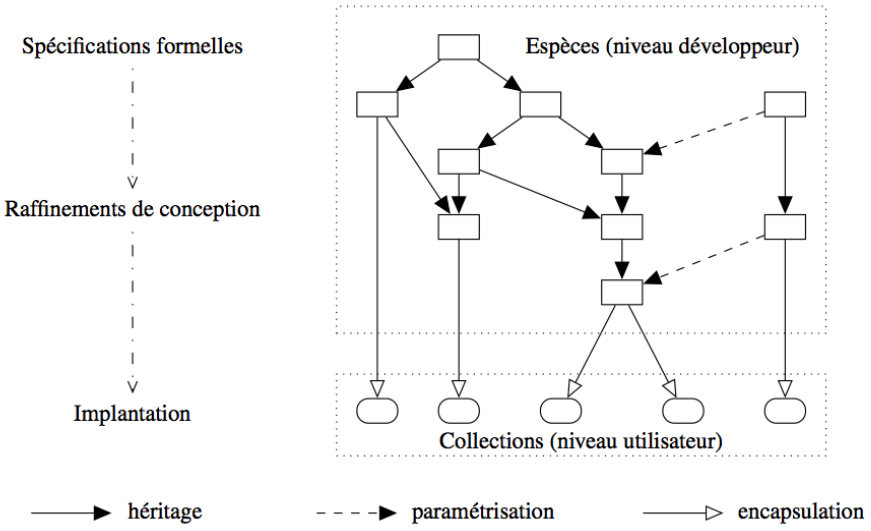

Figure 1: Construction d'un programme FoCaLiZe

démonstrateur automatique Zenon ([BDD07]) puis vérifiées par Coq [Coq10].

Un programme FoCaLiZe est la donnée d'une hiérarchie d'espèces. Une espèce peut être vue comme un ensemble de méthodes, identifiées par leur nom et une représentation de données. Chaque méthode peut être soit déclarée (constantes, opérations et propriétés) soit définie (implantations des opérations et preuves des théorèmes), et peut être opérationnelle ou logique. Une espèce définit ainsi des opérations sur des valeurs d'un type support que l'on nomme entités. Nous utiliserons ici le mot individu pour ne pas interférer avec les entités des systèmes sécurisés. Ces individus sont ensuite encapsulés dans un type abstrait de données que l'on nomme collection. Une collection est créée par une espèce (dite complète) dont toutes les fonctionalités sont implantées.

- Les méthodes opérationnelles sont les constantes ou les fonctions de toute collection implantant l'espèce. Les méthodes qui ne sont que déclarées sont appelées des signatures. Le langage utilisé pour définir ces méthodes est similaire à celui du noyau fonctionnel d'OCaml, avec une construction permettant d'appeler une méthode d'une espèce donnée.

- Les méthodes logiques décrivent les propriétés des méthodes opérationnelles. Dans ce contexte, la déclaration d'une méthode logique est simplement l'énoncé d'une propriété, tandis que la définition d'une méthode logique est la preuve que cette propriété est vraie.

- Le langage utilisé pour les énoncés de propriétés est composé des connecteurs logiques de base et des quantificateurs existentiels et universels portant sur les individus d'une collection. Ces propriétés sont donc des formules de logique du premier ordre, contenant les noms des méthodes liées par déclaration/définition, héritage ou paramétrisation.

La figure 1 illustre la construction d'une hiérarchie d'espèces FoCaLiZe. La compilation d'un programme FoCaLiZe s'effectue en plusieurs étapes. Le fichier source est analysé par le compilateur, qui effectue un calcul de dépendances permettant d'éliminer des cycles dans les dépendances (condition suffisante pour éviter des incohérences logiques), de gérer l'héritage multiple et la liaison tardive (relier un nom de méthode à sa définition la plus récente), d'effacer les preuves dépendant des définitions

de fonctions en cas de redéfinition de celles-ci. Puis, le fichier est traduit vers un fichier OCaml, qui correspond, une fois compilé, à un programme exécutable. Le fichier est également compilé vers un source Coq, contenant toute la structuration du programme ainsi que toutes les preuves. Chaque preuve écrite en FoCaLiZe est analysée par l'outil Zenon ([BDD07]) et transformée en un terme ou un script de preuve Coq. Enfin, le fichier est compilé vers un format intermédiaire de documentation, permettant de générer du XML, du LaTeX, de l'UML, etc. Un outil permettant de démontrer la terminaison des fonctions récursives est en cours d'intégration. FoCaLiZe propose également un outil de génération automatique de test et un outil graphique de représentation des dépendances entre espèces et méthodes.

3. Systèmes et politiques : implantation avec FoCaLiZe

Nous présentons ici une architecture de développement de systèmes sécurisés dans l'environnement FoCaLiZe. Dans un premier temps, nous présentons une implantation de la notion classique de systèmes de transition, puis nous enrichissons cette notion de manière à pouvoir introduire une notion d'état sûr à partir de laquelle nous définissons un système sécurisé. Dans un deuxième temps, nous montrons comment la notion de politique de sécurité peut être définie et utilisée pour définir la notion d'état sûr. Nous illustrons ces notions avec un exemple issu du domaine du contrôle d'accès. Nous ne donnons ici que les extraits significatifs du code : nous omettons donc certaines spécifications, définitions et preuves, remplacées par "..." (l'ensemble des preuves a été facilement obtenu avec Zenon).

3.1. Systèmes sécurisés

3.1.1. Systèmes de transition

Les systèmes que nous contraignons afin qu'ils vérifient des propriétés de sécurité sont représentés par des systèmes de transitions étiquetées (LTS, *Labelled Transition System*). Formellement, un LTS est un quadruplet $\mathbb{S} = (\Sigma, \Sigma^0, L, \delta)$ où Σ est l'ensemble des états du système, $\Sigma^0 \subseteq \Sigma$ est l'ensemble des états initiaux, L est l'ensemble des étiquettes et $\delta \subseteq \Sigma \times L \times \Sigma$ est une relation de transition (on écrira $\sigma_1 \xrightarrow{l}_\delta \sigma_2$ au lieu de $(\sigma_1, l, \sigma_2) \in \delta$). Pour implanter la notion de LTS avec FoCaLiZe, on introduit les espèces `State`, `Label` et `Transition_system`. Une collection implantant l'espèce `State` représente à la fois Σ et Σ^0 (la méthode `is_initial` permet de spécifier les états initiaux). Cette espèce hérite de l'espèce `Setoid` qui spécifie les ensembles non vides munis d'une relation d'équivalence. Une collection implantant l'espèce `Label` représente l'ensemble L : cette espèce n'a ni propriété ni méthode, exceptées celles de l'espèce des sétoïdes dont elle hérite, laissant ainsi libre la définition des étiquettes. Un individu d'une collection implantant l'espèce `Transition_system` représente un LTS : cette espèce est paramétrée par les états, les étiquettes, la collection implantant les triplets de $\Sigma \times L \times \Sigma$ et déclare une méthode correspondant à la fonction caractéristique de la relation de transition δ. On introduit également la notion d'accessibilité : un état est accessible ssi il existe une suite finie de transitions depuis un état initial jusqu'à cet état.

$$\sigma \text{ est accessible} \Leftrightarrow \exists l_1, \cdots, l_n \in L \ \exists \sigma_0 \in \Sigma^0 \quad \sigma_0 \xrightarrow{l_1}_\delta \sigma_1 \xrightarrow{l_2}_\delta \sigma_2 \xrightarrow{l_3}_\delta \cdots \xrightarrow{l_n}_\delta \sigma_n$$

On utilise ici une structure de liste et le quantificateur existentiel pour spécifier la propriété d'accessibilité. Enfin, on introduit deux définitions logiques permettant de caractériser les systèmes déterministes et complets :

- un système $\mathbb{S}$ est déterministe ssi : $\forall \sigma, \sigma_1, \sigma_2 \in \Sigma \ \ \forall l \in L \ \ (\sigma \xrightarrow{l}_\delta \sigma_1 \wedge \sigma \xrightarrow{l}_\delta \sigma_2) \Rightarrow \sigma_1 = \sigma_2$
- un système $\mathbb{S}$ est complet ssi : $\forall \sigma_1 \in \Sigma \ \ \forall l \in L \ \ \exists \sigma_2 \in \Sigma \ \ \sigma_1 \xrightarrow{l}_\delta \sigma_2$

Leur description en FoCaLiZe utilise une construction logique qui permet de définir des énoncés.

```
species State =
 inherit Setoid;
 signature is_initial : Self -> bool;
end;;
species Label = inherit Setoid; end;;
species Transition_system (S is State, L is Label, SLS is Abstract_triple(S, L, S)) =
 inherit Basic_object;
 signature delta : S -> L -> S -> bool;
 logical let is_deterministic = all x y z : S, all l : L,
    ( delta (x, l, y) /\ delta (x, l, z) ) -> S!equal (y, z);
 logical let is_complete = all x : S, all l : L, ex y : S, delta (x, l, y);
 let rec is_path_from_init_to(p, x) = match p with
     | [] -> S!is_initial (x)
     | h :: q -> S!equal (SLS!third(h), x) && delta (SLS!first(h), SLS!second(h), x)
                 && is_path_from_init_to (q, SLS!first(h)) ;
 logical let is_reachable (x) = ex p : list (SLS), is_path_from_init_to (p, x);
end;;
```

Lorsqu'ils sont déterministes et complets, les systèmes de transition sont souvent définis à partir d'une fonction de transition $\tau : \Sigma \times L \to \Sigma$. Nous introduisons donc l'espèce `Operational_transition_system` qui hérite de l'espèce `Transition_system`. Cette espèce déclare une fonction de transition `transition` à partir de laquelle il est possible de définir la relation de transition δ, accompagnée des preuves de déterminisme et de complétude de cette relation. Nous ne montrons ici que les indications de preuve du déterminisme fournies à Zenon pour faire la preuve automatiquement. Les autres preuves s'obtiennent sans problème de manière similaire.

```
species Operational_transition_system (S is State, L is Label, SLS is Abstract_triple(S, L, S)) =
 inherit Transition_system (S, L, SLS);
 signature transition : S -> L -> S;
 property transition_equal_compat: all s_1 s_2: S, all l: L,
    S!equal(s_1, s_2) -> S!equal(transition(s_1, l), transition(s_2, l));
 let delta(s_1, l, s_2) = S!equal(transition(s_1, l), s_2);
  (* this serves as specification if delta is redefined *)
  theorem transition_is_valid: all x y: S, all l: L, S!equal(transition(x, l), y) <-> delta(x, l, y)
   proof = by definition of delta
              property S!equal_reflexive;
 theorem deterministic : is_deterministic
  proof = by property transition_is_valid
             definition of is_deterministic
             property S!equal_reflexive, S!equal_symmetric, S!equal_transitive, transition_equal_compat;
 theorem complete : is_complete
  proof = ...
end;;
```

Exemple : Systèmes d'accès Nous montrons ici comment obtenir une implantation d'un système d'accès à partir des espèces que nous venons d'introduire. Un système d'accès est un système dont les états représentent des ensembles finis d'accès effectués par des sujets (appartenant à un ensemble fini $\mathcal{S}$) sur des objets (appartenant à un ensemble fini $\mathcal{O}$) selon certains modes (appartenant à un ensemble fini $\mathcal{A}$). Chaque accès est représenté par un triplet de la forme $(s, o, a) \in \mathcal{S} \times \mathcal{O} \times \mathcal{A}$ exprimant que le sujet s accède à l'objet o selon le mode d'accès a. Nous introduisons donc tout d'abord les trois espèces suivantes, qui héritent toutes de l'espèce spécifiant les ensembles finis.

```
species Subject = inherit Finite_set; end;;
species Object = inherit Finite_set; end;;
species Access_mode = inherit Finite_set; end;;
```

L'espèce `Accesses` spécifiant les ensembles finis d'accès peut alors être définie : un individu d'une collection implantant cette espèce correspond à une partie de $\mathcal{S} \times \mathcal{O} \times \mathcal{A}$. Cette espèce est donc

paramétrée par les espèces spécifiant les sujets, les objets, les modes d'accès et le produit cartésien $\mathcal{S} \times \mathcal{O} \times \mathcal{A}$, et introduit (avec leurs spécifications) une méthode `alpha` permettant d'associer la fonction caractéristique de l'ensemble d'accès correspondant à un individu, ainsi que les méthodes permettant de considérer un ensemble vide d'accès `empty`, l'ajout `add` et le retrait `release` d'un accès à un ensemble d'accès, une méthode `to_list` permettant d'obtenir une liste contenant les accès appartenant à un individu, une méthode `is_contained` définissant l'inclusion sur les ensembles d'accès. La définition de l'égalité et les preuves des propriétés de cette relation sont définies à partir de `is_contained`. Cette espèce est par la suite implantée en utilisant des listes.

```
species Accesses(S is Subject, O is Object, A is Access_mode, SOA is Abstract_triple(S, O, A)) =
 inherit Finite_set;
 signature alpha : Self -> SOA -> bool;
 signature empty : Self;
 signature add : Self -> SOA -> Self;
 signature release : Self -> SOA -> Self;
 signature to_list : Self -> list(SOA) ;
 signature is_contained : Self -> Self -> bool;
 property empty_spec : all t : SOA, ~ alpha (empty, t);
 property add_spec : all x : Self, all t_1 t_2 : SOA,
    alpha (add (x, t_1), t_2) <-> SOA!equal(t_1, t_2) \/ alpha(x, t_2);
 property release_spec : all x : Self, all t_1 t_2 : SOA,
    alpha(release(x, t_2), t_1) <-> SOA!different (t_1, t_2) /\ alpha(x, t_1);
 property is_contained_spec : all x y : Self,
    is_contained (x, y) <-> ( all t : SOA, alpha (x, t) -> alpha (y, t) );
 theorem is_contained_reflexive : all x : Self, is_contained (x, x)
  proof = by property is_contained_spec;
 theorem is_contained_transitive : all x y z : Self,
    (is_contained (x, y) /\ is_contained (y, z)) -> is_contained (x, z)
  proof = by property is_contained_spec;
 let equal(x, y) = is_contained(x, y) && is_contained(y, x);
 proof of equal_reflexive = by definition of equal
                            property is_contained_reflexive;
 proof of equal_transitive = by definition of equal
                             property is_contained_transitive;
 proof of equal_symmetric = by definition of equal;
end;;
```

Il est maintenant possible d'implanter un système d'accès $\mathbb{S}_{ac} = (\Sigma_{ac}, \Sigma^0_{ac}, L_{ac}, \delta_{ac})$, où Σ_{ac} contient les ensembles d'accès possibles, et correspond donc à l'ensemble des parties de $\mathcal{S} \times \mathcal{O} \times \mathcal{A}$, Σ^0_{ac} est le singleton contenant l'ensemble vide (nous supposons qu'à l'état initial, aucun accès n'est effectué), L_{ac} est l'ensemble des étiquettes de la forme $\langle +, s, o, a\rangle$ et $\langle -, s, o, a\rangle$ exprimant la demande d'accès $(+)$ ou la demande de relâchement d'accès (-) du sujet s sur l'objet o selon le mode a, et δ_{ac} est la relation de transition définie par :

$$\delta_{ac} = \left\{ A \xrightarrow{\langle +,s,o,a\rangle}_{\delta_{ac}} A \cup \{(s,o,a)\} \right\} \cup \left\{ A \xrightarrow{\langle -,s,o,a\rangle}_{\delta_{ac}} A \setminus \{(s,o,a)\} \right\}$$

L'espèce des états peut donc se définir comme suit :

```
species State_ac(S is Subject, O is Object, A is Access_mode, SOA is Abstract_triple(S, O, A))=
 inherit State, Accesses (S, O, A, SOA);
 let is_initial (x) = equal (x, empty);
 theorem initial_is_empty : all x : Self, is_initial (x) <-> equal (x, empty)
 proof = by definition of is_initial;
end;;
```

Cette espèce hérite à la fois de l'espèce des états et de l'espèce des ensembles finis d'accès. La méthode `is_initial` caractérise l'ensemble vide d'accès et est accompagnée de sa spécification qui est prouvée (cette spécification "paraphrase" la définition afin de contraindre l'utilisateur à maintenir cette propriété en cas de redéfinition [PJ03]). L'espèce des étiquettes, qui hérite de l'espèce `Label`, fixe la

représentation d'une étiquette par un quadruplet et introduit les "accesseurs" aux composants de ces quadruplets.

```
type action = | Add | Release;;
species Label_ac (S is Subject, O is Object, A is Access_mode) =
 inherit Label;
 representation = action * S * O * A;
 let label (x, s, o, a) : Self = (x, s, o, a);
 let action (l : Self) = match l with | (x, _, _, _) -> x;
 let subject (l : Self) = match l with | (_, s, _, _) -> s;
 let object_ac (l : Self) = match l with | (_, _, o, _) -> o;
 let access_mode (l : Self) = match l with | (_, _, _, a) -> a;
 theorem label_spec : all x : action, all s : S, all o : O, all a : A,
    action(label(x, s, o, a)) = x             && S!equal(subject(label (x, s, o, a)), s)
 && O!equal(object_ac(label(x, s, o, a)), o)  && A!equal(access_mode(label(x, s, o, a)), a)
  proof = ... ;
end;;
```

Enfin, le système $\mathbb{S}_{ac}$ peut être introduit par héritage de l'espèce `Transition_system` en définissant la relation de transition accompagnée de sa spécification et des preuves que cette relation est déterministe et complète. Puisqu'un individu d'une collection implantant cette espèce est entièrement déterminé par les paramètres et la définition de la relation de transition (et qu'aucune transformation n'est définie sur les systèmes de transition), aucune représentation n'est utile, ce qui s'exprime en fixant la représentation à `unit`.

```
species Transition_system_ac(S is Subject, O is Object, A is Access_mode,
 SOA is Abstract_triple(S, O, A), St is State_ac(S, O, A, SOA), L is Label_ac(S, O, A),
 SLS is Abstract_triple(St, L, St)) =
 inherit Transition_system (St, L, SLS);
 representation = unit;
 let delta(x, l, y)=
 ((L!action(l)=Add && St!equal(y, St!add(x, make(L!subject(l), L!object_ac(l), L!access_mode(l)))))
 || (L!action(l)=Release
     && St!equal(y, St!release(x, make(L!subject(l), L!object_ac(l), L!access_mode(l))))));
 theorem deterministic : is_deterministic ;
  proof = ...
 theorem complete : is_complete ;
  proof = ...
end;;
```

Les théorèmes `determinist` et `complete` permettent aussi d'envisager la définition d'un système opérationnel comme suit :

```
species Operational_transition_system_ac(S is Subject, O is Object, A is Access_mode,
   SOA is Abstract_triple(S, O, A), St is State_ac(S, O, A, SOA), L is Label_ac(S, O, A),
  SLS is Abstract_triple(St, L, St)) =
 inherit Transition_system_ac(S, O, A, SOA, St, L, SLS), Operational_transition_system(St, L, SLS);
 let transition(x, l) = match L!action(l) with
    | Add -> St!add (x, make(L!subject (l), L!object_ac (l), L!access_mode (l)))
    | Release -> St!release (x, make(L!subject (l), L!object_ac (l), L!access_mode (l)));
 proof of transition_is_valid = by type Action
                                definition of transition ;
end;;
```

Cette espèce hérite à la fois de `Transition_system_ac` et de `Operational_transition_system` qui sont des espèces qui introduisent toutes les deux une définition pour la méthode `delta`. Dans ce cas, avec FoCaLiZe, c'est la définition de la dernière méthode introduite dans l'héritage multiple qui est prise en compte : la définition de la méthode `delta` de l'espèce `Operational_transition_system_ac` sera donc celle provenant de l'espèce `Operational_transition_system`.

3.1.2. Systèmes sécurisés

Etats sûrs La notion de système sécurisé repose sur celle d'états sûrs. Nous montrons donc comment, à partir d'un ensemble d'états Σ, nous construisons un ensemble d'états Σ' sur lequel est défini un prédicat Ω caractérisant les états sûrs. Pour cela, nous introduisons l'espèce `Secure_State`, paramétrée par un ensemble d'états et héritant de l'espèce `State`. Cette espèce introduit une méthode `is_secure` caractérisant les états sûrs (un élément de Σ' qui satisfait le prédicat Ω) et restreint les états initiaux aux états sûrs. Un individu d'une collection implantant l'espèce `Secure_State` correspond à un état d'un système sécurisé (un élément de Σ') et sera obtenu à partir d'un individu de l'espèce `State` (un élément de Σ) : intuitivement la notion d'état sûr s'obtient en enrichissant une notion d'état (en ajoutant généralement de l'information permettant de définir la méthode `is_secure`). Nous montrerons dans la suite comment cette construction peut être obtenue à partir de la notion de politique de sécurité.

```
species Secure_state (S is State) =
 inherit State;
 signature state : Self -> S;
 signature is_secure : Self -> bool;
 signature make : Self -> S -> Self;
 property make_spec : all x: Self, equal(make(x, state(x)), x);
 let is_initial (x : Self) = S!is_initial (state (x)) && is_secure (x);
 theorem is_initial_spec : all x : Self,
   is_initial (x) <-> (S!is_initial (state (x)) /\ is_secure (x))
  proof = by definition of is_initial;
end;;
```

La méthode `state` permet, étant donné un état de l'espèce `Secure_State`, de connaître l'état de l'espèce `State` qui lui correspond, c'est-à-dire celui à partir duquel il a été obtenu. Il s'agit donc d'une fonction de projection $I_S : \Sigma' \to \Sigma$. Réciproquement, la méthode `make` permet, étant donnés un individu σ' d'une collection implantant `Secure_State` et un individu σ d'une collection Σ implantant `State`, de construire un individu σ'' d'une collection implantant `Secure_State` obtenu à partir de σ en l'enrichissant de la même manière que lors de la construction de σ'. En d'autres termes, si σ' a été obtenu à partir d'un état σ_0, alors $\texttt{make}(\sigma', \sigma) = \sigma''$ signifie que l'information "ajoutée" pour transformer σ_0 en σ' est la même que celle "ajoutée" pour transformer σ en σ''. La propriété `make_spec` fournit la spécification de cette construction. Enfin, le théorème `is_initial_spec` permet d'imposer que toute redéfinition de la méthode `is_initial` respecte cette spécification.

Construction d'un système sécurisé La notion d'état sûr qui peut être associée aux états d'un système de transition $\mathbb{S}$ permet de construire, à partir de $\mathbb{S}$, un nouveau système dont les états accessibles sont à la fois issus d'états accessibles avec $\mathbb{S}$ et sûrs. Un tel système est dit sécurisé. Plus formellement, à partir du système $\mathbb{S} = (\Sigma, \Sigma^0, L, \delta)$, d'un ensemble Σ' d'états (construit à partir de Σ) associé à :

- une fonction de projection $I_S : \Sigma' \to \Sigma$,
- un prédicat Ω défini sur Σ' caractérisant les états sûrs,

on définit le système sécurisé $\mathbb{S}_\Omega = (\Sigma', \Sigma^0_\Omega, L, \delta_\Omega)$ où :

$$\Sigma^0_\Omega = \left\{\sigma \in \Sigma' \mid I_S(\sigma) \in \Sigma^0 \wedge \Omega(\sigma)\right\} \quad \delta_\Omega = \left\{\sigma_1 \xrightarrow{l}_{\delta_\Omega} \sigma_2 \mid I(\sigma_1) \xrightarrow{l}_{\delta} I(\sigma_2) \wedge \Omega(\sigma_1) \Rightarrow \Omega(\sigma_2)\right\}$$

Cette construction permet donc de supprimer les transitions d'un système qui à partir d'un état sûr conduisent à un état non sûr. On montre donc facilement (par induction) que cette construction garantit que tout état σ accessible avec le système $\mathbb{S}_\Omega$ est sûr (i.e., est tel que $\Omega(\sigma)$). Pour construire la

preuve, on procède par induction sur la liste établissant l'accessibilité d'un état : on prouve séparément les preuves pour le cas de base (`secure_base`) et pour l'étape inductive (`secure_induction`), puis Zenon construit la preuve finale (`all_secure`) à partir de ces deux preuves. La propriété souhaitée s'obtient alors simplement par définition de la notion d'accessibilité.

```
species Secure_transition_system (St is State, L is Label, SLT is Abstract_triple(St, L, St),
 Sys is Transition_system (St, L, SLT), Ss is Secure_state (St), SLS is Abstract_triple(Ss, L, Ss)) =
 inherit Transition_system (Ss, L, SLS);
 let delta (x : Ss, l : L, y : Ss) =
    Sys!delta (Ss!state (x), l, Ss!state (y)) && ( ~~ Ss!is_secure (x) || Ss!is_secure (y) );
 theorem secure_base: all x: Ss, is_path_from_init_to([], x) -> Ss!is_secure (x)
    proof = by definition of is_path_from_init_to
               property Ss!is_initial_spec ;
 theorem secure_induction: all l: list(SLS),
   (all z: Ss, is_path_from_init_to(l, z) -> Ss!is_secure(z)) ->
   all x: Ss, all t:SLS, is_path_from_init_to((t :: l), x) -> Ss!is_secure(x)
  proof = by definition of is_path_from_init_to, delta
            property SLS!is_first, SLS!is_second, SLS!is_third, SLS!equal_spec, delta_equal_compat ;
 theorem all_secure : all l: list(SLS), all x: Ss, is_path_from_init_to(l, x) -> Ss!is_secure (x)
  proof = by property secure_base, secure_induction;
 theorem reachable_is_secure: all x: Ss, is_reachable(x) -> Ss!is_secure(x)
  proof = by definition of is_reachable property all_secure;
end;;
```

Puisque cette construction permet de sécuriser un système en supprimant des transitions "non sûres", la sécurisation d'un système défini à partir d'une fonction de transition conduit à un système dont la relation de transition n'est plus complète (seule la propriété de déterminisme est conservée). Pour palier à ce problème, une solution consiste à remplacer toute transition "non sûre" à partir d'un état σ par une transition de σ vers σ, ce qui permet de conserver une relation de transition complète[1]. Plus formellement, la fonction de transition $\tau_\Omega : \Sigma' \times L \to \Sigma'$ est obtenue à partir $\tau : \Sigma \times L \to \Sigma$ comme suit :

$$\tau_\Omega(\sigma_1, l) = \begin{cases} \texttt{make}(\sigma_1, \tau(I_S(\sigma_1), l)) & \text{si } \Omega(\sigma_1) \Rightarrow \texttt{make}(\sigma_1, \tau(I_S(\sigma_1), l)) \\ \sigma_1 & \text{sinon} \end{cases}$$

```
species Secure_operational_transition_system(St is State, L is Label, SLT is Abstract_triple(St, L, St),
 Sys is Operational_transition_system (St, L, SLT), Ss is Secure_state (St),
 SLS is Abstract_triple(Ss, L, Ss)) =
inherit Secure_transition_system (St, L, SLT, Sys, Ss, SLS), Operational_transition_system (Ss, L, SLS);
 let transition(x, l) =
     let s = Sys!transition(Ss!state(x), l) in let y = Ss!make(x, s) in
     if ((~~ Ss!is_secure(x)) || Ss!is_secure(y)) then y else x;
end;;
```

On remarquera que les deux espèces dont hérite l'espèce `Secure_operational_transition_system` fournissent une définition de la méthode `delta`. Avec le mécanisme d'héritage multiple de FoCaLiZe, c'est la définition provenant de l'espèce `Operational_transition_system` qui est prise en compte et permet de ne pas invalider de preuves.

3.2. Systèmes sécurisés par une politique de sécurité

Nous montrons ici comment la construction d'un système sécurisé peut être obtenue à partir de la notion de politique de sécurité. Afin de permettre une plus grande modularité, la notion de politique que nous introduisons ici est "indépendante" des systèmes sur lesquels elle est susceptible de s'appliquer. L'interface entre systèmes et politiques sera obtenue *via* une fonction d'interprétation.

[1] Une autre solution aurait pu consister à introduire un état "puits" $\sigma_\perp$, considéré comme sûr, et à remplacer toute transition non sûre à partir de σ par une transition de σ vers $\sigma_\perp$.

3.2.1. Politiques de sécurité

Une politique de sécurité permet de spécifier les propriétés de sécurité d'un ensemble d'entités, appelées les cibles de sécurité (les "choses contrôlées"), selon certaines informations de sécurité, appelées configurations de sécurité (les "choses contrôlantes"). Formellement, on représente une politique de sécurité $\mathbb{P}$ par le triplet $\mathbb{P} = (\mathbb{A}, \mathcal{C}, \Vdash)$ où $\mathbb{A}$ est l'ensemble des cibles de sécurité, $\mathcal{C}$ est l'ensemble des configurations de sécurité et $\Vdash \subseteq \mathcal{C} \times \mathbb{A}$ est la relation spécifiant si une cible est sécurisée dans une certaine configuration. L'implantation de cette notion de politique avec FoCaLiZe s'obtient en introduisant l'espèce `P_policy` qui est paramétrée par deux collections qui implantent les espèces `Target` et `Configuration` correspondant respectivement aux cibles et aux configurations et héritant de l'espèce des sétoïdes. L'espèce `P_policy` hérite de l'espèce des relations binaires (spécifiées *via* leur fonction caractéristique). Un individu d'une collection implantant l'espèce `P_policy` représente donc une politique de sécurité $\mathbb{P}$. Cette espèce déclare une méthode `secure` correspondant à la relation $\Vdash$ de $\mathbb{P}$ à partir de laquelle la méthode `relation` de l'espèce `Relation` peut être définie.

```
species Relation (A is Setoid, B is Setoid) =
 inherit Basic_object;
 signature relation : A -> B -> bool;
end;;
species Target = inherit Setoid; end;;
species Configuration = inherit Setoid; end;;
species P_policy (A is Target, C is Configuration) =
 inherit Relation (A, C);
 signature secure : A -> C -> bool ;
 let relation(a, c) = secure(a, c);
end;;
```

Application : Politiques de contrôle d'accès La notion de politique de sécurité que nous avons introduite est très générale et permet de considérer des politiques de sécurité dans de nombreux domaines : par exemple, pour spécifier des politiques de contrôle d'accès, il suffit d'instancier la notion de cible de sécurité par des ensembles finis d'accès. La notion de cible permet donc de spécifier le "domaine" sur lequel s'effectue le contrôle (ce domaine peut être instancié par des accès, des flots d'information, etc). Ainsi, pour définir l'espèce des politiques de contrôle d'accès, on écrira :

```
species Target_ac (S is Subject, O is Object, A is Access_mode, SOA is Abstract_triple(S, O, A)) =
 inherit Target, Accesses (S, O, A, SOA);
end;
species P_policy_ac (S is Subject, O is Object, A is Access_mode, SOA is Abstract_triple(S, O, A),
 T is Target_ac (S, O, A, SOA), C is Configuration) =
 inherit P_policy (T, C);
end;;
```

A partir de cette espèce, il est possible d'introduire (par héritage) plusieurs espèces différentes de politiques de contrôle d'accès, selon les configurations utilisées. Par exemple, les politiques discrétionnaires dont les configurations décrivent explicitement un ensemble d'accès autorisés peuvent être spécifiées par l'espèce suivante :

```
species Configuration_ac (S is Subject, O is Object, A is Access_mode, SOA is Abstract_triple(S, O, A))=
 inherit Configuration, Accesses (S, O, A, SOA);
end;
species P_policy_hru (S is Subject, O is Object, A is Access_mode, SOA is Abstract_triple(S, O, A),
 T is Target_ac(S, O, A, SOA), C is Configuration_ac(S, O, A, SOA)) =
 inherit P_policy_ac (S, O, A, SOA, T, C);
 let secure (t, c) = included(T!to_list(t), C!to_list(c));
end;
```

Une cible de cette politique correspond donc à un ensemble d'accès A qui, étant donnée une configuration c représentant un ensemble d'accès autorisés, est sécurisée ssi $A \subseteq c$ (la méthode `secure` est définie en utilisant une fonction prédéfinie `included` sur les listes).

Il est également possible de raffiner l'espèce `P_policy_ac` en considérant par exemple un ensemble de modes d'accès contenant les deux modes de lecture et d'écriture.

```
species Access_mode_rw =
  inherit Access_mode;
  representation = | Read | Write;
  let read = Read;
  let write = Write;
end;;
species P_policy_ac_rw (S is Subject, O is Object, A is Access_mode_rw, SOA is Abstract_triple(S, O, A),
 T is Target_ac(S, O, A, SOA), C is Configuration) =
 inherit P_policy_ac (S, O, A, SOA, T, C);
end;
```

Les deux méthodes `read` et `write` de l'espèce `Access_mode_rw` permettent de désigner les deux modes d'accès "en dehors" de la collection, puisque lors du mécanisme d'encapsulation de la collection, la représentation est masquée. La notion d'héritage multiple de FoCaLiZe permet alors de raffiner l'espèce `P_policy_hru` comme suit :

```
species P_policy_hru_rw (S is Subject, O is Object, A is Access_mode_rw, SOA is Abstract_triple(S, O, A),
 T is Target_ac(S, O, A, SOA), C is Configuration_ac(S, O, A, SOA)) =
 inherit P_policy_hru (S, O, A, SOA, T, C), P_policy_ac_rw (S, O, A, SOA, T, C);
end;
```

Lorsque l'ensemble des modes d'accès contient les deux modes "read" et "write", on peut également introduire la politique de Bell & LaPadula, dont les configurations décrivent un ensemble fini partiellement ordonné $(\mathcal{L}, \preceq)$ de niveaux de sécurité associés aux sujets et aux objets (*via* deux fonctions $f_s : \mathcal{S} \to \mathcal{L}$ et $f_o : \mathcal{O} \to \mathcal{L}$). Pour cela, on définit une espèce `Security_level` correspondant à $(\mathcal{L}, \preceq)$ qui hérite à la fois de l'espèce spécifiant les ensembles partiellement ordonnés et de l'espèce spécifiant les ensembles finis. L'espèce `Configuration_blp`, qui hérite de l'espèce `Configuration`, introduit alors les deux méthodes correspondant aux fonctions f_s et f_o.

```
species Security_level = inherit Partial_order , Finite_set; end;;
species Configuration_blp (S is Subject, O is Object, L is Security_level) =
  inherit Configuration;
  signature fs : Self -> S -> L;
  signature fo : Self -> O -> L;
end;;
species P_policy_blp (S is Subject, O is Object, A is Access_mode_rw, SOA is Abstract_triple(S, O, A),
 L is Security_level, T is Target_ac(S, O, A, SOA), C is Configuration_blp(S, O, L)) =
 inherit P_policy_ac_rw (S, O, A, SOA, T, C);
 let mac(c, soa) = L!geq(C!fs(c, SOA!first(soa)), C!fo(c, SOA!second(soa)));
 let macstar(c, soa1, soa2) =
     ~~ ( A!equal(SOA!(third(soa1)), A!read) && A!equal(SOA!(third(soa2)), A!write))
     || L!leq(C!fo(c, SOA!second(soa1)), C!fo(c,SOA!second(soa2)));
 let secure(t, c) = (list_forall(fun soa -> mac(c, soa), T!to_list(t))) &&
                    (list_forall2(fun soa1 -> fun soa2 -> mac_star(c,soa1,soa2), T!to_list(t)));
end;
```

Une cible de cette politique correspond donc à un ensemble d'accès A qui, étant donnée une configuration c spécifiant les niveaux de sécurité associés aux sujets ($f_s : \mathcal{S} \to \mathcal{L}$) et aux objets ($f_o : \mathcal{O} \to \mathcal{L}$), est sécurisée ssi les deux propriétés suivantes sont satisfaites.

- Les sujets n'accèdent qu'à des objets de niveaux de sécurité inférieurs à leur propre niveau de sécurité : pour tout accès $(s, o, a) \in A$, $f_o(o) \preceq f_s(s)$ (cette propriété est définie dans la méthode `secure` qui utilise une fonction prédéfinie `list_forall` sur les listes qui vérifie que tout élément d'une liste est associé à `true` par la méthode `mac`).

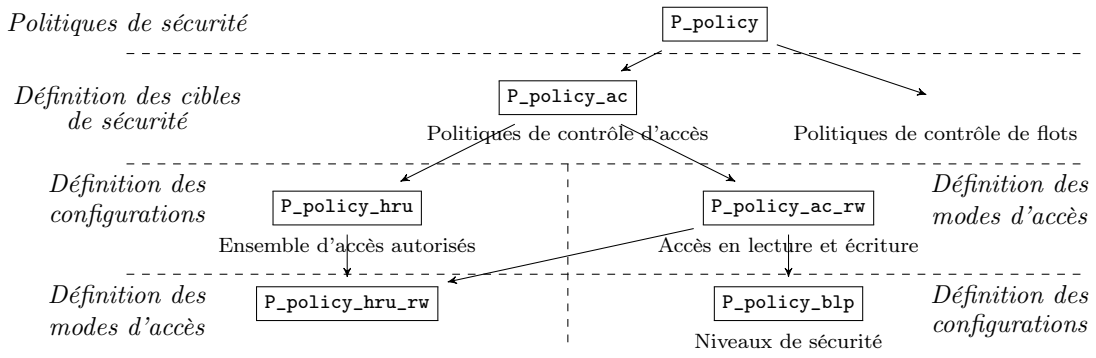

Figure 2: Hiérarchie de différentes politiques de sécurité

- Une information contenue dans un objet de niveau ℓ ne peut être "recopiée" que dans un objet dont le niveau est supérieur à ℓ (il n'y a pas de flots d'information descendants) : si $(s, o_1, read) \in A$ et $(s, o_2, write) \in A$ alors $f_o(o_1) \preceq f_o(o_2)$ (cette propriété est définie dans la méthode `secure` qui utilise une fonction prédéfinie `list_forall2` sur les listes qui vérifie que toute paire d'éléments d'une liste est associée à `true` par la méthode `macstar`).

Les exemples de politiques de contrôle d'accès présentés ici montrent l'aspect modulaire de l'architecture proposée (figure 2) qui permet de guider l'utilisateur dans son développement.

3.2.2. Etats sûrs obtenus à partir d'une politique de sécurité

La donnée d'un ensemble Σ d'états et d'une politique de sécurité $\mathbb{P} = (\mathbb{A}, \mathcal{C}, \Vdash)$ permet de définir un ensemble d'états $\Sigma_{\mathbb{P}} = \Sigma \times \mathcal{C}$ (on enrichit donc la notion d'état avec les informations décrites par les configurations de la politique) sur lequel un prédicat caractérisant les états sûrs peut être défini. L'ensemble $\Sigma_{\mathbb{P}}$ correspond donc à une collection implantant l'espèce `Secure_state` où :

- la fonction de projection $I_S : \Sigma_{\mathbb{P}} \to \Sigma$ est définie par $I_S((\sigma, c)) = \sigma$ pour toute paire $(\sigma, c) \in \Sigma_{\mathbb{P}}$,
- la fonction `make` est telle que `make`$((\sigma_1, c), \sigma_2) = (\sigma_2, c)$ pour toute paire (σ_1, c) et tout état σ_2.

Le prédicat Ω caractérisant les états sûrs de $\Sigma_{\mathbb{P}}$ peut alors être défini à partir d'une fonction d'interprétation $I : \Sigma \to \mathbb{A}$ permettant d'interpréter les états de Σ par des cibles de $\mathbb{P}$: un état $(\sigma, c) \in \Sigma_{\mathbb{P}}$ est sûr, ssi la configuration c "autorise" la cible associée à σ, c'est-à-dire ssi $c \Vdash I(\sigma)$ (i.e. $\Omega((\sigma, c)) \Leftrightarrow c \Vdash I(\sigma)$). Pour implanter cette construction avec FoCaLiZe, il suffit donc d'introduire une espèce `Secure_state_pol` qui hérite à la fois de l'espèce `Secure_state` et de l'espèce spécifiant le produit cartésien de deux setoïdes, dans laquelle la méthode `state` (I_S) est définie par un opérateur de projection sur les couples, qui déclare une méthode `interpretation` (I) permettant d'associer une cible de sécurité à tout état à partir duquel est construit un individu, et qui définit la méthode `secure` ($\Vdash$).

```
species Secure_state_pol (A is Target, C is Configuration, P is P_policy (A, C), S is State) =
  inherit Product_set (S, C), Secure_state (S);
  let state = first;
  let configuration = second;
  let make (ss, st) = pair(st, configuration);
  proof of make_spec = by definition of make;
  signature interpretation : S -> A;
  let is_secure (x : Self) = P!secure (interpretation (state (x)), configuration (x));
end;;
```

Par exemple, quand l'ensemble des états et des cibles sont des ensembles finis d'accès, on peut raffiner cette spécification en définissant la méthode `interpretation` comme suit :

```
species Secure_state_pol_ac(S is Subject, O is Object, A is Access_mode, SOA is Abstract_triple(S, O, A),
 St is State_ac(S, O, A, SOA), T is Target_ac(S, O, A, SOA), C is Configuration,
 P is P_policy_ac(S, O, A, SOA, T, C))=
 inherit Secure_state_pol(T,C,P,St) ;
 let interpretation(x) = x ;
end;;
```

3.2.3. Sécurisation d'un système à partir d'une politique de sécurité

Finalement, lorsque la notion d'état sûr est obtenue à partir d'une politique, la définition d'un système sécurisé s'obtient directement par héritage comme suit :

```
species Secure_transition_system_pol(A is Target, C is Configuration, P is P_policy(A, C),
  St is State, L is Label, SLS is Abstract_triple(St, L, St), Sys is Transition_system(St, L, SLS),
  Ss is Secure_state_pol(A, C, P, St), SsLS is Abstract_triple(Ss, L, SsLS))=
  inherit Secure_transition_system (St, L, SLS, Sys, Ss, SsLS);
end;;
```

Le mécanisme de sécurisation d'un système obtenu par cette construction permet donc, à partir d'une politique de sécurité $\mathbb{P}$, d'un système $\mathbb{S} = (\Sigma, \Sigma^0, L, \delta)$ et d'une fonction d'interprétation $I : \Sigma \to \mathbb{A}$, d'obtenir le système $\mathbb{S}_{\mathbb{P}} = (\Sigma_{\mathbb{P}}, \Sigma^0_{\mathbb{P}}, L, \delta_{\mathbb{P}})$ où $\Sigma_{\mathbb{P}} = \Sigma \times \mathcal{C}$ et :

$$\Sigma^0_{\mathbb{P}} = \{(\sigma, c) \mid \sigma \in \Sigma^0 \wedge c \Vdash I(\sigma)\} \quad \delta_{\mathbb{P}} = \left\{(\sigma_1, c) \xrightarrow{l}_{\delta_{\mathbb{P}}} (\sigma_2, c) \mid \sigma_1 \xrightarrow{l}_{\delta} \sigma_2 \wedge c \Vdash I(\sigma_1) \Rightarrow c \Vdash I(\sigma_2)\right\}$$

Par construction, tous les états accessibles de ce système satisfont la politique de sécurité. De même, l'obtention d'un système "opérationnel" sécurisé s'obtient directement par héritage comme suit :

```
species Secure_operational_transition_system_pol(A is Target, C is Configuration, P is P_policy(A, C),
 St is State, L is Label, SLS is Abstract_triple(St, L, St),
 Sys is Operational_transition_system (St, L, SLS),
 Ss is Secure_state_pol(A, C, P, St), SsLS is Abstract_triple(Ss, L, Ss)) =
  inherit Secure_transition_system_pol(A, C, P, St, L, SLS, Sys, Ss, SsLS),
          Secure_operational_transition_system (St, L, SLS, Sys, Ss, SsLS);
end;;
```

Ici, la relation $\delta_{\mathbb{P}}$ obtenue est définie par :

$$\begin{aligned}\delta_{\mathbb{P}} = \quad & \left\{(\sigma_1, c) \xrightarrow{l}_{\delta_{\mathbb{P}}} (\sigma_2, c) \mid \sigma_1 \xrightarrow{l}_{\delta} \sigma_2 \wedge c \Vdash I(\sigma_1) \Rightarrow c \Vdash I(\sigma_2)\right\} \\ \cup \quad & \left\{(\sigma_1, c) \xrightarrow{l}_{\delta_{\mathbb{P}}} (\sigma_1, c) \mid \sigma_1 \xrightarrow{l}_{\delta} \sigma_2 \wedge c \Vdash I(\sigma_1) \wedge \neg c \Vdash I(\sigma_2)\right\}\end{aligned}$$

Application Ces mécanismes de sécurisation peuvent être utilisés pour sécuriser un système d'accès avec une politique de contrôle d'accès. Par exemple, pour sécuriser un système "opérationnel" d'accès avec la politique de Bell & LaPadula, il suffira d'écrire :

```
species Secure_system_BLP(S is Subject, O is Object, A is Access_mode_rw, SOA is Abstract_triple(S, O, A),
 St is State_ac(S, O, A, SOA), L is Label_ac(S, O, A), SLS is Abstract_triple(St, L, St),
 T is Target_ac(S, O, A, SOA), Ll is Security_level, C is Configuration_blp(S, O, Ll),
 P is P_policy_blp(S, O, A, SOA, Ll, T, C),
 Sys is Operational_transition_system(St, L, SLS),
 Ss is Secure_state_pol_ac (S, O, A, SOA, St, T, C, P), SsLS is Abstract_triple(Ss, L, Ss)) =
 inherit Secure_transition_system_pol(T, C, P, St, L, SLS, Sys, Ss, SsLS),
         Secure_operational_transition_system_pol (T, C, P, St, L, SLS, Sys, Ss, SsLS);
end;;
```

4. Conclusion

Dans cet article, nous avons présenté une architecture de développement de systèmes sécurisés avec FoCaLiZe. L'expressivité des traits objets de FoCaLiZe permet d'obtenir un code à la fois simple, structuré en composants modulaires et donc facilement réutilisable dans divers contextes. Le cadre proposé permet, d'une part, de développer un système de transition permettant d'assurer les fonctionnalités désirées (par exemple un système d'accès), et, d'autre part, de définir une notion de sécurité sur les états de ce système. Un mécanisme opérationnel est alors introduit afin de restreindre les exécutions du système afin que seuls les états sûrs soient accessibles. Nous avons également présenté une notion de politique de sécurité permettant de définir la notion d'états sûrs d'un système. Par exemple, pour un système d'accès, nous avons vu comment définir des politiques de contrôle d'accès. Cette étude de cas illustre bien à quel point les concepts abstraits définis s'adaptent bien à des instanciations concrètes : plusieurs variantes de politiques de contrôle d'accès ont pu être facilement définies et l'instanciation des concepts correspond souvent à une simple composition des concepts existants.

Le développement suit pas à pas la construction "théorique" en utilisant les principaux traits de FoCaLiZe (héritage et paramétrisation). Ceci nous permet d'avoir deux types de raffinements : l'héritage avec implantation de méthodes déclarées (ou redéfinition de méthodes déjà définies) et l'instanciation de paramètre qui est plus classique en programmation certifiée. Il n'est pas toujours très simple de modéliser de cette manière puisque pour utiliser une méthode déclarée dans une espèce on peut soit hériter soit passer une collection en paramètre. Cependant, nous pensons que ce développement, utilisant à la fois l'héritage et la paramétrisation, est plus facile à aborder qu'un développement basé sur la seule abstraction de paramètres. En effet là où l'abstraction conduit naturellement à utiliser de l'ordre supérieur en paramétrant par des fonctions, nous spécifions ces fonctions dans une espèce, ce qui nous permet également de capturer ses propriétés dans la même espèce. Par exemple les fonctions d'interprétation ou les prédicats décrits de manière informelle sont implantés par des méthodes attachées aux valeurs FoCaLize d'une collection implantant une espèce.

Quoi qu'il en soit, l'approche formelle établie "sur le papier" pour décrire les politiques de sécurité et les systèmes sécurisés a pu être complètement implantée avec FoCaLiZe sans aucune modification de la construction théorique. Ce développement constitue par ailleurs un exemple assez complet d'utilisation des traits de FoCaLiZe. Ces traits permettent différents raffinements entre spécifications et implantations et conduisent donc à une architecture facilement réutilisable. Par exemple, la notion d'états sûrs sur laquelle repose la construction de systèmes sécurisés est indépendante de la notion de politique de sécurité et il est donc tout à fait envisageable d'utiliser une autre notion de politique pour spécifier cette notion d'états sûrs. Mais bien d'autres choix restent aussi possibles en utilisant ce développement (choix des cibles, choix des configurations, etc) et la possibilité d'appliquer une politique sur un système *via* une fonction d'interprétation des états du système par des cibles, permet l'application de plusieurs politiques différentes sur un même système, fournissant ainsi un mécanisme permettant une forme de composition de politiques sur un système. Enfin, ce cadre permet de faire le lien entre les différentes variantes des formalismes que nous avons proposés par le passé pour spécifier et comparer des politiques de sécurité.

Remerciements. Nous remercions Gabriel Szekely pour sa participation au développement du code FoCaLiZe lors de son stage de Master. Nous remercions aussi Thérèse Hardin, à l'origine avec Renaud Rioboo de la création de FoCaLiZe, pour les nombreuses discussions que nous avons eues avec elle sur ce travail. Nous remercions également les relecteurs anonymes pour leurs nombreuses remarques permettant d'améliorer cet article.

Bibliographie

[BCJK11] T. Bourdier, H. Cirstea, M. Jaume, and H. Kirchner. Formal specification and validation of security policies. In *Foundations & Practice of Security, FPS*, volume 6888 of *LNCS*. Springer. To appear, 2011.

[BDD07] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th Int. Conf., LPAR*, volume 4790 of *LNCS*, pages 151–165. Springer, 2007.

[BM07] J. Blond and C. Morisset. Un moniteur de référence sûr d'une base de données. *Technique et Science Informatiques*, 26(9):1091–1110, 2007.

[Coq10] Coq. *The Coq Proof Assistant, Tutorial and reference manual*. INRIA – LIP – LRI – LIX – PPS, 2010. Distribution available at: `http://coq.inria.fr/`.

[DÉDG06] D. Delahaye, J.F. Étienne, and V. Donzeau-Gouge. Certifying airport security regulations using the Focal environment. In *FM 2006: 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 48–63. Springer, 2006.

[Foc10] Focalize. *Focalize, Tutorial and reference manual*. LIP6 – INRIA – CEDRIC, 2010. Distribution available at: `http://focalize.inria.fr`.

[Jau10] M. Jaume. Security rules versus security properties. In *Information Systems Security - 6th Int. Conf., ICISS*, volume 6503 of *LNCS*, pages 231–245. Springer, 2010.

[JM07] M. Jaume and C. Morisset. Contrôler le contrôle d'accès. In *AFADL, Approches Formelles dans l'Assistance au Développement de Logiciels*, 2007.

[JM08] M. Jaume and C. Morisset. Un cadre sémantique pour le contrôle d'accès. *Technique et Science Informatiques*, 27(8):951–976, 2008.

[JTM10] M. Jaume, V. Viet Triem Tong, and L. Mé. Contrôle d'accès versus contrôle de flots. In *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL*, 2010.

[JTM11] M. Jaume, V. Viet Triem Tong, and L. Mé. Flow based interpretation of access control: Detection of illegal information flows. In *Information Systems Security - 7th International Conference, ICISS*, volume 7093 of *LNCS*, pages 72–86. Springer. To appear, 2011.

[PJ03] V. Prevosto and M. Jaume. Making proofs in a hierarchy of mathematical structures. In *11th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus 2003*, pages 89–100. Aracne, 2003.

[Rio09] R. Rioboo. Invariants for the focal language. *Ann. Math. Artif. Intell.*, 56(3-4):273–296, 2009.

[SP07] N. Stouls and M.L. Potet. Security policy enforcement through refinement process. In *B 2007: Formal Specification and Development in B, 7th Int. Conf. of B Users*, volume 4355 of *LNCS*, pages 216–231. Springer, 2007.