A formal approach to implement access control models

Mathieu Jaume[1], Charles Morisset[1]

[1]SPI LIP6 University Paris 6, 8 rue du Capitaine Scott
75015 Paris, France
{ Mathieu.Jaume , Charles.Morisset }@lip6.fr

Abstract: Access control software must be based on a security policy model. Flaws in them may come from a lack of precision or some incoherences in the policy model or from inconsistencies between the model and the code. In this paper, we present a formalisation of access control models, based on the algebra of security models introduced by J.McLean [10], together with a description of its implementation in an environment, namely Focal [17], which provides a language with object-oriented features that allows to write formal specifications, proofs and programs at the same level. Then, we show how such formal development can be used to obtain a particular security model: the Bell and LaPadula security model. Last, as an example, we show how such a program can be integrated for secure databases.

Keywords: access control policies, formal methods, Focal

1 Introduction – Motivations

Information systems'security becomes a major problem for society and a well-established field of computer science. Research themes involve access control mechanisms, modeling of information flow and its applications to confidentiality policies [11, 12], mobile code security, cryptographic protocols, etc. The methods used to consider these questions are evolving, as are the ones used in safety areas, where ad-hoc and empirical approaches were progressively replaced by more formal methods. High levels of safety require that the requirement/specification phase is done using mathematical models, allowing mechanized proofs of the required properties. In the same way, assurance in system security asks for the use of true formal methods along the process of software development, starting at the specification level.

Computer information security is usually seen as a combination of three classes of properties: confidentiality (denying unauthorised accesses), integrity (denying unauthorised modifications of information) and availability. Security evaluation criterias have been elaborated by government agencies to promote the conception of trusted systems, for example the Trusted Computer Security Evaluation Criterias (1983) (TCSEC), the Information Technology Security Evaluation Criterias (1991) (ITSEC [5]) and the Common Criterias (1999), which are a collection of normative documents. These criterias provide both a framework for the software industry to ensure that software has been carefully designed and a referentie for its customers. A product evaluation and certification against the common criteria's framework is built according to two hypothesis. The first one is the "protection profile", that is, the conditions under which the evaluated product is supposed to be used. The second one is its level of assurance, which is simply the level of trust the system can be granted, according to the development process. A good security level can be reached if the product is evaluated at an assurance level greater than EAL-5 against a convincing protection profile such as the one given in [14]. Such a profile requires the use of a mandatory formal policy model in order to achieve an acceptable level of confidentiality. Mandatory access control (MAC), contrary to discretionary access control (DAC) such as access control list (ACL), is managed and enforced by the underlying system rather than by an authorized user.

In this paper, we focus on access control. Access control is any mechanism by which a system can grant or revoke the right for active entities (the subjects) to access some passive entities (the objects) or perform some actions. Our long-term aim is to develop a formal library of access control policies together with their implementations within a programming environment, namely Focal [17], in which specifications, proofs and programs are written at the same level. To achieve this goal, it is first necessary to formalise in a precise and unambiguous way the access control policies we want to implement and to prove. Then, since many access control policies share some common definitions, properties and proofs, as it is widely recognised, it seems desirable to deal with an abstract generic framework in order to ease and speed implementations by reusing. Indeed, having an abstract formalism would allow to obtain an implementation which is well-suited for a given context just by instantiating parameters.

Hence, in this paper, we address the three following problems:

- the evaluation of security: reaching high levels of safety by using the Focal programming environment,
- the full formalisation of access control policies,
- the definition of an abstract framework in which many access control policies can be described.

This paper extends [7] and is organized as follows. The end of this introduction gives some supplementary motivations about the formalisation of access control policies. Then section 2 briefly presents the Focal programming environment used to implement access control policies. Section 3 describes in detail the implementation of Mc Lean's algebra of security within Focal, thus showing how features of Focal can be used. Last, before the conclusion, section 4 illustrates the use of such an implementation to obtain the classical Bell and LaPadula model and its application to manage access in a relational database.

On formalising access control policies An extensive literature on access control exists now and one can wonder about the usefulness (or the difficulties) of formalising access control policies. However, many papers describing such policies are rather informal and/or present a particular access control mechanism through examples without any formalisation (or generalisation) of the concepts involved in the policy. Of course, such papers are very useful to understand how a particular access control works but they provide few help to implement it in another context. Furthermore, even for papers containing specifications, definitions and properties expressed in a mathematical way, such formalisations are not always done at the level of detail required to obtain a "computer-assisted formalisation". Indeed, even if having a mathematical model drawn by hand is a very serious way to increase confidence, this is not enough. First, nothing is said about how these abstract notions can be implemented: in this paper we

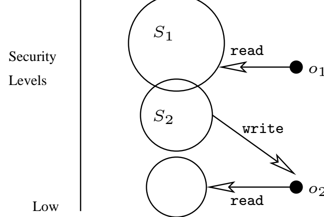

Figure 1: Violation of the ⋆-security property

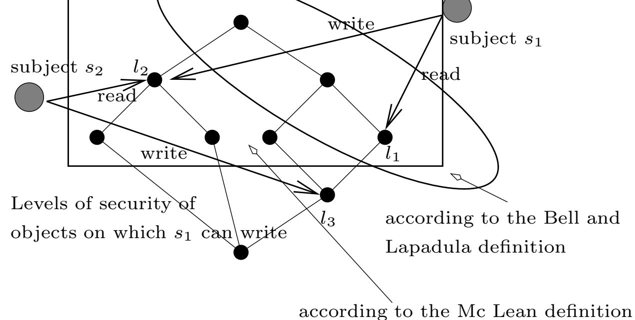

Figure 2: Definitions of the ⋆-security property

present a way of formalising and implementing such notions. Furthermore, when attempting to check proofs done by hand with a proof assistant, many of them have been discarded. Often, the errors are introduced by points considered as evident details or by forgotten cases. Last, even if proofs done by hand are correct, nothing formally ensures that the implementation meets the "formal" specification done "on the paper".

As a typical example of details that can be crucial, let us consider the classical ⋆-security property introduced to prevent the copy of an object to a lower security level by a malicious subject. In [1], Bell and LaPadula define this property as follows:

$$\forall s \in \mathcal{S}\ \forall o_1, o_2 \in \mathcal{O}$$
$$((s, o_1, \mathsf{read}) \in m \wedge (s, o_2, \mathsf{write}\ ) \in m) \Rightarrow f_o(o_1) \preceq f_o(o_2)$$

where $\mathcal{S}$ (resp. $\mathcal{O}$) is a set of subjects (resp. objects), m is a set of access containing elements of the form (s, o, a) expressing that a subject s has an access to an object o according to the access mode a, and where f_o is a mapping that gives the security level of objects. If we deal with joint access of groups of subjects over objects, we can "generalize" this property as follows:

$$\begin{array}{l}\forall S_1, S_2\ \forall o_1, o_2 \in \mathcal{O}\\ \quad ((S_1, o_1, \mathsf{read}) \in m \wedge (S_2, o_2, \mathsf{write}\ ) \in m \wedge S_1 \cap S_2 \neq \emptyset)\\ \Rightarrow \quad f_o(o_1) \preceq f_o(o_2)\end{array} \tag{1}$$

where S_1 and S_2 are sets of subjects (now, we write (S, o, a) to express that subjects in the set S have a joint access to o according to a). Figure (1) illustrates the situation we want to avoid. In [10], this property is stated as follows:

> "*a state is ⋆-secure if for any subjects* S_1, S_2 *and objects* o_1, o_2*, if* $(S_1, o_1, \mathsf{read}) \in m$ *and* $(S_2, o_2, \mathsf{write}\ ) \in m$ *and the classification of* o_1 *dominates that of* o_2*, then* $S_1 \cap S_2 = \emptyset$ "

A direct translation of this sentence leads to the following logical formula:

$$\begin{array}{l}\forall S_1, S_2\ \forall o_1, o_2 \in \mathcal{O}\\ \quad ((S_1, o_1, \mathsf{read}) \in m \wedge (S_2, o_2, \mathsf{write}\ ) \in m \wedge f_o(o_2) \prec f_o(o_1))\\ \Rightarrow \quad S_1 \cap S_2 = \emptyset\end{array}$$

which is logically equivalent (by contraposition) to:

$$\begin{array}{l}\forall S_1, S_2\ \forall o_1, o_2 \in \mathcal{O}\\ \quad ((S_1, o_1, \mathsf{read}) \in m \wedge (S_2, o_2, \mathsf{write}\ ) \in m \wedge S_1 \cap S_2 \neq \emptyset)\\ \Rightarrow \quad \neg(f_o(o_2) \prec f_o(o_1))\end{array} \tag{2}$$

Clearly, since $\preceq$ is only a partial order defining the lattice of security levels, (1) and (2) are not equivalent (they become equivalent when $\preceq$ is a total order, most of times this is not the case). Thus, confusions may arise because these properties in fact define different policies. This example points out that different authors give subtley different definitions of the ⋆-security property, and shows that even if these two definitions seems to be similar, by expressing them in a formal way, we obtain two different security policies as illustrated in figure (2). This figure clearly shows that if a subject s_1 has a read access on an object whose security level is l_1, then the set of objects on which s_1 can write according to (1) (i.e. objects whose security levels are greater or equal than l_1) is strictly included in the set of objects on which s_1 can write according to (2) (i.e. objects whose security levels are not lower than l_1). Similarly, if we adopt the property (2), a subject s_2 having a read access on an object whose security level is l_2, such that l_1 and l_2 are not comparable with $\preceq$, can have a write access on a object whose security level is l_3 with $l_3 \prec l_1$. Hence, as we can see on this example, (2) does not prevent the copy of an object to a lower security level. In fact, it is now well-known that formalising specifications, definitions and proofs brings us at a level of detail that is often left to the reader, and by taking into account these details, often considered as minor in informal presentations, definitions and proofs are sometimes getting a little more complicated.

2 The Focal programming environment

The Focal project[15, 17][1] attempts to build a new programming environment providing a language based on firm theoretical results, with clear semantics and which permits an efficient implementation – via translation to OCaml [9]. It has functional and object-oriented features and provides means for the programmers to write formal proofs of their code, and to have them verified by a proof checker (Zenon and Coq [16]). The Focal language also provides features allowing to write programs, specifications and proofs implementation in an incremental way. Indeed, inheritance and refinement mechanisms are powerful features which are particularly well-suited to develop a library for secure applications since they allow to make several refinements of a specification until providing an executable code. Thus, Focal avoids separately treating the formal modeling work and the development.

In Focal, species are the nodes of the hierarchy of structures that makes up the library. A species can be seen as a set of methods, which are identified by their names. Each method can be either declared (primitive constants, operations and properties) or defined (implementation of operations and proofs of theorem). Moreover, we can distinguish three kinds of methods: the carrier type, the programming methods and the logical methods.

Carrier type The carrier, or representation type (`rep` for short), is the type of the representation of the underlying set of the structure defined by the species. The carrier is represented by the keyword `self`, so that we identify the set with the structure, as it is usual. Each species must have one unique carrier, but as all the other methods, it can be either declared or defined. A declared carrier is simply an abstract data type, while a defined one is bound to a concrete type.

Programming methods These methods represent the constants and the operators of the structure. Declared methods are called signatures and are introduced through the keyword `sig`, defined methods are introduced through `let` and recursive definitions must be explicitely flagged with the keyword `rec`. The language used for the definitions is similar to the functional core of OCaml [9], with the addition of a construction to call a method from a given structure. More precisely,

[1] `http://focal.inria.fr`

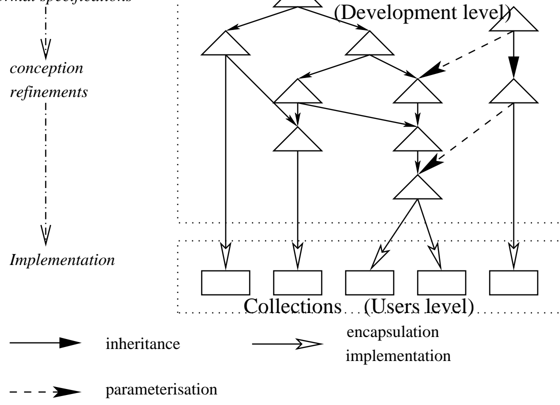

Figure 3: Focal's features

the main syntactic constructions of the language are the following[2]:

- abstraction with respect to a variable `fun x -> ...`
- application of a function `f(x,y)`
- call of a method `m` from a structure `c`: `c!m`
- call of a method `m` of the structure we are currently building `self!m`

Logical methods Last, we can find in a species methods which represent the properties of programming methods. In this context, the declaration of such a method is simply the statement of the property, while the definition is a proof of this statement. In the first case, we speak of properties (that are still to be proved later in the development), while in the second case we speak of theorems. The language used for the statements is composed of the basic logical connectors `and`, `or`, `->`, `<->`, `not`, and universal `all` and existential `ex` quantification over a Focal type.

Currently, a proof in Focal consists in either a Coq script interpreted in the context of the species where it is done[3] or in a more or less detailed proof expressed with Cutter, a declarative language on which the theorem prover Zenon is based[4].

The main (object-oriented) features of Focal are illustrated on figure (3). Using frameworks involving inheritance requires to perform analysis in order to check coherence properties (inheritance lookup, resolution of multiple-inheritance conflicts, dependency analysis, type-checking ...). In Focal, classical object-oriented features have been restricted in order avoid unsound constructions, such as open recursion, which can lead to inconsistencies when used carelessly.

A species can inherit the declarations and definitions of one or several already defined species and is free to define or even redefine an inherited method (but must not change its type). Besides inheritance, the other important feature of Focal resides in the possibility to parameterize a species by another one.

A collection is built upon a completely defined species. This means that every method must be defined. In other words, in a collection, every operation has an implementation, and every theorem is formally proved. In addition, a collection is "frozen". Namely, it cannot be used as a parent of a species in the inheritance graph.

[2] See [17] for a complete description of the syntax

[3] Some syntactic constructions have been added to tell the Focal compiler what are the dependencies of such a proof, so that the compiler can set up the appropriate environment when translating a Focal program into Coq.

[4] Zenon allows to easily obtain formal proofs and translates them in order to be able to check them by Coq.

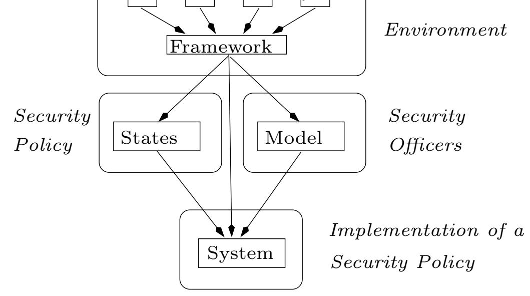

Figure 4: Mc Lean's Algebra of Security

Last, note that the computational part of Focal is validated by its computer algebra library, mostly developed by R. Rioboo [3], which implements mathematical structures up to multivariate polynomial rings and includes complex algorithms with performance comparable to the best computer algebra systems in existence. As we will see, such library is very useful when formalising the algebra of security models (we will use implementations of lattices and boolean algebras).

3 Implementation of the "algebra of security" models

The algebra of security models has been introduced by J.McLean in [10]. As illustrated in figure (4), it contains three levels of specifications: frameworks, models and systems.

3.1 Frameworks

A framework $\mathcal{F}$ is a quadruple $(\mathcal{S}, \mathcal{O}, \mathcal{L}^S, \mathcal{A})$ where $\mathcal{S}$ is a set of subjects (users, programs, ...), $\mathcal{O}$ is a set of objects (data, files, ...), $\mathcal{L}^S = (\mathcal{L}, \preceq, \curlyvee, \curlywedge)$ is a complete lattice of levels of security and $\mathcal{A}$ is a set of access modes, such as read, write, etc.

So we first introduce the species of subjects and objects :

```
species 𝔖 inherits setoid = end
species 𝔒 inherits setoid = end
```

The species `setoid` defines a non empty set with an equivalence relation over its elements.

While a subject is an element of the species $\mathfrak{S}$, every element of the species $\mathfrak{S}$ is not necessarily a subject of a framework.

As we will see, we will need to deal with finite sets of subjects and objects. Hence, we define the notion of sets of subjects and sets of objects with the following species :

```
species 𝔖⋆ (s is 𝔖)
inherits finite_powerset(s) = end
species 𝔒⋆ (o is 𝔒)
inherits finite_powerset(o) = end
```

An element of the species `finite_powerset(s)` is finite set and every element of this set is an element of the parameter `s` (which is a species).

Since levels of security form a lattice, we can define the species of levels of security by inheritance from the species `lattice` defined in the library of Focal. This species contains definitions and properties defining a lattice.

```
species ℒ^S inherits lattice = end
```

We also define the species of access modes :

```
species 𝒜 inherits setoid = end
```

Since we consider that a subject can access to an object in several access modes, we define the notion of set of access modes :

```
inherits finite_powerset (m) = end
```

Each framework introduces a set $\mathcal{S}^+$ containing every set of subjects that can potentially perform a joint access over an object. The notion of joint access is not always available in classical access control policy, but is very useful in some specific domains, such as the military system. For instance, in order to launch a missile, two different people have to push the button at the same time. However, as we will see, $\mathcal{S}^+$ is often reduced to the set of singleton sets of $\mathcal{S}$. Since $\mathcal{S}^+$ is a set of set of subjects, it is an element of the species defined as follows :

```
species 𝔖** (s is 𝔖, sp is 𝔖*(s))
inherits finite_powerset(sp) = end
```

Furthermore, as we will see when defining the notion of model, a framework involves a set $\mathcal{S}^{+\star}$ of sets of sets of subjects which is an element of the species :

```
species 𝔖*** (s is 𝔖, sp is 𝔖*(s),spe is 𝔖**(sp))
inherits finite_powerset(spe) = end
```

Now, we can first define the species $\mathbb{F}$ of frameworks containing the declaration of these sets.

```
species 𝔽
   (s is 𝔖, sp is 𝔖*(s), spe is 𝔖**(s, sp),
    spee is 𝔖***(s, sp, spe), o is 𝔒,
    so is 𝔒*, n_s is ℒ^𝒮, m is 𝒜,  m_a is 𝒜*(m))
inherits cartesian(sp, so) =
  let 𝒮 in self -> sp = fun x -> #first(x);
  let 𝒪 in self -> so = fun x -> #scnd(x);
  sig 𝒮+ in self -> spe;
  sig 𝒮+* in self -> spee;
end
```

The species `cartesian` defines the cartesian product of two setoids: an element of $\mathbb{F}$ is a pair of a set of subjects and a set of objects. For example, the method $\mathcal{S}$, defining the set of subjects, is obtained as the first element of the pair. Methods `first` and `scnd` are global methods (they are prefixed by `#`) and returns respectively the first and the second element of a pair.

At this level, $\mathcal{S}^+$ or $\mathcal{S}^{+\star}$ are not defined, but we have to specify that $\mathcal{S}^+$ and $\mathcal{S}^{+\star}$ are built upon elements of $\mathcal{S}$. So we add the two following properties in $\mathbb{F}$:

```
property build_𝒮+ : ∀ f ∈ self ∀ x ∈ 𝔖*,
   𝔖**!is_in(x, !𝒮+(f)) ⇒
   𝔖*!is_subset(x,!𝒮(f))

property build_𝒮+* : ∀ f ∈ self ∀ x ∈ 𝔖**,
   𝔖***!is_in(x, !𝒮+*(f)) ⇒
   𝔖**!is_subset(x,!𝒮+(f))
```

The methods `is_in` and `is_subset` are defined in the species `finite_powerset`: `is_in(x, S)` returns `true` if the element `s` belongs to the finite set `S`, `false` otherwise, and `is_subset`(S_1 , S_2) returns `true` if S_1 is a subset of S_2, `false` otherwise. The keyword `!` is a shortcut for `self!`, which means that we call the method of the current species.

When the species $\mathbb{F}$ is implemented by a collection, the developer has to prove these properties, so he has to define $\mathcal{S}^+$ and $\mathcal{S}^{+\star}$ in such a way that they respect them.

Then, we introduce a species $\mathbb{F}^g$ which inherits from $\mathbb{F}$ in which:

- $\mathcal{S}^+$ is specified as the finite powerset of $\mathcal{S}$ without the empty set (an access is initiated by at least one subject).
- $\mathcal{S}^{+\star}$ is specified as the finite powerset of $\mathcal{S}^+$.

This is done as follows :

```
property spec_𝒮+ : ∀ f ∈ self ∀ x ∈ 𝔖*,
   (¬ (𝔖*!equal(x, 𝔖*!vide))) ⇒
   𝔖*!is_subset(x,!𝒮(f)) ⇒
   𝔖**!is_in(x, !𝒮+(f));
```

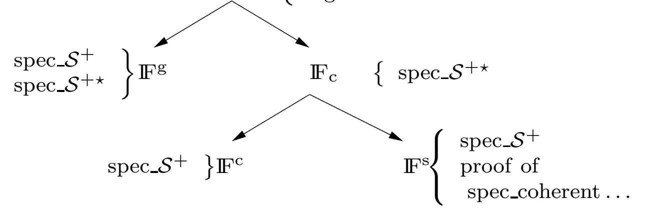

Figure 5: Hierarchy of frameworks

```
property spec_𝒮+* : ∀ f ∈ self ∀ x ∈ 𝔖**,
   𝔖**!is_subset(x,!𝒮+(f)) ⇒
   𝔖***!is_in(x, !𝒮+*(f));
```

As the properties `build_`$\mathcal{S}^+$ and `build_`$\mathcal{S}^{+\star}$, these two properties imply for the developer of the final collection to define $\mathcal{S}^+$ and $\mathcal{S}^{+\star}$ in a way that respects these conditions. This species represents the most general kind of framework, because no constraints are required over the sets $\mathcal{S}^+$ and $\mathcal{S}^{+\star}$.

We also define a species $\mathbb{F}_c$, for coherent frameworks, which inherits from $\mathbb{F}$ and specifies $\mathcal{S}^{+\star}$ as the set:

$$\{X \subseteq \mathcal{S}^+ \mid \forall y, z \in \mathcal{S}^+ \ (y \in X \wedge y \subseteq z) \Rightarrow z \in X\}$$

```
property spec_coherent_𝒮+* : ∀ f ∈ self ∀ x ∈ spe,
    𝔖***!is_in(x, !𝒮+*(f)) ⇔
   (𝔖**!is_subset(x, !𝒮+(f)) ∧
   (∀ y z ∈ sp, 𝔖**!is_in(z, !𝒮+(f)) ∧
    𝔖**!is_in(y, x) ∧ 𝔖*!is_subset(y, z)
    ⇒ 𝔖**!is_in(y, x)));
```

A coherent framework ensures that if a set of subjects is granted to change the security level of a subject or an object, then it can do it together other subjects. This will be explained later when defining models. Note that at this level, nothing is said about $\mathcal{S}^+$.

Last, we introduce the species $\mathbb{F}^c$ which inherits from $\mathbb{F}_c$ such that $\mathcal{S}^+$ is specified as the finite powerset of $\mathcal{S}$ without the empty set, as in $\mathbb{F}$, and the species $\mathbb{F}^s$ which also inherits from $\mathbb{F}^c$ and specifies $\mathcal{S}^+$ as the set of singleton set. By this constraint, $\mathbb{F}^s$ is clearly coherent (and then the property over $\mathcal{S}^{+\star}$ can be proved in this species) and is the most used in the literature (where no joint access are usually taken into account). The figure (5) describes the hierarchy of species of frameworks.

3.2 Models

Given a framework $\mathcal{F}$, a model is defined by two functions c_s and c_o, respectively from $\mathcal{S}$ and $\mathcal{O}$, to $\mathcal{S}^{+\star}$. These functions respectively associate to each subject and object every set of subjects allowed to change the security level of the subject or the object. As the range of these functions is $\mathcal{S}^{+\star}$, we see that by constraining $\mathcal{S}^{+\star}$ like we did for coherent frameworks, we can express that if a (set of) subject is granted to change the security level of an entity, then it can also do it together another subjects.

First, we define the species $\mathbb{M}_{\mathbb{F}}$ introducing c_s and c_o as follows :

```
species 𝕄𝔽
   (s is 𝔖, sp is 𝔖*(s), spe is 𝔖**(s, sp),
    spee is 𝔖***(s, sp, spe), o is 𝔒, eo is 𝔒*,
    n_s is ℒ^𝒮,m is 𝒜,  m_a is 𝒜*(m),
    f is 𝔽 (s , sp, spe, spee,o, n_s, m,  m_a),
    f1 in f)
inherits basic_object =
  rep = ((s -> spe) * (o -> spe));
  let mc_s in self -> s -> spe = fun m -> #first(m);
  let mc_o in self -> o -> spe = fun m -> #scnd(m);
```

As we can see, the carrier type of this species is a pair of functions. Here, the range of methods mc_s and mc_o is `spe` and not $\mathcal{S}^{+\star}$, as

not possible to use an entity as a type. So we have to define two invariants in order to express that the range of mc_s and mc_o is $\mathcal{S}^{+\star}$:

```
letprop inv_spec_mcs (x in self) = ∀ s1 ∈ s,
  𝔖⋆!is_in(s1,f!𝒮(f1)) ⇒
  𝔖⋆⋆⋆!is_in(!mcs(x, s1),f!𝒮+⋆(f1));

letprop inv_spec_mco (x in self) = ∀ o1 ∈ o,
  𝔒⋆!is_in(o1,f!𝒪(f1)) ⇒
  𝔖⋆⋆⋆!is_in(!mcs(x, o1),f!𝒮+⋆(f1));

letprop inv_spec_c (x in self) =
  !inv_spec_mcs(x) ∧ !inv_spec_mco(x);
```

We do not define these invariants with `property` since it is possible to create models that do not respect these properties, and so it would be impossible to prove them. The keyword `letprop` allows to define a property as a predicate. We will see later how to use such a predicate.

From two models, we can create another one by doing operations over sets returned by mc_s and mc_o.

Let $M_1 = (\mathcal{F}, c_s^1, c_o^1)$ and $M_2 = (\mathcal{F}, c_s^2, c_o^2)$ be two models, then:

$$M_1 \sqcup M_2 = (\mathcal{F}, c_s, c_o) \text{ with } \forall x \in \mathcal{S},\, c_s(x) = c_s^1(x) \cup c_s^2(x) \text{ and } \forall x \in \mathcal{O},\, c_o(x) = c_o^1(x) \cup c_o^2(x)$$

$$M_1 \sqcap M_2 = (\mathcal{F}, c_s, c_o) \text{ with } \forall x \in \mathcal{S},\, c_s(x) = c_s^1(x) \cap c_s^2(x) \text{ and } \forall x \in \mathcal{O}\quad c_o(x) = c_o^1(x) \cap c_o^2(x)$$

Since the functions c_s and c_o return a set of sets of subjects, inclusion over sets easily leads to an ordering between models defined as follows:

$$M_1 \leq M_2 \Leftrightarrow \left(\forall x \in \mathcal{S}\ c_s^1(x) \subseteq c_s^2(x) \wedge \forall x \in \mathcal{O}\ c_o^1(x) \subseteq c_o^2(x)\right)$$

So we can declare this two methods along with their specifications :

```
sig ⊓ in self -> self -> self;

property ⊓_spec_mcs : ∀ x ∈ s, ∀ m1 m2 ∈ self,
spe!equal(!mcs(!⊓(m1,m2),x),
          spe!inter(!mcs(m1,x),!mcs(m2,x)));

property ⊓_spec_mco : ∀ x ∈ o, ∀ m1 m2 ∈ self,
spe!equal(!mco(!⊓(m1,m2),x),
          spe!inter(!mco(m1,x),!mco(m2,x)));
```

The method $\sqcup$ is defined in the same way with `union` instead of `inter`.

Hence, we introduce the species $M_{\mathbb{F}}$ which inherits from $\mathbb{M}_{\mathbb{F}}$ that contains the declaration and specification of $\leq$, together with the proofs that $\sqcap$ and $\sqcup$ are respectively the operators of greatest lower bound (glb) and least upper bound (lub) for $\leq$. Since it can be easily shown that $\sqcap$ and $\sqcup$ are distributive, and that there exists two models which are the minimum and the maximum, the species $M_{\mathbb{F}}$ also inherits from the species of complete distributive lattice defined in the Focal library.

```
species MF
  (s is 𝔖, sp is 𝔖⋆(s), spe is 𝔖⋆⋆(s, sp),
  spee is 𝔖⋆⋆⋆(s, sp, spe), o is 𝔒, eo is 𝔒⋆,
  n_s is ℒ𝒮,m is 𝒜,  m_a is 𝒜⋆(m),
  f is 𝔽 (s , sp, spe, spee,o, n_s, m,  m_a),
  f1 in f)
inherits
  𝕄F  (s, sp, spe, spee, o, so, n_s, m, m_a, f, f1),
  distributive_lattice, bounded_lattice =

  sig leq in self -> self -> bool;
  property spec_leq : ∀ x ∈ s, ∀ y ∈ o, ∀ m1 m2 ∈ self,
    !leq(m1, m2) ⇔
     spe!is_subset(!c_s(m1, x), !c_s(m2,x)) ∧
     spe!is_subset(!c_o(m1, y), !c_o(m2,y));
```

```
          (!leq(!⊓(m1,m2), m1) ∧
          !leq(!⊓(m1,m2), m2) ∧
          (!leq(m, m1) ∧
          !leq(m, m2))
    ⇒ !leq(m, !⊓(m1,m2)))
  proof : by !spec_leq, !⊓_spec_mcs, !⊓_spec_mco, ...
```

The order $\leq$ corresponds to the intuitive notion of stringness: for example, the minimum model is such that $c_s(s) = c_o(o) = \emptyset$ for all subject s and object o and then is clearly the more stringent model (no-one can change the security level of an entity).

So we can introduce the minimum model :

```
let min =
   let cs = fun x -> spe!empty_set in
   let co = fun x -> spe!empty_set in
   #crp(cs, co);
```

The maximum model is defined by :

```
let max =
   let cs = fun x -> f!𝒮+(f1) in
   let co = fun x -> f!𝒮+(f1) in
   #crp(cs, co);
```

Since min and max obviously respects the invariant, we can define the two following theorem :

```
theorem min_is_inv :
  !inv_spec_c(!min)
  proof : def !inv_spec_c, !min;

theorem max_is_inv : !inv_spec_c(!max)
  proof : by !build_𝒮+⋆ def !inv_spec_c, !min, ...;
```

We can then introduce the species $M_{\mathbb{F}^c}$, where the involved framework is refined into a coherent framework :

```
species MFc
   (s is 𝔖, sp is 𝔖⋆(s), spe is 𝔖⋆⋆(s, sp),
    spee is 𝔖⋆⋆⋆(s, sp, spe), o is 𝔒, eo is 𝔒⋆,
    n_s is ℒ𝒮,m is 𝒜,  m_a is 𝒜⋆(m),
    f is 𝔽c  (s , sp, spe, spee,o, n_s, m,  m_a),
    f1 in f)
inherits
   MF  (s, sp, spe, spee, o, so, n_s, m, m_a, f, f1) =

end
```

Note that $\mathcal{S}^{+\star}$ is defined in $\mathbb{F}_c$, and we prove that for all subsets S_1 and S_2 of $\mathcal{S}^{+\star}$, $S_1 \cup S_2$ and $S_1 \cap S_2$ belong to $\mathcal{S}^{+\star}$. Then it follows that $\sqcup$ and $\sqcap$ respect the invariant :

```
theorem ⊓_is_inv : ∀ x y ∈ self,
   !inv_spec_c(x) and !inv_spec_c(y) ⇒
   !inv_spec_c(!⊓(x, y))
 proof : ...

theorem ⊔_is_inv : ∀ x y ∈ self,
   !inv_spec_c(x) ∧ !inv_spec_c(y) ⇒
   !inv_spec_c(!⊔(x, y))
 proof : ...
```

Now, we introduce the species $M_{\mathbb{B}}$ which inherits from $M_{\mathbb{F}}$ and defines the complement of a model $\mathcal{M} = (\mathcal{F}, c_s, c_o)$ as the model $\bar{\mathcal{M}} = (\mathcal{F}, \bar{c_s}, \bar{c_o})$ where:

$$\forall x \in \mathcal{S}\ \bar{c_s}(x) = \mathcal{S}^+ \setminus c_s(x)\ \wedge\ \forall y \in \mathcal{O}\ \bar{c_o}(y) = \mathcal{S}^+ \setminus c_o(y)$$

During inheritance, we will prove supplementary results allowing this species to inherit from the species of boolean algebra (defined in the Focal library).

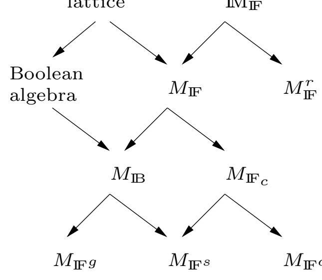

Figure 6: Hierarchy of models

```
species M_B
   (s is 𝔖, sp is 𝔖*(s), spe is 𝔖**(s, sp),
    spee is 𝔖***(s, sp, spe), o is 𝔒, eo is 𝔒*,
    n_s is L^S,m is A,  m_a is A*(m),
    f is F (s , sp, spe, spee,o, n_s, m,  m_a),
    f1 in f)
  inherits
    M_F  (s, sp, spe, spee, o, so, n_s, m, m_a, f, f1),
    boolean_algebra =

let complement (mod) =
 let cs = fun x -> spe!diff(f!S+(f1), !mc_s(mod, x)) in
 let co = fun x -> spe!diff(f!S+(f1), !mc_o(mod, x)) in
     #crp(cs, co);
```

The method `#crp` is used to create a pair. At this level, it's not possible to prove the required properties of boolean algebra since such a proof depends on the definitions of $\mathcal{S}^+$ and $\mathcal{S}^{+\star}$ which are not yet specified. So, we define two species $M_{\mathbb{F}^g}$ and $M_{\mathbb{F}^s}$ which inherits from $M_{\mathbb{B}}$: $M_{\mathbb{F}^g}$ is defined from a framework in $\mathbb{F}^g$ while $M_{\mathbb{F}^s}$ is defined from a framework in $\mathbb{F}^s$. Furthermore, $M_{\mathbb{F}^s}$ also inherits from the species $M_{\mathbb{F}_c}$, since $\mathbb{F}^s$ inherits from coherent frameworks. Because $\mathcal{S}^+$ and $\mathcal{S}^{+\star}$ are defined in $M_{\mathbb{F}^g}$ and $M_{\mathbb{F}^s}$, we can apply the invariants to $\sqcap$ and $\sqcup$, as we did in $M_{\mathbb{F}_c}$. Last, we define the species $M_{\mathbb{F}^c}$ which inherits from $M_{\mathbb{F}_c}$ for models built upon a coherent framework. In this case, we just have a complete distributive lattice (the complement operation does not preserve the property for $\mathcal{S}^{+\star}$). The figure (6) describes the hierarchy of species of models.

A more general (pre)ordering can be defined over models. Indeed, naming convention of subjects and objects may change from a model to another, and we want to be able to compare them independently of this naming convention. For this, J. McLean introduces the following relation: we say that $M_1 \lhd M_2$ iff there exists two one-to-one functions $I_s : \mathcal{S} \to \mathcal{S}$ and $I_o : \mathcal{O} \to \mathcal{O}$ such that:

$$\forall x \in \mathcal{S}\ \forall y \in \mathcal{O} \quad \left(\begin{array}{rl} & c^1_s(x) \subseteq (I^+_s)^{-1}(c^2_s(I_s(x))) \\ \wedge & c^1_o(y) \subseteq (I^+_s)^{-1}(c^2_o(I_o(y))) \end{array} \right)$$

where

$$\begin{array}{l} \forall X \in \mathcal{S}^{+\star} \quad I^+_s(X) = \{K(Z) \mid Z \in X\} \\ \text{with} \quad K(Z) = \{I_s(x) \mid x \in Z\} \end{array}$$

In fact, $\lhd$ is just a preorder from which we can define an equivalence relation $\equiv$ and an order relation $\lesssim$ over equivalence classes of models. As $\lhd$ generalises $\leq$, we try to generalize the operators $\sqcap$ and $\sqcup$. However, the set of equivalence classes of models is not a lattice but for each pair of models M_1 and M_2, we can define the following set containing every least upper models of M_1 and M_2:

$$\Psi^+(M_1, M_2) = \left\{ \begin{array}{ll} M \in M^r_{\mathbb{F}} \mid & [M_1] \lesssim [M] \wedge [M_2] \lesssim [M] \wedge \\ & \forall M' \quad [M_1] \lesssim [M'] \wedge \\ & [M_2] \lesssim [M'] \Rightarrow \neg([M'] < [M]) \end{array} \right\}$$

where $<$ is the strict order induced by $\lesssim$ and where $[M]$ denotes the equivalence class of the model M. However, $\Psi^+(M_1, M_2)$ cannot be computed efficiently and we have only implemented the following operator:

$$M_1 \Delta M_2 = \left\{ \begin{array}{ll} M \in M^r_{\mathbb{F}} \mid & \exists M_3, M_4 \in M^r_{\mathbb{F}} \quad M_3 \equiv M_1 \wedge \\ & M_4 \equiv M_2 \wedge M = M_3 \sqcup M_4 \end{array} \right\}$$

presented in detail in [13] and are introduced in the species $M^r_{\mathbb{F}}$ which inherits from $\mathbb{M}_{\mathbb{F}}$.

3.3 Systems

Last comes the notion of system, where the security policy and the access control function must be specified. Given a framework and a model, a system can be viewed as an abstract state machine. The set Σ of states is defined as the product:

$$\Sigma = \mathbb{F} \times \mathbb{A} \times \mathbb{A}$$

where:

- $\mathbb{F}$, often called classification vector, is the product $\mathbb{F}_s \times \mathbb{F}_o$ with:
  - $\mathbb{F}_s = \{f_s : \mathcal{S} \to \mathcal{L}^S\}$ (security level associated to a subject)
  - $\mathbb{F}_o = \{f_o : \mathcal{O} \to \mathcal{L}^S\}$ (security level associated to an object)
- $\mathbb{A} = \wp(\mathcal{S}^+ \times \mathcal{O} \times \mathcal{A}^\star)$ is used to describe granted and current access of subjects over objects.

Hence, a state $\sigma = ((f_s, f_o), d, m)$ defines mandatory rights (f_s and f_o), discretionary rights (d) and current access (m).

So we first define the species of access :

```
species access (s is 𝔖, sp is 𝔖*(s), o is 𝔒,
                m is A, ma is A*(m))
inherits setoid =
   rep = sp * (o * ma); ...
```

Then we can introduce the species of states :

```
species Σ(
  s is 𝔖, sp is 𝔖*(s), spe is 𝔖**(s, sp),
  spee is 𝔖***(s, sp, spe), o is 𝔒, eo is 𝔒*,
  n_s is L^S, m is A,  m_a is A*(m),
  f is F (s , sp, spe, spee,o,eo n_s, m,  m_a),
  f1 in f, a is access (s, sp, o, m, m_a),
  d is set_access (s,sp,o,m,m_a,a),
  mat is set_access (s,sp,o,m,m_a,a))
inherits setoid =
  rep = ((s -> n_s) * (o -> n_s)) * (d * mat);
```

where `set_access` is the species of finite sets of access.

In order to specify joint access security policy, we extend f_s as follows:

$$\begin{array}{ll} f^\curlywedge_s : \mathcal{S}^+ \to \mathcal{L}^S & f^\curlyvee_s : \mathcal{S}^+ \to \mathcal{L}^S \\ f^\curlywedge_s(S) = \curlywedge\{f_s(s) \mid s \in S\} & f^\curlyvee_s(S) = \curlyvee\{f_s(s) \mid s \in S\} \end{array}$$

At this level, we can define the security properties over states. Since we have implemented the Bell and La Padula system as an example of use of our development, we specify in this species the three classical following properties:

- a state is said to be simple secure iff:

$$\forall (S, o, A) \in m \ \mathsf{read} \in A \Rightarrow f^\curlywedge_s(S) \succeq f_o(o)$$

```
letprop simple_sur (e in self) = ∀ x ∈ a,
    mat!is_in(x, !mca(e)) ⇒
    m_a!is_in(m!read, a!mode(x)) ⇒
    n_s!order_inf(!f_o(e, a!objet( x)),
                  !f_s_sup(e, a!sujet(x)));
```

- a state is said to be $\star$-secure iff:

$$\forall S_1, S_2 \in \mathcal{S}^+\ \forall o_1, o_2 \in \mathcal{O}$$

$$\left(\begin{array}{ll} (S_1, o_1, \mathsf{read}) \in m & \wedge \\ (S_2, o_2, \mathsf{write}\ ) \in m & \wedge \\ S_1 \cap S_2 \neq \emptyset & \end{array} \right) \Rightarrow f_o(o_1) \preceq f_o(o_2)$$

```
    ((mat!is_in(x, !mca(e)) ∧
      mat!is_in(y, !mca(e)) ∧
      m_a!is_in(m!read, a!mode(x)) ∧
      m_a!is_in(m!write, a!mode(x)) ∧
      ¬ sp!equal(sp!inter(a!sujet(x), a!sujet(y)),
                 sp!vide)) ⇒
      n_s!order_inf(!f_o(e, a!objet(x)),
                    !f_o(e, a!objet(y))));
```

- a state is said to be ds-secure iff:

$$\forall S \in \mathcal{S}^+ \ \forall o \in \mathcal{O},\ (S, o, x) \in m \Rightarrow (S, o, x) \in d$$

```
letprop ds_sur (e in self) = ∀ x ∈ a,
    mat!is_in(x, !mca(e)) ⇒
    d!is_in(x, !md(e));
```

A state satisfying these three properties is said to be secure.

```
letprop sur (e in self) =
   !simple_sur(e) ∧ !mac_star(e) ∧ !ds_sur(e);
```

Now, we can introduce the species of systems by specifying a transition function $\tau : \mathcal{S}^+ \times \mathcal{R} \times \Sigma \rightarrow \mathcal{D} \times \Sigma$ between states where $\mathcal{R}$ is a set of requests and $\mathcal{D}$ is a set of answers. In this species, we define a property over such transition functions: τ is said to be secure iff

$$\tau(S_1, r, \sigma_1) = (d, \sigma_2) \Rightarrow \begin{pmatrix} \forall s \in \mathcal{S}\ f_s^1(s) \neq f_s^2(s) \Rightarrow S_1 \in c_s(s) \\ \forall o \in \mathcal{O}\ f_o^1(o) \neq f_o^2(o) \Rightarrow S_1 \in c_o(o) \end{pmatrix}$$

That is, if the security level of an entity has changed, then subjects that perform the change are granted to do it (by the model).

```
species system (
  s is 𝔖, sp is 𝔖⋆(s), spe is 𝔖⋆⋆(s, sp),
  spee is 𝔖⋆⋆⋆(s, sp, spe), o is 𝔒, eo is 𝔒⋆,
  n_s is ℒ^𝒮, m is 𝒜,  m_a is 𝒜⋆(m),
  f is 𝔽 (s , sp, spe, spee,o,eo n_s, m,  m_a),
  f1 in f,
  mod is 𝕄_𝔽 (s, sp, spe, spee, o, eo,
             n_s, m, m_a, f, f1),
  m1 in mod, a is access (s, sp, o, m, m_a),
  d is set_access (s,sp,o,eo,m,m_a,a),
  mat is set_access (s,sp,o,eo,m,m_a,a),
  r is request, dc is decision)
inherits Σ (s, sp, spe, spee, o, eo, n_s,
              m, m_a, f, f1, a, d, mat) =
  sig τ in sp -> r -> e -> (dc*e);
end
```

Last, we introduce the species of secure systems which are system with a secure initial state and a secure transition function mapping every secure state into a secure state (the well-known "Basic Security Theorem" of Bell and La Padula).

```
species system_bst (...)
inherits system (...) =
  property bst : ∀ req ∈ r, ∀ x ∈ sp,
     ∀ st ∈ self, !sur(st) ⇒
     !sur(#scnd(!tau(x, req,st)));
end
```

In this way, we have formalised in Focal the whole algebra of security of McLean. Now, we are going to see how to instantiate this algebra in order to obtain the Bell and LaPadula model in a first time, and this model in a database in a second time.

4 Applications

In this section, we briefly describe the instanciation of our framework in order to obtain the Bell and La Padula system, and, then, we show how this program can be used to manage access into a relational database.

In order to implement the Bell and La Padula system, we build a system from a framework in $\mathbb{F}^s$ in which $\mathcal{L}^S$ is the product lattice $T_c \times T_k$ where:

- $T_c = (\mathcal{C}l, \leq, \sqcup_{cl}, \sqcap_{cl})$ is a lattice of classifications
- and $T_k = (\wp(\mathcal{K}), \subseteq, \cup, \cap)$ is the powerset lattice of a set $\mathcal{K}$ of needs-to-know

This product is obtained by using the implementation of product between lattices in the Focal library. The set of access modes is $\mathcal{A} = \{\mathsf{read}, \mathsf{write}\ , \mathsf{execute}\ , \mathsf{append}\ , \mathsf{control}\ \}$. The Bell and La Padula system is built from a model such that $c_s(s) = c_o(o) = \mathcal{S}^+$ for all subject s and object o. Now, in order to obtain a complete system, we have just to define requests, answers and the transition function. This has been done by following the paper of Bell and La Padula [8] and by reusing a formalisation within the Coq system [16] of a similar work [6]. Indeed, in [6], we have implemented a formal description of the Bell and La Padula model together with the formal proof of the "Basic Security Theorem", checked by the theorem prover Coq.

4.2 Access control in a database

The previous development provides an abstract system, with the Bell and La Padula transition function and the proof that this function satisfies the "Basic Security Theorem". Now, we want to refine this system in order to apply the Bell and LaPadula security policy in a database. We do not want to define the complete database in Focal, but rather to define a module and to plug it into an existing database, such as MySQL. So we have to catch every SQL query sent to the server and translate it terms of access, thus providing requests for the access control system. Hence, a (formal) semantics of SQL has been defined in terms of access. Then these requests are executed by our Focal program, which returns an answer. If this answer is yes, then the SQL query is executed by the SQL server, else an error message is returned to the user (see figure (7)). The initial state (of the Bell and LaPadula's system) is loaded from the private part of the database when the program starts and stored in this part at the end.

Since we already have the definition of the system, we only have to define the species of subjects and objects.

We implement the species of subjects by the user of the database, and the security level of a subject is given at the creation of the user.

There are several ways to define the objects. The first is to consider each cell of a table as an object. This solution has the disadvantage to double the size of the table. Indeed, we have to store for each cell its level of security. Another solution is to consider each row as an object. This solution, as the above one, implies to execute every SQL query in order to know rows accessed. Such a way to proceed can compromise the integrity of the database. Indeed, let us imagine that a user submit a `DELETE` query. If we have to execute this query before authorizing it or denying it, the user could delete some information he is not allowed to. So, defining objects as rows implies the use of a transaction mechanism, which is not implemented in every database server.

Finally, we have to define an object as a whole table rather than columns, mostly for reasons of simplicity. Of course, more sophisticated security models exist for databases, but our aim was to show that our "formal program" can be used in concrete contexts. This whole development can be found in [2].

5 Conclusion

In this paper, we have described ways of mapping general security algebras (in the sense of McLean's algebras) into the Focal programming environment, to provide the ability to produce validated implementations. Such an approach allows to avoid incoherences in the security model and inconsistencies between the model and the code. We think it can help to trust the implementation.

This development shows that even from a practical point of view, formal methods can be used to increase the security of software.

under development.

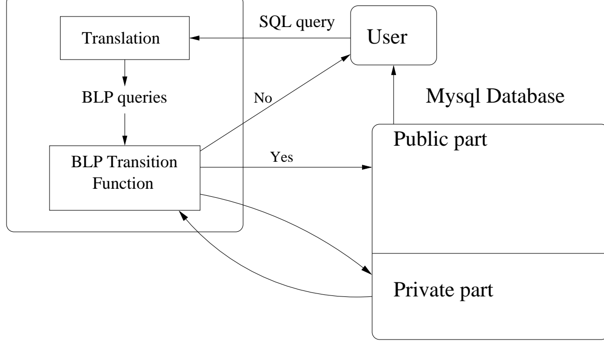

Figure 7: Access control in a database

Indeed, in this work, starting from a mathematical presentation of the algebra of security models, we have completely formalised this notion within the Focal environment and designed the architecture of its implementation.

Moreover, Focal allows us to be very general during the development. For example, when defining the species of frameworks, models and systems, nothing is said about subjects and objects (no concrete datatype is associated to the abstract types of subjects and objects). This is only done at the end when we define a collection for a specific system (e.g. database access control). In this work, we have obtained a complete specification of a system with access control and a transition function: we can apply it to any specific system, as a file manager or a database.

Another useful feature provided by Focal is the theorem prover Zenon. Indeed, such a prover allows to implement proofs in a very natural and declarative way without any background about the Coq system. Thus, it becomes possible for a large public to make proofs within Focal.

Note that different kinds of users are involved in such a development. First, the Focal-developer supplies frameworks, models and systems, and possibly specific systems, like the Bell and La Padula system. This developer is in charge to give a sound specification and proofs that the implementation meets this specification. Then, the system administrator picks one framework, one model and one system, and gives an implementation of subjects and objects. The system administrator does not need to make any proof, but just has to use the correct formal system in regards to his specific system. Last, the system user can access to the data of the system with methods provided by the system administrator, which are generally the standard methods provided with this kind of specific system (e.g. open_file, read_file, etc), and of course these operations are sound in regards to the security policy.

Of course, since this paper does not introduce any new access control policy and deals with well-known results (algebra of security and Bell and LaPadula's security policy), one could think that it does not contain any new material. However, our aim was to describe access control policies at a level of formalisation that is very close of a formal specification and/or implementation. Hence, one of our motivation was to fill the gap between (in)formal papers and formal implementations in order to ease our development and to avoid some bugs (as shown in the introduction). We think that these bugs may be dangerous because the engineers and scientists tend to trust the answers given by security systems, and the consequences of a wrong computation can range from a few days of time lost in tracking down the error, to a complete failure of the system designed by the engineers.

The access control policies actually implemented are rather simple but we think that the techniques prensented here are general enough to be used in a variety of other settings: this development can be viewed as a basis for several extensions. For example, it could be interesting to implement others security models for access control (Chinese Wall, Roles Based Access Control, ...) in this framework. However, we also believe that some parts of this development can be reused in different contexts: for example, work on

Current works As we said, our long-term aim is to develop a formal library of access control policies together with their implementations. For this, we have implemented the algebra of security introduced by Mc Lean in order to ease reusing and we have obtained an implementation of the Bell and LaPadula's security policy by instantiating this algebra. The Bell and LaPadula's security policy is significant in "military" contexts as the Chinese Wall security policy [4], introduced by D.F.C. Brewer and M.J. Nash, is to the financial world. Hence, we now would like to implement such a security policy in our formal library. The basic idea of the Chinese Wall security policy is that people are only allowed to access to information which is not held to conflict with any other information that they already posses. Hence, one of the main difference between the Bell and LaPadula's and the Chinese Wall security policies comes from the fact that while the first one constrains the access to an object o by a subject s by considering security levels associated with both o and s, the second one constrains the access by what data the subject already holds access right to.

In order to reuse our previous developments, we want to implement the Chinese Wall security policy by following a lattice based interpretation of this policy as introduced by Sandhu in [19, 18][5]. However, here again, the need of formalisation is strong if we want to reach hight level of safety. Indeed, in addition to express the formal definitions in an uniform framework, we also want to be able to formally express that such an interpretation is a correct embedding of the Chinese Wall policy.

Nevertheless, the algebra of security is not expressive enough to take such considerations into account. More generally, we need a formalism allowing to express in a very concise way access control policies and their implementations, but also to compare access control policies and implementations. For example, we would like a more general definition of frameworks allowing security parameters to be either a lattice of security level or a set of conflict-of-interest classes or anything else. Furthermore, we want to be able to express in an abstract way the property stating that an implementation i_1 of a policy p_1 is a correct and complete embedding of the implementation i_2 of a policy p_2.

Roughly speaking, a possible way to achieve this goal, is to define an access control policy as a predicate over states of an abstract machine depending on security parameters, and an implementation as a transition function between states. By following such an approach, many properties can be defined over access control policies and implementations.

Hence, our current works consists in defining a generic framework allowing to express many aspects of access control policies and their implementations, in instantiating such a framework with variants of classical access control policies, in proving properties about these instantiations, and last in defining embeddings of access control implementations into another ones and proving them correct and complete.

References

[1] D. Bell and L. LaPadula. Secure Computer Systems: a Mathematical Model. Technical Report MTR-2547 (Vol. II), MITRE Corp., Bedford, MA, May 1973.

[2] J. Blond and C. Morisset. Formalisation et implantation d'une politique de sécurité d'une base de données. In INRIA, editor, *17ème Journées Francophones des Langages Applicatifs, JFLA'06*, 2006.

[3] S. Boulmé, Th. Hardin, and R. Rioboo. Some hints for polynomials in the Foc project. In *Calculemus 2001 Proceedings*, June 2001.

[5] However, the Sandhu's lattice-based intepretation of the Chinese Wall policy requires notions (like the distinction between users, principals and subjects) that are not necessary and one of our current works is also the definition of another lattice-based intepretation of the Chinese Wall policy.

policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–214, 1989.

[5] European Economic Communuty. Information Technology Security Evaluation Criteria (ITSEC). Technical report, CEE, 1990.

[6] E. Gureghian, Th. Hardin, and M. Jaume. A full formalisation of the Bell and Lapadula security model. Technical Report 2003-007, Univ. Paris 6, LIP6, 2003.

[7] M. Jaume and C. Morisset. Formalisation and implementation of access control models. In *Information Assurance and Security (IAS'05) International Conference on Information Technology, ITCC*, pages 703–708. IEEE CS Press, 2005.

[8] L.J. LaPadula and D.E. Bell. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4:239–263, 1996.

[9] X. Leroy. *The Objective Caml system release 3.09. Documentation and user's manual.* INRIA-Rocquencourt, 2004.

[10] McLean. The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, 1988.

[11] J. McLean. Security models and information flow. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 180–187, 1990.

[12] J. McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. J. Wiley & Sons, 1994.

[13] C. Morisset. Formalisation et implantation d'un modèle de contrôle d'accès dans l'atelier Focal. Master's thesis, Université Paris 6, 2004.

[14] NSA. Protection Profile For Multilevel Operating Systems In Environments Requiring Medium Robustness. Technical report, National Security Agency, 2001.

[15] V. Prevosto and D. Doligez. Algorithms and proof inheritance in the foc language. *Journal of Automated Reasoning*, 29(3-4):337–363, dec 2002.

[16] Logical Project. *The Coq Proof Assistant Reference Manual Version 8*. INRIA-Rocquencourt, 2004.

[17] R. Rioboo, D. Doligez, V. Prevosto, M. Jaume, M. Maarek, C. Dubois, S. Fechter, V. Ménissier-Morain, O. Pons, D. Delahaye, V. Viguié, and T. Hardin. *Focal, version 0.3.1 Tutorial and reference manual*. LIP6 – INRIA – CNAM, sept 2004. Distribution available at: `http://focal.inria.fr`.

[18] R. S. Sandhu. A lattice interpretation of the chinese wall policy. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pages 329–339, 1992.

[19] R. S. Sandhu. Lattice-Based Access Control Models. *IEEE Computer*, 26(11):9–19, November 1993.

Acknowledgements Many thanks to Julien Blond, Damien Doligez, Emmanuel Gureghian, Thérèse Hardin, Virgile Prevosto and Renaud Rioboo for enlightening discussions on this subject. We also thank anonymous referees of the conference version of this paper for their very usefull remarks.

Author Biographies

First Author Mathieu Jaume is lecturer at the LIP6 at the University Pierre and Marie Curie in Paris, France. Its research interests include security policies, formal methods and semantics of programming languages.

Second Author Charles Morisset is a PhD student at the LIP6 at the University Pierre and Marie Curie in Paris, France. Its research interests include formalisation and modelisation of security policies and formal methods.