Secure Software within Focal

Philippe Ayrault\textsuperscript{1}, Matthieu Carlier\textsuperscript{2}, David Delahaye\textsuperscript{3}, Catherine Dubois\textsuperscript{2}, Damien Doligez\textsuperscript{4}, Lionel Habils\textsuperscript{1}, Thérèse Hardin\textsuperscript{1}, Mathieu Jaume\textsuperscript{1}, Charles Morisset\textsuperscript{5}, François Pessaux\textsuperscript{1}, Renaud Rioboo\textsuperscript{2}, and Pierre Weis\textsuperscript{1}

\textsuperscript{1} Université Pierre et Marie Curie-Paris 6, SPI-LIP6, 104 avenue du Président Kennedy, Paris 75016, France,

\texttt{Firstname.LastName@lip6.fr}

\textsuperscript{2} ENSIIE-CEDRIC-CNAM

1 Square de la résistance, 91025 Evry Cedex, France,

\texttt{Firstname.LastName@ensie.fr}

\textsuperscript{3} CNAM-CEDRIC

292 rue Saint Martin, 75003, Paris, France

\texttt{Firstname.LastName@cnam.fr}

\textsuperscript{4} INRIA

Bat 8. Domaine de Voluceau, BP 105, F-78153 Le Chesnay, France,

\texttt{Firstname.LastName@inria.fr}

\textsuperscript{5} United Nations University, International Institute for Software Technologies, UNU-IIST, P.O. Box 3058, Macao SAR,

\texttt{Lastname@iist.unu.edu.fr}

\textbf{Abstract.} This paper describes the Integrated Development Environment Focal together with a brief proof of usability on the formal development of access control policies. Focal is an IDE providing powerful functional and object-oriented features that allow to formally express specification and to go step by step (in an incremental approach) to design and implement while proving that the implementation meets its specification or design requirements. These features are particularly well-suited to develop libraries for secure applications.

1 Introduction

Since at least forty years, an important area of software engineering is concerned with safety of industrial systems as those critical systems use more and more software components. Since twenty years, security problems of information systems spread from military domain to all society activities. Some fifteen years ago, safety and security of systems were usually considered of separated concern. Now, although their concerns are different, it is recognized that, in most cases, these two families of requirements must be considered together to receive satisfactory solutions (see the new rules for SCADA systems for example). Moreover, some methods to deal with safety requirements can be adapted to security requirements and conversely. Whatever is the domain, their methods are evolving, ad-hoc and empirical approaches being replaced by more formal methods. For example, for high levels of safety, formal models of the requirement/specification
phase are more and more considered as they allow mechanized proofs, test or static analysis of the required properties. In the same way, high level assurance in system security asks for the use of true formal methods along the process of software development and is often required for the specification level.

To ease developing high integrity systems with numerous software components, an Integrated Development Environment (IDE) must provide tools to formally express specifications, to describe design and coding and to ensure that specification requirements are met by the corresponding code. This is not enough. First, standards of critical systems ask for pertinent documentation which has to be maintained along all the revisions during the system life cycle. Second, the evaluation conformance process of software is by nature a sceptical analysis. Thus, any proof of code correctness must be easily redone at request and traceability must be eased. Third, design and coding are difficult tasks. Research in software engineering has demonstrated the help provided by some object-oriented features as inheritance, late binding and early research works on programming languages have pointed out the importance of abstraction mechanisms such as modularity to help invariant maintaining. There are a lot of other points which should also be considered when designing an IDE for safe and/or secure systems to ensure conformance with high Evaluation Assurance or Safety Integrity Levels (EAL-5,7 or SIL 3,4) and to ease the evaluation process according to various standards (e.g. IEC61508, CC, ...): handling of non-functional contents of specification, handling of dysfunctional behaviors and vulnerabilities from the true beginning of development and fault avoidance, fault detection by validation testing, vulnerability and safety analysis.

The main aim of this paper is to present an IDE, called Focal[22, 7] (freely distributed at http://focal.inria.fr), dedicated to the complete development of high integrity software, which attempts to give a positive solution to the three requirements identified above, the other items being currently under consideration. It provides means for the developers to formally express their specifications and to go step by step (in an incremental approach) to design and implementation while proving that such an implementation meets its specification or design requirements. It also provides some automation of documentation production and management.

Focal has already been used to develop huge examples. First, a computer algebra library was developed by Rioboo [25], it offers full specification and implementation of usual algebraic structures up to multivariate polynomial rings with complex algorithms. The point was to measure how Focal can help to render mathematical specifications and also to measure efficiency of the produced code, which is comparable (even little better) to the best computer algebra systems in existence. Such a library is very useful when formalising the algebra of access control models, using implementations of orderings, lattices and boolean algebras (presented below). Focal was also very successfully used to specify the security policy of an airport[5].
Focal semantics was initially specified in Coq, which brings a satisfactory confidence in the language’s correctness. On the other side, the correction of the compiler against Focal’s semantics is proved (by hand) [21].

In a second part, we will expose the usability of Focal through the formalization and development of access control policies. We present how formal methods can be used in practice to obtain trusted implementations of an access control policy. The library of access control policies offers a generic specification of the model and several developments based on it, giving implementations of classic access control policies.

2 The Focal philosophy

Since our aim is to have a unique framework from specification to implementation, from code to proofs of requirements, Focal provides a unique and manifold language to express all these aspects of software development.

Before really entering inside Focal, we briefly remind a few well-known concepts in term of software engineering.

Specifications of a system describe its functionalities, without referring to any particular practical solution. On the other side, the implementation gives an explicit, algorithmic, solution making the system running.

The Statement of Work of critical systems require the holding of certain properties (often called “requirements”) to ensure that the system will be indeed operational. All along the development cycle, such requirements must be expressed and verified. Moreover, each development stage may introduce its own requirements. For instance, at specification-time, a railway system may require that doors of a train can’t be opened while running. At implementation-time, this requirement must still hold, but some extra ones, due to implementation constraints may arise. For instance, the fact that a square root function is only called on positive numbers in the speed computation. This requirement was previously hidden since the speed comparison was not expressed finely enough to make the square root function appearing.

Considering these concepts, Focal allows management of declarations (specification), algorithms (implementation), properties (requirements) and proofs (demonstrations that requirements hold).

2.1 The basic brick

The primitive entity of a Focal development is the species. It can be viewed as a record grouping “things” related to a same concept. Like in most modular design systems (i.e. objected oriented, algebraic abstract types) the idea is to group a data structure with the operations to process it. Since in Focal we don’t only address data type and operations, among these “things” we also find the specification of these operations, the representation of requirements (properties) and their proofs.

We now describe each of these “things”, called methods.
- The method introduced by the keyword rep gives the data representation (carrier) that the species embeds. It is a type called the carrier type and defined by a type expression. The carrier may be not-yet-defined in a species, meaning that the real structure of the datatype the species embeds does not need to be known at this point. In this case, it is represented by a type variable. However, to obtain an implementation, the carrier has to be defined later either by setting rep = exp where exp is a type expression or by inheritance (see below). Type expressions in Focal are roughly ML-like types (variables, basic types, inductive types, record types) plus species carrier types, denoted by keyword Self inside the species and by the name of their species outside of them.

Each species has a unique method rep and thus, a unique carrier. This is not a restriction compared to other object-oriented languages where an object can own several private variables representing the internal state, hence the data structure of the object. In such a case, the carrier type can simply be the tuple grouping all these variables that were disseminated all along the object.

- Declarations (signature) introducing a name and a type allows to announce a method to be defined later, i.e. to only specify its type, without implementation yet. Such methods are especially dedicated for specification or design purposes since it allows to use the name of the introduced method to define others methods while delaying the choice of its implementation. The type provided by the signature allows Focal to ensure via type-checking that the method is used in contexts compatibles with this type. The late-binding and the collection mechanisms, further introduced, ensure that the definition of the method will be effectively known when needed.

- Definitions (let) made of a name, a type and an expression introduce functions, i.e. computational operations. The core language used to implement them is roughly ML-like expressions (let-binding, pattern matching, conditional, higher order functions, ...) with the addition of a construction to call a method from a given species. Mutually recursive definitions are introduced by let rec.

- Statements (property) introduce a name and a first-order formula. A property may serve to express requirements (i.e. facts that the system must hold to conform to the Statement of Work delivered by the customer) and then can be viewed as a specification purpose method, like signatures were for let-methods. It will lead to a proof obligation later in the development. A property may also be used to express some “quality” information of the system (soundess, correctness, ..) also submitted to a proof obligation. Formulae are written with usual logical connectors, universal and existential quantifications over a Focal type, and names of methods known within the species’s context. For instance, a property telling that if a speed is non-null, then doors can’t be opened could look like:

\[
\text{all } v \text{ in Speed, } v \not< \text{Speed!zero} \rightarrow \sim \text{doors.open}
\]
In the same way as signatures, even if no proof is yet given, the name of the property can be used to express other ones and its statement can be used as an hypothesis in proofs. Focal late binding and collection mechanisms ensure that the proof of a property will be ultimately done.

- Theorems (theorem) made of a name, a statement and a proof are properties together with the formal proof that their statement holds in the context of the species. The proof accompanying the statement will be processed by Focal and ultimately checked with the theorem prover Coq.

Like in any formal development, one severe difficulty before proving is obviously to enunciate a true interesting and meaningful statement. For instance, claiming that a piece of software is “formally proved” as respecting the safety requirements system_ok “since its property is demonstrated” is a lie if this property was, for instance, 1 = 1 \rightarrow \text{system\_ok}. This is obviously a nonsense since the text of the property is trivial and does not link system\_ok with the rest of the software (see [8] for less trivial examples).

We now make concrete these notions on an example we will incrementally extend. We want to model some simple algebraic structures. Let’s start with the description of a “setoid” representing the data structure of “things” belonging to a set, which can be submitted to an equality test and exhibited (i.e. one can get a witness of existence of one of these “things”).

```focal
species Setoid =
  signature ( = ) : Self \rightarrow Self \rightarrow bool ;
  signature element : Self ;

  property refl : all x in Self, x = x ;
  property symm : all x y in Self, x = y \rightarrow y = x ;
  property trans : all x y z in Self, x = y and y = z \rightarrow x = z ;

  let different (x, y) = basics#not\_b (x = y) ;

end ;;
```

In this species, the carrier is not explicitly given (no rep), since we don’t need to set it to be able to express functions and properties our “setoid” requires. However, we can refer to it via Self and it is in fact a type variable. In the same way, we specify a signature for the equality (operator =). We introduce the three properties that an equality (equivalence relation) must conform to.

We complete the example by the definition of the function different which use the name = (here basics#not\_b stands for the function not\_b, the boolean and coming from the Focal source file basics.foc). It is possible right now to prove that different is irreflexive, under the hypothesis that = is an equivalence relation (i.e. that each implementation of = given further will satisfy these properties).

It is possible to use methods only declared before they get a real definition thanks to the late-binding feature provided by Focal. In the same idea, redefining a method is allowed in Focal and, it is always the last version which is kept as the effective definition inside the species.
2.2 Type of species, Interfaces and collections

The type of a species is obtained by removing definitions and proofs. If `rep` is still a type variable say \( \alpha \), then the species type is prefixed with an existential binder \( \exists \alpha \). This binder will be eliminated as soon as the `rep` will be instantiated (defined) and must be eliminated to obtain runnable code. The species types remain implicit in the concrete syntax.

The interface of a species is obtained by abstracting the `rep` type in all the method types of the species type and this abstraction is permanent (see the paragraph Collections). No special construction is given to denote interfaces in the concrete syntax, they are simply denoted by the name of the species underly- ing them. Interfaces can be ordered by inclusion, a point providing a very simple notion of subtyping.

A species is said to be complete if all declarations have received definitions and all properties have received proofs.

When complete, a species can be submitted to an abstraction process of its carrier to create a collection. Thus the interface of the collection is built out of the type of its underlying species. A collection can hence be seen as an abstract data type, only usable through the methods of its interface, but having the guarantee that all methods/theorems are defined/proved.

2.3 Combining bricks by inheritance

A Focal development is organised as a hierarchy which may have several roots. The upper levels of the hierarchy are built during the specification stage while the lower ones correspond to implementations. Each node of the hierarchy, i.e. each species, is a progress to a complete implementation. On the previous example, forgetting different, we typically presented a kind of species for “specification” since it expressed only signatures of functions to be later implemented and properties to which, later, give proofs.

We can now create a new species, may be more complex, by inheritance of a previously defined. We say here “may be more complex” because it can add new operations and properties, but it can also only bring real definitions to signatures and proofs to properties, adding no new method.

Hence, in Focal inheritance serves two kinds of evolutions. In the first case the evolution aims making a species with more operations but keeping those of its parents (or redefining some of them). In the second case, the species only tends to be closer to a “run-able” implementation, providing explicit definitions to methods that were previously only declared. A strong constraint in inheritance is that the type of inherited, and/or redefined methods must not change. This is required to ensure consistence of the Focal model, hence of the developed software.

Continuing our example, we want to extend our model to represent “things” with a multiplication and a neutral element for this operation.

```
species Monoid inherits Setoid =
signature ( * ) : Self -> Self -> Self ;```
We see here that we added new methods but also gave a definition to element, saying it is the application of the method \(*\) to one twice, both of them being only declared. Here, we used the inheritance in both the presented ways: making a more complex entity by adding methods and getting closer to the implementation by explicitly defining element.

Multiple inheritance is available in Focal. For sake of simplicity, the above example uses simple inheritance. In case of inheriting a method from several parents, the order of parents in the inherits clause serves to determine the chosen method.

The type of a species built using inheritance is defined like for other species, the methods types retained inside it being those of the methods present in the species after inheritance is resolved.

### 2.4 Combining bricks by parametrisation

Until now we are only able to enrich species (may be “refine”, even if we do not address the notion of “refinement” of the Atelier B[1]). However, we sometimes need to use a species, not to take over its methods, but rather to use it as an “ingredient” to build a new structure. For instance, a pair of setoids is a new structure, using the previous species as the “ingredient” to create the structure of the pair. Indeed, the structure of a pair is independent of the structure of each component it is made of. A pair can be seen as parametrised by its two components. Following this idea, Focal allows two flavors of parametrisation.

**Parametrisation by collection parameters** We first introduce the collection parameters. They are collections that the hosting species may use through their methods to define its own ones.

A collection parameter is given a name C and an interface I. The name C serves to call the methods of C which figure in I. C can be instantiated by an effective parameter CE of interface IE: CE is a collection and its interface IE must contain I. Moreover, the collection and late-binding mechanisms ensure that all methods appearing in I are indeed implemented (defined for functions, proved for properties) in CC. Thus, no runtime error, due to linkage of libraries, can occur and any property stated in I can be safely used as a hypothesis.

Calling a species’s method is done via the “bang” notation: !\text{meth} or \text{Self!meth} for a method of the current species (and in this case, even simpler: \text{meth}, since the Focal compiler will resolve scoping issues). To call collection parameters’s method, the same notation is used: A!\text{element} stands for the method \text{element} of the collection parameter A.

To go on with our example, a pair of setoids has two components, hence a species for pairs of setoids will have two collection parameters. It is itself a setoid, a fact which is simply recorded via the inheritance mechanism: inherits Setoid gives to Setoid\_product all the methods of Setoid.
species Setoid_product (A is Setoid, B is Setoid) inherits Setoid =
  rep = (A × B) ;
let (=) (x, y) =
  and _b
  (A! (=) (first (x), first (y)),
   B! (=) (scnd (x), scnd (y))) ;
let create (x, y) in Self = basics#crp (x, y) ;
let element = Self!create (A!element, B!element) ;
proof of refl = by definition of (=) ;
end ;;

We express the carrier of the product of two setoids as the cartesian product
of the carriers of the two parameters. In A × B, × is the Focal type constructor
of pairs, A denotes indeed the carrier type of the first collection parameter, and
B the one of of the second collection parameter.

Next, we add a definition for = of Setoid_product, relying on the methods
= of A (A! (=)) and B (which are not yet defined). Similarly, we introduce a
definition for element by building a pair, using the function create (which calls
the predefined function basics#crp) and the methods element of respectively
A and B. And we can prove that = of Setoid_product is indeed reflexive, upon
the hypothesis made on A! (=) and B! (=). The part of Focal used to write
proofs will be shortly presented later, in section 2.6.

This way, the species Setoid_product builds its methods relying on those of
its collection parameters. Note the two different uses of Setoid in our species
Setoid_product, which inherits of Setoid and is parametrised by Setoid.

Why such collection parameters and not simply species parameters? There
are two reasons. First, effective parameters must provide definitions/proofs for
all the methods of the required interface: this is the contract. Thus, effective
parameters must be complete species. Then, we do not want the parametrisation
to introduce dependencies on the parameters’ carrier definitions. For example,
it is impossible to express “if A!rep is int and B!rep is bool then A+B is a list
of boolean values”. This would dramatically restrict possibilities to instantiate
parameters since assumptions on the carrier’s structure, possibly used in the
parametrised species to write its own methods, could prevent collections having
the right set of methods but a different internal representation of the carrier to
be used as effective parameters. Such a behaviour would make parametrisation
too weak to be usable. We choose to always hide the carrier of a collection
parameter to the parametrised hosting species. Hence the introduction of the
notion of collection, obtained by abstracting the carrier from a complete species.

Parametrisation by entity parameters Let’s imagine we want to make a
species working on natural numbers modulo a certain value. In 5 modulo 2 is 1,
both 5 and 2 are natural numbers. To be sure that the species will consistently
work with the same modulo, this last one must be embedded in the species.
However, the species itself doesn’t rely on a particular value of the modulo.
Hence this value is clearly a parameter of the species, but a parameter in
which we are interested by its value, not only by its carrier and the methods acting on it. We call such parameters entity parameters, their introduction rests upon the introduction of a collection parameter and they denote a value having the type of the carrier of this collection parameter.

Let’s first have a species representing natural numbers:

```haskell
species IntModel =
  signature one : Self ;
  signature modulo : Self -> Self -> Self ;
end ;
```

Note that IntModel can be later implemented in various ways, using Peano’s integers, machine integers, arbitrary-precision arithmetic ... We now build our species “working modulo …”, embedding the value of this modulo like:

```haskell
species Modulo_work (Naturals is IntModel, n in Naturals) =
  let job1 (x in Naturals) in .... =
    .... Naturals!modulo (x, n) .... ;
  let job2 (x in Naturals, ...) in .... =
    .... .... Naturals!modulo (x, n) .... ;
end ;
```

Using the entity parameter n, we ensure that the species Modulo_work will work for any value of the modulo, but will always use the same value n of the modulo everywhere inside the species.

### 2.5 The final brick

As briefly introduced in 2.2, a species needs to be fully defined to lead to executable code for its functions and checkable proofs for its theorems. When a species is fully defined, it can be turned into a collection, that can roughly be seen as an instance of the species. Hence, a collection represents the final stage of the inheritance tree of a species and leads to an effective structure of the carrier with executable functions processing it.

For instance, providing that the previous species IntModel turned into a fully-defined species MachineNativeInt through inheritance steps, with a method from_string allowing to create the natural representation of a string, we could get a related collection by:

```haskell
collection MachineNativeIntColl implements MachineNativeInt ;
```

Next, to get a collection implementing arithmetic modulo 8, we could extract from the species Modulo_work the following collection:

```haskell
collection Modulo_8_work implements Modulo_work
  (MachineNativeIntColl, MachineNativeIntColl!from_string ('8')) ;
```

As seen by this example, a species can be applied to effective parameters by giving their values with the usual syntax of parameter passing.

As said before, to ensure modularity and abstraction, the carrier of a collection turns hidden. This means that any software component dealing with a collection will only be able to manipulate it through the operations (methods)
2.6 Properties, theorems and proofs

Focal aims not only to write programs, it intends to encompass both the executable model (i.e. program) and properties this model must satisfy. For this reason, “special” methods deal with logic instead of purely behavioral aspects of the system: theorems, properties and proofs.

Stating a property expects that a proof that it holds will finally be given. For theorems, the proof is directly embedded in the theorem. Such proofs must be done by the developer and will finally be sent to the formal proof assistant Coq who will automatically check that the demonstration of the property is consistent. Writing a proof can be done in several ways.

It can be written in “Focal’s proof language”, a hierarchical proof language that allows to give hints and directions for a proof. This language will be sent to an external theorem prover, Zenon[6] developed by D. Doligez. This prover is a first order theorem prover based on the tableau method incorporating implementation novelties such as sharing. Zenon will attempt, from these hints to automatically generate the proof and exhibit a Coq term suitable for verification by Coq. Basic hints given by the developer to Zenon are: “prove by definition of a method” (i.e. looking inside its body) and “prove by property” (i.e. using the logical body of a theorem or property”. Surrounding this hints mechanism, the language allows to build the proof by stating assumptions (that must obviously be demonstrated next) that can be used to prove lemmas or parts for the whole property.

The detailed presentation of Zenon, its internals and its language (the one we called “Focal’s proof language”) is outside the scope of this presentation. We however show below an example of such demonstration.

```
theorem order_inf_is_infimum: all x y i in Self,
  !order_inf(i, x) -> !order_inf(i, y) ->
  !order_inf(i, !inf(x, y))
proof:
  <1>1 assume x in Self, assume y in Self,
  assume i in Self, assume H1: !order_inf(i, x),
  assume H2: !order_inf(i, y),
  prove !order_inf(i, !inf(x, y))
  <2>1 prove !equal1(i, !inf(!inf(i, x), y))
by hypothesis H1, H2
property inf_left_substitution_rule,
  equal_symmetric, equal_transitive
definition of order_inf
<2>9 qed
by step <2>1
property inf_is_associative, equal_transitive
definition of order_inf
<1>2 qed.
```

The important point is that Zenon works for the developer: it searches the proof itself, the developer does not have to elaborate it formally “from scratch”.

it provides. This point is especially important since it prevents other software components from possibly breaking invariants required by the internals of the collection.
Like any automatic theorem prover, Zenon may fail finding a demonstration. In this case, Focal allows to write verbatim Coq proofs. In this case, the proof is not anymore automated, but this leaves the full power of expression of Coq to the developer.

Finally, the assumed keyword is the ultimate proof backdoor, telling that the proof is not given but that the property must be admitted. Obviously, a really safe development should not make usage of such “proofs” since they bypass the formal verification of software’s model. However, such a functionality remains needed since some of “well-known” properties can never be proved for a computer. For instance, \( \forall x \in \mathbb{N}, x + 1 > n \) does not hold in a computer with native integers: in this case arithmetic works modulo the number of bits of the machine word! However, in a mathematical framework, this property holds and is needed to carry out other proofs! On another side, a development may be linked with external code, trusted or not, but for which properties can’t be proved inside the Focal part since it does not belong to it. Expressing properties of the Focal part may need to express properties on the imported code, that can’t be formally proved, then must be “assumed”.

2.7 Around the language

In the previous sections, we presented Focal through its programming model and shortly its syntax. We especially investigated the various entities making a Focal program. We now address what becomes a Focal program once compiled. We recall that Focal supports the redefinition of functions, which permits for example to specialize code to a specific representation of the carrier (for example, there exists a generic implementation of integer addition modulo \( n \) but it can be redefined in arithmetics modulo 2 if boolean values are used to represent the two values). It is also a very convenient tool to maintain software.

Consistency of the software All along the development cycle of a Focal program, the compiler keeps trace of dependencies between species, their methods, the proofs, … to ensure that modifications of an entity will be detected on any of those depending of the former.

Focal considers two types of dependencies:

- The decl-dependency: a method \( A \) decl-depends on a method \( B \), if the declaration of \( B \) is required to write \( A \).
- The def-dependency: a method (and more especially, a theorem) \( A \) def-depends on a method \( B \), if the definition of \( B \) is required to write \( A \) (and more especially, to prove the property stated by the theorem \( A \)).

The redefinition of a function may invalidate the proofs that use properties of the body of the redefined function. All the proofs which truly depend of the definition are then erased by the compiler and must be done again in the context updated with the new definition. Thus the main difficulty is to choose the best level in the hierarchy to do a proof. In [23], Prevosto and Jaume propose a coding
style to minimize the number of proofs to be redone in the case of a redefinition, by a certain kind of modularisation of the proofs.

**Code generation** Focal currently compiles programs toward two languages, OCaml to get an executable piece of software, and Coq to have a formal model of the program, with theorems and proofs.

In OCaml code generation, all the logical aspects are discarded since they do not lead to executable code.

Conversely, in Coq, all the methods are compiled, i.e. “computational” methods and logical methods with their proofs. This allows Coq to check the entire consistence of the system developed in Focal.

Since Focal’s compilation model is based on record types (i.e. struct à la C) code generation toward most of the current programming languages can be imagined. A C backend is currently under study, which should allow to manage once for all the compilation of high-level constructs found in ML-like languages and reused in Focal (pattern-matching, higher-order functions, variant types . . .).

**Tests** A tool for Integration/Validation testing of Focal developments is announced. We have no room in this paper to present it. It is described in [18].

**Documentation** The tool called FOCDOC [17] automatically generates documentation, thus the documentation of a component is always coherent with respect to its implementation.

This tool uses its own XML format that contains information coming not only from structured comments (that are parsed and kept in the program’s abstract syntax tree) and Focal concrete syntax but also from type inference and dependence analysis. From this XML representation, FOCDOC generates HTML files or Latex files. Although this documentation is not the complete safety case, it can helpfully contribute to its elaboration.

Focal’s architecture is designed to easily plug third-parties analyses that can use the internal structures elaborated by the compiler from the source code. This allows, for example, to make dedicated documentation tools for custom purposes, just exploiting information stored in the Focal program’s abstract syntax tree, or extra information possibly added by extra processes, analyses.

3 Access control : a Case Study within Focal

Access control is any mechanism by which a system grants or revokes the rights for active entities, the subjects, to access some passive entities, the objects, or perform some action. In this section we present a brief survey of several approaches that we have followed in order to build a formal library of access control policies. Such a work needs to formalise in a precise and unambiguous way the access control policies we want to implement, and requires the use of an IDE that allows to reach high Evaluation Assurance Levels. Many papers describing
security 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. 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”. Last, even if proofs done by hand are correct, nothing formally ensures that the implementation meets the “formal” specification done “on the paper”. We illustrate here our approach by considering well-known access control policies.

Full formalisation of BLP within Coq In [9], we have formalised within Coq [24] the Bell & LaPadula (BLP) model by following the original paper [16]. The system is represented by an abstract machine containing a state that operations (or requests) can change. A state of the system is a tuple \( (m, D, f_s, f_o) \) where \( m \) is the set of current accesses, \( D \) is the set of access rights and \( f_s : S \rightarrow \mathcal{L} \) (resp. \( f_o : \mathcal{O} \rightarrow \mathcal{L} \)) defines security levels associated with subjects (resp. objects).

Three classical security properties are defined over states: each current access must belong to \( D \), each read access over and object \( o \) must be done by a subject whose security level is greater than the security level of \( o \), and the famous “no write-down” property which prevents copying of an object to a lower security level. Given the notion of states, we can define transition functions as mappings from \( \mathcal{R} \times \Sigma \) to \( \mathcal{D} \times \Sigma \) where \( \mathcal{R} \) is a set of requests, \( \mathcal{D} = \{ \text{yes}, \text{no} \} \) is a set of answers and \( \Sigma \) is the set of states. Then, we say that a transition function \( \tau \) is secure iff \( \tau \) preserves the security properties. The approach presented in the initial paper of BLP consists in the definition of the 10 rules of transition which are easily encoded into the definition of a transition function \( \tau_{BLP} \). We have formalised within Coq the proof of the “Basic Security Theorem” [16] stating that \( \tau_{BLP} \) is secure. Then, by using the program extraction (from a proof) mechanism of Coq, we have obtained a certified implementation of this access control policy. Such development formally ensures that the program we have obtained satisfies the desired security properties, thus providing a greater level of confidence. However, it is rather technical and time-consuming and requires some backgrounds within Coq. Furthermore, this formalisation cannot be easily reused if we want to implement another policy. Indeed many policies share some definitions and properties and, 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. These questions are addressed in the following.

BLP as an instance of McLean Algebra In order to ease the reusing of formal developments of access control policies, we have used Focal to implement the algebra of security model introduced by J.McLean in [19]. Such algebra
provides a generic framework to specify policies. From this implementation, we have instantiated the framework in order to obtain a formal development of the BLP model. Then, the obtained Focal program has been used to manage accesses in a relational database. All this work is described in [11, 12, 3, 4].

McLean algebra of security The algebra of security models contains three levels of specifications (see fig. 1). First, frameworks are parameterized by sets \( S \) (subjects), \( O \) (objects), \( L \) (lattice of security levels) and \( A \) (access modes) and specify what are the groups of subjects that can jointly initiate an operation. At this level, we only specify “who can initiate”, but we don’t specify if the security policy allow the operation. Depending on these specifications, several kinds of frameworks are possible. This leads to define a hierarchy of frameworks. Then, given a framework, we can define the notion of models. A model specifies who can modify the security level of some subject or object. Here again, there is a hierarchy of models. Next, the notion of state of the system is defined from a framework and allows to specify the security policy considered. Last, given a model and the notion of state, a system specifies what is a transition function over states (requests, answers, ...) and what abstract properties such functions have to satisfy. Then, an implementation is just an instantiation of a system.

BLP as an instance of McLean algebra In order to implement the original BLP policy within Focal, we have instantiated a system by considering a framework for which no joint access is allowed and a model for which each subject can ask to modify security levels of objects or subjects (of course, this does not mean that each subject is allowed to perform such a modification), by specifying the security properties, by implementing the transition function \( \tau_{BLP} \) and by proving that \( \tau_{BLP} \) is secure.

Application to BD In order to illustrate the practical applicability of the previous development, we have refined this system to manage accesses into a relational database. To achieve this goal, we do not want to define the complete database in Focal, but rather to define a reference monitor 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 in terms of access, thus providing requests for the access control system. 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 (2)). 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.

Definition of an abstract framework to specify, implement and compare access control policies Many access control models are described in the literature. Each of them provides a description their own notion of information system together with a specification of the granted accesses in this system. In fact, such approaches cannot be easily reused when considering new models
or even variants of these models. Indeed, although many access control policies can now be found in the literature, their description often suffer of a lack of precision: formal and mathematical specification are rare. Furthermore, these policies are not described within a common framework and it is rather difficult to extract from these developments some methodological guidelines. Unfortunately, the framework of the “algebra of security” is not enough expressive to achieve and we have defined and implemented within Focal a specification of access control policies at a deeper level.

Hence, we have addressed the question of “what is an access control model?” regardless of any specific context, by defining a general semantical framework allowing to specific and to define access control models. Such a framework provides several levels of specification. First, the considered information system can be described by specifying the security parameters and by defining how to represent states of this system (what is the security information describing a state of the system). Then an access control policy is defined as a means to permit or deny a subject (users, processes, ...) the use of an object (files, processes, ...). This can be done by introducing a predicate allowing to characterise secure states of the system (which are the states satisfying the policy). The next step consists in specifying the syntax and the semantics of a language of requests allowing the system to move from one state to another state. The specification of a policy and a language of requests leads to the notion of access control model. Lastly, our framework allows to describe what are the implementations of a model and what security properties these implementations have to satisfy (the main properties are concerned with the policy and the semantics of requests). Furthermore, since several implementations can be defined for an access control model, we introduce a way to compare such implementations.

Although there exist some papers focusing on comparisons and translations between policies, these developments are not expressed within a common framework. Hence, it is rather difficult to compose (or to compare) these translations.
Hence, we have also addressed the question of “how to compare two access control models?”. Our approach, based on the notion of simulation of implementations, provides some formal tools to express reusing of implementations and some abstract notions to help the comparison of models.

To summarise, our framework, described in [13–15,20], has been used to formalise, to implement within Focal and to compare the BLP model [20], the Chinese Wall model [20], the RBAC (role based access control) model [10], and discretionary policies (HRU model, trust management model, unforgeable tickets model) [2].

4 Conclusion

This paper is mostly devoted to Focal which was conceived from the beginning to help constructions of systems highly concerned with safety and security. Its development is based on strong theories such as type theories, denotational and operational semantics, rewriting. But, our methodology for Focal development is to incorporate only parts of these theories which are mandatory for our purposes. For example, the Focal language is indeed a dependent type language but some dependencies available for example in the proof assistant Coq are not offered: for instance, a function cannot depend of a proof. If such a possibility is retained, then treatment of partiality of functions can perhaps be improved but we would have to manage possible logical clashes when redefining a function. It seems to us that this management would be too difficult for engineers and we reject this possibility.

The claim behind Focal is that formal developments tend to increase the confidence in the final code. One of the main characteristics of critical software is that it is subject to the approval of a safety/security authority before its commissioning. These authorities have defined requirements explaining what should be an acceptable software and its related life cycle process for their own domain. For this reason, getting a high confidence in produced code, and making possible for the safety/security authority to acquire this confidence is an important task, for which Focal brings solutions. Very important is the possibility in Focal to have one unique language for specification, implementation and proofs, since it eliminates the errors introduced between each layer, each switch between languages, during the development cycle.

Other frameworks like Atelier B[1] also aims to implement tools for making formal development a reality. Focal doesn’t follow the same path, trying to keep the mean of expression close to what engineers usually know: something close to a programming language. Moreover, instead of having its own system for proofs validation, Focal makes use of external tools, leaving the task of handling proof automation and verification outside its scope and keeping benefits from researches performed aside in these specific domains.

Acknowledgements This work was partially supported by the french SSURF ANR project ANR-06-SETI-016.
References