Trusted Software within Focal
Philippe Ayrault, Matthieu Carlier, David Delahaye, Catherine Dubois, Damien Doligez, Lionel Habib, Thérèse Hardin, Mathieu Jaume, Charles Morisset, François Pessaux, et al.

To cite this version:
Philippe Ayrault, Matthieu Carlier, David Delahaye, Catherine Dubois, Damien Doligez, et al.. Trusted Software within Focal. C
ESAR 2008 - Computer
Electronics Security Applications Rendez-vous, Dec 2008, Rennes, France. pp.162-179. hal-01125667v2

HAL Id: hal-01125667
https://hal.archives-ouvertes.fr/hal-01125667v2
Submitted on 11 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 mechanism 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 [24, 9] (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 [27], 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 general 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 formalise airport security regulations [7].
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) [23].

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 theses 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 compatible 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 (soundness, 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:

```
all v in Speed, v <> Speed!zero -> ¬ 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 [10] 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”).

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

The \textit{interface} of a species is obtained by abstracting the \texttt{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 \textit{interfaces} in the concrete syntax, they are simply denoted by the name of the species underlying them. Interfaces can be ordered by inclusion, a point providing a very simple notion of subtyping.

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

When \textit{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 \texttt{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 \textit{species}, is a progress to a complete implementation. On the previous example, forgetting \texttt{different}, we typically presented a kind of \textit{species} for “specification” since it expressed only \textit{signatures} of functions to be later implemented and properties to which, later, give \textit{proofs}.

We can now create a new \textit{species}, may be more complex, by \textit{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 \textit{signatures} and \textit{proofs} to \textit{properties}, adding no new \textit{method}.

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

\begin{verbatim}
species Monoid inherits Setoid =
  signature ( * ) : Self -> Self -> Self ;
\end{verbatim}
signature one : Self ;
let element = one * one ;
end ;;

We see here that we added new methods but also gave a definition to element, saying it is the application of the method \( \ast \) 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 Parameterisation

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 parameterised by its two components. Following this idea, Focal allows two flavors of parameterisation.

**Parameterisation 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 an hypothesis.

Calling a species’s method is done via the “bang” notation: \(!meth\) 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 parameter’s method, the same notation is used: \( A!element\) stands for the method 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.
We express the carrier of the product of two setoids as the Cartesian product of the carriers of the two parameters. In \( A \times B \), \( \times \) 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 parameterised 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 parameterisation to introduce dependencies on the parameters’ carrier definitions. For example, it is impossible to express “if \( A!rep \text{ is } \text{int} \) and \( B!rep \text{ is } \text{bool} \) then \( A*B \text{ is a list of boolean values} \)”. This would dramatically restrict possibilities to instantiate parameters since assumptions on the carrier’s structure, possibly used in the parameterised 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 parameterisation too weak to be usable. We choose to always hide the carrier of a collection parameter to the parameterised hosting species. Hence the introduction of the notion of collection, obtained by abstracting the carrier from a complete species.

**Parameterisation by Entity Parameters** Let us 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 us first have a species representing natural numbers:

```plaintext
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:

```plaintext
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:

```plaintext
collection MachineNativeIntColl implements MachineNativeInt ;;
```

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

```plaintext
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)
it provides. This point is especially important since it prevents other software components from possibly breaking invariants required by the internals of the collection.

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 [5, 8] 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))
    by hypothesis H1, H2
<2>1 prove !equal(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”.

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 cannot 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 track 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 [25], 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** Focal incorporates a tool named FocalTest [20] for Integration/Validation testing. It allows to confront automatically a property of the specification with an implementation. It generates automatically test cases, executes them and produces a test report as an XML document. The property under test is used to generate the test cases, it also serves as an oracle. When a test case fails, it means a counterexample of the property has been found: the implantation does not match the property; it can also indicate an error in the specification.

The tool FocalTest automatically produces the test environment and the drivers to conduct the tests. We benefit from the inheritance mechanism to isolate the testing harness from the components written by the programmer.

The testable properties are required to be broken down into a precondition and a conclusion, both executable. FocalTest proposes a pure random test cases generation: it generates test cases until the precondition is satisfied, the verdict of the test case is given by executing the postcondition. It can be an expensive process for some kind of preconditions. To overcome this drawback, a constraint based generation is under development: it allows to produce directly test cases for which the precondition is satisfied.

**Documentation** The tool called FOCDOC [19] 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 and thanks to some XSLT stylesheets, it is possible to generate HTML files or LATEX files. Although this documentation is not the complete safety case, it can helpfully contribute
to its elaboration. In the same way, it is possible to produce UML models [6] as means to provide a graphical documentation for Focal specifications. The use of graphical notations appears quite useful when interacting with end-users, as these tend to be more intuitive and are easier to grasp than their formal (or textual) counterparts. This transformation is based on a formal schema and captures every aspect of the Focal language, so that it has been possible to prove the soundness of this transformation (semantic preservation).

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 the library of access control models developed within Focal.

Several approaches have been followed in order to build a formal library of access control models. First, in [11], we have formalised within Coq [26] the Bell & LaPadula (BLP) model by following the original paper [18] : definition of the policy, of the transition function between states of the system and formal proof of the “Basic Security Theorem” [18] stating that the transition function preserves the policy. 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. Hence, in a second time, we have used Focal to implement the algebra of security model introduced by J.McLean in [21]. 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 [13, 14]. 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 1). 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. All this work is described in [3, 4].

Unfortunately, the framework of the “algebra of security” is not enough expressive for some access control models 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 specify 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 transition functions (definitions of reference monitors) of a model and what security properties these transition functions have to satisfy (the main properties are concerned with the policy and the semantics of requests). Furthermore, since several transition functions can be defined for an access control model, we introduce a way to compare such transition functions.

As a usage of Focal, we are developing an access control library based on the framework previously described. The architecture of the framework’s implementation within Focal is shown in figure 2. As we consider the systems on which we apply access control as state transition systems (reference monitors are tran-
sition functions), we see that the species `states` is the central species of the implementation. Indeed, it is parameterised by the basic components of an access control system (subjects, objects, security parameter, ...), and is inherited by the species defining the access control policy (`policies`), the semantics of requests (`semantics_requests`) and the transition function (`models`). We also see how we use the framework’s implementation in order to implement a particular access control model, this is done by refining the framework’s species thanks to inheritance. Those species defining an access control model are the ones ending with an `*`, where `*` can be replaced by the name of the access control model implemented. For instance, the library currently offers the Bell & LaPadula, RBAC, HRU, Unix-like and Unforgeable Tickets (capabilities based) models. Hence, this architecture shows some of the strength of Focal which are parameterisation and refinement through inheritance that allow us to follow a quite complex formalisation and to implement access control models by first considering what is common to every models and then focusing on what is particular to a specific one (this way of developing also ease code reusing).

Another major feature of Focal is to allow to write properties and to prove them in order to show the correctness of an implementation. This has currently be done in the access control library for the RBAC and HRU models (our aim is to certify all the implemented models), and we show here an example of such a theorem concerning the correctness of the RBAC transition function. Indeed, the following theorem `tau_rbac_acc_secure` states the correctness of the RBAC transition function according to the RBAC security predicate (`omega`). For the sake of simplicity, we only consider here the requests allowing to ask to get or to release an access (the correctness proof taking into account the administrative requests of the RBAC model has also be done).

```plaintext
theorem tau_rbac_acc_secure :  
  all st1 st2 in Self , all r1 in Req , all d1 in Dec ,  
  tau_rbac_acc(r1 , st1) = (d1 , st2) ->  
  omega(st1) ->  
  omega(st2)
```

We now show the architecture of its proof. The first step consists to consider the hypothesis needed to do the proof and to express what we want to prove, as we would do in any mathematical proof.

```plaintext
proof :  
<br>1>1 assume st1 in Self ,  
assume st2 in Self ,  
assume r1 in Req ,  
assume d1 in Dec ,  
assume H1 : tau_rbac_acc(r1 , st1) = (d1 , st2) ,  
assume H2 : omega(st1) ,  
prove omega(st2)
```

Then, mainly, we need to consider two cases corresponding to the two possible requests, "get an access" (`is_get`) or "release an access" (`is_rel`) and prove our goal. Finally, we are able to conclude this proof thanks to the property `get_or_rel` stating that the requests we consider are either "get an access" or "release an access" requests.
Fig. 2. Architecture of the framework’s and access control models’ implementation
To summarize, our framework, described in [15–17, 22], has been used to formalise, to implement within Focal and to compare the BLP model [22], the Chinese Wall model [22], the RBAC (role based access control) model [12], 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.

Acknowledgments This work was partially supported by the French SSURF ANR project ANR-06-SETI-016 (Safety and Security Under Focal). Many thanks to James Geater for his careful proofreading.

References


