Development of a Generic Voter under FoCaL*

Philippe Ayrault[1], Thérèse Hardin[2], and François Pessaux[2]

[1] Etersafe, 43, Allée du pont des beaunes F-91120 Palaiseau
philippe.ayrault@etersafe.com
[2] Semantics, Proofs and Implementation, Laboratoire Informatique de Paris 6
Pierre & Marie Curie University, F-75005 Paris
therese.hardin|francois.pessaux@lip6.fr

Abstract. Safety and security are claimed major concerns by the formal FoCaL development environment. In [7] we introduced a safety development cycle customised to FoCaL. In this paper, we examine how to specify and implement a concrete example following this cycle. We show that indeed it is feasible and we present how FoCaL features fit with software best practises like modularity, reuse, fault confinement and maintenance.

Key words: formal methods, development cycle, voter, FoCaL

1 Introduction

Development of safety related systems has to be strictly compliant with applicable standards. This is required by any safety authority before commissioning of a system. Safety demonstration is greatly helped by using a formally based framework to express requirements, design, code and to ensure that this code meets specification requirements. Unfortunately, this is not sufficient. Safety authorities also require an independent verification process which must follow the different stages of a strict development cycle. Thus, any tool claiming to be dedicated to formal developments should not only provide a formal paradigm to model and prove the system but must also strongly support a well defined development cycle and must produce adequate documentation at each stage of the development. Moreover, such a tool should help to identify impacts of modifications during the whole life of the system (frequently more than 20 years). The purpose of the FoCaL tool is to bring some answers to this problem area. FoCaL provides a unified language to express requirements (declaration and properties) as well as source code and proofs while only using concepts largely accessible to engineers.

In [7], we introduced a safety development cycle customised for FoCaL. Our problem now is to study its feasibility and to understand how practically FoCaL can help to answer safety and good practise requirements. We have chosen to fully treat the example of a voter. Indeed a voter is a central equipment of all redundant architectures, widely used for safety related systems. A voter is a process or a device whereby a number of similar signals are monitored for discrepancies and are voted upon to obtain the

* This work is supported in part by the *Agence Nationale de la Recherche* under grant ANR-06-SETI-016 for the *SSURF* Project (Safety and Security UndeR FoCaL).

selected output which is most probably correct. It is highly safety critical because in many cases, the voter is the last barrier able to eliminate failure effects and answers a part of the safety principles underlining the whole system. As these ones strongly depend of the aim and domain of the system, for the re-usability sake, a voter should be conceived as a very generic equipment. Thus choosing a voter seems a good balance between the exemplary nature of the development and the length of the paper.

The rest of the paper is organised as follows. We present the main features of the FoCaL tool in Section 2. Section 3 exposes the rules of a voter and its textual specification. The Section 4 comments the development of the voter using the FoCaL tool. We conclude and comment possible further works in Section 5.

2 The FoCaL environment

We give here an informal presentation of near all FoCaL features, to help further reading of this paper. For more details and the new release [3].

2.1 The Basic Brick

The primitive entity of a FoCaL development is the *species*. Like in most modular systems (i.e. objected oriented, algebraic abstract types), it can be viewed as a record grouping a data structure with its related operations. Since FoCaL does not only address data type and operations, species may contain the declarations (specifications) of these operations, some properties (which may represent requirements) and their proofs. All these components of a species are called *methods* and we briefly describe them.

- The *method* introduced by the keyword `representation` gives the data representation of entities manipulated by the *species*. It is defined by a type expression, which is roughly a ML-like pure type (with restricted polymorphism). The *representation* may be not-yet-defined in a *species*, meaning that the real structure of the data-type the *species* embeds does not need to be known at this point. However, to obtain an implementation, the *representation* has to be defined later either explicitly or by inheritance.
- Declarations (keyword `signature` followed by a name and a type) introduce *methods* to be defined later: they only specify types without implementation yet. Declarations serve to express specifications, properties. Thanks to late-binding, as soon as a name is declared, it can be used in definitions.
- Definitions (keyword `let`, followed by a name, a type and an expression) introduce constants or functions, i.e. computational operations. The expressions are roughly pure ML-like expressions with an auxiliary construction (`S!m`) to call the *method* `m` from a given *species* `S`.
- Statements (keyword `property` followed by a name and a first-order formula) may serve to express requirements (i.e. facts that the system must hold to conform to the Statement of Work) and then can be viewed as a specification purpose *method*, like *signature*s were for `let`-*method*s. They entail a proof obligation later in the development. Like *signatures*, even if no proof is yet given, the name of the *property* can be used to prove other theorems or lemmas.

[3] `see : http://focalize.inria.fr`

– 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*. This proof will be processed by FoCaL and ultimately checked with the theorem prover Coq.

In addition, FoCaL provides a powerful mechanism for documentation by allowing special kind of commentaries (called annotations) kept along the compiler process.

2.2 Type of Species, Interfaces and Collections

The *type* of a *species* is obtained by removing definitions and proofs. Thus, it is a kind of record type, made of all the method types of the species. If the `representation` is still a type variable say α, then the *species* type is prefixed with an existential binder $\exists\alpha$. This binder will be eliminated as soon as the `representation` will be instantiated (defined) and must be eliminated to obtain executable code. Species types remain totally implicit to users and serve only to introduce interfaces.

The *interface* of a species is obtained by abstracting the *representation* type in the *species type* and this abstraction, which hides the representation, is permanent. Interfaces play an important role. They are simply denoted by the species name. Interfaces can be ordered by inclusion, a point providing a very simple notion of sub-typing.

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 representation to create a *collection*. Thus the *interface* of the collection is just the *interface* of the complete species underlying it. A collection can hence be seen as an abstract data-type, only usable through the methods of its interface, but with the guarantee that all methods/theorems are defined/proved.

2.3 Combining Bricks

A FoCaL development is organised as a hierarchy which may have several roots. Usually the upper levels of the hierarchy are built during the specification stage while the lower ones correspond to implementation. Each node of the hierarchy, i.e. each *species* (or *collection* as terminal ends), is a progress to a complete implementation. There are two ways to build new species from previously built species: inheritance and parametrisation.

In FoCaL **inheritance** serves two kinds of evolutions, which can be freely mixed. One may create a new *species* by extending the inherited ones with new operations and properties while keeping those of the inherited ones (or redefining some of them). One may create a new *species* by giving explicit definitions to *signatures* and *proofs* to *properties* of the inherited species, to be closer to a “executable” implementation.

Multiple inheritance is available in FoCaL. 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.

A strong constraint in inheritance is that the type of inherited, and/or redefined *methods* must not change. This is required to ensure logical consistency of the FoCaL model.

FoCaL allows two flavors of parametrisation: parametrisation by *collection parameters* and parametrisation by *entity parameters*. For instance a pair is a structure which is built upon its two components and is described by a *species* parametrised by its two components. *Entity parameters* are not introduced as we do not use them within this paper.

The parametrised *species* can use *collection parameters' 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. Note that species (without abstraction) are not allowed as parameters. Indeed, if an incomplete species were used as an effective parameter run-time error due to linkage of libraries can occur and properties stated in I can not be safely used as an hypothesis. In contrast, the collection and late-binding mechanisms ensure that all methods appearing in I are indeed implemented in CE.

2.4 The Final Brick

As briefly introduced, a *species* needs to be *complete* to lead to executable code for its functions and checkable proofs for its theorems and then, can be turned into a *collection*. Hence, a *collection* represents the final stage of the inheritance tree of a *species* and leads to an effective data representation with executable functions processing it. As said before, to ensure modularity and abstraction, the *representation* 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*) its interface provides. This point is especially important since it prevents other software components from possibly breaking invariants required by the internals of the *collection*.

2.5 Properties, Theorems and Proofs

FoCaL intends to encompass both the executable model (i.e. program) and properties this model must satisfy. For this reason, *theorems*, *properties* and *proofs* are *methods* dealing with logical instead of purely behavioural aspects of the system. Stating a *property* entails that a *proof* of it will be finally built. For *theorems*, the *proof* is directly embedded in the *theorem*. The compilation process submits proofs to the formal proof assistant Coq which automatically checks that they are consistent.

No syntax is offered to express high-order properties as they are rather difficult to manage by engineers not experts in logic theory. But special instances for induction and termination proofs are to be provided.

Proofs are done by the developer as follows. It can be written in "FoCaLProof Language", a hierarchical proof language that allows to give hints and directions for a proof. This script is submitted to an external theorem prover, Zenon[4] developed by D. Doligez. Zenon is a first order theorem prover based on the Tableaux method incorporating implementation novelties such as sharing[4]. From these hints Zenon attempts

[4] `see : http://focal.inria.fr/zenon/`

to automatically generate a proof and if it succeeds, expresses its proof as a Coq term verified by Coq during the compilation process. 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 statement 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) which can be used to prove lemmas or parts for the whole property. We show below an example of such a 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 !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>f qed
        by step <2>1
           property inf_is_associative, equal_transitive
           definition of order_inf
    <1>f conclude
  ;
```

Like any automatic theorem prover, Zenon may fail finding a demonstration. In this case, FoCaL allows to write verbatim Coq proofs. The compiler provides a Coq script "with a hole" which can be filled with the proof done by hand, then imported back to the FoCaL source code as verbatim Coq code.

Finally, the `assumed` keyword is the ultimate proof backdoor, telling that no proof is given thus the property is considered as an axiom. Obviously, a really safe development should not make usage of such "proofs" since they bypass the formal verification of software's model and can break the global consistency. However, a development may use external (trusted or not) code no property of which can be proved. Moreover, whatever the reason, the user may choose to admit some properties. But any "assumed" lemma should always be at least receive a textual justification inside an annotation. A good practise is to submit such lemma to the FoCaL test tool to increase confidence.

2.6 Around the Language

All along the development cycle of a FoCaL program, the compiler keeps track of dependencies between *species*, their *methods*, the *proofs*, ... to ensure any modification of component will be detected and its impact will be reported on those depending of it.

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 express 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 state 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 on the definition are

then erased by the compiler and must be done again in the context updated with the new definition. Hence an important point is the choice of the most interesting level in the hierarchy where to write a proof.

FoCaL currently supports two target languages: OCaml[10] and Coq[11]. Code generation towards OCaml allows to build an executable: all the logical aspects are discarded since they can't be expressed in this language and don't lead to executable code. Code generation towards Coq provides a formal model of the program, including computational and logical aspects: all computational *methods* and logical *methods* with their proofs are compiled. Thus the consistence of the whole FoCaL development can be checked by Coq.

However the compilation model (i.e. the structure of a collection, of a species, of a method) remains very simple. It is the same in both target languages and uses a few set of basic constructs: records (i.e. structures), functions and simple modules (not even functors).

Note that references are not currently allowed as it is not so easy to handle memory management at the proof level. However functional views of memory are used in several developments. We currently consider to add data-flow programming features and their logical counterparts to ease reactive systems development.

The tool called FOCDOC [3] automatically generates documentation, ensuring that the documentation of a component is always consistent 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 style-sheets, it is possible to generate HTML or LaTeX files. In the same way, it is possible to produce UML models [5] as means to provide a graphical documentation (class diagrams) for FoCaL specifications. The use of graphical notations appears quite useful when interacting with end-users, as these tend to be more intuitive and easier to grasp than their formal (or textual) counterparts. This transformation is based on a formal scheme and captures every aspect of the FoCaL language, so that it has been possible to prove the soundness of this transformation (semantic preservation).

Although this documentation is not the complete safety case, it can helpfully contribute to its elaboration.

As a conclusion of this presentation, FoCaL is not at all a "All-in-One" language, it helps to define strict boundaries between phases of a development. Consistency between phases is ensured by a powerful dependency calculus and a proof system. FoCaL user is guided to maintain the global consistency of a complete development, even during the maintenance phase.

There exist several systems and languages having the same purposes like B[16], CASL[13], RAISE[15]... True comparison with these systems is out of the scope of this paper (but is considered in a forthcoming paper). It seems to us that the originality of FoCaL is its unified language along the different phases which allows to provide the assessor a complete package built in conformity with the related standards.

Differently from CASL[13] that is strongly oriented towards ADT, hence without effective representation of data and computation, FoCaL lets the user go until speci-

fying them (and adds an abstraction layer to prevent users from having access to internal effective implementation). FoCaL produces executable code and related proofs whereas CASL emphasises on the specification phase despite the fact that institutions are provided to link specifications to some SML or Java code[14]. RAISE[15] provides a large range of specification and programming concepts. In contrast, FoCaL offers a limited set of concepts but they can be altogether translated to Coq to ensure a global consistency of the whole development.

2.7 FoCaL development cycle

As recalled in the introduction, the development of a critical system must follow a strict development cycle, compliant with standards. In [7] we proposed a development life cycle taking advantages of all features of FoCaL and compliant with the main Standards in the field of critical software development ([1], [2],...). It is based on a V-cycle, decomposed into five (mandatory) phases: requirements/specification, architecture/design, implementation and low level testing, integration/validation testing and the longest one, the maintenance phase. It also covers some transverse processes like generation of the documentation or formal traceability between phases. Transverse processes are processes that should be applied once during all phases. The main characteristics of this life cycle are :

- a strong boundary between phases, especially between the specification phase and the architecture/design one. Specification phase ends when all safety requirements can be proved using the functional requirements and the glue assumptions (see below). Architecture/design phase ends with the implementation of the functional requirements into executable code.
- the implementation phase consists in assembling collections and proving the glue assumptions made on the parametrised species.
- the use of the FoCaL dependencies calculus and documentation generation to generate formal traceability and to help maintenance.

In this paper, we mainly focus on requirements specification, architecture/design and implementation phases as we build the voter. Integration/validation testing phase is considered in the work of M. Carlier [6].

The development cycle is strongly based on the establishment of the specification through requirements expression. We distinguish four kinds of requirements:

- *Functional requirements* describe the relations between inputs and outputs of the system and what is expected on the behaviour of the system without referring to any specific solution.
- *Non-functional requirements* describe all constraints that the system must meet, like time and space bounds, safety integrity levels to achieve, portability needs... These requirements are pretty difficult to express by a first-order formula. They are put inside annotations, so are kept along the compilation process and figure in the compiled documentation.
- *Safety requirements* coming from the results of the safety studies. They ensure that the functional requirements will never trigger a Feared Event. They are considered as requirements on functional requirements.

– As a species can be parametrised, proofs of functional and/or safety requirements sometimes need assumptions on the functions and properties of the collection parameters. These assumptions are called *glue assumptions*. They are proved at the coding level just before final building of the whole system.

All the requirements but some non-functional ones are expressed as FoCaL properties.

3 Overview of the voter

3.1 Generic definition

Sensors may exhibit various kinds of errors like bias offset, scale factor or transient faults due to sensitivity to spurious or environmental factors (temperature, pressure,...). Redundancy is one of the major techniques used to guard safety critical systems against such transient faults. There exists many kinds of redundancy, depending on which characteristics (safety, reliability or both) should be privileged for the system. Roughly speaking, each redundant component performs the same work and, when one of them fails, the voter has to detect it and to select an output value among the other, then has to go on providing the service.

Usually, a *voter* is used to elaborate the output from the input values given by the redundant components. Voters are used, for example, for temperature acquisition by multiple sensors in a boiler, or elaboration of the emergency brake signal of a train from several computer replicas... The basic principle of a voter is to compare its input values according to a given consistency relation, and then to output one value depending on a voting policy. The point is that, in redundant systems, the voter is *the* component that must be perfect (as far as possible obviously). A failure of the voter is considered as a major weakness of the system.

A voter system must fulfil three main requirements:

– reliable and correct choice of one non faulty input among its n inputs [5]
– detection of errors on inputs
– localisation of the source of the error and report of a diagnosis related to it[6]

The elaboration of the output value follows a two-stages process:

1. the *inputs comparison*, which takes 2 or more inputs and compares them according to a "consistency law". There are many kinds of such consistency laws: strict equality, equality within a certain tolerance, most recent input, max or min values ...
2. the *arbitration*, by a voting policy algorithm which produces the output value. This algorithm is the heart of a voter and determines its classification and its main properties (majority vote, selection of the most restrictive vote, selection of the most recent value ...).

[5] We wilfully limit the voter specification to a function that returns one of its inputs. Other more complex voters can be found in [8].

[6] The third requirement is sometimes optional depending on which dependability characteristic is emphasised.

A pre-processing of the inputs can also be performed by a *filtering process*. It analyses input values by in-line "acceptance tests" and eliminates values recognised wrong, a way to eliminate some transient faults. The inputs that succeed the filtering process are then sent to the voter[7].

In the following, we focus on voters with 3 inputs. Our choice is motivated by the fact that 3 inputs is the minimal number for a voter to deal both with safety and reliability characteristics. Taking the case where agreement between 2 sensors is sufficient to ensure the required safety level of a system. Using a voter with 2 inputs meets the safety requirements. But, in case of one failure on a sensor, you need to stop or run your application in a degraded mode. Adding a third sensor permits to continue the service in case of one fault. Most of the time, the cost of the third sensor is far less than the unavailability of the whole system. This kind of voter is a widespread voter architecture in safety critical systems (i.e. Triple Modular Redundancy).

Our choice could be even more generic. It is indeed possible to specify an "n out of m" voter and then to instantiate n and m with the needed value. But this solution needs to highly complicate the voter specification and implementation for a small added value. Indeed only some n and m values are generally used; 1oo2 for reliability, 2oo2 for safety, 2oo3 or 2oo4 for safety and reliability and the concerns are so different that this "sharing by genericity" is just useless.

3.2 The 2oo3 voter specification

The 2oo3 voter, used for our example, selects one value from three independent inputs if at least two of them are consistent. Moreover, we also want to detect the faulty value. So, a second output is added to the voter in order to qualify the first result. Table 1 summarises all cases, described as follows:

- **perfect_match**: the three inputs are consistent, the value and index of one of them are returned.
- **partial_match**: two of the three inputs are consistent together, but the third one is not. One of the consistent values and the index of the inconsistent one are returned. This enables identifying a failure on this input.
- **range_match**: One input is consistent with the two others which are mutually inconsistent. The consistent value and the associated index are returned. This can arise when the consistency law is not transitive (i.e. equality within a tolerance). In this case, the system can go on working with the most plausible value.
- **no_match**: all the inputs are inconsistent two per two. The voter cannot take a decision since the majority rule is not applicable.

The specification of the **no_match** case seems, at first sight, satisfactory: no value is output as there is no good candidate. At the specification level, this behaviour is acceptable, but a choice has to be made during the design phase: the component connected to the voter is waiting for two values (the index of the component and the flag). It will be its own concern to decide what to do with the first output, according to the second.

[7] Filtering is different from the input comparison as it works on one single input in opposition to the consistency law which compare at least two inputs.

Consistency between inputs			Returned	Diag	
v1 and v2	v1 and v3	v2 and v3	Value	Index	Qualifier
Yes	Yes	Yes	v1	sensor_1	perfect_match
Yes	Yes	No	v1	sensor_1	partial_match
Yes	No	Yes	v2	sensor_2	partial_match
No	Yes	Yes	v2	sensor_3	partial_match
Yes	No	No	v1	sensor_3	range_match
No	Yes	No	v3	sensor_2	range_match
No	No	Yes	v2	sensor_1	range_match
No	No	No	?	?	no_match

Table 1. Transfer function

3.3 The 2oo3 properties

Functional requirements should describe the voting policy. Each line of the table 1 is transposed in a functional requirement. For example, for a perfect match, the requirement is:

$$\forall v1, v2, v3\ in\ value, consistency(v1, v2) \wedge$$
$$consistency(v1, v3) \wedge consistency(v2, v3)$$
$$\Rightarrow voter(v1, v2, v3) = (v1, 1, perfect\_match)$$

Table 1 makes also assumptions on symmetry and reflexivity of the consistency law[8]. This leads to define the following glue assumptions:

$$\forall v1, v2\ in\ value, consistency(v1, v2) \Rightarrow consistency(v2, v1)$$
and
$$\forall v\ in\ value, consistency(v, v)$$

A voter should also meet some safety requirements. Whatever is the order of the input values, the voter has to return a compatible output value. Thus a notion of "compatible output value" is introduced by properties P1 and P2. P1 says that if input values can be compared and output values are consistent and qualifiers are the same then the output values are compatible. P2 says that by default all output values are compatible for inconsistent input values (i.e. there is no choice made for inconsistent input values).

P1 : $\forall val_a, val_b\ in\ value, \forall qual\ in\ qualifier,$
$qual \neq no\_match \wedge consistency(val_a, val_b)$
$\Rightarrow compatible(val_a, val_b, qual, qual)$
P2 : $\forall val_a, val_b\ in\ value,$
$compatible(val_a, val_b, no\_match, no\_match)$
and
$\forall v1, v2, v3, val_a, val_b\ in\ value, qual_a, qual_b\ in\ qualifier$
$voter(v1, v2, v3) = (val_a, \_, qual_a) \Rightarrow$
$voter(v2, v1, v3) = (val_b, \_, qual_b) \Rightarrow$
$compatible(val_a, val_b, qual_a, qual_b)$

[8] Note that the voter does not need transitivity for the consistency law. Use of a transitive consistency law will remove the partial_match qualifier.

Another safety requirement allocated to a voter is that the output value is always one of the inputs.

$$\forall v1, v2, v3\ in\ value,$$
$$vote(v1, v2, v3) = v1 \vee vote(v1, v2, v3) = v2 \vee vote(v1, v2, v3) = v3$$

4 Development of the voter

4.1 Global architecture

The voter specification closely follows the description given in Section 3. Indeed separation of the voting policy and the inputs management eases reuse and independent evolution of parts of the voter. The whole specification of the voter is thus split into two major parts. The part concerning the voting policy is introduced by the species `Voter` (see fig. 1) which specifies the voting policy algorithm and the glue assumptions made on the inputs of the voter, represented by parameters. The second part, which concerns values, is itself split into two parts, the first one describing what is supposed/needed on the "basic" types (like naturals, integers, booleans) and functions on them, the second one (the species `Values`) specifying inputs as "complex values" (like integers with tolerance, integers modulo n) built upon basic types and ensuring that glue assumptions could be satisfied. Consistency between the two parts is guaranteed by the FoCaL dependency and proof mechanisms.

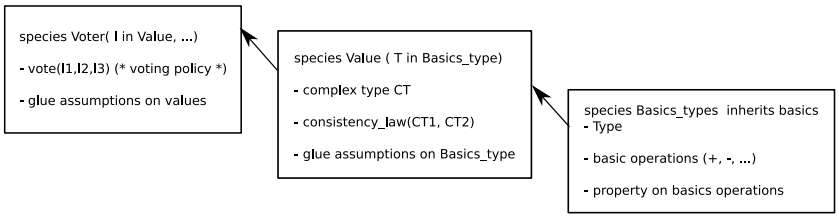

Fig. 1. Voter decomposition

4.2 The voting policy

Specification phase The aim is to specify the data-flow interfaces and the requirements for the system without referring to any specific solution. In FoCaL, data-flow interfaces are encoded by functions declarations and requirements by properties on these functions.

Specification of the voting policy is performed with a two levels approach. The first level corresponds to the specification of a most general voter. It gives the signatures and the generic properties of all 2oo3 voters, making no assumption on the voting policy.

The second level, derived from the first one, specifies the voting policy and gives the proof of the generic properties.

The specification of the generic 2oo3 voter is represented by a species "Voter_2oo3" with two parameters: a collection `V` for the values submitted to the vote, a collection `Diag` for diagnosis prescriptions. The species provides a single signature `vote`. It also gives the safety requirements allocated to the voter. In order to ease reading, we add a shortcut to extract the value from of output value.

```
species Gen_2oo3_voter( V is Value, Diag is Basics_object) =

signature vote in V -> V -> V -> (V * Diag);

(* Shortcut to extract the value *)
let output_value(p in V * Diag) in V = basics#fst(p);

(* Safety requirements of a voter *)
property voter_returns_an_input_value:
all v1 v2 v3 in V,
   output_value(vote(v1, v2, v3)) = v1
\/ output_value(vote(v1, v2, v3)) = v2
\/ output_value(vote(v1, v2, v3)) = v3
end;;
```

A species can be derived from this generic species for any choice of a voting policy. These derived species define the "voting policy", the glue properties required on the input values (through the collection parameter of interface `Value`) and give a proof of the generic properties. As an example, see the specification of a majority voter with identification of the faulty inputs as defined in section 3.

```
(* Specification of the diagnostics output *)
species My_diag(C is Sensor, Q is Qualifier) inherits Basics_object =
... end;;

(* Specification of the majority voter *)
species Majority_voter(V is Value, C is Sensor, Q is Qualifier, Diag
is My_diag(C, Q)) inherits
Gen_2oo3_voter(V, Diag) =

(* Functional requirements of the majority vote               *)
(* Vote with 3 consistent values returns a perfect_match and  *)
(* the value of the first sensor.                             *)
property perfect_vote :
  all v1 v2 v3 in V,
   (V!consistency_law(v1, v2) /\  V!consistency_law(v2, v3) /\
    V!consistency_law(v1, v3))
  ->
  (output_value(vote( v1, v2, v3)) = v1) /\
  (output_diag(vote( v1, v2, v3)) = Diag!constr(C!sensor_1, Q!perfect_match)))
...
(* Glue assumptions on parametrized species *)
property consistency_law_is_symmetric :
  all v1 v2 in V,
  V!consistency_law(v1, v2) -> V!consistency_law(v2, v1);
...
(* Proof of the safety requirements *)
proof of voter_returns_an_input_value =
...
end;;
```

To create the specification of the "majority voter", we use two main features of the FoCaL language. First, we use the FoCaL inheritance mechanism to create a species

having all declarations, definitions and properties defined in the species `Gen_2oo3_voter`. Then, we instantiate parameters of `Gen_2oo3_voter` with a more specific species (i.e. `My_diag`). Note that, the dependency calculus ensures consistence between the different occurrences of formal and effective parameters.

The new species contains the functional specification of the 2oo3 majority voter, the glue assumptions on parameters and the proof of safety properties under those assumptions (done with Zenon). We can compile the FoCaL model to an OCaml file and a Coq file. The first one contains only typed functions declarations that can be used as an external interface for a development. The second one contains a proof term of the safety properties that can already be checked by the Coq prover.

Architecture/design phase This phase introduces the architectural choices to answer the specification requirements. It provides definition of the functions and the representation of the species. From the majority voter specification, we have sufficient information to propose an implementation of the functional requirements and to provide a proof that this code fulfils them.

```
species Imp_Majority_voter(V is Value, C is Sensor, Q is Qualifier,
Diag is My_diag(C, Q)) inherits Majority_voteur(V, C, Q, Diag) =
(* Implementation of the vote function *)
let vote( v1 in V, v2 in V, v3 in V) in V * Diag =
  let c1 = V!consistency_law( v1, v2) in
  let c2 = V!consistency_law( v1, v3) in
  let c3 = V!consistency_law( v2, v3) in
  if c1 then
    if c2 then
      if c3 then
        (v1, Diag!constr(C!sensor_1, Q!perfect_match))
...
(* Proof of the vote property *)
proof of perfect_vote  =
  by property V!consistency_law_reflexive, Diag!equal_reflexive
     definition of vote, output_diag, output_value;
end;;
```

Transition between specification and design is also made using inheritance. Here parameters are kept while a definition is provided to the declaration, using the FoCaL functional language. Proofs of the functional properties are made using Zenon. Giving a FoCaL interface (and thus at least a specification species) for the input values, the sensor and the qualifier, this species can be compiled: type-checking can be done and a translation of the whole species contents (including Zenon proofs) into a Coq term can be obtained to be immediately assessed. Thus we have a species that implements a 2oo3 majority voter, which can be used with any kind of input values respecting the required glue properties.

Multiple inheritance can be used at design level to provide an existing representation to a specification species. For instance the implementation species `Imp_vote_status` is a "merge" of two species: the specification species `Sp_vote_status` which provides the functional requirements and the species `Integers` which provides the representation and its basic properties. The dependency calculus ensures compatibility between these inherited species.

```
(* Specification species for a set of vote status *)
species Sp_vote_status inherits Setoid =
```

```
  (** The 3 values are inconsistent *)
  signature no_match : Self;
  (** 1 value is consistent with the two other which are mutually inconsistent *)
  signature range_match : Self;
...
end;;

(* Design of the vote status *)
species Imp_vote_status inherits Sp_vote_status, Integers =

(* Definition of the set elements *)
let no_match = 0;;
...
end;;
```

Definitions can also be changed at any level to fit new needs. In this case, the FoCaL compiler computes all definitions and properties impacted by the redefinitions and asks to provide a new proof of the impacted properties.

4.3 Building a voter

At this stage, we have several species representing all components of the voter. In order to obtain runnable code a species should be transformed into a collection. Thus a voter collection has to be created from the voter species and collections representing the values and the diagnosis. Firstly, we choose the nature of input values (here the collection `Coll_int_with_tol`), finalise the diagnosis (here the collection `Coll_my_diag`). Secondly, we create a species which inherits from the implementation of the voter applied to effective parameters and we provide a proof for the glue assumptions. This step ensures "compatibility" between species. Then, as the new species is now complete, we can create a collection.

```
species Majority_voter_on_int_with_tol inherits
Imp_Majority_voter(Coll_value, Coll_sensor, Coll_int_with_tol, Coll_my_diag) =
  proof of consistency_law_is_symmetric =
    by property Coll_int_with_tol!consistency_law_symmetric;
...
end ;;

collection Coll_int_imp_vote_tol implements Sp_int_imp_vote_tol ;;
```

The collection `Coll_int_imp_vote_tol` is ready to use.

```
let s = Coll_int_imp_vote_tol!output_value(voter( 1, 3, 5);;

(* Results of several calls *)
Voter for integer with a tolerance of 2
v1 : 1, v2 : 3, v3 : 5 --> val : sensor_2 , res : partial_match
v1 : 1, v2 : 2, v3 : 5 --> val : sensor_3 , res : range_match
v1 : 4, v2 : 5, v3 : 5 --> val : sensor_1 , res : perfect_match
v1 : 1, v2 : 4, v3 : 7 --> val : sensor_1 , res : no_match
```

When a component is a COTS (Commercial Off The Shelf) or a very low level component as I/O drivers, one cannot produce a proof of its functional properties. In the same way, we do not want to prove well known or highly used components. In order to consolidate our confidence level of such components, works are currently performed by M. Carlier[6] to generate test cases from FoCaL functional requirements. These test cases can be ran on the external component to validate it. Verification of the voter using

M. Carlier tool has been achieved. It shows that the FoCaL life cycle we propose is fully compliant with the validation of external components.

4.4 Re-usability

Many other implementations can fit the generic voter. For example, S. Dajani-Brown proposes a 2oo3 voter for avionics purpose[9]. This voter is based on a classical 2oo3 architecture with an high availability (voter provides an output even when only one input is available). Following our development cycle, we implemented this voter within FoCaL by changing the voting policy and the value representation. Then we carried out successfully all the proof of the generic voter properties (voter always returns one of the input values, voter is insensitive to inputs order).

5 Conclusion and further works

This paper illustrates how a safety life cycle can be developed using FoCaL features like inheritance, parametrisation by collections, properties and proofs. This development process respects boundaries between development phases, produces a certified and efficient code, is compliant with standards and thus, eases the assessment process required before commissioning. Several other examples have been developed following the same methodology like hierarchical automata's, physical input acquisition... These developments give similar results on the development process.

In this paper, due to a lack of space, we only gave a short glimpse on FoCaL testing. However we used the testing tool (still in development), not only to do classical tests on outputs of the voter but also when encountering some difficulties in proofs, to validate expression of some requirements. We appreciated a lot to have at our disposal, at any stage of the development cycle, a proof tool and a test tool working on the same expressions. This is indeed a FoCaL feature which is of great help when expressing some requirements, when wondering about the validity of some lemmas, when some proofs are out of reach. In this last case, some of the corresponding statements are considered as axioms during proof process, a point which weaknesses the logical approach but is unavoidable in practice, thus increasing confidence by testing the statement is welcomed.

More generally, we emphasise the great facilities given by the coordination of FoCaL programming features (typing, inheritance, late binding, etc.) and FoCaL logical features (statements, proofs) through syntactical means controlled by dependency calculus. Indeed, having the possibility to introduce a statement using names not yet associated to a definition and to prove it under hypothesis submitted to a delayed proof obligation, allows to detect anomalies sooner in the life cycle and the diagnostic provided by the compiler helps to repair them. Yet it is possible to criticise the choice of a functional language for source code. Having no notion of internal state for species can be considered as a weakness. But first, there is no difficulty to translate, if needed, FoCaL definitions into some imperative language: the compiler uses only very basic features of programming languages (simple records and modules). Second, a lot of static analysers rebuild a functional version of imperative code to perform their analyses. In FoCaL this functional version is directly at hand, a point which will be exploited to integrate static

analysers in FoCaL, in the near future as the new compiler was conceived to easily include such extensions.

Using the theorem prover Coq as an assessor while having near all proofs quite automatically done with Zenon and having the choice of either doing the remaining ones directly in the Coq environment build by the FoCaL compiler or assuming their statement appears as a good compromise between the confidence level and the cost of the development. Yet the choice of Coq can be questioned. As the compiler does not use truly specific features of Coq, other theorem provers based on type theories and a flavor of Curry-Howard isomorphism can be chosen.

References

1. "*Railway Applications - Communications, Signalling and Processing Systems - Software for Railway Control and Protection Systems*", Standard Cenelec EN 50128, 1999
2. "*Functional safety of electrical/electronic/programmable electronic safety-related systems*", Standard IEC-61508, International Electrotechnical Commission, 1998
3. Manuel Maarek and Virgile Prevosto, "*FoCDoC: the documentation system of FoC*", in Proceedings of Calculemus, September 2003
4. Richard Bonichon, David Delahaye and Damien Doligez, *Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs*", LPAR, pages 151–165, 2007
5. D. Delahaye, J.F. Etienne and V. Donzeau-Gouge. "*Producing UML Models from Focal Specifications: An Application to Airport Security Regulations*" in 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering pp. 121-124, 2008
6. M. Carlier and C. Dubois, "*Functional Testing in the Focal environment*", in Test And Proof (TAP'2008), B. Beckert and R. Hahnle eds, LNCS 4966, pages 84–98, April 2008
7. P. Ayrault, T. Hardin, F. Pessaux, "*Development life cycle of critical software under FoCaL*" in TTSS'08, harnessing Theories for Tools Support in Software, Istanbul 2008
8. Paul R. Lorczack, et al, "*Theoretical Investigation of Generalized Voters for Redundant Systems*", Digest of Papers FTCS-19: The Nineteenth International Symposium on Fault-Tolerant Computing, 1989, pp. 444 - 451.
9. Samar Dajani-Brown, Darren Cofer, Amar Bouali, "*Formal verification of an avionics sensor voter using SCADE*", Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, 2004, pp. 5 - 20.
10. Xavier Leroy with Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon. "*The Objective Caml system, documentation and user manual. release 3.11*.", Documents include with the Objective Caml distribution, INRIA 11/2008
11. Yves Bertot, Pierre Castéran "*Coq'Art: The Calculus of Inductive Constructions*", Series: Texts in Theoretical Computer Science. An EATCS Series
12. Catherine Dubois, Thérèse Hardin and Véronique Viguié Donzeau-Gouge, "*Building a certified component within* FoCaL", Trends in Functional Programming Vol. 5, Intellect, 2006, pp. 33 - 48.
13. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P.D. Moss, D. Sannella, A. Tarlecki, "*CASL: the Common Algebraic Specification Language*", Theorical Computer Science 286, 2002, pp 153 - 196
14. D. Aspinall and D. Sannella, "*From specification to code in CASL*" AMAST, 2002
15. The RAISE Method Group, "*The RAISE Development Method*", Prentice Hall International, 1995
16. J.R. Abrial, "*The B-Book - Assigning Programs to meanings*", Cambridge University press, 1996.