Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Electronic Notes in Theoretical Computer Science 243 (2009) 15–31
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Development Life-cycle of Critical Software Under FoCaL [1]

Philippe Ayrault

Etersafe
43, Allée du pont des beaunes
F-91120 Palaiseau
philippe.ayrault@etersafe.com

Thérèse Hardin - François Pessaux

Semantics, Proofs and Implementation
Laboratoire Informatique de Paris 6
Pierre & Marie Curie University
104, Avenue du Président Kennedy
F-75016 Paris
therese.hardin|francois.pessaux@lip6.fr

Abstract

Before their installation, critical systems must be assessed by an independent authority, who ensures that software components are really compliant with a set of requirements described in standards. Such standards describe the framework and the rules to be strictly followed along the development process. Moreover high levels of safety highly recommand the use of formal methods.
In this paper, we examine how the FoCaL development environment can help to fulfil these requirements and to ease assessment. This tool aims to help all stages of critical software development, at least when formal methods are required (step-by-step specification and implementation, properties expressed by first-order formulae, proofs helped by automatic tool). Upon our experience as either software safety assessor or researchers in software engineering and formal methods, we propose a development life cycle adapted to the FoCaL specificity and compliant with independent assessment requirements, through a complete example. We show how features such as inheritance, late binding, redefinition, parametrisation, encapsulation and declarations/definitions, properties/theorems, whole development checked by an independent proof assistant and partially automatic documentation can be used to improve the global safety and the re-use of software components.

Keywords: formal methods, assessment, software life-cycle, FoCaL

1 Introduction

Software development process is usually presented as the cooperation between two major actors: the end user, whose describes the "what" and the software devel-

[1] 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).

doi:10.1016/j.entcs.2009.07.003

oper which answers the "how". The development of safety critical software needs to integrate a third participant, the governmental regulatory bodies (authorities). Their main task is to assert the compliance of each stage of the development with national and/or international Standards before commissioning. Most of them, such as the future DO-178C [2] [2] for avionics or Cenelec EN 50128 [1] for railway systems, are dedicated to a specific domain whereas some others, such as IEC-61508 [4] or Common Criteria for security [3], have a much more generic purpose. These standards, usually built upon a classical view of development cycle of products, provide requirements about activities to be performed, contents of outputs to be produced, verifications to carry out along each phase of this cycle and describe some mandatory support activities such as Change and Configuration Management.

For non vital software, Verification and Validation (V&V) is a rather informal process usually performed by the software engineer himself. But Critical systems cannot escape a full *assessment process* by a third-party expert. Indeed, standards prescribe the organisational independence level between the different teams involved in the development. The *assessor* has to evaluate the full development process from the system level to the embedded software. Note that, being legally liable in case of damages caused by the system, the assessor will refuse commissioning of the system under evaluation, if he/she has any doubt regarding the safety/security/reliability demonstration. For these reasons, developers of critical systems have the greatest interest in easing the assessment process of their products and particularly the verification of the life-cycle compliance with the prescriptions of the standards.

When high criticality or insurance levels must be reached, standards require the use of formal methods during the software development cycle. These methods usually offer a non-ambiguous language and some features to express and to reason upon properties. They can be proof or verification-based, but they can be usefully applied only if the system concepts, its purposes and its potential hazards are described clearly, precisely and as completely as possible.

The first author has 15-years experience in assessment of critical systems, either developed via formal or conventional methods, and is a "third party" expert in railway systems. The two next authors work on the theory, design and implementation of the FoCaL environment [3] (presented within this paper), which aims to ease development of safety critical software through a proof-based formal approach. The contents of the paper are issued from the assessment experience and some experiments using FoCaL features to answer needs of development and assessment of software parts of critical systems. We mainly focus in this paper on the software life-cycle.

As is commonly known, a software life-cycle is a sequence of steps describing how a development team specifies, designs, implements, tests, and maintains a piece of software. Each stage is described by its required inputs, performed activities and expected outputs, together with documentation, required properties, etc. There are

[2] DO-178B introduces formal methods in a "soft" tone, but DO-178C is discussing it more directly.
[3] homepage : http://focal.inria.fr

different presentations of software cycles (V-cycle, Waterfall, ...). Unfortunately, rather often in practise some phases, such as specification or maintenance, are not fully accomplished.

In the case of critical embedded software, the cycle is usually a V-cycle, mainly decomposed into five (mandatory) phases: requirements specification, architecture design, implementation and low level testing, integration/validation testing and the longest one, the maintenance phase. Moreover, standards ask for strict boundaries between phases together with traceability between phases, and the assessment process must state that these demands are fully achieved. Each software requirement must be linked to software components implementing it and to the set of tests validating this requirement. Similarly, each line of code should be linked to a need (software requirements or architecture design choices). Indeed many anomalies come from errors inserted during the transformation from one phase to its successor, especially during the transition from the specification to the design phase.

Usually each phase is treated with a dedicated formalism, which changes from one phase to another. Most of the tools helping traceability between the different phases are not powerful enough to formally ensure that unexpected behaviours have not been inserted when developing the next phase. One possible solution to this problem is to use the same language/frame along the whole software life-cycle and to choose it according to its formal capabilities to meet the high level safety/security requirements. There exist indeed powerful tools, like the different environments based on the B system, introduced by J-R Abrial [21] upon set theory, which are able to treat a large part of software cycle. The FoCaL environment, based on type theory, was in fact partly inspired by some work and discussions with B designers and developers. Yet B and FoCaL have strong differences (see [23] for some comparison). Building upon our common knowledge about B and type theory, and upon the strong first author experience of assessing software produced in a B environment, we explain in this paper how to handle, within FoCaL, most of the requirements mandated by standards upon the assessment of a software development life-cycle. Benefits of using a unique formal language are two fold: first, ease of traceability since the expression of the semantics of each phase does not need to be re-iterated. Second, verification between phases may be performed by mechanised proofs. But difficulties do not totally disappear. The main risk is to mix and overlap phases and therefore to have fuzzy boundaries between phases. We recall that only software developed following a well formed life-cycle can be easily assessed by a third-party.

The assessment of life-cycle depends also upon the produced documentation during each phase of life-cycle. These artefacts are also subject to strong requirements by standards concerning content, traceability and maintenance. The FoCaL tool provides some features to automate the generation of the software documentation (see section 3).

We illustrate our approach with the development of a voter, a sufficiently tiny example to comply with paper size limitations. However, safety device as such voters are used in safety critical systems as guard against transient faults.

Several industrial projects rest upon formal methods to build safety critical systems/software. Usually, the development life-cycle as well as the related requirements are treated according to the specific features of the used formal method. Then, arguments are provided to convince independent assessors of the compliance of the final product. Our approach departs from those ones as the development of FoCaL itself and the proposed methodology deeply embed these requirements. We reap the benefit of this approach with a better view on the relations between critical development cycle and FoCaL features.

The rest of the paper is organised as follows. We present the textual specification of the voter in section 2. We sketch the main characteristics of FoCaL in section 3. Section 4 presents assessment needs for a safety critical development and their instantiation in the FoCaL tool, written like a Software Quality Plan would be. We detail the modelisation of the voter in section 5. We finally conclude and present further works in section 6.

2 Overview of the voter

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,...), that is, transient faults. Redundancy is one of the major techniques used to guard safety critical systems against such transient faults. There exist many kinds of redundancies, 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 fails, the others detect it and 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 predefined rule. There are two parts in a voter:

- its consistency law, describing the comparison policy between input values (strict equality, equality within a certain tolerance...).
- its algorithm, describing the choice rules for the output values (majority vote, identification of the faulty input, most restrictive vote, most recent value...)

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.

The redundancy called "2 out of 3" (aka 2oo3) participates in obtaining a safe and reliable system based on 3 identical (or functionally identical) components connected to a majority voter. From voter results, the system can determine the actions to take (filtering failure, sending alarms, or system shutdown) in order to remain within safety conditions.

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 as follows:

- **perfect_match**: the three inputs are consistent, the index of one of them is returned (first one for example).
- **partial_match**: two of the three inputs are consistent together, but the third one is not. The index of the inconsistent one is returned. This enables identifying a failure on this input.
- **range_match**: One input is consistent with the two others which are mutually inconsistent. The index of the consistent value is returned. This can arise when the consistency law is not transitive (ie. 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.

3 The FoCaL environment

The FoCaL project was launched in 1998 by T. Hardin and R. Rioboo [7] with the objective of helping all stages of development of critical software within safety and security framework, at least when formal methods are required or chosen. The idea was to elaborate a development environment able to provide high-level and justified confidence to users. On the other hand, this system had to remain easy to use by well-trained engineers.

Currently, FoCaL can be seen as still a prototype of an Integrated Development Environment (IDE), for a language providing high level mechanisms such as inheritance, late binding, redefinition, parametrisation, etc. Confidence in proofs submitted by developers relies on formal proof verification. This support language was formally described and studied [6,5,22].

A FoCaL development is organised as a hierarchy that may have several roots. The upper levels of the hierarchy are built along the specification stage while the lower ones correspond to implementation. Each node of the hierarchy corresponds to a progress toward a complete implementation. We call here *refinement* the process of building a top-down hierarchy. The FoCaL refinement process has been formally studied in [6]. Its formal comparison with other notions of refinement such as those of B or TLA remains to be done.

Species

A node of the hierarchy is called a *species*. A species is a kind of record composed of fields, called *methods*, which may be:

- The method introduced by the keyword `rep`, exposing the data representation of values of the species. This type is called the *carrier type*. Available types are roughly: type variables, ML-like types (with restricted polymorphism) and species carrier types (i.e. the carrier of a species can depend on the carrier of other species). This method is mandatory and can be explicitly given or obtained by inheritance. It may be undefined (i.e. be just a type variable) along some inheritance stages but will have to be defined once and only once in the final inheritance branch (ie. collection).
- Declarations, introduced by the keyword `signature` followed by a name and a type (no computational body provided).
- Definitions introduced by the keyword `let` made of a name, a type and an expression. Mutually recursive definitions are introduced by `let rec`. The syntax of expressions is very close to those of usual ML family languages.
- Statements, introduced by the keyword `property` followed by a name and a first-order formula built with the usual connectors (`not`, `and`, $\forall \ldots$).
- Theorems, introduced by the keyword `theorem` followed by a name, a statement and a proof ultimately checked by the Coq proof assistant [14] (see below).

Statements, definitions and proofs can freely use names of other methods of the species (denoted by `Self!m` or shorter `!m`, or even shorter `m` if no local function named `m` is in the scope). The FoCaL compiler (see below) performs various analyses to ensure that dependencies between properties, definitions and proofs do not lead to cycles or logical inconsistencies.

A `let rec` definition is not considered as leading to a dependence cycle. Here is a strong difference with most usual object-oriented languages where methods are considered as mutually recursive without any restriction. Hence, the FoCaL "object" model is different since species are not classes, nor objects according to the usual meanings in the object oriented terminology. Species are not first class-values, only values living in carriers are so. Method types are type expressions *à la* ML. The expression `Self` in a type denotes the `rep` of the species and should be understood as `Self!rep`.

The *elements* of the species are called *entities*, to emphasise the fact that they are defined not only by their representation, but also by the functions manipulating them and their properties.

Inheritance

We say that a species A_2 "refines" a species A_1 if A_2 inherits from A_1 and "adds precision" to A_1. That is, the methods introduced in A_1 and/or the carrier type of A_1 are made more concrete (more defined) in A_2 (it is not the **B** refinement concept). In case A_2 multiply inherits from two species S_i having a method with

the same name, then the types of the two methods must be compatible (they must have a most general unifier [10]). If both methods are defined, the last inherited definition is kept (as in OCaml[9]) in A_2.

In addition, new methods can be added in A_2 and already given definitions can be redefined. However, once fully defined by a concrete type, `rep` cannot be redefined, for logical consistency reasons.

Interfaces

The *type of a species* is obtained by erasing definitions and proofs. If the carrier (`Self!rep`) is not defined, it will silently be considered as a type variable (e.g. α) and the species type will also be silently prefixed by an existential binder $\exists\alpha$. This binder will be eliminated as soon as the `rep` gets instantiated (defined) and must be eliminated to obtain runnable code. The *interface* of a species is obtained by abstracting the `rep` type in all the method types of the species type and this abstraction is permanent (see the paragraph Collections).

While species types remain totally implicit to users, interfaces are simply denoted by the species name. Interfaces can be ordered by inclusion, a point providing a very simple notion of sub-typing.

Collections

A species is said to be *complete* if all declarations have received definitions and all properties have received proofs. When complete, a species can be submitted to an abstraction process of its carrier to create a *collection*. Thus the interface of the collection is built out of the type of its underlying species. A collection can hence be seen as an abstract data type, only usable through the methods of its interface, but having the guarantee that all methods/theorems are defined/proved and that invariants used to build the underlying species cannot be broken by using this "component on the shelf".

Parametrised species

Species can be parametrised by collections. The formal parameter is introduced by a name C and an interface I. Any collection CC having an interface including I can be used as actual parameter for C.Methods and statements figuring in I are denoted by `C!m` in the species body (`C!rep` allows to use the rep of C as an abstract type). As any CC is issued from a complete species no "link error" can arise at run-time and properties of CC can be used as lemmas.

FoCaL allows very simple dependent types via parametrisation: a collection parameter being already introduced, an entity parameter denoting a value of the collection parameter carrier can be introduced. The syntax forbids dependence cycles between parameters and the compiler ensures consistency between the parameters.

Late-binding and redefinitions

As previously mentioned, methods may be only declared. This means that the operation is only equipped with a signature (`signature`) and properties (`property`)

but no effective implementation. However definitions of other methods may refer to such *only declared* methods. This is indeed perfectly safe as runnable code can be obtained only from collections. Thus these only declared methods will need to be *defined* later in the inheritance hierarchy in order to allow construction of collections. Now, late-binding brings the supplementary possibility to link together methods according to their latest definition. This way, it is possible to rely at any stage, while defining or proving properties of a method m, on a not yet implemented method p, which functional model is enough to work on m. Once a computational body is provided to p, the resolution mechanism will automatically link p and m's bodies. This mechanism also enables to redefine a method at any inheritance level, always keeping the latest defined one in the final collection extracted from the species.

On the proof side, if a proof π of m uses only the type of p, π remains valid when p is defined or redefined. If the definition of p is needed to achieve the proof π, clearly, π is no more valid after redefinition of p. The compiler invalidates all proofs depending on an "old" definition and prompts the user to redo the proofs. Thus, redefinition is fully safe.

Proof and automation

As properties must be proved, at latest before the collection level, a strong effort is done to have automated tools collaborating with FoCaL in order to get these proofs automatically done by theorems provers. FoCaL is designed to be open to whatever kind of such provers: it currently supports Zenon [17] which is an automated theorem prover developed by D. Doligez [18]. When Zenon succeeds, it produces a Coq proof term. Moreover, some experiments have been done with the rewriting based prover CiMe [15,16]. These external provers are dedicated to some specific kinds of proofs and may discharge the developer from making proofs by hand.

Compilation

Compilation of FoCaL sources leads, today, to both OCaml [9] and Coq codes. The generated OCaml code provides the executable form of the development. On the other side, the Coq code is produced from both the species source code, the proof terms of Zenon and the Coq proofs directly given by the user. Indeed, getting proofs is a quite satisfactory thing, but one must ensure these proofs are indeed correct. Obviously, no human being would stand reading pages of demonstrations to get convinced of this correctness! The Coq code will be checked by the Coq theorem prover who will act as assessor, not only on all the proofs contained in the development but also on the whole consistency of the model.

On the executable side, work is currently performed in the FoCaLteam to generate C code. Generation towards most of programming languages is possible since required features are mainly record data structures.

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 was proved in [5].

Documentation

Safety developments must contain various documentation items to be assessed. Creation of this documentation can be assisted by FoCaL by extraction of comments that can (and should) be embedded within the development (code) and from the code itself. As previously stated, these comments may describe non-functional requirements and will be available for traceability purpose. These comments are kept during the compilation process, linked to FoCaL entities forming the software, and can be automatically processed by (internal or external) tools processing the abstract syntax tree of the program. FoCaL's architecture allows to easily interface analyses/tools with the internals of the compiler to take advantage of the analyses it already performs. In combination with the properties and operations of the species (i.e. theorems and functions), it is possible to automatically generate (in the simplest case) a document (HTML for instance) describing the system. Moreover, UML diagrams can be automatically extracted from a Focal Model [20]. This permits to the developer to have a graphical view of its model and also facilitate the communication between the developer, the client and the assessor.

4 Software life-cycle within FoCaL

As said in the introduction, the life-cycle of critical software is based on a 5 steps V-cycle similar to the V-cycle of classical software development cycle. The main difference is that it is submitted to a traceability analysis by an independent verification team after each phase while another independent validation team performs the software testing. This implies that the boundary between each life-cycle phase must be clearly identified when a unique formalism is used along the whole development.

As presented in section 3, a FoCaL model is made of *species* which are used to describe all the phases of the cycle. Properties can be expressed as first-order formulae, a choice which is recognised as a good compromise between the expressiveness of the logic framework and its ease of use. Indeed FoCaL is intended to be used not only by computer researchers or mathematicians, but also by smart engineers with minimal background in logic.

Currently FoCaL provides no syntactical categories to distinguish between the different stages of a life-cycle and we are not sure that such distinctions will be beneficial for engineers as they could perhaps add too much rigour. Instead, we propose to provide *species templates* dedicated for each phase of the development life-cycle. Since each phase addresses a different view of the system, these templates will be defined by the kind and the forms of methods (declarations, definitions, properties, proofs) which should be used (or not) during the considered phase. They will help to identify clear boundaries between phases, which is mandatory for the verification process.

The following figure depicts the classical V life-cycle for software development.

In this paper, we consider only requirements specification, architecture design,

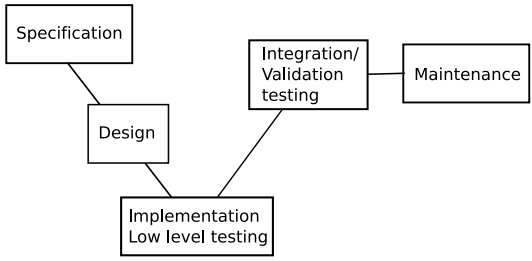

Fig. 1. The classical V life-cycle

coding and maintenance phases. The phase "Integration/validation testing" is considered in the work of C. Dubois and M. Carlier [12].

4.1 Specification phase

The aim is to specify the interface and the requirements for the software based on the Statement Of Work (SOW) delivered by the customer. In fact, the requirement phase is a way to re-express the customer needs with a software engineering view. Requirements are usually split into two parts:

- *Functional requirements* focus on describing the expected behaviour of the system/software without referring to any specific solution. These requirements describe the relations between inputs and outputs of the software.
- *Non-functional requirements* describe all constraints of the software, like time and space bounds, safety integrity levels to achieve, portability needs... These requirements are pretty difficult to express in term of software design models but are mandatory to issue an industrial system.
- For critical software, there is a third kind of requirements called *Safety requirements*. Safety requirements are coming from the results of the safety studies performed on the previous phases. They ensure that the functional requirements will never trigger a Feared Event. They can be considered as requirements on the two first kinds of requirements.

A species describing a component of a specification will follow the *species for specification* template. It is a species containing only specification requirements expressed as follows:

- A functional requirement is represented by a signature (name and type) and functional properties (described below).
- A safety requirement is also represented by properties on these signatures. However, their shape is slightly different from the functional properties since they do not express property on the function behaviour, but rather characteristics of the function.
- Non functional requirements which cannot be – easily – expressed by a first-order formula are put inside commentaries and, as they are kept along the compilation process, they can receive separate treatments (see section 3). This is a way to

ensure their traceability through the development.

- Proofs that the safety requirements are respected by the functional requirements, under the hypothesis that the functional properties hold.
- In addition, since a component can be parametrised by other ones, one must express the mandatory properties these other components must comply with. We call them *"glue" assumptions*.

Functional and safety requirements are both encoded as FoCaL properties. We distinguish between them by introducing a template for the functional ones as follows.

Since a functional property expresses the relation between inputs and outputs of the component, we propose the following template:

```
signature foo in t_1 -> t_2  -> ... -> t_n -> Self ;

(* Requirement 1 for foo. *)
property foo_1:
  all i_1 in t_1, i_2 in t_2,... i_n in t_n,
  (* Pre-condition on the signature. *)
  r1 (i_1, i_2, ..., i_n)
     ->
     (* Post-condition on the signature. *)
     r2 (i_1, i_2, ..., i_n, foo (i_1, i_2, ..., i_n)) ;
```

in which `r_1` and `r_2` are properties on signatures (local or imported by parametrisation) and "glue" assumptions.

At this stage of the life-cycle, proofs can be made that safety requirements are entailed by the functional requirements on functions and by the glue assumptions on the imported functions. Indeed, late-binding and collection mechanisms in FoCaL allow to already perform proofs of the safety requirements, without having definitions of functions declared in the species, nor proofs from imported components already done. The system ensures that such definitions will be given and proofs of their properties will be done before the creation of the collections defining the system. In the same way, the user may be sure that the components passed as actual parameters later in the life-cycle will have these proofs done, hence he may safely assume them at this stage.

The phase ending criterion is when all the safety requirements are proved. Usually no definition is introduced during the specification phase. However, some definitions of mathematical functions can help to express requirements: it is easier to rely on the definition of the absolute value function instead of expressing it only through its characteristic properties. We currently consider the possibility of adding a new syntactic category (logical let) to handle such definitions which may be useless in the following phases (for example, by in-lining of the absolute value function).

4.2 Architecture and design phase

This stage is dedicated to introduce the architectural choices to answer the specification requirements. It describes:

- the software breakdown into components,

- the behaviour of each component,
- the data exchanged between components ,
- the inter-relations between components (inheritance, parametrisation...),
- the scheduling of components.

For critical software, the design should follow *design rules*. These rules are guidelines to apply in order to achieve and ease the safety demonstration and description of all constraints coming from the hardware supporting the software (built-in-tests to perform, hardware performance and reliability...) or pre-existing software (Operating System, COTS...).

According to the presentation of this phase, we propose to give the following structure for a *species for design*:

- Breakdown of the software must answer 2 main issues: to propose efficient algorithms and to respect the functional requirements expressed in the previous phase. FoCaL's parametrisation allows acquisition of existing or future (i.e. not yet implemented) dedicated species offering such algorithms and inheritance can be used to keep traceability between the two phases.
- Definition of the behaviour via a progressive carrier choice and via the implementation of the signatures (possibly working on the carrier's structure). FoCaL authorises re-definition of methods previously implemented. This feature may be handy to specialise algorithms for a specific data representation of the carrier.
- Proofs of the functional requirements ensure that, through the breakdown steps and the introduction of definitions, the functional properties given in the *species for specification* are indeed fully implemented. This achieves the traceability between phases.

The refinement process between specification and design may span on several inheritance steps. Late-binding enables the use of a method not yet defined, and to be sure that once defined, the compiler will find and use it. These points are especially crucial to perform successive and incremental refinements. Hence, it is possible to incrementally implement some concepts, enunciate properties, make proofs, and all of them will be kept (inherited) from the parent(s), providing that the compiler didn't detect redefinitions breaking parent definitions. This last point ensures that at any inherited stage properties and implementations remain consistent even in case of redefinition. Moreover, following an incremental refinement, it is possible to derive by inheritance several species from a same parent. The parent may provide a default implementation of a method and each child will be free to redefine or not this implementation according to its own constraints.

Obviously, industrial projects rarely start from a blank page. It is usual to reuse external components (middleware, operating system primitives, COTS...). For this reason, FoCaL provides a way to make these external components available in the model. However, a safe development cannot on one hand simply assume that such components are safe, and on the other hand cannot prove their properties as these COSTS are usually black boxes. Our solution, in term of methodology,

is to verify by testing that these properties hold and then to assert the functional properties required on these components by issuing a proof reduced to the keyword `admitted`. This solution is certainly unsatisfactory for purists, but this is the only possible choice when using external COTS. The compiler inserts the fact that these properties have been `admitted` in the documentation. So the reader is warned about that and may review the test results [12].

Like any development environment, FoCaL provides a set of fully proved "standard" libraries on which developments can safely rely. It is worth mentioning that there is a library for symbolic computation on multivariate polynomials, on all ordered-based mathematical structures and a generic one on access control by which most of the usual access control policies received a FoCaL implementation.

The software breakdown is iterated until obtaining the proof of all the software items. A software item is defined as the smallest piece of software that can be compiled and tested alone. The phase ending criterion is when the proofs of the functional requirements are done based on the definition of the software components, on the properties of the reuse external components and on the "glue" assumptions.

4.3 Implementation phase

This stage aims to produce the source code that implements the components. For critical software, specific coding rules should be followed. These rules describe the *safe subset* of the language that may be used. Low-level testing is also performed during this stage. It permits verification that each software item is in line with its documentation and does not contain systematic bugs (table overflow, division by zero,...). At this stage, the software items are "clear", i.e. seen as *white boxes*.

We propose to give the following structure to a *species for implementation*:

- Final assembly of species. The aim is to produce a complete species, in which all the methods are implemented or externally linked (i.e. there are no more signatures without their related implementation).
- The proofs induced by the final assembly. They enable the user to ensure that the "glue" assumptions set out during the specification phase hold.

Once such a *complete species* is obtained, it is possible to turn it into a collection, which corresponds to an effective piece of software able to compute and give back results.

From this collection the FoCaL compiler generates target code of the software build. The phase ending criterion is obviously when the executable is produced. Although FoCaL relies upon object-oriented flavours for specification and design, it produces no object-oriented target code. This is very important because the produced code fulfils the safe language subset recommended by standards and is easily traceable to the model by independent assessors. Finally this permits FoCaL backends to most languages since it only relies on common and widespread programming language features (mostly modules and records).

4.4 Maintenance phase

This stage is dedicated to keep the consistency of the model despite any required or imposed evolutions (hardware obsolescence or changes, enhancement requests, new regulations from authorities or even bug fixes). The mainly used mechanisms are redefinition, inheritance and late-binding just at the needed level in the hierarchy defining the system. Standards require that each modification must be traced and its impact on the whole system has to be clearly identified. Because FoCaL provides highly efficient dependencies calculus [5], proofs to be redone are automatically exhibited and strictly limited to the performed modifications. Since software is formally developed using a well established life-cycle, impacts of modifications are under control.

4.5 Transverse processes

Transverse processes are tasks performed on every life-cycle phase. For a FoCaL development, there are two transverse processes:

- *proofs* to be performed at each phase. These proofs have 2 aims: correctness of the provided definition and correctness of the refinement between phases. The last point ensures the formal traceability between phases.
- the *generation of the documentation* associated to each phase.

Proof of a FoCaL model is assisted by theorem prover (see paragraph 'proof' on section 3). Of course, some proofs cannot be fully automated and remain in charge of the engineer. However, the experience proves that automated tools strongly help in effective FoCaL developments. FoCaL's standard library (more that 7500 lines of formal mathematics in FoCaL) is mostly proved thanks to Zenon!

FoCaL is strongly connected to the Coq proof assistant [14] since it acts as its final assessor. Coq is a well established system already used in the industrial and academic domains [13], which allows to put enough confidence in its verifications. The use of a formal demonstration checker acting as an assessor of the proofs avoids having to perform these verifications by hand and especially prevents errors during these verifications. The choice of using Coq obviously implies that any automated prover collaborating with FoCaL must be able to provide as output a Coq proof trace.

Benefits of a unique language come from the ease of traceability since there is a unique semantics for each phase. Figure 2 summarises FoCaL's main features used during the life-cycle phases.

Documentation of the development is not missed: during each phase, documentation can be extracted from the FoCaL model (see paragraph 'documentation' on section 3). All these documentations are valuable for the assessor to verify the safety issues and for the testing team in order to produce the validation tests. Note that even if the software is formally developed, an independent validation testing is mandatory. Finally, this documentation can be included in the artifacts for the system safety-case.

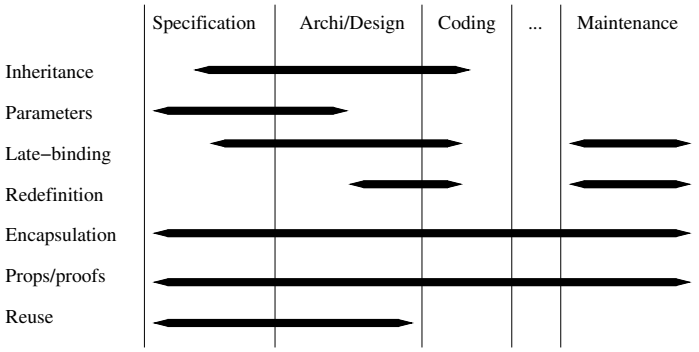

Fig. 2. FoCaL features along the life-cycle

5 Voter implementation

We now present an implementation of a voter in FoCaL following the development life-cycle.

Voter specification phase

Species *Sp_voter* presents the signature of the *vote* function, the functional requirements of *vote* and the constraints that imported components must respect for the voter to be proved. Here we force the imported comparison function *comp_value* representing the consistency law to be symmetric. This avoids the need to consider all comparisons combinations.

```
(* Specification species for the vote algorithm part. *)
species Sp_voter (Q is Sp_qualifier, Si is Sp_sensor_index, V is Value)
  inherits Basic_object =

  (* Declaration of the vote function:
     it takes 3 values and returns a sensor (index * qualifier). *)
  signature vote : V -> V -> V -> (Si * Q) ;
  signature sensor in (Si * Q) -> Si ; (* Get the first component of the
      voter. *)
  signature state in (Si * Q) -> Q ; (* Get the second component of the voter
      . *)

  (* Functional requirement 1: vote between 3 compatible values. *)
  property vote_perfect :
    all v1 v2 v3 in V,
    ((V!comp_value (v1, v2) /\ V!comp_value (v2, v3) /\ V!comp_value (v1, v3)
        ) ->
    (Si!equal (sensor (vote (v1, v2, v3)), Si!capt_1) /\
     Q!equal (state (vote (v1, v2, v3)), Q!perfect_match))) ;
  ...

  (* Constraint on imported functions: function comp_value must be symmetric.
      *)
  property comp_value_is_symmetric :
    all v1 v2 in V, V!comp_value (v1, v2) -> V!comp_value (v2, v1) ;
  ...
```

Voter design phase

In this phase, we give an effective implementation to the *vote* function and we prove that it satisfies the functional requirements (c.f. $vote\_perfect$).

```
(* Design phase of the voter algorithm. *)
species Imp_voter (Q is Sp_qualifier, Q is Sp_qualifier, Si is
    Sp_sensor_index,
    V is Value)
  inherits Sp_voter (Si, Q, V) =
```

```
  rep = unit ;            (* Definition of the carrier. *)

  (* Definition of the function. *)
  let vote (v1 in V, v2 in V, v3 in V) in (Si * Q) =
    let c1 = V!comp_value (v1, v2) in
    let c2 = V!comp_value (v1, v3) in
    let c3 = V!comp_value (v2, v3) in
    if c1 then
      if c2 then
        if c3 then (Si!capt_1, Q!perfect_match)
        else ...

  (* Proof of the functional requirements. *)
  proof of vote_perfect =
    by property Si!comp_transitive, Q!comp_transitive definition of vote
  ...
```

Voter implementation phase

The concrete species is built by providing effective collections to each voter's parameters. At this stage, "glue" assumptions are proved (c.f. *comp_value_is_symmetric*).

```
(* Definition of the closed species. *)
species Concrete_coll_int_imp_vote_tol inherits
   Imp_vote (Coll_etat_vote, Coll_capteur, Coll_int_imp_value_tol) =

   (* Proof of the properties of the imported function. *)
   proof of comp_value_is_symmetric =
     by Coll_int_imp_value_tol!comp_value_symmetric ;
  ...
end ;;

(* Final point: get the frozen and run-able component. *)
collection Coll_int_imp_vote_tol implements Concrete_coll_int_imp_vote ;;
```

6 Conclusion and further works

FoCaL is a living demonstration that an academic development framework can be created, integrating simultaneously strong expressiveness and semantics, efficient target code and industrial needs to assess the produced software. Indeed, the FoCaL tool provides a usable compromise between the ease to develop and the constraints imposed by standards.

Several examples have been developed following the present methodology like hierarchical automata's, physical input acquisition,...

Ten years after its birth, the language is now mature enough to add enhancements, and to bring stability and openings to external tools. A complete rewriting of the tool is currently performed to improve compilation and to facilitate integration of new features. Some other system paradigms (like synchronous features, higher-order parametrisation, certified C back-end...) are currently studied, always keeping in mind that the produced software must be assessed by an independent authority before its commissioning.

References

[1] "*Cenelec EN 50128, Railway Applications - Communications, Signalling and Processing Systems - Software for Railway Control and Protection Systems*", Cenelec, 1999

[2] "*DO-178C/ED12C, Software considerations in airborne systems and equipment certification*", RTCA/EUROCAE, To appear

[3] " *CC, Common Criteria for Information Technology Security Information*", Sept. 2006 `http://www.commoncriteriaportal.org/`

[4] "*Functional safety of electrical/electronic/programmable electronic safety-related systems*", International Electrotechnical Commission, 1998

[5] Virgile Prevosto, "*Conception et Implantation du langage FoC pour le développement de logiciels certifiés*", PhD Thesis, Paris 6 University, September 2003

[6] Sylvain Boulmé, "*Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*", PhD Thesis, Paris 6 University, 2000

[7] Hardin, Thérèse and Rioboo, Renaud, "*Les Objets des Mathématiques*", in RSTI - L'objet, (Hermes Science / Briot, Jean Pierre), October 2004

[8] Manuel Maarek and Virgile Prevosto, "*FoCDoC: the documentation system of FoC*", in Proceedings of Calculemus, September 2003

[9] Xavier Leroy, Jérôme Vouillon, Damien Doligez and others "*The Objective Caml system*", 2007, `http://caml.inria.fr/ocaml/`

[10] Luis Damas and Robin Milner, "*Principal type-schemes for functional programs*", POPL82, ACM, Pages 207–212, 1982

[11] Robin Milner, "*A theory of type polymorphism in programming*", JCSS, 1978, Volume 17 Number 3 Pages 348–375

[12] M. Carlier and C. Dubois, "*Functional Testing in the Focal environment*", in Test And Proof (TAP'2008), B. Berckert and R. Hahnle eds, LNCS 4966, pages 84–98, April 2008

[13] C. Parent "*Developing certified programs in the system Coq - The Program tactic*", in Proceedings of Types'93 ed. H. Barendregt and T. Nipkow 1994

[14] "*The Coq proof assistant*", Software and documentations freely available at `http://coq.inria.fr/`

[15] "*The CiME Rewrite Tool*", Software and documentations freely available at `http://cime.lri.fr/`

[16] E. Contejean, P. Courtieu, J. Forest, O. Pons and X. Urbain, "*Certification of automated termination proofs*", Proc. of 6th International Symposium on Frontiers of Combining Systems, Lecture Notes in Artificial Intelligence, Springer-Verlag, 2007

[17] Damien Doligez, "*The Zenon tool*", Software and documentations freely available at `http://focal.inria.fr/zenon/`

[18] Richard Bonichon, David Delahaye and Damien Doligez, *Zenon : An Extensible Automated Theorem Prover Producing Checkable Proofs*", LPAR, Pages 151–165, 2007

[19] David Delahaye, Jean-Frédéric Etienne and Véronique Donzeau-Gouge. "*Reasoning about airport security regulations using the Focal environment*", in 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation, Isola 2006

[20] David Delahaye, Jean-Frédéric Etienne and Véronique 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, Pages 121-124, 2008

[21] J.R. Abrial, " *The B-Book - Assigning Programs to meanings*", Cambridge University press, 1996.

[22] Virgile Prevosto and Sylvain Boulmé, "*Proof contexts with late binding*", In Typed Lambda Calculi and Applications, volume 3461 of LNCS, pages 324-338. Springer, April 2005

[23] D. Delahaye and J.-F Étienne and V. Viguié, " *Certifying Airport Security Regulations using the Focal Environment*", in FM2006: Formal Methods, volume 4085 of LNCS, pages 48-63. Springer, August 2006