Development life cycle of critical software under FoCal\textsuperscript{1}

Philippe Ayrault - Thérèse Hardin - François Pessaux

Pierre & Marie Curie University - Paris - SPI - LIP6

GDR-LTR 2009 - 21st of October 2009

\textsuperscript{1}This work is supported in part by the \textit{Agence Nationale de la Recherche} under grant ANR-06-SETI-016 for the SSURF Project (Safety and Security UndeR FoCal).
Plan

Context and objectives

The FoCal tool

Development cycle

A simple example : a voter

Voter under FoCal

Conclusion
Plan

Context and objectives

The FoCal tool

Development cycle

A simple example: a voter

Voter under FoCal

Conclusion
Critical systems standards

Mandatory certification by independent authority before commissioning.

- Certification checks compliance of development with the requirements of applicable standards.
- Certification emits a judgement, on the conformity of the product with intended purpose.
Critical systems standards

Mandatory certification by independent authority before commissioning.

- Certification checks compliance of development with the requirements of applicable standards.
- Certification emits a judgement, on the conformity of the product with intended purpose.

Requires:

- A sharp breakdown of the development in phases (objectives, inputs, activities, outputs)
- Criteria to close a phase and strict boundaries between phases

Needs a development methodology covering all development phases from the specification to the system commissioning.
Objectives

Development life cycle of critical software under FoCal
Objectives

Development life cycle of critical software under FoCal

- Propose “FoCal templates” dedicated to each phase
- Study transcription of normative requirements with their aims into FoCal using features like inheritance, late binding, redefinition, parametrisation...
- Consider automation of documentation to prepare evaluation
Objectives

Development life cycle of critical software under FoCal

- Propose “FoCal templates” dedicated to each phase
- Study transcription of normative requirements with their aims into FoCal using features like inheritance, late binding, redefinition, parametrisation...
- Consider automation of documentation to prepare evaluation

Apply it on concrete samples
Plan

Context and objectives

The FoCal tool

Development cycle

A simple example: a voter

Voter under FoCal

Conclusion
FoCal

- Launched in 1998 by T. Hardin and R. Rioboo
- IDE for a language offering high level mechanisms
- Conceived to be easy to use by well-trained engineers
- Unique framework embedding code, proofs, tests and documentation from specification to implementation
- Provide high-level and justified confidence to users
The core language 1/2

- **Species**: fundamental entity grouping:
  - **Representation**: the internal data-structure of the species,
  - **Signatures**: prototype of computational functions,
  - **Functions**: implementation of signatures using a ML-like language,
  - **Properties**: first order logical expressions that must hold and will be proved,
  - **Theorems**: properties with their proof.

- Multiple inheritance between species, late binding and redefinition.

- Parameterization of species by collections (flavour of species polymorphism).
The core language 2/2

- *Complete species*: species with effective data representation, executable functions and all theorems proved.
- *Collections*: abstraction of the representation of a complete species. It can be seen as a kind of abstract data-type, only usable through its signatures and properties.
The core language 2/2

- **Complete species**: species with effective data representation, executable functions and all theorems proved.

- **Collections**: abstraction of the representation of a complete species. It can be seen as a kind of abstract data-type, only usable through its signatures and properties.

- Consistency between species is ensured by a powerful dependency calculus and a proof system. FoCal compiler keeps track of dependencies between species, their functions, their properties, the proofs... to ensure detection of any modification by redefinition.
The proof language

Several ways to make a proof for the Coq theorem prover:

1. Easy but discouraged: consider the proof as “assumed”.
3. By “hand” but tricky: write the Coq code of the proof manually.
Species are translated to OCaml and Coq using a unique compilation model. The compilation model uses simple features available in any usual programming languages (structure and first order-modules).

- Computational methods ⇒ OCaml “runable” code and Coq definitions.
- Logical properties ⇒ Coq statements.
- FoCal proofs ⇒ Coq proofs

The whole development is sent to the Coq theorem prover who acts as an assessor.
“Documentation” (parsed and kept comments, signatures and properties) are compiled via the FocDoc tool to XML, HTML...

“UML class diagram” are automatically generated from FoCal development

The tool FocalTest automatically produces the test environment and the drivers to conduct the tests.
Plan

Context and objectives

The FoCal tool

Development cycle

A simple example: a voter

Voter under FoCal

Conclusion
V life cycle

- Specification
- Design
- Implementation
  - Low level testing
- Integration/Validation testing
- Maintenance
V life cycle

This presentation is focusing on the specification, architecture/design, coding and maintenance phase. All other phases dealing with tests are covered by the work of M. Carlier et C. Dubois.
Requirements

**Objectives**: Define the requirements and external interfaces of the system from the needs expressed by the customer and the Hazard Analysis.
Requirements

Objectives: Define the requirements and external interfaces of the system from the needs expressed by the customer and the Hazard Analysis.

Three kinds:

- *Functional requirements* providing the behaviour of the system independently of any specific design.
Requirements

**Objectives** : Define the requirements and external interfaces of the system from the needs expressed by the customer and the Hazard Analysis.

Three kinds :

- *Functional requirements* providing the behaviour of the system independently of any specific design.
- *Non-functional requirements* addressing design constraints (response time, memory space available, safety level, COTS to re-use, ...).
Requirements

**Objectives**: Define the requirements and external interfaces of the system from the needs expressed by the customer and the Hazard Analysis.

Three kinds:

- *Functional requirements* providing the behaviour of the system independently of any specific design.
- *Non-functional requirements* addressing design constraints (response time, memory space available, safety level, COTS to re-use, ...).
- *Safety requirements* ensuring that the functional requirements will never trigger a Feared Event. They can be considered as requirements on the two first kinds of requirements.
Requirements in FoCal

Requirements are expressed as far as possible as properties in FoCal.

- *Functional requirements* expressed as properties
- *Non-functional requirements* usually expressed as comments
- *Safety requirements* expressed as properties of any form
- *Glue assumptions* to express properties expected on parameters of species
In FoCal: each “object/part/entity” of the system leads to a species only made of:

- **Signatures** of functions
- **Properties** expressing functional requirements (i.e. relations between inputs and outputs of each function)
- **Properties** expressing the safety requirements
- **Properties** expressing assumptions on species used as parameters (i.e. ”glue” assumption)
In FoCal: each “object/part/entity” of the system leads to a species **only** made of:

- **Signatures of functions**
- **Properties** expressing functional requirements (i.e. relations between inputs and outputs of each function)
- **Properties** expressing the safety requirements
- **Properties** expressing assumptions on species used as parameters (i.e. "glue" assumption)
- End of specification phase: all safety requirements are proved using the functional requirements and the glue assumptions as hypothesis
Architecture/Design phase (1/2)

**Objectives:** Provide a decomposition of specification into computable and efficient components.

Decomposition leads to:
- Specification of new components (c.f. previous phase)
- Scheduling these new components.

**Decomposition must respect the functional (and non-functional) requirements of the specification phase.**

This process is performed iteratively until obtaining each system/software primitive components.
In FoCal: Provide a decomposition, into species, using inheritance and parametrisation.
For each component of the system:
  ▶ Inheritance of the species specification in order to share the requirements
  ▶ Definition of the data representation
  ▶ Provide definitions for signatures
  ▶ Re-use existing species via inheritance and parametrisation. (FoCal Components On The Shelf or components from some foreign languages).

Proofs of the functional requirements using the definitions and the properties of external (i.e. parameters and inherited) species.
Coding phase

**Objectives**: Produce the source code of the completely defined components in the target language

In FoCal, for each component of the system:

- Production of the *collections* (i.e. opaque species)
- Generation of the target source code
- Proofs that the “glue” properties hold between the parametrised species.
Maintenance phase

Objectives: Maintain the consistency of the system (especially safety properties) during each evolution of the system.

In FoCal, for each component of the system:

- Modification of the species to integrate the system evolutions
- Generation of the target source code.
- Proofs required by the FoCal tool in order to maintain the consistency of the system.
Plan

Context and objectives

The FoCal tool

Development cycle

A simple example: a voter

Voter under FoCal

Conclusion
Generic definition

Elaboration of an output from input values given by redundant components

Basic principles

▷ to compare its input values according to a given consistency relation

▷ to output one value depending on a voting policy
Generic definition

Elaboration of an output from input values given by redundant components

**Basic principles**

- to compare its input values according to a given consistency relation
- to output one value depending on a voting policy

**Functional requirements**

- Reliable and correct choice of one non faulty input among its $n$ inputs
- detection of faulty inputs
- Localisation of the source of the error and report of a diagnosis related to it
Generic definition

Voter with 3 inputs

Value1 -> Value
Value2 -> Diag
Value3 ->
Safety requirements

A voter returns one of its input values or no value can be chosen

\[ \forall v_1, v_2, v_3 \text{ in value}, s = \text{vote}(v_1, v_2, v_3) \rightarrow \]
\[ \text{value}(s) \in \{v_1, v_2, v_3\} \lor \neg \text{valid}(\text{diag}(s)) \]
Safety requirements

A voter returns one of its input values or no value can be chosen
\[ \forall v1, v2, v3 \text{ in value}, s = \text{vote}(v1, v2, v3) \rightarrow \]
\[ \text{value}(s) \in \{v1, v2, v3\} \lor \neg \text{valid(diag}(s)) \]

Returned values for different input value orders are compatible
\[ \forall v1, v2, v3 \text{ in value}, s1 = \text{vote}(v1, v2, v3) \land s2 = \text{vote}(v3, v1, v2) \]
\[ \rightarrow \text{compatible}(s1, s2) \]

Outputs are compatible if values fit consistency rule or values are not valid
\[ \forall (v1, d1), (v2, d2) \text{ in (value * diag)}, \]
\[ \text{compatible}((v1, d1), (v2, d2)) \rightarrow \]
\[ (\text{valid}(d1) \land \text{valid}(d2) \land \text{consistency_rule}(v1, v2)) \lor \]
\[ (\neg \text{valid}(d1) \land \neg \text{valid}(d2)) \]
2003 voter specification

- Perfect_match, v1
- Partial_match, v1
- Range_match, v1
- No_match
### 2003 voter specification

<table>
<thead>
<tr>
<th>Consistency between inputs</th>
<th>Returned Value</th>
<th>Diag</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Value</td>
<td>Index</td>
</tr>
<tr>
<td>v1 and v2</td>
<td>v1 and v3</td>
<td>v2 and v3</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
2003 architecture

A 2 steps algorithm:

- the *inputs comparison*, which takes 2 or more inputs and compares them according to a "consistency law"

- the *arbitration*, voting policy algorithm which produces the output value.

We focus on the arbitration part of the voter. The complete development is available as a FoCal’s contrib.
Plan

Context and objectives

The FoCal tool

Development cycle

A simple example: a voter

Voter under FoCal

Conclusion
Generic voter specification architecture

species Gen_voter( l in Gen_Value, ...)  
signature vote(i1,i2,i3)  
Generic/Safety requirements  
Glue assumptions

species Gen_diag inherits Setoid  
signature valid;

species Gen_value inherits Setoid  
signature consistency law  
Generic/safety requirements
The generic voter requirements in FoCal

```
species Gen_voter( V is Gen_value, Diag is Gen_diag) =

  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);
  let output_diag(p in V * Diag) in V = basics#snd(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
  \ ~(Diag!valid (output_diag (voter (v1, v2, v3))))

  (* Glue assumption on generic values *)
  property consistency_rule_is_reflexive:
    all v1 in V, V!consistency_rule (v1, v1);
  end;;
```
The 2oo3 voter requirements

- species Voter inherits from the generic voter
- defines the 2oo3 voter functional requirements
- provides proofs of the safety properties based on voter functional requirements and glue assumptions
2003 specification architecture

species Gen_voter( I in Gen_Value, ...)
- voting policy requirements
- proof of safety requirements

species 2003_Voter( I in Value, ...)

species diag( I in sensor, ...)
- validity requirements
- proof of generic requirements

species Gen_diag inherits Setoid

species Value ( T in Basics_type)
- complex type CT
- consistency_law(CT1, CT2)
- glue assumptions on Basics_type

species Gen_value inherits Setoid

species Basics_types inherits basics
- Type
- basic operations (+, -, ...)
- property on basics operations
species Voter
(E is Sp_etat_vote, C is Sp_capteur, V is Gen_value, P is
   Diag_2003 (E, C)) =
   inherit Gen_voter (V, P);

(* Functional definition of the majority vote
   Vote with 3 equivalent values returns a perfect_match
   and the value of the first sensor. *)
property vote_perfect : 
   all v1 v2 v3 in V,
      (V!consistency_rule (v1, v2) \/
       V!consistency_rule (v2, v3) \/
       V!consistency_rule (v1, v3)) ->
      ((value (voter (v1, v2, v3)) = v1) \/
       (diag (voter (v1, v2, v3)) =
        P!constr (C!capt_1, E!perfect_match)));

proof of voter_returns_an_input_value =
   ...
end ;;
The 2oo3 voter implementation

- species Imp_Voter inherits from the 2oo3 voter
- implements the 2oo3 voter functional requirements
- provides proofs of the voter properties based on voter definition and glue assumptions
The 2003 voter implementation in FoCal

species Imp_vote
    (E is Sp_etat_vote, C is Sp_capteur, V is Gen_value, P is Diag_2003 (E, C)) =

    inherit Voteur (E, C, V, P);

    let voter (v1 in V, v2 in V, v3 in V) in V * P =
    let c1 = V!consistency_rule (v1, v2) in
    let c2 = V!consistency_rule (v1, v3) in
    let c3 = V!consistency_rule (v2, v3) in
    if c1 then ...

    proof of vote_perfect =
    by property V!equal_reflexive, P!equal_reflexive
    definition of voter, diag, value;
...
end;;
Getting the 2oo3 voter collection

- complete species Imp_vote by providing proofs of the glue assumptions of the parameters
- build collections for the parameters
- collection Coll_Voter implements the complete species
The 2003 voter collection in FoCal

species Sp_int_imp_vote_tol =

  inherit
    Imp_vote (Coll_etat_vote, Coll_capteur,
    Coll_int_imp_value_tol, Coll_diag);

(* proof of the glue assumptions *)
proof of consistency_rule_is_symmetric =
  by property Coll_int_imp_value_tol!
    consistency_rule_symmetric;
proof of consistency_rule_is_reflexive =
  by property Coll_int_imp_value_tol!
    consistency_rule_reflexive;

end;;

collection Coll_int_imp_vote_tol =

  implement Sp_int_imp_vote_tol;
end;;
Complete architecture

species Gen_value inherits Setoid

species Value (T in Basics_type)
- complex type CT
- consistency_law(CT1, CT2)
- glue assumptions on Basics_type

species Basics_types inherits basics
- Type
- basic operations (+, -, ...)
- property on basics operations
The 2oo3 voter execution

Voter on integers with a margin of error of 2
1, 3, 5 --> val : 3 , diag : (capt_2, partial_match)
1, 1, 5 --> val : 1 , diag : (capt_3, range_match)
4, 5, 5 --> val : 4 , diag : (capt_1, perfect_match)
1, 4, 7 --> val : 1 , diag : (capt_1, no_match)

Voter on integer with a validity bit
( 23, valid), ( 45, valid), ( 23, valid)
  --> val : ( 23, valid) , diag : (capt_2, range_match)

( 23, invalid), ( 45, valid), ( 23, valid)
  --> val : ( 23, invalid) , diag : (capt_1, no_match)
Other works in FoCal

Realistic works with FoCal:

- Standard library providing common data structures,
- Formal calculus library (R. Rioboo)
- Certification of an airport security regulation (J.F. Etienne)
- Access control Library (M. Jaume, C. Morisset)
- Nested automatas (P. Ayrault)
Plan

Context and objectives

The FoCal tool

Development cycle

A simple example: a voter

Voter under FoCal

Conclusion
Contributions

- Definition of an usable methodology to apply to critical software development taking advantage of one language for covering all phases.
Contributions

- Definition of an usable methodology to apply to critical software development taking advantage of one language for covering all phases.
- Experimentation that FoCal is easy to use and to understand by engineers without knowledge on higher order features.
Contributions

- Definition of an usable methodology to apply to critical software development taking advantage of one language for covering all phases.
- Experimentation that FoCal is easy to use and to understand by engineers without knowledge on higher order features
- Demonstration that FoCal offers a good compromise between development expressivity and compliance with normative requirements
Thank you for your attention.

FoCal tool can be downloaded from
http://focalize.inria.fr/.
Thank you for your attention.

FoCal tool can be downloaded from
http ://focalize.inria.fr/.