Development of a generic voter under FoCal[1]

Philippe Ayrault - Thérèse Hardin - François Pessaux

Pierre & Marie Curie University - Paris - SPI - LIP6

TAP 2009 - 2nd of July 2009

[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).

Plan

Plan

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

Define a development methodology using FoCal

Objectives

Define a development methodology using 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
- Consider testing coverage

Objectives

Define a development methodology using 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
- Consider testing coverage

Apply it on concrete samples

Plan

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".
2. Helped : a structured proof in the FoCal Proof Language. Each step possibly proved automatically by Zenon which produces a Coq proof term.
3. By "hand" but tricky : write the Coq code of the proof manually.

FoCal compilation 1/2

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 $\Rightarrow$ OCaml "runable" code and Coq definitions.
- Logical properties $\Rightarrow$ Coq statements.
- FoCal proofs $\Rightarrow$ Coq proofs

The whole development is sent to the Coq theorem prover who acts as an assessor.

FoCal compilation 2/2

- "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

Requirements

Three kinds :

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

Requirements

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

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 of the form

$$\forall\ i_1\ \mathit{in}\ t_1, \ldots, i_n\ \mathit{in}\ t_n,$$
$$r1(i_1, i_2, ..., i_n) \rightarrow r2(i_1, i_2, ..., i_n, foo(i_1, i_2, ..., i_n))$$

- *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

Development cycle

- End of specification phase : all safety requirements are proved using the functional requirements and the glue assumptions as hypothesis
- End of architecture/design phase : all functional requirements are proved using glue assumptions as hypothesis. Needs to introduce the corresponding definitions
- Implementation phase consists in creating and assembling collections, proving the glue assumptions
- Testing phases : help in validating non functional requirements, partially proved properties and external components based on their properties
- Maintenance phase using late binding and redefinition

Use of the FoCaL dependencies calculus and documentation generation to generate formal traceability between phases.

Plan

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

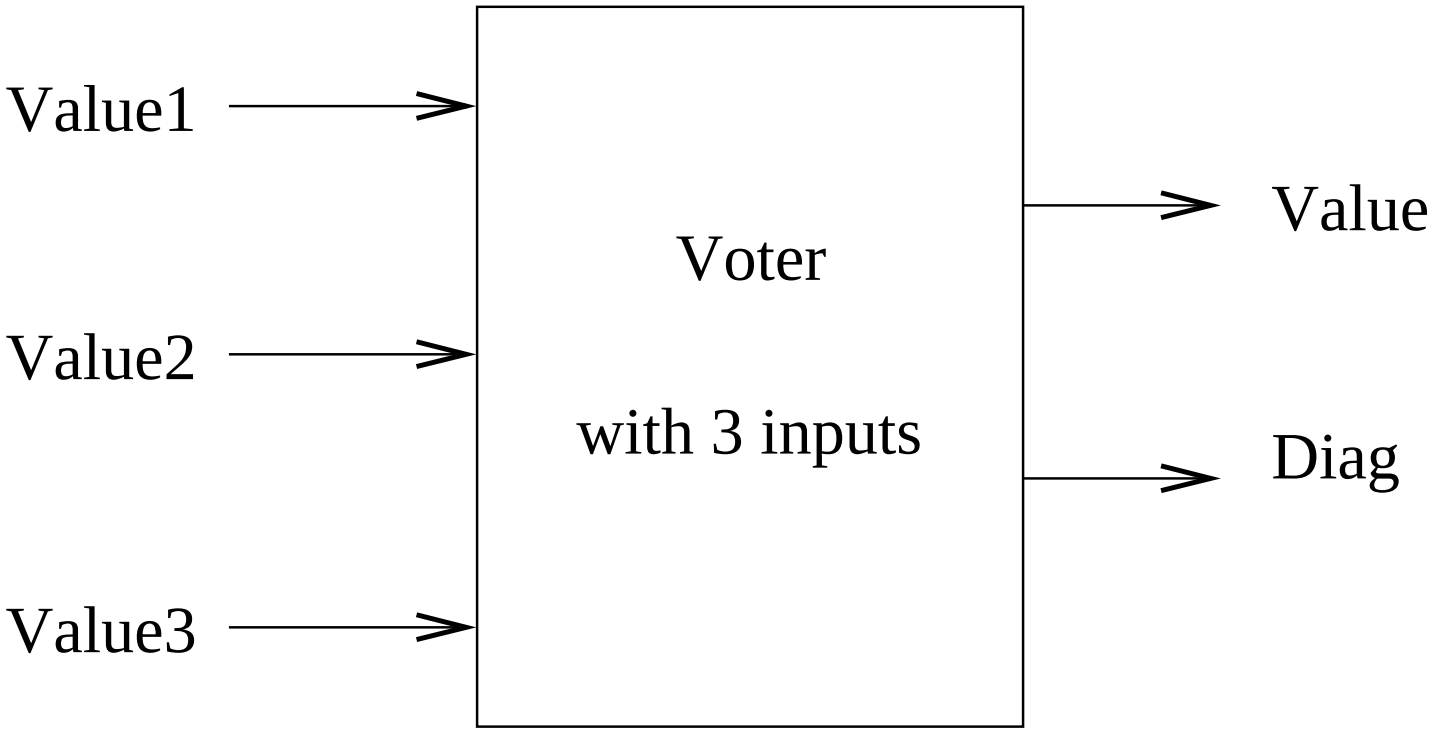
Basic principles

- 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



Safety requirements

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

$\forall v1, v2, v3 \ in \ value, s = vote(v1, v2, v3) \rightarrow$
$value(s) \in \{v1, v2, v3\} \vee \neg valid(diag(s))$

Safety requirements

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

$\forall v1, v2, v3\ in\ value, s = vote(v1, v2, v3) \rightarrow$
$value(s) \in \{v1, v2, v3\} \vee \neg valid(diag(s))$

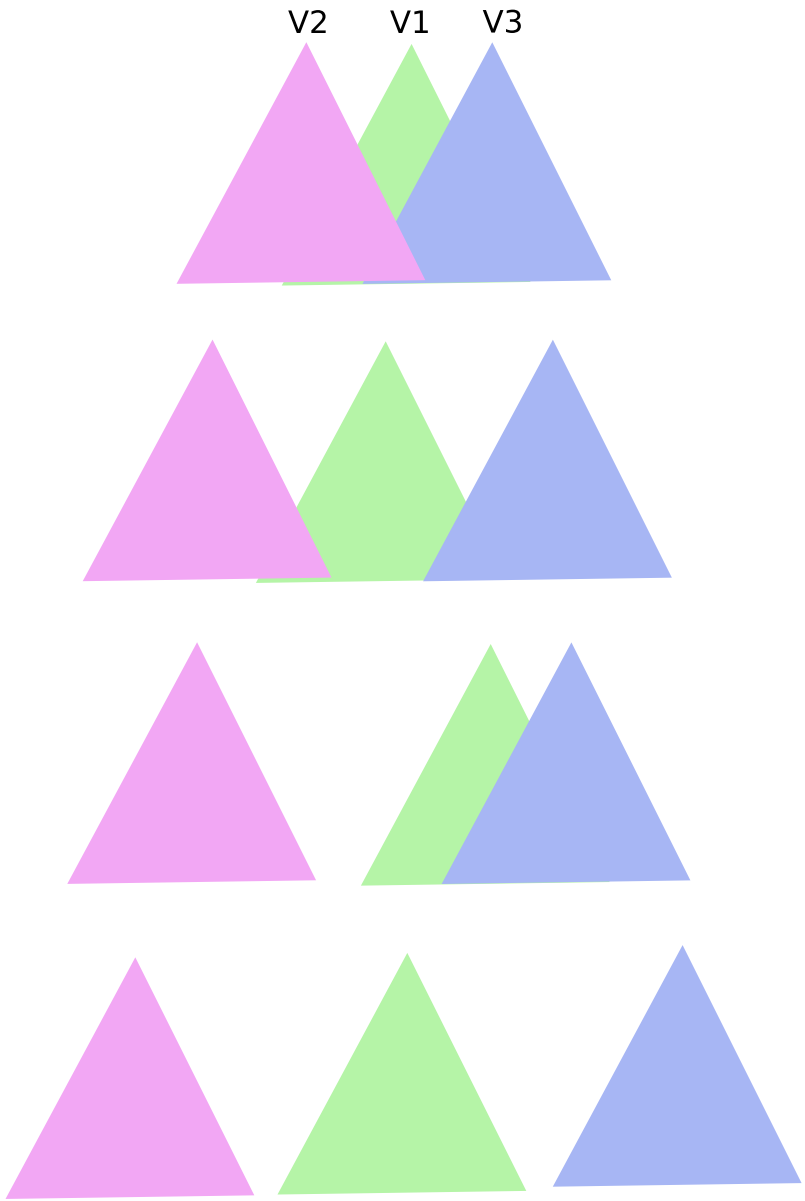
Returned values for different input value orders are compatible

$\forall v1, v2, v3\ in\ value, s1 = vote(v1, v2, v3) \wedge s2 = vote(v3, v1, v2)$
$\rightarrow compatible(s1, s2)$

Outputs are compatible if values fit consistency rule or values are not valid

$\forall (v1, d1),\ (v2, d2)\ in\ (value * diag),$
$compatible((v1, d1), (v2, d2)) \rightarrow$
$(valid(d1) \wedge valid(d2) \wedge consistency\_rule(v1, v2)) \vee$
$(\neg valid(d1) \wedge \neg valid(d2))$

2oo3 voter specification



Perfect_match, v1

Partial_match, v1

Range_match, v1

No_match

2oo3 voter specification

Consistency between inputs			Returned Value	Diag	
v1 and v2	v1 and v3	v2 and v3		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

2oo3 architecture

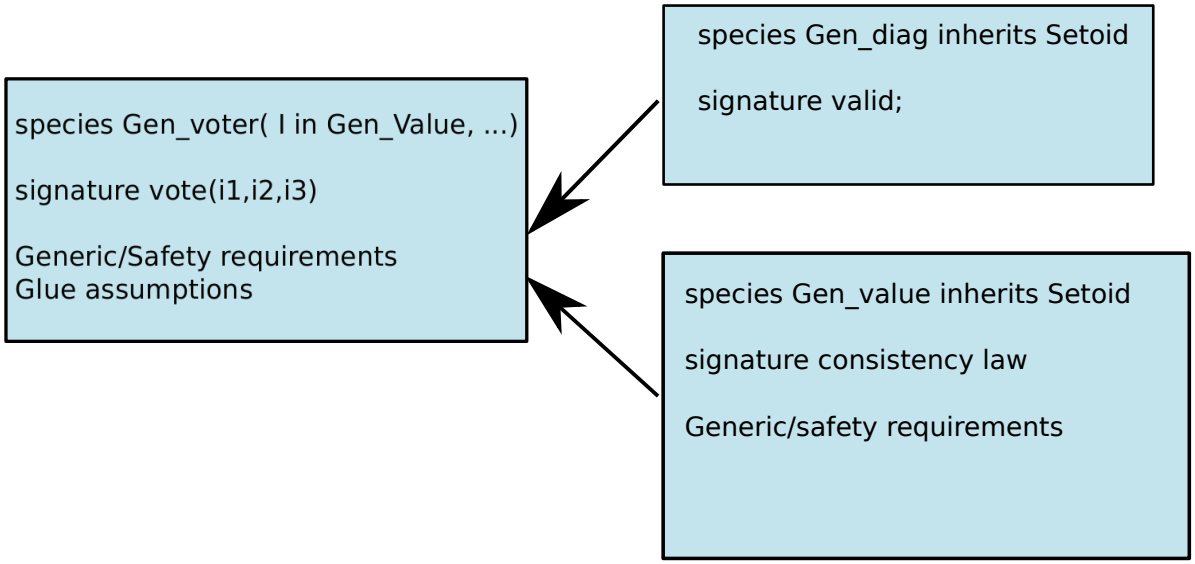
A 2 steps algorithm :

- 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

Generic voter specification architecture



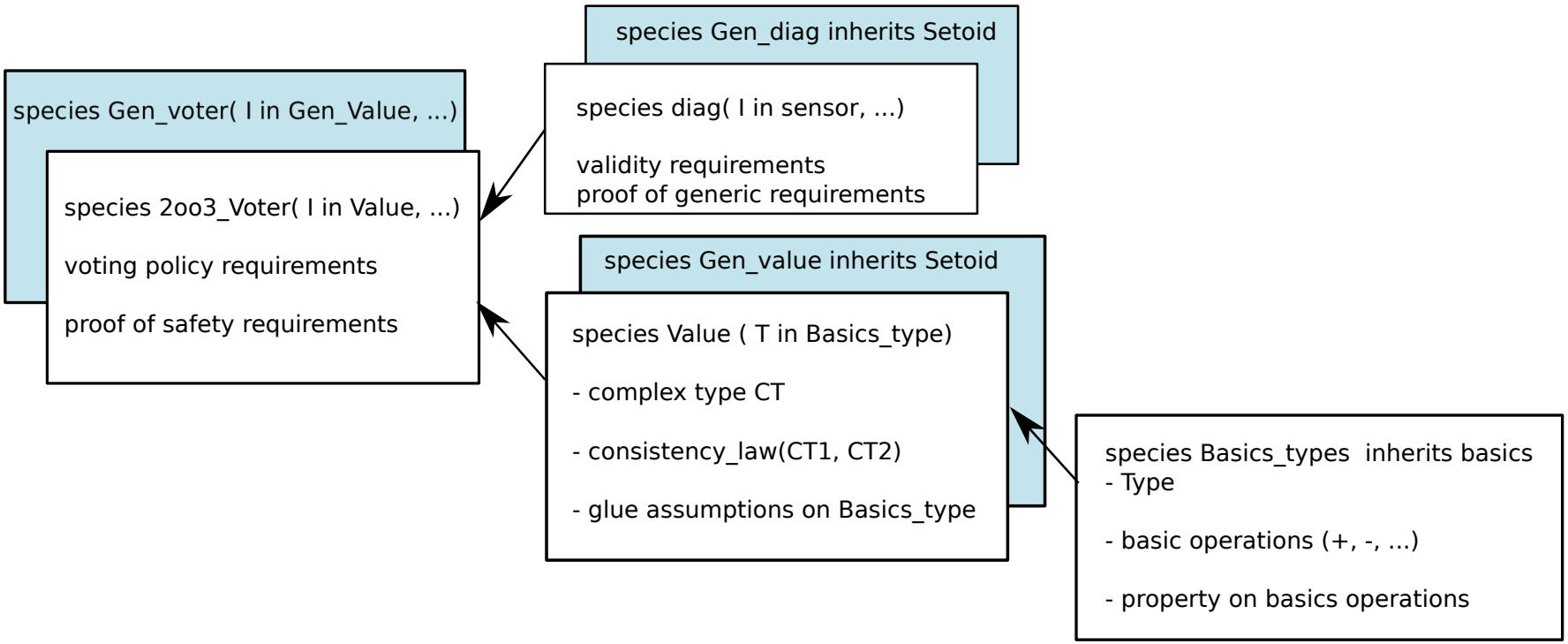
The generic voter requirements in FoCal

```
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

2oo3 specification architecture



The 2oo3 voter requirements in FoCal

```
species Voter
 (E is Sp_etat_vote, C is Sp_capteur, V is Gen_value, P is
     Diag_2oo3 (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 2oo3 voter implementation in FoCal

```
species Imp_vote
 (E is Sp_etat_vote, C is Sp_capteur, V is Gen_value, P is
     Diag_2oo3 (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 2oo3 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;;
```

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

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

Future works

Based on this normative development cycle for FoCal developments, we explore other features required by standards like :

- dysfunctional analysis based on dataflow
- impact analysis
- reactive language features

Tap 2009

Thank you for your attention.

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

Tap 2009

Thank you for your attention.

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