A Software Product Line Approach for Semantic Specification of Block Libraries in Data Flow Languages

Arnaud Dieumegard[1] Andres Toom[2,1,3] Marc Pantel[1]

[1]IRIT/ENSEEIHT - 2 rue Charles Camichel, 31000 Toulouse
first.last@enseeiht.fr

[2]Institute of Cybernetics at Tallinn University of Technology
Akadeemia tee 21, EE-12618 Tallinn, Estonia

[3]IB Krates OÜ, Mäealuse 4, EE-12618 Tallinn, Estonia
andres@krates.ee

TUT CS PhD Seminar IXX9603

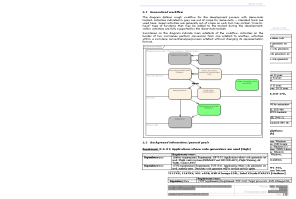
24.09.14

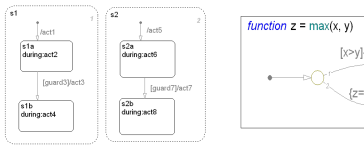
Outline

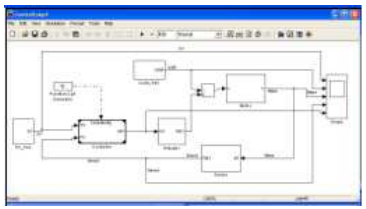
1. Context
   - Embedded SW development for (safety) critical and high-integrity systems
   - Current work
2. Block libraries in graphical dataflow languages
   - Graphical dataflow languages
3. Specifying block libraries
   - Natural language specification
   - Mathematical specification
   - Model-based specification
4. Applications
   - Tooling
   - Verification of the specification
   - Verification of generated code
   - Other applications

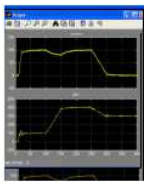
Embedded SW development for *(safety) critical* and *high-integrity* systems

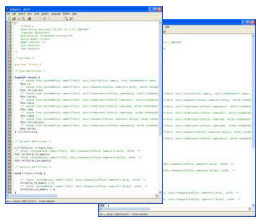
- Critical systems development is often organized as a V-Cycle
- V-Cycle is specified in several IEC standards (general safety, railway)
- Well suited also for DO-178C / ED-12C (avionic)





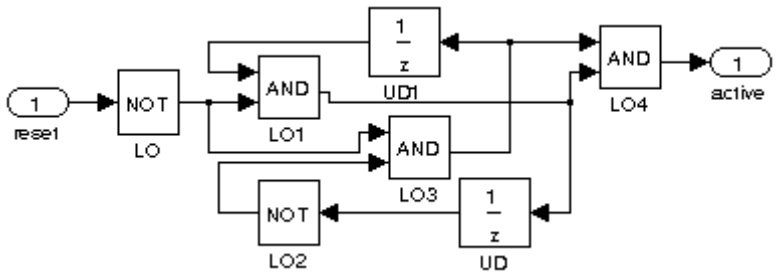






Current work

- Graphical modelling languages are widely used for the development of complex systems
- Control and and command algorithms often specified in
  - Data flow style: Simulink, Scicos, Xcos, SCADE ...
  - State chart/machine style: StateMate, Stateflow, SCADE ...
- P/Hi-MoCo and its predecessor Gene-Auto
  - Development of an open multi-domain *code generation toolkit* (C, Ada) for high-integrity embedded systems
  - Certification according to the avionic *software qualification* standard DO-178 C and similar standards of other domains
- Current work: Structure and formalize the block library specification

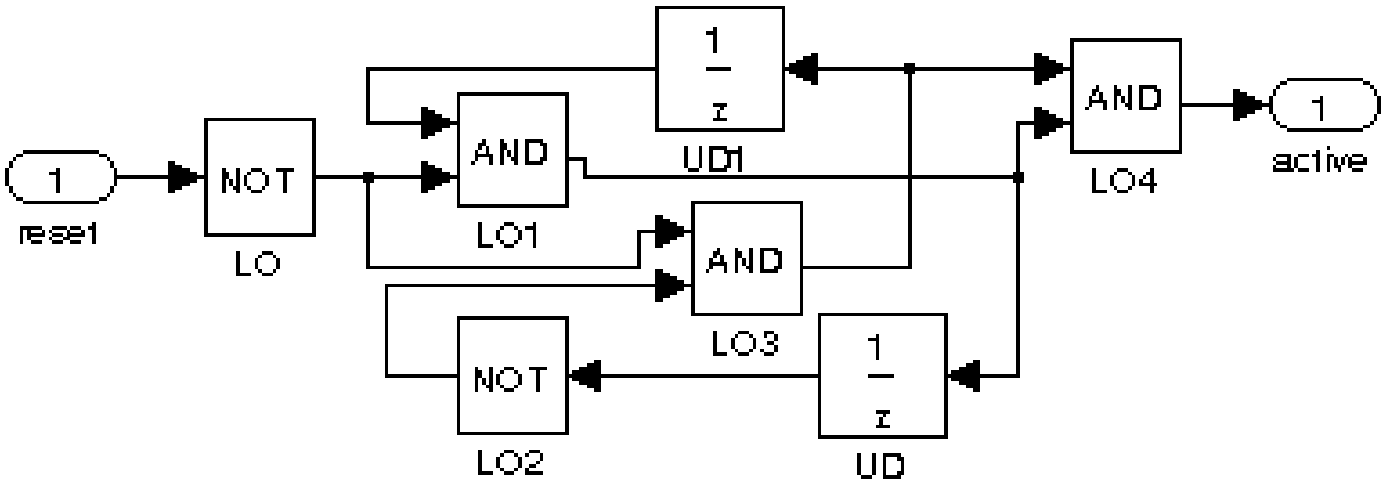


Outline

1. Context
   - Embedded SW development for (safety) critical and high-integrity systems
   - Current work
2. Block libraries in graphical dataflow languages
   - Graphical dataflow languages
3. Specifying block libraries
   - Natural language specification
   - Mathematical specification
   - Model-based specification
4. Applications
   - Tooling
   - Verification of the specification
   - Verification of generated code
   - Other applications

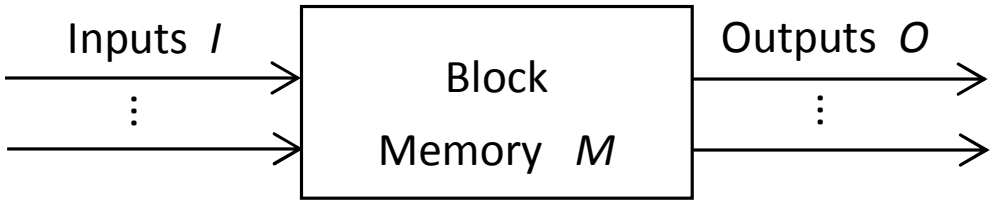
Graphical dataflow languages and block libraries

- Widely-used tools: Simulink, Scicos, Xcos, SCADE ...
- Main elements are *blocks* (nodes) and *signals* (data and control flows)
- Often used in the design of embedded control and command algorithms
- Lot of functionality encapsulated in blocks / block libraries
  - (Simulink >300 standard blocks + XX toolboxes)
- User extendable - Encapsulation of proprietary industrial IP

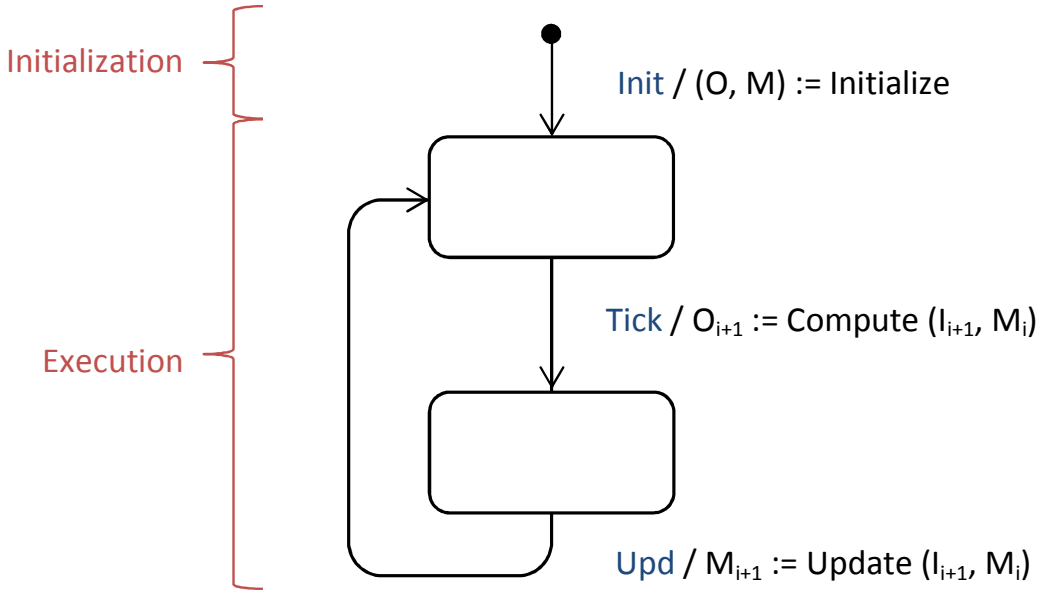


Block semantics

- Logical structure of a block

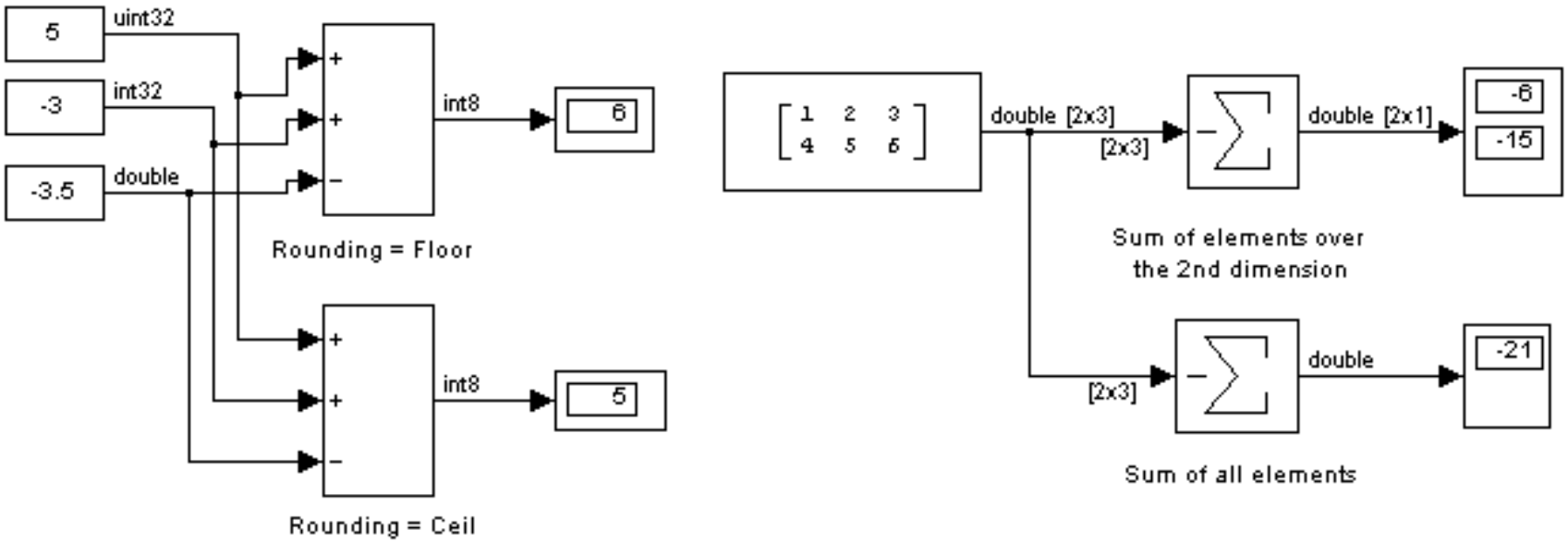


- Operational semantics in synchronous dataflow languages



Polymorphism / Configurability

- Blocks are often polymorphic wrt. data types and arity (scalars, vectors, matrices, ...)
- Behavior can be configured via a number of static parameters (number of inputs, rounding, saturation, ...)
- E.g. the Simulink Sum block
  - Original documentation: 19 pages on the meaning of parameters



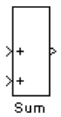
Outline

1. Context
   - Embedded SW development for (safety) critical and high-integrity systems
   - Current work
2. Block libraries in graphical dataflow languages
   - Graphical dataflow languages
3. Specifying block libraries
   - Natural language specification
   - Mathematical specification
   - Model-based specification
4. Applications
   - Tooling
   - Verification of the specification
   - Verification of generated code
   - Other applications

Specification in a natural language

- Pros
  - Easy to read
  - Natural language
- Cons
  - Not formal
  - Lengthy
  - Ambiguities/errors

4.51 Sum



4.51.1 Simulink equivalence

Simulink equivalence for this block is **Sum** block.

4.51.2 Optimizable

Yes, if inputs are scalars.

4.51.3 Characteristics

A purely functional block, serving to sum the block's inputs, element by element (element-wise) in case of multiple inputs and to sum the blocks's input elements in case of single input.

4.51.4 Inputs

Unlimited number of inputs, but must be of the same data type.

In case of different signal types, there can be a mix between scalar and vector signals in which case the scalar value will be summed to each element of the vector, or scalar and matrix signals in which case the scalar value will be summed to each element of the matrix but IMPOSSIBLE to mix vector and matrix signals.

Vector (resp. matrix) input signals must have the same dimension.

If only one input is given then it must be of scalar or vector signal type.

4.51.5 Output

Single output, of the same signal type and data type as the inputs having as value :

- If scalar : the sum of its scalar inputs or the sum of its vector input elements
- If vector : the sum vector of its vector inputs, element by element (element-

4.51.6 Pseudo-code

The associated pseudo-template code is :

- If scalar :
```
%o1 = %i1 + %i2 + … //scalar input(s)
%o1 = %i1[0] + %i1[1] + … //single input (obligatorily vector)
```
- If vector :
```
{
  int index = 0;
  for (index = 0; index < vector_size; index ++){
    %o1[index] = %i1[index] + %i2[index] + … + %i5 + …;
  }
}
```
- If matrix :
```
{
  int index_x = 0;
  int index_y = 0;
  for (index_x = 0 ; x < matrix_x_size ; index_x ++)
    for (index_y = 0; index_y < matrix_y_size; index_y ++)
      %o1[index_x][index_y] = %i1[index_x][index_y] + %i2[index_x][index_y] + … + %i5 + …;
}
```

with *vector_size*, *matrix_size_x* (and *matrix_size_y*) signify respectively the size of the output vector, the output matrix lines number, and the output matrix columns number. Here, we presume that we have mixed input signals with the 5th signal being a scalar.

4.51.7 Parameters

Block input	Parameter name	Possible values
-	Number of inputs	Any integer value.

Mathematical specification

- Based on a mathematical representation of the elements to specify
- Rely on standardised Math formalisms (MathML,. . .)

- Pros
  - Reading
  - Understanding
  - Writing
  - Formal
- Cons
  - General purpose language
  - Too much freedom in specification writing
  - No dedicated variability management

$\mathcal{P}(Inputs)$ The $\mathcal{P}(Inputs)$ parameter is defined as the following :

- $\forall i \in [1, n_{in}], \exists op_i \in [+, -], \mathcal{P}(Inputs) = \{op_i\}$

Input must be one of the following :

1. $\mathcal{I} = \{\forall i \in [1, n_{in}], X_i \in \mathbb{T}\}$
2. $\mathcal{I} = \{n_{in} = 1, X \in \mathcal{V}_n(\mathbb{T}_{\setminus \mathbb{B}})\}$
3. $\mathcal{I} = \left\{ n_{in} > 1, \left\{ \begin{array}{l} X_i \in \mathcal{V}_n(\mathbb{T}_{\setminus \mathbb{B}}) \\ X_i \in \mathbb{T}_{\setminus \mathbb{B}} \Rightarrow X_i \leftarrow \mathcal{V}_n(\mathbb{T}_{\setminus \mathbb{B}}) \wedge a_1 = \ldots = a_n \end{array} \right\} \right\}$
4. $\mathcal{I} = \left\{ n_{in} > 1, \left\{ \begin{array}{l} X_i \in \mathcal{M}_{n,m}(\mathbb{T}_{\setminus \mathbb{B}}) \\ X_i \in \mathbb{T}_{\setminus \mathbb{B}} \Rightarrow X_i \leftarrow \mathcal{M}_{n,m}(\mathbb{T}_{\setminus \mathbb{B}}) \wedge a_{1,1} = \ldots = a_{n,m} \end{array} \right\} \right\}$

Model-based specification

- Inspired from Feature Modeling [Kang 90] and Software Product Line (SPL) engineering
- Domain Specific Language specification
- Purpose-oriented model/language

- Pros
  - Focused on the goal to achieve
  - Ease reading, writing and understanding
  - Structured
  - Formal
- But also
  - Based on standardized formalism (MOF, Ecore)
  - Available and relatively easily extendable tooling
  - Model-based verification
  - Model-based transformations

- Cons
  - More difficult to apprehend
  - Not a "standard" language

Model-based specification

- Inspired from Feature Modeling [Kang 90] and Software Product Line (SPL) engineering
- Domain Specific Language specification
- Purpose-oriented model/language

- Pros
  - Focused on the goal to achieve
  - Ease reading, writing and understanding
  - Structured
  - Formal
- Cons
  - More difficult to apprehend
  - Not a "standard" language
- But also
  - Based on standardized formalism (MOF, Ecore)
  - Available and relatively easily extendable tooling
  - Model-based verification
  - Model-based transformations

Feature Modeling [Kang 90]

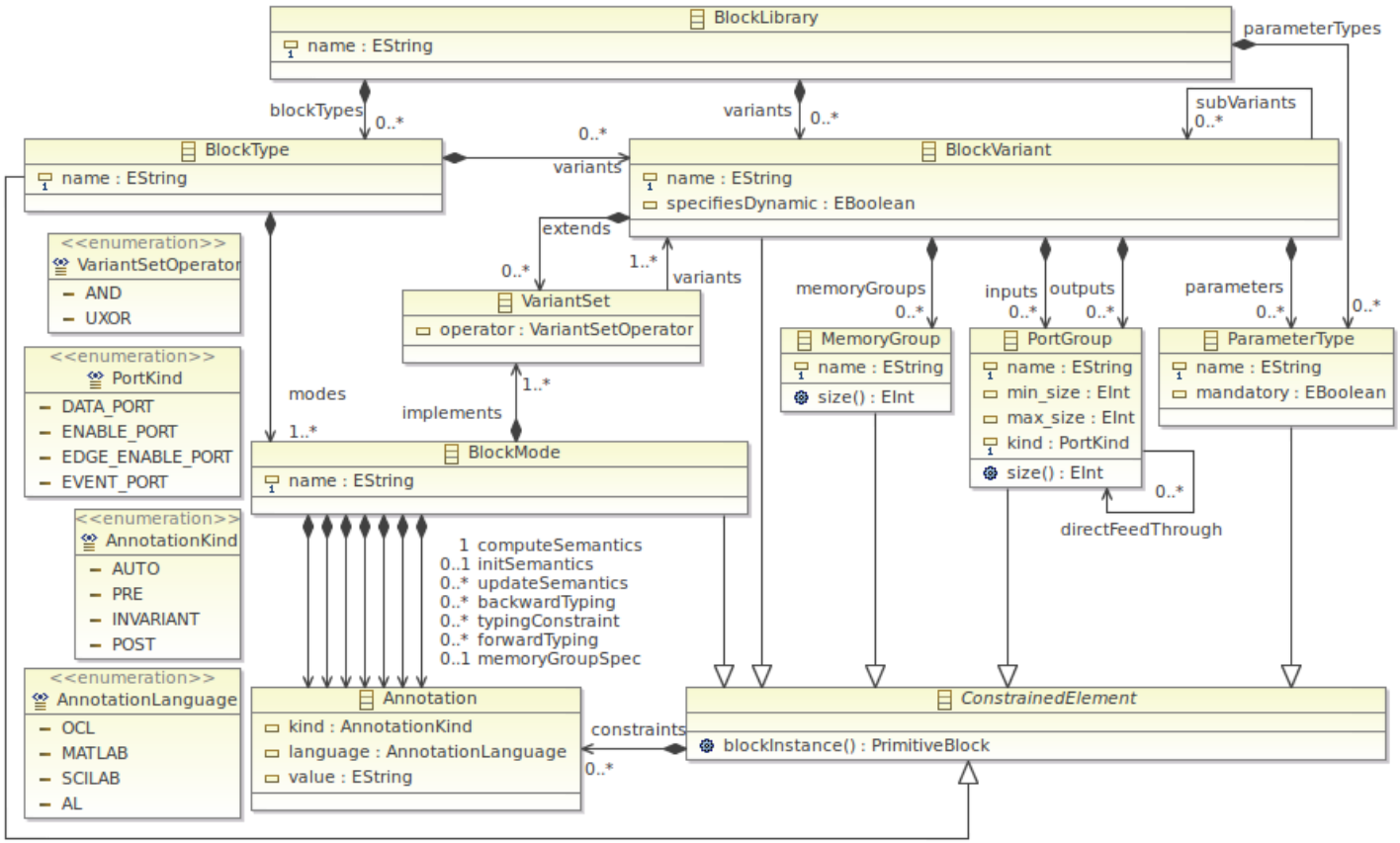
- All the different products of a Software Product Line (SPL) are represented in terms of *features*
- Typically in the form of a tree, where sub-features can be:
  - Mandatory
  - Optional
- and sub-feature groups can have
  - OR-relation (at leat one)
  - XOR/Alternative-relation (at most one)
- Additional cross-tree constraits can be specified
- Valid *feature configuration* $\equiv$ Member of the SPL

Block Library DSL

- BlockLibrary
  - Container for BlockTypes and "Generic" BlockVariants
- BlockType
  - Holds a block specification including BlockVariants and BlockModes
- BlockVariant
  - Parameters, ports and memories
  - Relations between BlockVariants: product line engineering approach
- BlockMode
  - Configuration of a block (set of block variants)
  - Typing constraints/rules
  - Computational semantics
- Structural correctness as first order logic properties on BlockModes, BlockVariants, Parameters, Ports and Memories
- Computational semantics of BlockModes expressed
  - axiomatically (pre/post conditions)
  - operationally (a simple imperative action language)

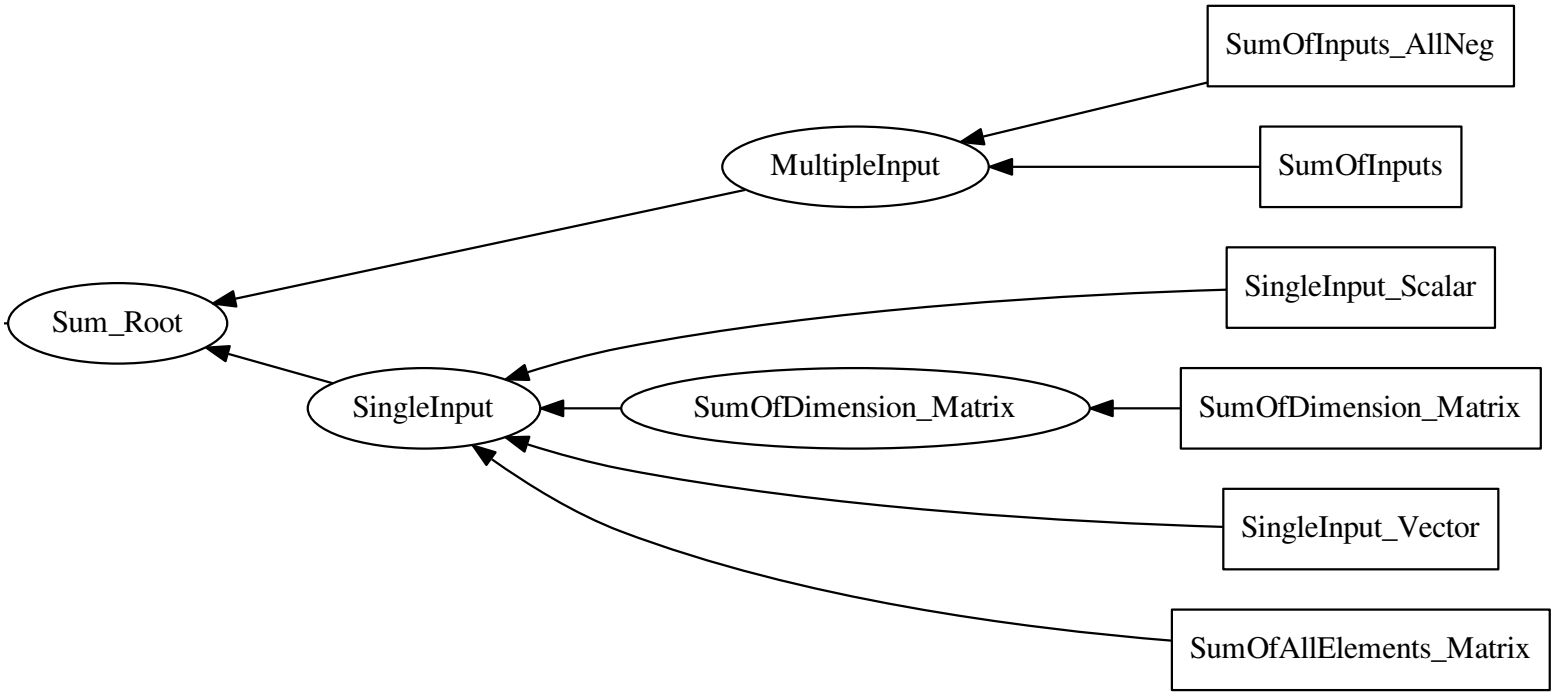
Block Library Meta-model

- MOF/Ecore: well accepted in the industry and standardized
- + OCL constraints for static semantics



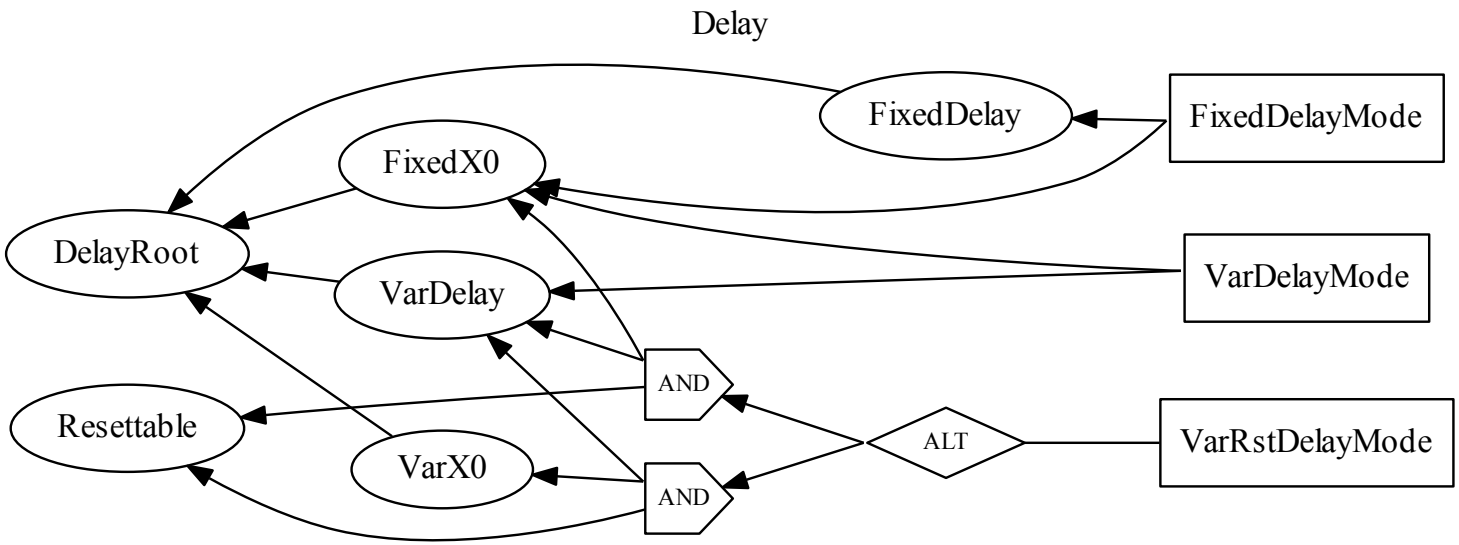
Specification decomposition examples

- The variablity graph helps to understand dependencies between BlockVariants and BlockModes



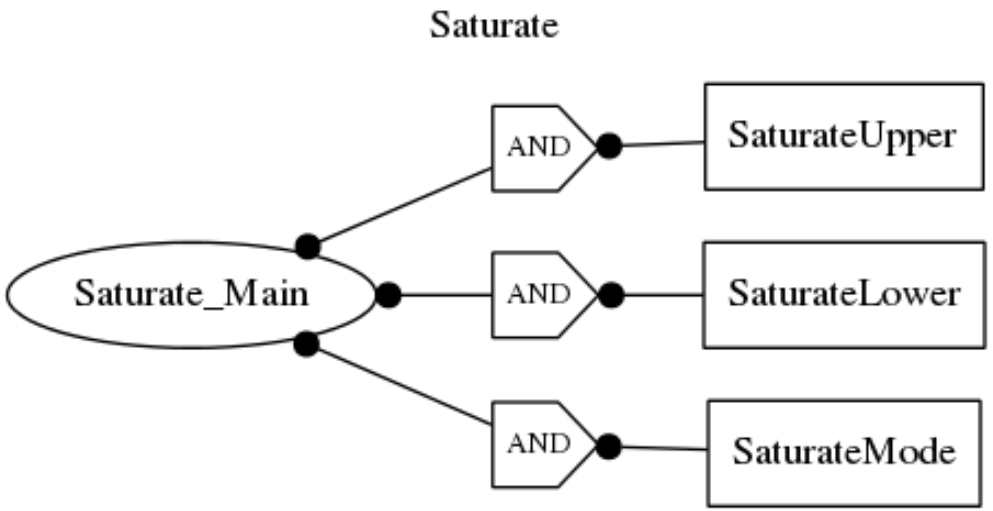
Specification decomposition examples (2)

- The variablity graph helps to understand dependencies between BlockVariants and BlockModes
- Multiple inheritance and reusable specification parts are possible



Specification decomposition examples (3)

- The variablity graph helps to understand dependencies between BlockVariants and BlockModes
- Multiple inheritance and reusable specification parts are possible
- *Dynamic* BlockVariants and BlockModes can be specified in terms of run-time values (as opposed to *static* configuration)



Outline

1. Context
   - Embedded SW development for (safety) critical and high-integrity systems
   - Current work
2. Block libraries in graphical dataflow languages
   - Graphical dataflow languages
3. Specifying block libraries
   - Natural language specification
   - Mathematical specification
   - Model-based specification
4. Applications
   - Tooling
   - Verification of the specification
   - Verification of generated code
   - Other applications

Tooling

MOF/Ecore: Provides tooling capabilities

- Textual editor
- Textual/Form editor
  (Guillaume Babin's work)

and generation capabilities

- Documentation/Requirements
- Verification
- Test cases

Website for the BlockLibrary DSL and applications:

`http://block-library.enseeiht.fr/html`

```
library Simulink_Basic {
  typeDef realDouble double
  typeDef enum onoff {'ON', 'OFF'}
  typeDef array h1type of double [16,2]
  blockType OneDInter {
    variant OneDInter_Root isDynamic {
      out data s1 : double
      in data e1 : double
      parameter exth1 : onoff
    }
    variant H1asParameter extends and OneDInter_Root {
      parameter h1 : h1type {
        struct as fol # forall i:int. (0 <= i < 15) ->
                        h1[i][0] <. h1[i+1][0] #
      }
      struct as eml # exth1 = OFF #
    }
    variant H1asInput extends and OneDInter_Root {
      in data h1 : h1type {
        struct as fol # forall i:int. (0 <= i < 15) ->
                        h1[i][0] <. h1[i+1][0] #
      }
      struct as eml # exth1 = ON #
    }
    mode OneDInter_inf implements uxor  H1asInput,
                                        H1asParameter {
      struct as fol # e1 <. h1[0][0] #
      compute as fol # s1 = h1[0][1] #
    }
    mode OneDInter_sup implements uxor H1asInput,
                                        H1asParameter {
      struct as fol # e1 >=. h1[15][0] /\ e1 >=. h1[0][0] #
      compute as fol # s1 = h1[15][1] #
    }
    mode OneDInter_mid implements uxor H1asInput,
                                        H1asParameter {
      struct
        as fol # e1 <. h1[15][0] /\ e1 >=. h1[0][0] #
        as fol #   exists i:int. 0 <= i <= 14 ->
                     h1[i][0] <=. e1 <. h1[i+1][0] #
      compute as fol
        # s1 = h1[i][1] +.
          ((e1-h1[i][0])*.((h1[i+1][1]-.h1[i][1])/.(h1[i+1][0]-.h1[i][0]))) #
    }
```

Tooling (2)

MOF/Ecore: Model to model or model to text transformations

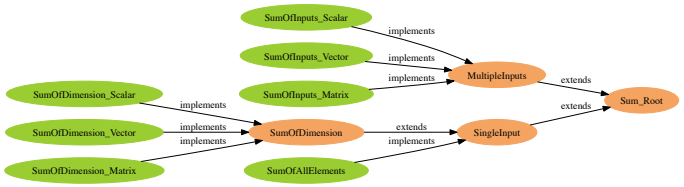
- Documentation/Requirements generation

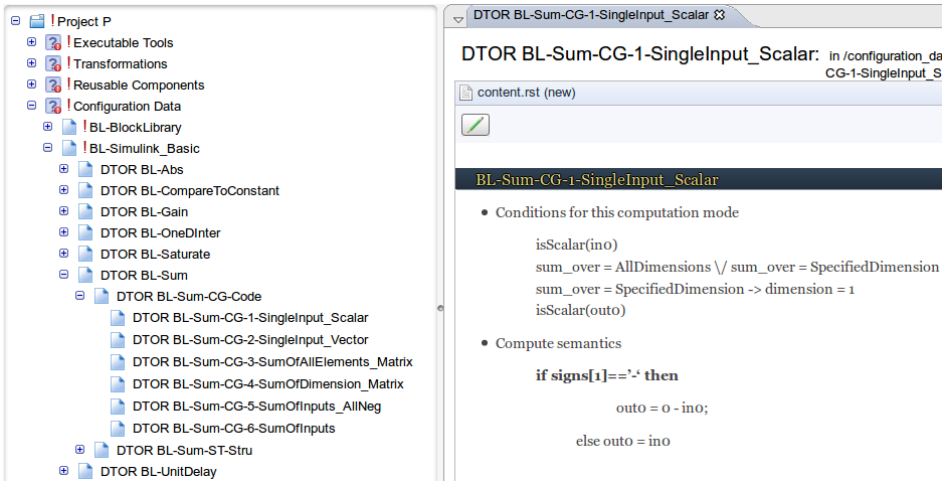
2 Sum

2.1 Parameters

Parameter name	Possible values or DataType
Signs	string
IntegerRoundingMode	['Ceiling', 'Convergent', 'Floor', 'Nearest', 'Round', 'Simplest', 'Zero']
SaturateOnIntegerOverflow	bool
SumOver	['AllDimensions', 'SpecifiedDimension']
Dimension	int

2.2 Modes hierarchy





2.3 Modes

2.3.1 Mode SumOfAllElements

- Variant hierarchy

- Input Ports

Port name	Port rank	Port multiplicity	Port data type
in0	0	0	null

- Output Ports

Structural well-formedness

Completeness and *disjointness* of the block's specification (variability aspect)

- A Signature of a BlockMode is a set of BlockVariants it implements in one configuration
- The Signature of a BlockType is the conjunction of the Signatures of its BlockModes

$$SIG_{BT} = \bigwedge_{i=1}^{n} SIG_{BM_i}$$

- The $\bigoplus$ operator specifies that exactly one boolean expression is true:

$$S = \{a_1, a_2, ..., a_n\}, \forall i \in [1..n], a_i \in \mathcal{B}$$

$$!\bigoplus_{i=1}^{n} S(i) = \left(\bigvee_{i=1}^{n} S(i)\right) \wedge \left(\bigwedge_{i,j=1}^{i,j\leq n, i\neq j} \neg(S(i) \wedge S(j))\right)$$

- Definition of the signature of a BlockMode:

$$SIG_{BM} = \left(\bigwedge_{i=1}^{|SC_{BM}|} SC_{BM}(i)\right) \wedge \left(\left(\bigwedge_{i=1}^{|IV^{\&}_{BM}|} SIG_{IV^{\&}_{BM}(i)}\right) \wedge \left(\bigwedge_{i=1}^{|IV^{!\oplus}_{BM}|} \left(!\bigoplus_{j=1}^{|IV^{!\oplus}_{BM}(i)|} SIG_{IV^{!\oplus}_{BM}(i,j)}\right)\right)\right)$$

- Definition of the signature of a BlockVariant:

$$SIG_{BV} = \left(\bigwedge_{i=1}^{|SC_{BV}|} SC_{BV}(i)\right) \wedge \left(\left(\bigwedge_{i=1}^{|EV^{\&}_{BV}|} SIG_{EV^{\&}_{BV}(i)}\right) \wedge \left(\bigwedge_{i=1}^{|EV^{!\oplus}_{BV}|} \left(!\bigoplus_{j=1}^{|EV^{!\oplus}_{BV}(i)|} SIG_{EV^{!\oplus}_{BV}(i,j)}\right)\right)\right)$$

Completeness
$\bigvee_{i=1}^{n} SIG_{BM_i} = \top$

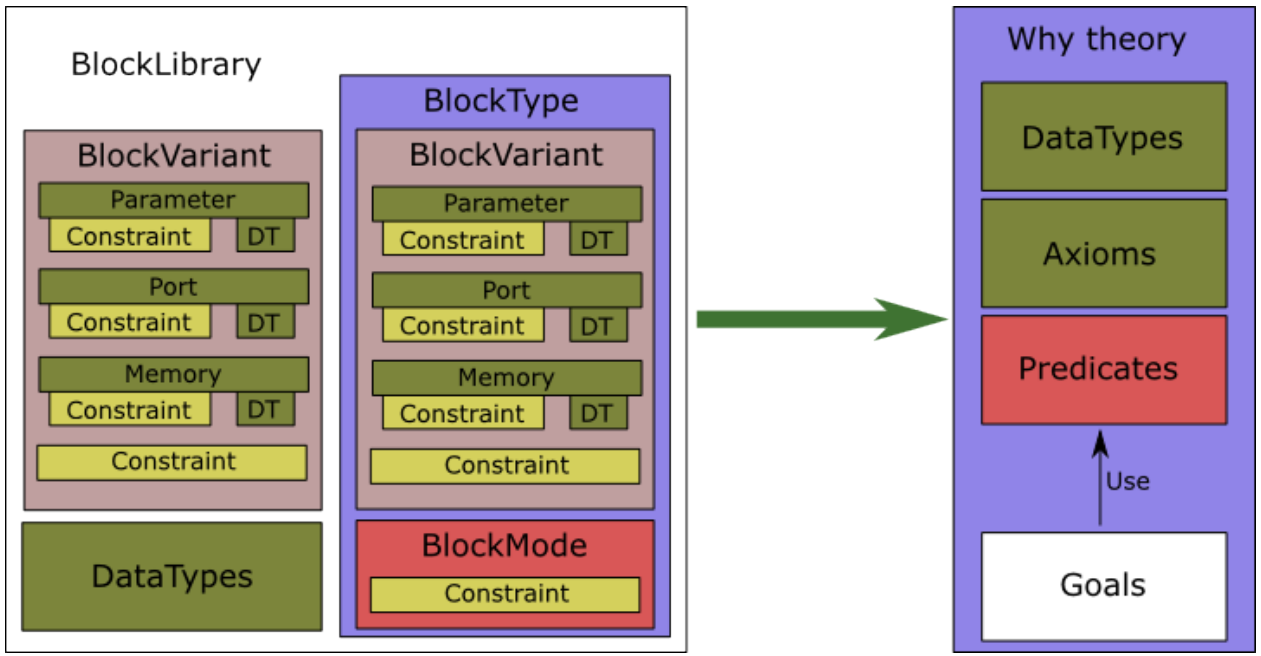
Disjointness
$!\bigoplus_{i=1}^{n} SIG_{BM_i} = \top$

Semantic consistency

- The set of *BlockSignatures* cover all valid (static or dynamic) configurations of a block type
- *BlockSignatures* with identical behaviour are represented by a *BlockMode*
- A *BlockMode* is associated with behaviour specification:
  - spec. of the *init*, *compute* and *update* semantic functions
- These specifications can be given either:
  - *axiomatically* via *pre* and *post*-conditions (in some logic) and/or
  - *operationally* via providing the function *fun* (in an operational language: OCL, Matlab, ...)
- The axiomatic and operational specifications are complementary and allow different usage
- When both are given, they should form consistent Hoare triples (for each semantic function):
  - *{pre} fun {post}*

Formalisation and verification in Why

- The BlockLibrary DSL and its structural well-formedness and semantic consistency properties have been formalised in the Why3 language
- The specific theories corresponding to each block type are automatically generated from a BlockLibrary instance
  - Implemented as an ATL transformation by Guillaume Babin
- Verification of the properties is performed automatically
  (or semi-automatically in the Why3 toolset using available SMT solvers



BlockLibrary instance example (in Why)

```
use import int.Int
use import real.RealInfix
use import datatype.GeneAuto

(* BlockLibrary DataTypes *)
type onoff_type = ON | OFF
type numeric_type = tRealDouble
type h1type_type = array (array (tRealDouble))

(* ParameterTypes constraints *)
axiom param_h1_pre1:
  forall h1:h1type_type. forall i:int. (0 <= i <
       15)
    -> h1[i][0] <. h1[i+1][0]

(* InPortGroup constraints *)
axiom in_port_h1_pre1:
  forall h1:h1type_type. forall i:int. (0 <= i <
       15)
    -> h1[i][0] <. h1[i+1][0]

(* BlockMode Signature checking *)
predicate onedinter_inf (e1:numeric_type) (h1:
      h1type_type)
      (exth1:onoff_type) =
  e1 <. h1[0][0] /\ (exth1 = ON \/ exth1 = OFF)
  /\ not (exth1 = ON /\ exth1 = OFF)
end
```

```
(* Completeness *)
goal OneDInter_completeness:
  forall e1:numeric_type. forall h1:h1type_type.
       forall exth1:onoff_type.
    (onedinter_inf e1 h1 exth1)
     \/ (onedinter_sup e1 h1 exth1)
     \/ (onedinter_mid e1 h1 exth1)
(* Disjointness *)
goal OneDInter_disjointness:
  forall e1:numeric_type. forall h1:h1type_type.
       forall exth1:onoff_type.
    not ((onedinter_inf e1 h1 exth1) /\ (
         onedinter_sup e1 h1 exth1))
     /\ not ((onedinter_inf e1 h1 exth1) /\ (
         onedinter_mid e1 h1 exth1))
     /\ not ((onedinter_sup e1 h1 exth1) /\ (
         onedinter_mid e1 h1 exth1))
```

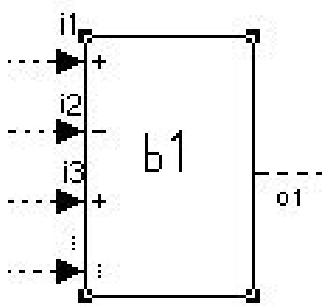
- Result

```
OneDInter.why OneDInter
      OneDInter_completeness_union : Valid
      (0.02s)
OneDInter.why OneDInter
      OneDInter_completeness_disjoint : Valid
       (0.01s)
OneDInter.why OneDInter OneDInter_consistency
      : Valid (0.01s)
```

Going further – Generation of formal annotations[1]

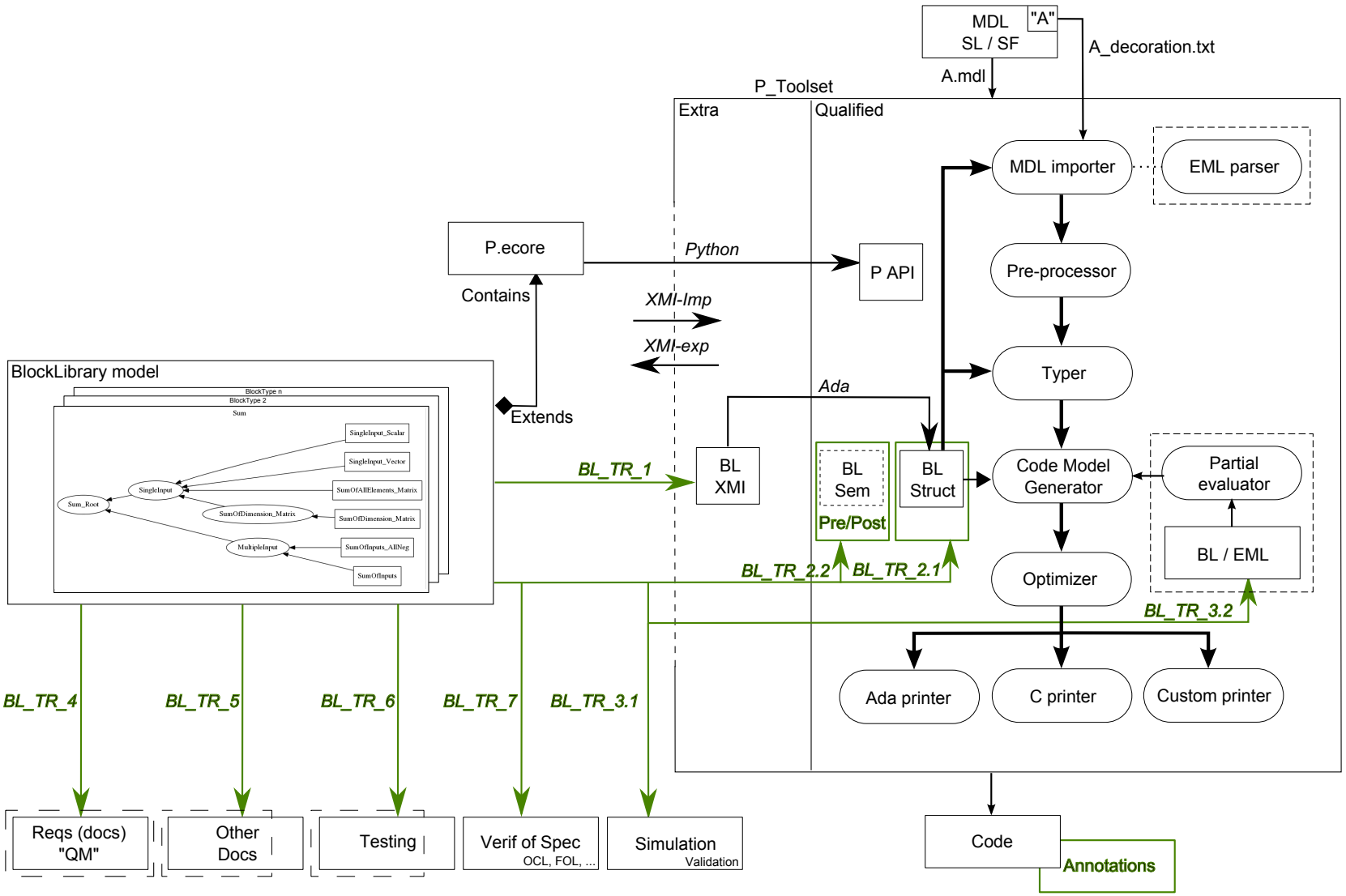
- Formal annotations in some (dedicated) language: ACSL, Ada Spark, Ada 2012 etc. Run-time verification of logical contracts (*inv*, *pre*, *post*)
- Annotations for instructions generated from blocks
  - Conditionals: behavior annotations
  - Loops: loop invariants / variants
  - Simple instructions: asserts
- Annotations for assignments generated from signals
- Annotations for functions generated from (Sub)Systems
  - Pre/post conditions
  - Behavior annotations

```
/*@ loop invariant 0 <= index <= vector_size;
    loop invariant \forall integer m; 0 <= m < index ==>
      b1.o1[m] == b1.i1[m] - b1.i2[m] + ...;
    loop variant vector_size - index; */
for (int index = 0; index < vector_size; index++){
    b1.o1[index] = b1.i1[index] - b1.i2[index] + ...;
}
```

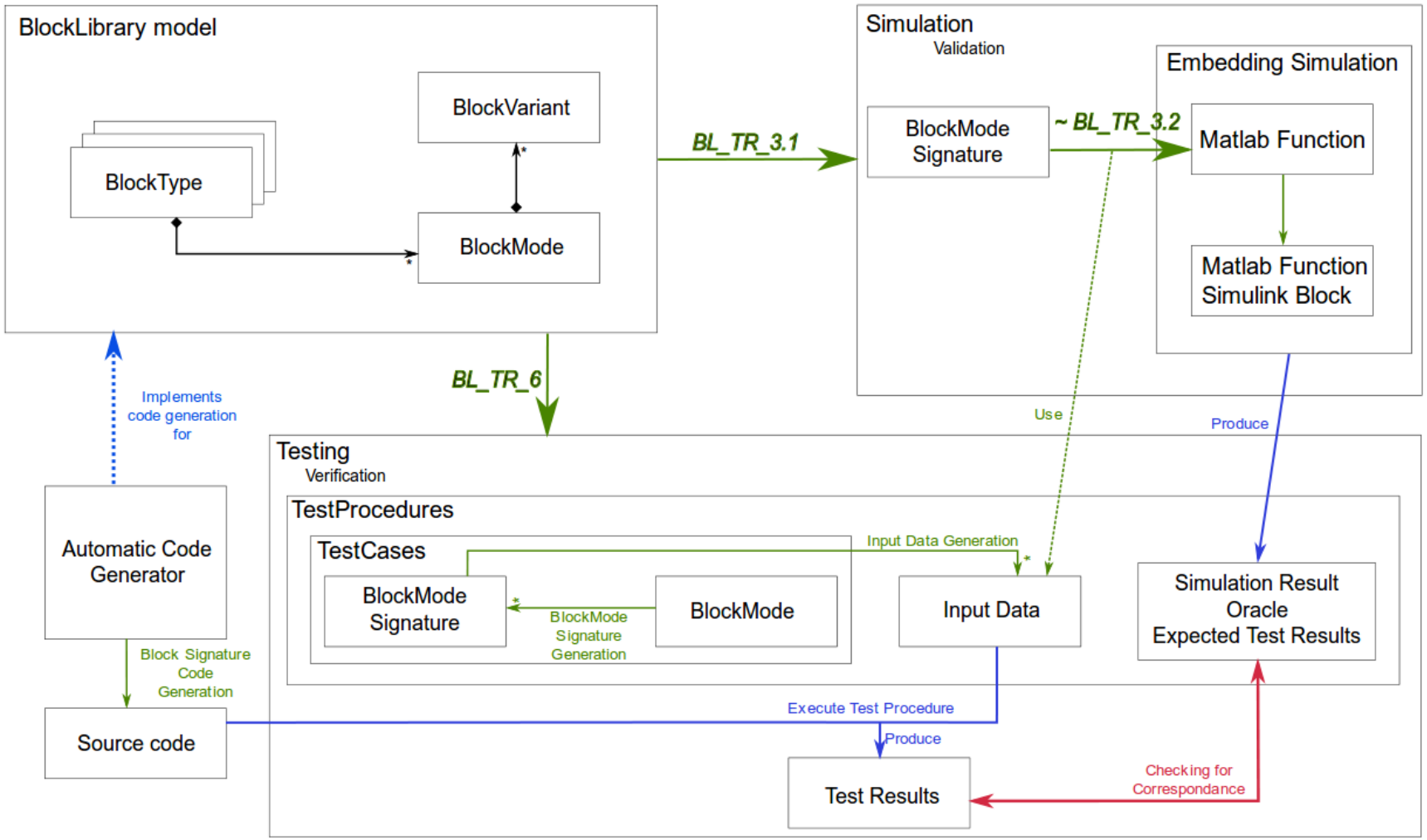


[1] Work by Arnaud Dieumegard

From BlockLibrary to P Toolset – *Possible applications*



Generation of test cases – *One possible approach*



Conclusions

- MDE-based DSL for formal specification of a block library
  - Textual editor with structural verification, documentation generation, etc
- Complexity of block types managed by a *feature modeling*-like approach
- Semantic specification via *constraints* + *behavioural fragments*
- Verification of the specifications' structural and semantic consistency
- Automation of the verification
  - Generation of block type specific theories and verification goals – automatic (ATL)
  - Proving the goals – automatic / semi-automatic (Why + SMT solvers)
- Currently formalised ca 11 Simulink block types (with some limitations)
- Possible to generate observers / test oracles / test cases
- Generation of part of the block library configuration for the P Model Compiler
- But, need to consider also *independence* between:
  - *specification - implementation - verification*
- Work in progress ...

Future work

- Extend the approach and implement more applications
- Refine and explore other formal verification possibilities
- Test case generation
- Integrate more tightly with the P Toolset
- Specify a larger block set
- ...

Thank you!

- A paper on SPLC 2014:
  Software Product Line Conference, Florence (Sept 15-19, 2014)
  `http://www.splc2014.net`
- Block Library DSL and applications:
  `http://block-library.enseeiht.fr/html`
- Project P:
  `http://www.open-do.org/projects/p`
- Gene-Auto:
  `http://www.geneauto.org`