Automatic code generation in certified system development: a model driven specification and verification approach

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

[1]IRIT/ENSEEIHT, Université de Toulouse, France

[2]IB-Krates, Institute of Cybernetics, Tallinn University of Technology, Tallinn, Estonia

27 November 2012

PROJECT P
Model Compilers for Safety-Critical Systems

Plan

Model Compilers for Safety-Critical Systems

Application Context

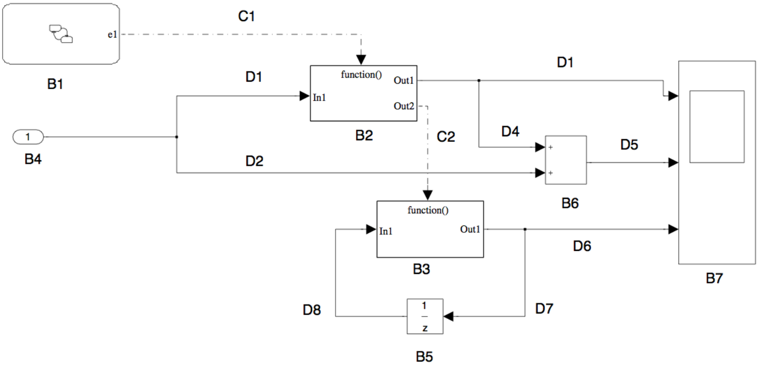
- Context
  - Critical embedded system development
    - Certification (DO-178, ECSS, ISO-26262, IEC-61508)
  - Automatic code generation
    - Tools must be qualified according to DO-330 (adapted from DO-178C)
    - Formal methods use allowed by DO-333 (supplement to DO-178C)
    - Model-based techniques use allowed by DO-331 (supplement to DO-178C)

- Purpose
  - Verification of the functional correctness of the generated code according to the specification
  - Reduce source code verification testing
  - Reduce the risk of anomalies in source code

PROJECT P
Model Compilers for Safety-Critical Systems

Simulink-Stateflow/Scicos

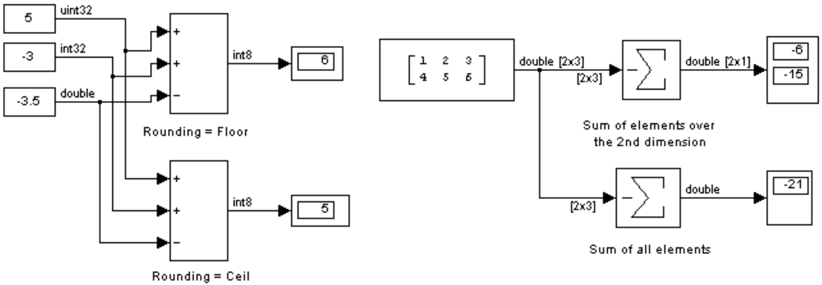
- Discrete control and command algorithm specification
  - Blocks : Elementary (language operation), Hierarchical, User functions
  - Signals between Block Ports (data flow/event based control flow)



PROJECT P
Model Compilers for Safety-Critical Systems

The Sum block specification

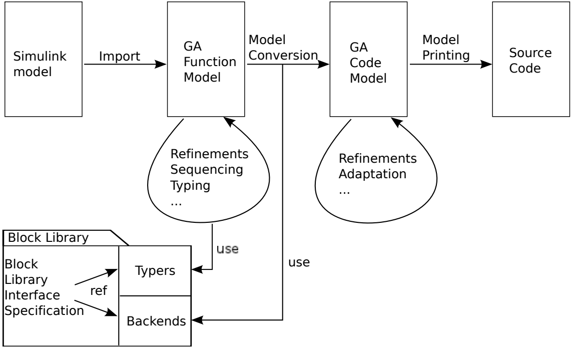
- Simulink version of the Sum block has quite a lot of variability : sum of inputs, sum of elements, sum of dimension, saturation, rounding, scalar expansion, etc
- Original documentation : 19 pages on the meaning of parameters
- Contains many ambiguities and errors



PROJECT P
Model Compilers for Safety-Critical Systems

GeneAuto

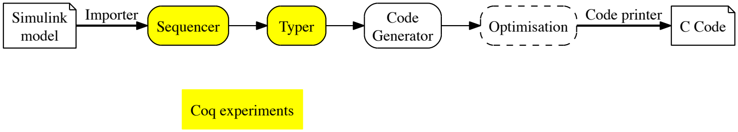
- Open source embedded code generator developed in project GeneAuto. Extended in projects P and Hi-MoCo.
- Takes Simulink-Stateflow/Scicos models as input
- Outputs C, Ada or Java code



PROJECT P
Model Compilers for Safety-Critical Systems

GeneAuto

- Verification process combining formal (proof assistant) and classic approaches (test, proofreading)
- Some of the elementary tools were developed using the proof assistant Coq



Needs to be addressed

- Formal specification of the semantics of the code generator input languages
  - Suited for use by industrial system and software engineers
- Verification (preferably automatic)
  - Soundness and completeness of each elementary block specification
  - Providing artifacts for the generated code verification
- Generated code verification
  - Conformance of elementary blocks specification and implementation

Plan

PROJECT P

Model Compilers for Safety-Critical Systems

Block library specification [1]

- Contains specifications of blocks that are supported by the given code generator configuration
- Block specification consists of
  - Number and type of input and output ports
  - Required state variables and their types
  - Parameters and their types
  - Allowed type combinations
  - Semantics of the block
- Complexity lays in each block semantics and the combination of inputs/outputs/parameters/state variables types and their multiplicities
- Leads to sophisticated variability and complex specification

PROJECT P Model Compilers for Safety-Critical Systems

1. In association with A.Toom (IB-Krates/IRIT)

Mathematical representation

- Pros
  - Easily to read, understand and write by engineers
- Cons
  - General purpose language
  - Too much freedom in specification writing
  - Hard to maintain
  - No dedicated variability management

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

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

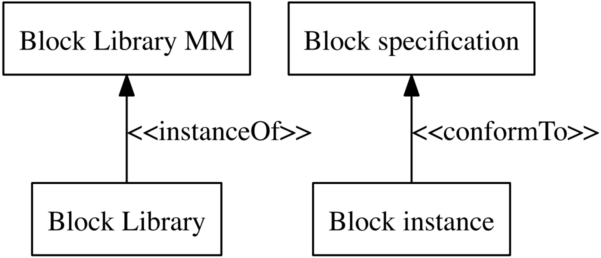
Input must be one of the following :

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

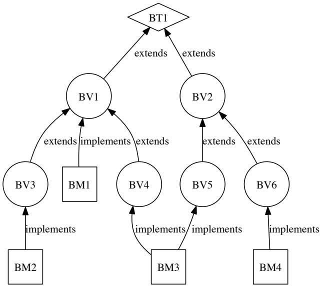
PROJECT P
Model Compilers for Safety-Critical Systems

We found a solution

- General purpose language → use DSLs
- Too much freedom in specification writing → use DSLs
- Hard to maintain → DSLs ease maintenance
- No dedicated variability management → A DSL with variability management



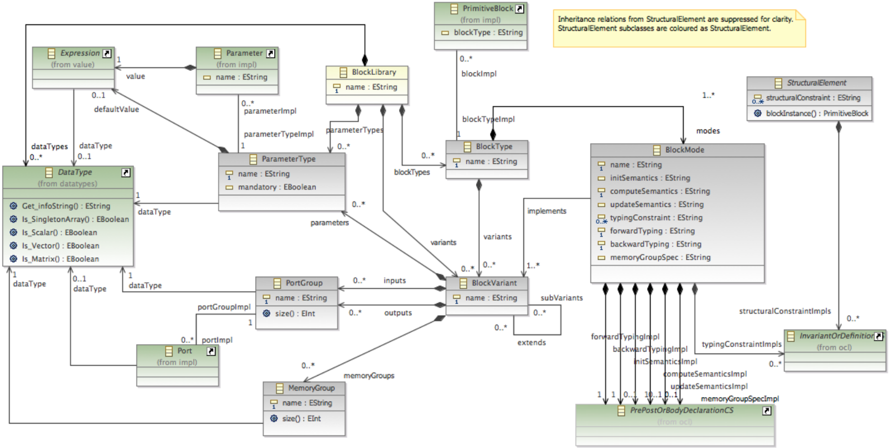
DSL-based specification



- MOF/Ecore-based formalism : well accepted in the industry and standardized
  - Variant : set of related parameters and associated block ports
  - Mode : valid configuration (set of block variants)
  - Structural correctness as OCL invariants on modes and variants
  - Typing/Computational semantics as OCL pre/post conditions on modes

PROJECT P
Model Compilers for Safety-Critical Systems

DSL-based specification - Meta-Model



PROJECT P
Model Compilers for Safety-Critical Systems

Textual specification : Types, Variants

```
library GeneAuto {
    type bool = enum {'true', 'false'};
    type signsChar = enum {'+','-','|'};
    type stringSigns = signsChar[1,256];
    block Sum {
        variant Default {
            parameter Signs : stringSigns ;
            parameter IntegerRoundingMode : enum {'Ceiling', 'Convergent',
                'Floor', 'Nearest', 'Round', 'Simplest', 'Zero'
            } ;
            parameter SaturateOnIntegerOverflow : bool ;
        }
        variant SingleInput extends Default {
            in in0;
            out out0;
            invariant signsSize : Signs.size() = 1
            invariant signsValue : Signs = "+" or Signs = "-"
        }
        variant MultipleInputs extends Default {
            in inGroup[2,-1];
            out out0;
            parameter SumOver : enum {'AllDimensions', 'SpecifiedDimension'};
            parameter Dimension : int;
            invariant signsSize : Signs.size() = inGroup->size()
            invariant signsValue :
                Signs.split("|")->forAll(st|
                    st.split("+")->forAll(st1|
                        st1.split("-")->forAll(st2|st2.size()=0)
                    )
                )
            invariant dimensionValue : Dimension = 1 or Dimension = 2
        }
```

PROJECT P
Model Compilers for Safety-Critical Systems

Textual specification : Modes, Semantics

```
mode SumOfAllElements implements SingleInput {
    invariant SumOverValue : SumOver = 'AllDimensions'
    specification compute output_computation :
        if (in0.isScalar) then
            if (Signs = "+") then
                out0 = in0
            else
                out0 = 0 - in0
        else
            if (in0.isVector) then
                if (Signs = "+") then
                    out0 = in0->sum()
                else
                    out0 = 0 - in0->sum()
            else
                if (in0.isMatrix) then
                    if (Signs = "+") then
                        out0 = in0->collect(in0_elem | in0_elem->sum())->sum()
                    else
                        out0 = in0->collect(in0_elem | 0 - in0_elem->sum())->sum()
}
```

PROJECT P
Model Compilers for Safety-Critical Systems

Associated tools

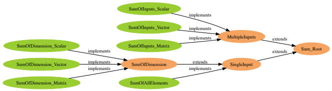
- Textual editor :
  - Auto-completion
  - Static semantics check
- Documentation generation :
  - Mathematical view of the specification
  - Graphical representation of the variants/modes hierarchy
  - Adaptable/easy to maintain

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

3

PROJECT P
Model Compilers for Safety-Critical Systems

Future applications

- Verification of the soundness/completeness of the block specification (variability aspect)
  - OCL constraints on the BlockLibrary metamodel (structural correctness)
  - Generation of inductive data types for verification in a proof assistant
  - Embedded OCL constraints in block specification (dynamic semantics)
    - Generate predicates for dependent types
    - High level properties check in other kinds of proof environments
- Generation of typing and code generation functions from the formal block specification
- Generation of test cases
- Conformance checking between block instance and block library specification

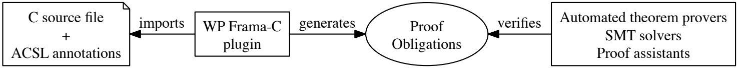
Plan

Questions to answer

- How to verify an automated code generation ?
- How to handle the industrial constraints ?
- How to reduce the costs of the use of formal verification for common engineers ?

Static analysis

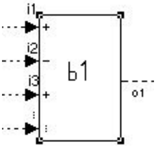
- Static analysis based verification (deductive kind)
- Subset of C compatible with Compcert-C [2]
- Hoare triples using ACSL [3]
  - Pre/Post conditions
  - Variants and Invariants (loops)
  - Annotations for the data flows between blocks
- Frama-C : Weakest preconditions calculus combined with automated SMT solvers (boolean Satisfaction Modulo Theories)



2. http://compcert.inria.fr/
3. ANSI/C Specification Language

Generation of an annotated code for a block

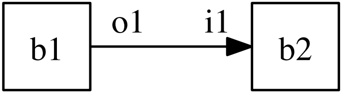
```
/*@ requires \forall integer m; 0 <= m < vector_size ==>
          \valid(&b1.o1+m) && ... ;
    ensures \forall integer m; 0 <= m < vector_size ==>
      b1.o1[m] == b1.i1[m] - b1.i2[m] + ... ; */
{
    /*@ loop invariant 0 <= index <= vector_size;
        loop invariant \forall integer m; 0 <=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



Model Compilers for Safety-Critical Systems

Annotations for the data flows between blocks

```
/*@  ensures \forall integer n; 0 <= n < vector_size ==>
       b2.i1[n] == b1.o1[n];
*/
{
  /*@  loop invariant 0 <= index <= vector_size;
       loop invariant \forall integer n; 0 <= n < index ==>
         b2.i1[n] == b1.o1[n];
       loop variant 2-index;
  */
  for(index=0; index < vector_size; ++index){
    b2.i1[index] = b1.o1[index];
  }
}
```



PROJECT P
Model Compilers for Safety-Critical Systems

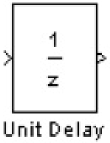
Annotations for a block initialization 1/2

```
typedef struct{
  double d_in[2];
  double d_out[2];
}t_struct;

typedef struct{
  double d_mem[2];
}t_struct_memory;

t_struct b;
t_struct_memory mem;

/*@ requires \valid(&mem.d_mem+(0..1));
    ensures \forall integer n; 0 <= n < 2 ==> mem.d_mem[n] == 0.0;
*/
void init(){
  /*@ loop invariant 0 <= i <= 2;
      loop invariant init_mem_invariant:
        \forall integer n; 0 <= n < i ==> mem.d_mem[n] == 0.0;
      loop variant 2-i;
  */
  for (int i=0;i<2;i++){
    mem.d_mem[i] = 0.0;
  }
}
```

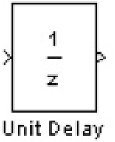


PROJECT P
Model Compilers for Safety-Critical Systems

Annotations for a block initialization 2/2

```
//@ ghost double pre_mem[2];

/*@ requires \valid(&b.d_in+(0..1)) && \valid(&b.d_out+(0..1)) &&
      \valid(&mem.d_mem+(0..1));
    ensures \forall integer n; 0 <= n < 2 ==>
      b.d_out[n] == pre_mem[n] &&
      mem.d_mem[n] == b.d_in[n];
*/
void compute(){
  /*@ loop invariant 0 <= i <= 2;
      loop invariant get_mem_invariant: \forall integer n; 0 <= n < i ==>
        b.d_out[n] == pre_mem[n];
      loop invariant set_mem_invariant: \forall integer n; 0 <= n < i ==>
        mem.d_mem[n] == b.d_in[n];
      loop variant 2-i;
  */
  for (int i=0;i<2;i++){
    b.d_out[i] = mem.d_mem[i];
    //@ ghost pre_mem[i] = mem.d_mem[i];
    mem.d_mem[i] = b.d_in[i];
  }
}
```



Model Compilers for Safety-Critical Systems

Using Frama-C

- WP/Jessie : proof obligations generation
- SMT solvers : assess the satisfaction of the proof obligations

Information | Messages | Console | Properties | WP Proof Obligations

Module	Function	Behavior	Origin	Model	Kind	Alt-Ergo	Coq	Z3	Simplify
sum_vector.c	f		loop invariant (0 ≤ i) ∧ (i ≤ 2);	store	Establishment	✓	✓	✓	✓
sum_vector.c	f		loop invariant (0 ≤ i) ∧ (i ≤ 2);	store	Preservation	✓			
sum_vector.c	f		loop invariant ∀ ℤ m; (i < m) ∧ (m < 2) ⇒ (vect.Sum[m] ≡ \at(vect.Sum[m],Pre));	store	Establishment	✓			
sum_vector.c	f		loop invariant ∀ ℤ m; (i < m) ∧ (m < 2) ⇒ (vect.Sum[m] ≡ \at(vect.Sum[m],Pre));	store	Preservation	✓			
sum_vector.c	f		loop invariant ∀ ℤ m; (0 ≤ m) ∧ (m < i) ⇒ (vect.Sum[m] ≡ vect.Const[m]+vect.Const1[m]);	store	Establishment				✓
sum_vector.c	f		loop invariant ∀ ℤ m; (0 ≤ m) ∧ (m < i) ⇒ (vect.Sum[m] ≡ vect.Const[m]+vect.Const1[m]);	store	Preservation				✓
sum_vector.c	f		loop variant 2-i;	store	Decreasing	✓			
sum_vector.c	f		loop variant 2-i;	store	Positive	✓			
sum_vector.c	f		ensures ∀ ℤ m; (0 ≤ m) ∧ (m < 2) ⇒ (vect.Sum[m] ≡ vect.Const[m]+vect.Const1[m])	store	Proof Obligation				✓

PROJECT P
Model Compilers for Safety-Critical Systems

Main issues

- Partially automatic generation of loops variants and invariants
  - Affordable complexity
  - Verification efficiency
- Introducing invariants in the generation
  - Synthesized by the tools (based on code patterns, pre/post-conditions)
  - Provided by the code generator developers

PROJECT P
Model Compilers
for Safety-Critical Systems

- Work done [4]
  - Extension of GeneAuto in order to handle annotation generation
    - Annotation model for generic annotation expressions
    - Extension of the CPrinter tool for annotation printing
- Work to do
  - Experiment about scaling
  - Extraction of loop invariants from block specification
- Expected result
  - Automated verification of the generated code according to the expected behavior (pre/post conditions for the blocks)

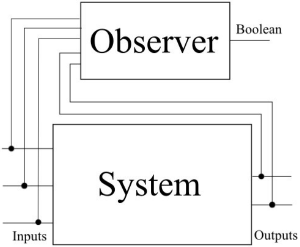
4. In association with T.Wang (GeorgiaTech)

Plan

Model Compilers for Safety-Critical Systems

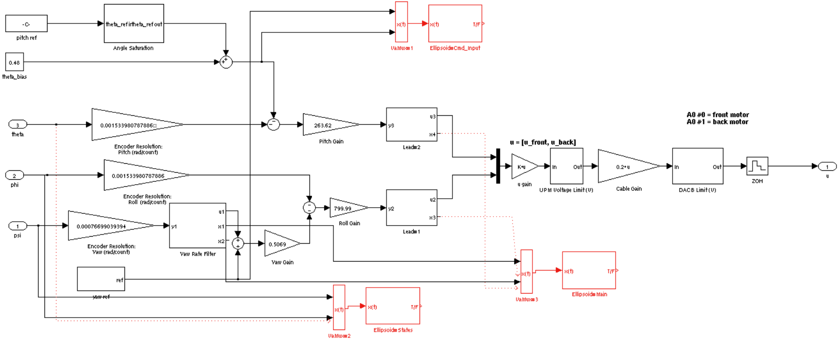
The observer approach

- Expressing the system expected behavior using system design language (system engineer level)
  - Related to execution context, model of the external world, simulations
- Observer ⇒ Behavioral property of the whole system
- Verify the generated verification conditions with Frama-C



Specific annotation blocks

- Specific blocks have been implemented for experimentation with GeneAuto
  - VAMux
    - Create ghost annotations for system data usage in annotations (bridge between system generated code and annotations)
  - Ellipsoid
    - Stability invariant used in control theory (Lyapunov function)



Model Compilers for Safety-Critical Syst

On going work

- Verification-container block to be implemented
  - Holds a sub-system used only for verification (annotation generation)
  - This sub-system is made of elementary Simulink/Scicos blocks
  - Generalisation of the observer block directly specified using domain blocks
- Extension of the block library specification method to handle specification of annotation blocks

Plan

Conclusion

- Block Library specification
  - DSL for formal specification of a block library
  - Complexity of the blocks managed using a feature modeling approach
  - OCL constraints expressions for semantics specification
- Applications
  - Textual editor / Documentation generation
  - Verification of the generated code semantics
    - Pre/Post conditions for the whole system
    - Pre/Post conditions for each block
    - Invariants for blocks and data flows

Future work

- Code generation
  - Invariants/Variants, Pre/Post conditions generation
    - According to code patterns
    - From specification
  - Integration in the development and qualification processes
- Block Library specification
  - Soundness/Correctness verification
  - Generation of
    - Code generator components (backends, typers)
    - Test cases

Thanks for your attention

BlockLibrary : http://dieumegard.perso.enseeiht.fr/blocklibrary
GeneAuto : http://www.geneauto.org
ProjectP : http://www.open-do.org/projects/p

PROJECT P
Model Compilers for Safety-Critical Systems