Software product line for semantic specification of block libraries in dataflow languages

Arnaud Dieumegard, Andres Toom, Marc Pantel

Software product line for semantic specification of block libraries in dataflow languages *

Arnaud Dieumegard[1]
[1]IRIT - ENSEEIHT
Université de Toulouse
2, rue Charles Camichel
31071 Toulouse, France
first.last@enseeiht.fr

Andres Toom[1,2,3]
[2]Institute of Cybernetics
Tallinn University of Technology
Akadeemia tee 21
EE-12618 Tallinn, Estonia

Marc Pantel[1]
[3]IB Krates OU
Maealuse 4
EE-12618 Tallinn, Estonia
first@krates.ee

ABSTRACT

Dataflow modelling languages such as SCADE or Simulink are the de-facto standard for the Model Driven Development of safety critical embedded control and command systems. Software is mainly being produced by Automated Code Generators whose correctness can only be assessed meaningfully if the input language semantics is well known. These semantics share a common part but are mainly defined through block libraries. The writing of a complete formal specification for the block libraries of the usual languages is highly challenging due to the high variability of the structure and semantics of each block. This contribution relates the use of software product line principles in the design of a domain specific language targeting the formal specification of block libraries. It summarises the advantages of this DSL regarding the writing, validation and formal verification of such specifications. These experiments have been carried out in the context of the GeneAuto embedded code generator project targeting Simulink and Scicos; and are being extended and applied in its follow ups projects ProjectP and Hi-MoCo.

*This work has been funded by the FUI Project P, EuroStars project Hi-MoCo and partly by the Campus France and Estonian Research Council's Parrot program and the Estonian Ministry of Education and Research target-financed research theme No. 0140007s12.

Keywords

Model Driven Engineering, Feature Modelling, Formal Specification, Software Qualification, Automated Code Generation, Simulink, Scicos, Xcos, Why3

1. INTRODUCTION

Model Driven Engineering (MDE) advocates the automation of routine transformations from design models to code relying on Automated Code Generators (ACGs). However, these ones are complex software themselves that also need to be verified in order to replace the human activities reliably. This task is further complicated, when both the source and target languages and the transformations don't have complete formal specifications, are constantly evolving and/or the associated tools are closed source.

Dataflow-style languages are widely used for the high-level specification and design of control and command algorithms, which are used in critical embedded systems. The main elements of such languages are *computation nodes* (blocks) and directed *dataflow connections* between them (signals). Variants of the same block are highly reused in the design of many systems and are parameterised and stored in block libraries, which provide an evolving basis of industrial software and know-how. It is then common for key industrials to have their own set of block libraries tailored to their domain and customers.

The current work was started in the context of the GeneAuto[1] project, where an open source embedded code generator for Simulink[2] and Sci-

[1]http://www.geneauto.org/
[2]http://www.mathworks.com/products/simulink/

COS[3] dataflow modelling languages was developed. The work is carried on and extended in follow up projects PROJECTP[4] and HI-MOCO[5]. One of their main goals is to achieve qualification of the tool-set according to the currently most stringent and detailed industrial software development guideline DO-178C that is mainly used in the civil avionics. Therefore, there is a significant focus on the specification and verification of all the aspects of development, including a rigorous specification of the code generator input languages.

This contribution presents a model-based formalisation for the block libraries of common dataflow languages relying on a Domain Specific Language (DSL) *BlockLibrary*. The key aspects of this language and some earlier work were presented in [1] and [2]. Here we develop the methodology further and show its relations with usual Software Product Line (SPL) principles. Then, we focus on the verification of the *BlockLibrary* SPL wrt. structural, variability and semantic properties. The paper is organised as follows. Section 2 gives insights into dataflow languages and block libraries. Section 3 discusses the motivations and alternatives for specifying the problem domain. Section 4 explains the *BlockLibrary* language and Section 5 discusses the related verification criteria and experiments. Finally, we conclude and point out future work.

2. DATAFLOW LANGUAGES

Apart from classical imperative programming that focuses on sequences of instructions, dataflow programs are sets of equations that describe *elementary computations* and *data dependencies* between them. In pure dataflow, an equation is computed as soon as the data that it depends on becomes available (computed). In this paper, we shall refer to these computation nodes as *blocks* and the dataflows between them as *signals*. Blocks can be either *atomic* (opaque) or *hierarchical* (compositions of other blocks and signals). An atomic block is combinatorial if its outputs only depend on the current values of its inputs. An atomic block is sequential if its outputs also depend on its input values from the past. LUSTRE [3] is a well-known textual dataflow language. Similar graphical languages are SCADE[6], SIMULINK and SCICOS. LUSTRE is a fully formal language developed in the academia and successfully transferred to the industry as the semantic backbone of the SCADE tool and language. SIMULINK is a commercial tool largely adopted in the industry and SCICOS is a similar open source alternative. Figure 1 displays an example of a SIMULINK diagram.

All these languages have a similar execution semantics (see for example the one of LUSTRE given in [3]). A program is executed periodically according to a *sample time*. Execution starts from an *init phase* - the state is initialised. At each sample time there is a *compute phase* - all the equations (blocks) are computed, which is followed by an *update phase* - state update is performed for each equation (block) that has an effect on memory (sequential blocks). The core semantics of SIMULINK, SCICOS and several other languages is similar. Often, the dataflow languages provide also some means to control the sequencing of blocks and execute them at different rates, conditionally or on-demand.

The semantics of blocks (computation nodes) is an important extension point of the core language as the functionality and extensibility of block libraries determine the practical usability of the language. Obviously, there is a large number of different computations to be done in a realistic system. But, in order to reduce the number of blocks in the library and ease their maintenance, the semantics of blocks are often tunable by a number of static parameters. These control, for example, the number and data types of inputs/outputs, their dimensions (scalar, vector, matrix) and the amount of memory that the block relies on. We will refer to the *inputs*, *outputs*, *parameters* and *memory* of a block as its *StructuralFeatures*.

As an example, Figure 1 shows some configurations of the *Sum* block from the SIMULINK standard library with different parameters, types, dimensions and number of inputs/outputs. This block can do summation of inputs ("multi input mode"), summation of all the elements of the single input ("single input-full summation mode") or summation of elements along a specified dimension of the single input ("single input-dimension summation mode"). Additional parameters allow to tune the signs at each input port, rounding and other computational details. The full specification of this block in the SIMULINK documentation is around 20 pages of natural language.

Such polymorphic variability makes the writing of

[3] `http://www.scicos.org/`
[4] `http://www.open-do.org/projects/p/`
[5] `http://www.eurekanetwork.org/project/-/id/6037`
[6] `http://www.esterel-technologies.com/products/scade-suite/`

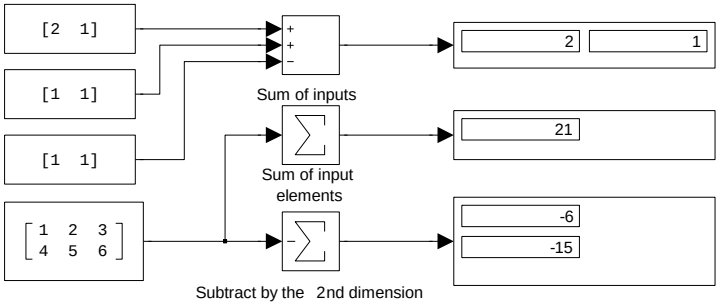

Figure 1: Simulink model with different configurations of the Sum block

a precise and complete specification of the block's semantics, its implementation, validation and verification quite challenging.

3. MOTIVATION FOR A DSL

In this section we discuss the chosen formalism for specifying the blocks structure and semantics. We illustrate both why we do not rely on already existing solutions and the advantages brought by the use of a DSL dedicated to our purpose relying on some SPL principles.

3.1 "Out of the box" solutions

There is a large number of formalisms for creating specifications. We limit our choice to those that are well known, expressive and are likely to be accepted by engineers from the industry. Modelling languages are a good candidate. For instance, the standard *class diagrams* extended with OCL constraints have simple formal semantics, are easy to use and are widely adopted in the industry. Unfortunately, these ones offer too much freedom in the writing of specifications. This complicates the analysis and systematic applicability of the specifications. Class diagrams can be specialised using profiles. However, this is no more an "out of the box" solution but the start of DSL design. In addition to that, there is no dedicated variability management, which would be required for our purpose.

The same elements led us to eliminate general purpose programming languages or databases for block specification. All these offer too little variability management functionality and allow too much freedom in specifications writing.

Feature Modelling was developed specifically for variability management. The methodology was defined in [4] and was given formal semantics in later publications, such as [5, 6]. Feature models are easy to understand and a lot of work has been done on their formal analysis and use. A comprehensive overview is given in [7]. Basic Feature Models (FM) contain features, a set of basic relations (mandatory, optional, alternative) and cross-tree constraints. An example of a feature model for the *Sum block* is provided in Figure 2. Parameters of a block can be represented either as features or as attributes of features (using attributed feature models). It is also often required to define cross-tree constraints to link features, such as the one in Listing 1. In this example, the constraints specify the relation between single matrix input configuration of the block and the parameter that defines the computation kind (see the two last examples in Figure 1).

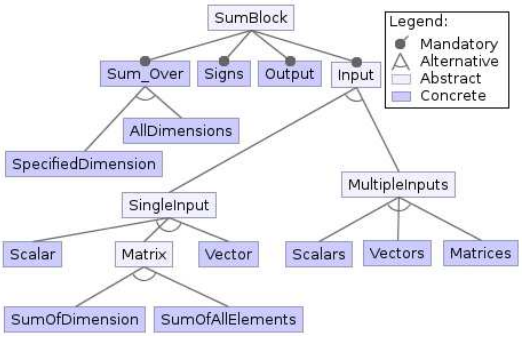

Figure 2: A simple feature model for the Sum block specification

```
SumOfDimension implies SpecifiedDimension
SumOfAllElements implies AllDimensions
(Scalar or Vector) implies AllDimensions
MultipleInputs implies AllDimensions
```

Listing 1: Sum block feature model cross-tree constraints

Such representation allows, for instance, to specify the structural variability of a block. However,

it is quite limited. We need to specify on each feature the selection conditions. For example, we want to choose between features *Vector* and *Matrix*. Depending on the input of a particular block instance we need to give meaning (semantics) to each feature of the model. We also want to be able to express conditions on any other *StructuralFeature* of the block in order for example to restrict their range of values. In this purpose we have to extend the representation, eventually leading to a dedicated DSL with feature modelling elements. A similar conclusion has been reached e.g. in [8] after looking at several alternatives.

Another alternative would be to use Δ-modelling for specifying blocks. In this setting, all the mandatory elements of the block should be defined in the main component and a delta defined for each variant of the block specification. This could be done, but it would still be required to specialise it to our domain and it would become cumbersome, when a block type captures very different behaviours, as in the example described earlier.

3.2 One DSL to rule them all

Feature models are a good starting point for variability management. However, in order to specify the domain more precisely and allow better use of the specification, we have chosen a combined DSL with feature modelling elements. Such an approach has also been promoted in [8] and [9]. A common way to develop tools around DSLs is to use the MDE methodology. We have used the Eclipse Modelling Framework[7] (EMF), which offers rich support for tool development and is very widely accepted both in the modelling community and by industrial users. We have developed a textual editor for our DSL using XTEXT and other tools around it. The DSL and its applications are presented in the next sections.

4. BLOCKLIBRARY MODEL

In MDE, defining a DSL starts from defining the metamodel. The *BlockLibrary* metamodel has been specified in Ecore, an EMF variant of the MOF[8] standard. The metamodel has been completed with OCL [10] constraints to make the structural and semantic constraints more precise. Such constraints can be automatically validated on BlockLibrary instances using standard EMF tools. The main concepts of the metamodel have been presented in Figure 3 and their definitions are given in the next subsections. A detailed

[7] `http://www.eclipse.org/emf/`
[8] `http://www.omg.org/mof/`

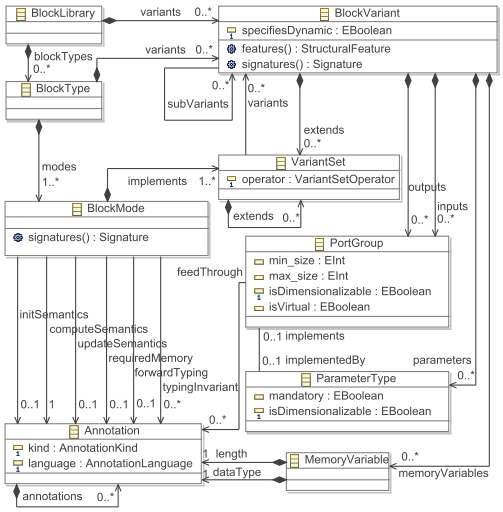

Figure 3: The *BlockLibrary* metamodel

version of the metamodel and related OCL constraints are available from the project's website [11].

4.1 Annotations

Formal annotations play an important role in the *BlockLibrary* DSL. We distinguish several kinds of annotations: *definition* (constant or function), *precondition*, *postconditon*, *invariant* and *mode invariant*. Mode invariants are specific to the DSL and will be explained later. We will also make use of *Hoare Triples*:

DEFINITION 1. *A HoareTriple HT is a 3-tuple of annotations* $(Pre, Fun, Post)$*, where Pre is a precondition, Fun is a behavior definition and Post is a postconditon.*

Annotations can be generally specified in any formal language. We chose to implement support for a subset of OCL as the general constraint and definition language and are working on adding a Matlab-like action language for more convenient specification of the semantics of blocks. For now, our action language allows common constructs found in simple imperative languages.

4.2 Structural elements

We shall present the main structural elements of the *BlockLibrary* DSL bottom-up. All structural elements have a name attribute and can hold local definitions and invariants. These and some less relevant details have been omitted below.

DEFINITION 2. *A ParameterType PT defines a static parameter that a block instance can or must have. It is a 3-tuple* $(\{DT\}, D, M)$*, where:* $\{DT\}$ *is a set of allowed data types; D specifies, whether the parameter is dimensionalisable and M specifies, whether the parameter is mandatory or not.*

DEFINITION 3. *A PortGroup PG defines a group of ports that a block instance can or should have. It is a 4-tuple* (Min, Max, D, V)*, where: Min and Max specify how many of such ports a block instance can have; D specifies, whether the ports are dimensionalisable and V specifies, whether port group is virtual (mapped to a parameter) or not.*

DEFINITION 4. *A MemoryVariable MV defines a state variable that a block instance must have in a given configuration. It is a 2-tuple* (λ_{DT}, λ_L)*, where:* λ_{DT} *is a function that determines the data type of the variable and* λ_L *a function that determines the amount of memory needed (depth of the past used in the sequential block).*

DEFINITION 5. *StructuralFeature SF is one of:*
$$PT \mid PG \mid MV$$

DEFINITION 6. *A BlockVariant BV specifies a variation point of a BlockType. It is a 7-tuple:* $(\{PT\}, \{PG\}_i, \{PG\}_o, \{MV\}, \{VS\}, Dyn, \{Inv_{mode}\})$*, where:* $\{PT\}$ *is a set of ParameterTypes;* $\{PG\}_i$ *a set of input PortGroups;* $\{PG\}_o$ *a set of output PortGroups;* $\{MV\}$ *a set of MemoryVariables;* $\{VS\}$ *a possibly empty set of VariantSet(s) that BV directly extends, Dyn specifies, whether the variant is dynamic and* $\{Inv_{mode}\}$ *are the mode invariants defined in BV.*

DEFINITION 7. *A VariantSet VS is a 3-tuple:* $(\{VS\}_{ext}, \{BV\}, Op)$*, where:* $\{VS\}_{ext}$ *is a possibly empty set of VariantSets that the current VariantSet extends;* $\{BV\}$ *a set of contained BlockVariants and* $Op = and_n \mid xor_n$*, which are the n-ary versions of the and and xor logical relations that specify how the BV are to be combined in the VS.* $n = |\{BV\}|$*. The VS corresponds to a set of constraint edges in the* FODA *terminology and to the consists-of relations in [12].*

DEFINITION 8. *A BlockMode BM represents one possible semantics of the block type. It is a 7-tuple* $(Init, Compute, Update, \{VS\}, \lambda_{MV}, Dyn, \{Inv_{mode}\})$*, where: Init, Compute and Update are HoareTriples HT specifying the respective semantic functions of the block in this mode;* $\{VS\}$ *is a non-empty set of implemented VariantSets;* λ_{MV} *is a function that returns the set of MemoryVariables required by the block in this mode, Dyn specifies, whether the variant is dynamic and* $\{Inv_{mode}\}$ *are the mode invariants defined in BM.*

DEFINITION 9. *A BlockType BT captures the full specification of a block type. It is a 2-tuple:* $(\{BV\}, \{BM\})$*, where:* $\{BV\}$ *is a set of defined BlockVariants and* $\{BM\}$ *a set of defined BlockModes.*

DEFINITION 10. *A BlockLibrary BL is a 2-tuple:* $(\{BT\}, \{BV\})$*, where:* $\{BT\}$ *is a set of defined BlockTypes and* $\{BV\}$ *a set of globally reusable BlockVariants.*

In terms of feature modelling, a *BlockType* can be seen as a root feature. *BlockVariants* and *BlockModes* are sub-features, related to the root feature or other features via *VariantSets*. The *BlockType* specification forms a Directed Acyclic Graph (DAG) with possibly multiple roots (reusable *BlockVariants*). *BlockModes* form the leaves of the DAG.

Mode invariants have a special role. They are used to distinguish between the semantic variation points of a *BlockType*. I.e. they are the selection conditions mentioned in Section 3.1. Mode invariants are specified in terms of static parameters and/or values at the input ports defined or inherited by a *BlockVariant* or *BlockMode*. There are multiple ways to decompose the specification of a *BlockType*. The primary way is to decompose according to the values of some key parameters that control the shape and behaviour of the block. However, more detailed decomposition is also possible by specifying *dynamic BlockVariants* or *BlockModes*, which decompose the behaviour further according to the run-time values of the block's inputs. It is mandatory to have at least one mode invariant in each *BlockVariant* and that all mode invariants in a *BlockType* are consistent.

4.3 BlockType specification examples

Variation graphs of two *BlockType* specifications are given in Figures 4 and 5. They show the structure of the specification of the *Sum* and *Delay* blocks. *BlockVariants* are depicted as ellipses, *BlockModes* as rectangles, **and** *VariantSets* as house shaped nodes and **xor** *VariantSets* as diamond shaped ALT nodes. A fragment of the textual specification of *Sum* has been given in Listing 3.

We shall explain the variation graph of the *Delay* BlockType specification more closely. The purpose of this block is to delay the input signal by either a fixed (*FixedDelay* BlockVariant) or variable bounded (*VarDelay* BlockVariant) amount of time. Depending on the values of static parameters a block instance can have from one to four

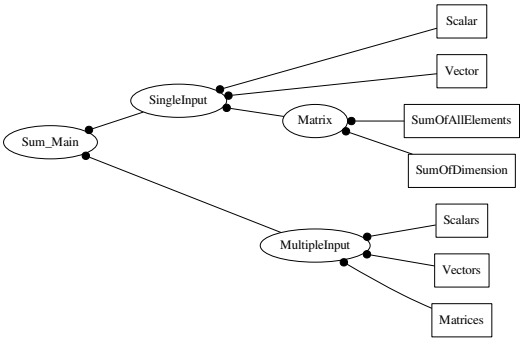

Figure 4: *BlockType* graph of *Sum*

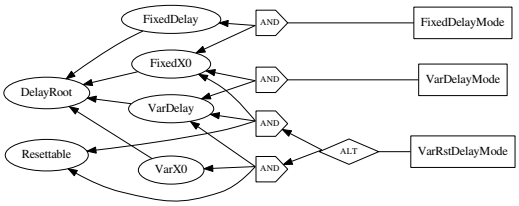

Figure 5: *BlockType* graph of *Delay*

input ports. The first two modes represent configurations with fixed initial value (*FixedXO* BlockVariant). The last mode (*VarRstDelayMode*) defines two additional leaf configurations of the block that both implement the *VarDelay* BlockVariant, which provides the ability to reset the block's output via a dedicated input port. This functionality is included into the specification by implementing the globally reusable *Resettable* BlockVariant, which is further extended by two BlockVariants providing either a fixed (*FixedXO*) or variable initial value (*VarXO*). This alternative choice between the last two configurations is modelled by an *xor* VariantSet.

4.4 Semantics

In the SPL approach, one of the basic tasks is to determine the set of all valid configurations, called *products*. In the *BlockLibrary* DSL a specification instance of a *BlockType* is called a *Signature*. It is composed of a *BlockMode* and one possible *BlockVariant* hierarchy that it implements. The *Signature* is *static*, when its *BlockMode* and all *BlockVariants* are static. Otherwise, it is *dynamic*. Because of multiple inheritance the variation graph is a DAG and it is possible that a *BlockMode* has multiple *Signatures*. Each valid instance of a block must be resolvable to exactly

```
/*@
  requires invariant(input_1); ...
  requires invariant(parameter_1); ...
  requires invariant(memory_1); ...
  requires mode_invariants(BlockVariant_1);
  requires mode_invariants(BlockMode);
  requires pre(Block_Mode);

  ensures post(memory_1); ...
  ensures post(Block_Mode);
*/
void Block_Mode_sigN (
  input_1, ..., input_n,
  output_1, ..., output_n,
  parameter_1, ..., parameter_n,
  memory_1, ..., memory_n)
{
  init_semantics(Block_Mode);
  compute_semantics(Block_Mode);
  update_semantics(Block_Mode);
}
```

Listing 2: Generic specification of a Signature

one static *Signature* or a set of disjoint dynamic *Signatures*.

A *BlockMode* corresponds to the behaviour of the block under the static or dynamic mode invariants for this mode. Its dataflow semantics is given by the Hoare triples of the semantic functions (init, compute, update) specified in the *BlockMode*. The semantics can be given axiomatically by providing the pre- and postconditions and/or operationally by providing the actual function definitions. All the invariants and structural properties inherited by the *BlockMode* transform logically to the primary preconditions of the semantic functions.

Using an imperative code language with annotations (like ACSL [13] or SPARK [14]), a *Signature* can be nicely mapped to a function contract. This function contract can be complemented with the function definition, if the specifier provides also the operational semantics of the block. The generic form of this function contract is given in Listing 2. This transformation completes the semantic specification of our *BlockLibrary* specification language by giving it an interpretation in the formal domain of function contracts.

```
library BlockLibrary {
  type enum TSum_over {AllDimensions,
                       SpecifiedDimension}
  blocktype Sum {
    variant Sum_Main {
      out data Out0 : TArrayDouble
      parameter Signs : TString {
        invariant ocl {
          Signs.value->forAll(s |
```

```
10              (s='+' or s='-') or s='|')
            }
            invariant ocl {
              Signs.value->size() > 0
            }
15        }
          parameter Dimension : TInt {
            invariant ocl {
              Dimension.value = 1 or
              Dimension.value = 2
20          }
          }
          parameter Sum_over : TSum_over
        }

25      variant MultipleInput extends Sum_Main {
          in data In0 : TArrayDouble [2 .. 0]
          modeinvariant ocl {
            Signs.value->select(s |
              s='+' or s='-')->size() =
30              In0.size()
          }
          modeinvariant ocl {
            Sum_over.value =
              !!TSum_over::AllDimensions
35        }
        }
        mode AllInputsScalar implements
          MultipleInput {
            modeinvariant ocl {
40            In0->
                forAll(e| e.value.isScalar())
              }
              definition eml =
                computeAllInputScalar {
45                var out0 = 0;
                  for (var i=1; i<size(In0);
                       i = i + 1) {
                    if (Signs.value[i] == '+')
                      out0 =
50                      out0 + In0[i].value;
                    else
                      out0 =
                        out0 - In0[i].value;
                  }
55                Out0.value = out0;
                }
                compute computeAllInputScalar
            }
    }
60  }
```

Listing 3: Extract of the Sum block textual specification

5. SPECIFICATION CORRECTNESS

BlockLibrary models are should be trustable data that is used as input for multiple development and verification activities. Confidence of the specification can be provided by performing formal verification of it. In this section, we illustrate our verification strategy through the following three aspects: a) syntactical and structural correctness; b) completeness and consistency of the specifications wrt. variability and finally c) correctness and verifiability of the specified block semantics.

5.1 Structural correctness

Structural correctness can be assessed by standard ECORE-MOF compliant tools that check, whether a *BlockLibrary* model conforms to the *BlockLibrary* metamodel and the associated OCL constraints. We have added the required elements to our tooling to ensure this verification.

5.2 Variability correctness

Each *Signature* forms an instance of the specification of a *BlockType*. It contains a distinct combination of *BlockVariants*, *StructuralFeatures* and *Annotations*.

5.2.1 Variability properties

Variability modelling targets the enumeration of all the possible products of a SPL ensuring that each product is unique. *Signatures* should satisfy the same property. We need to take into account the structure of the *BlockLibrary* and the specified constraints. We have split the verification of the set of *Signatures* to 1) **disjointness** - every *Signature* is different from the others; 2) **completeness** - the whole set of *Signatures* always contains a specification that is satisfiable.

5.2.2 Verification technique

The common practice to assess properties of DSLs is to translate its models to a formalism that supports formal verification methods and tools. These methods must be adapted to the kind of properties targeted for the DSL. There exist many formal verification methods and associated tools in the literature. For a non-recent but accurate overview, the reader can refer to [15] (chapter 2). In our case we are working on sophisticated type systems for blocks and want to assess properties based on these types. We decided to rely on theorem proving as it provides good capabilities regarding both automation of the verification and efficiency of the analysis.

We focused on a translation from the *BlockLibrary* language to the WHY3 [16] language. As a formal language, WHY3 provides foundations for formal assessment of properties using automated or assisted theorem proving. The Why platform relies on WHY3 as a pivot language that can be translated to a variety of automatic SMT solvers (Alt-Ergo, Simplify, Z3, CVC3, ...), proof assistants (Coq, PVS, ...) and other verification formalisms. Having bridges to both automatic SMT solvers and proof assistants is an advantage, as it allows to rely on the power and automation capabilities of the SMT solvers in most cases and

on the proof assistants for tackling complex and non-standard problems. Our goal is to automate the verification and avoid the need for proof assistants as much as possible.

A logical specification in WHY3 is written by defining theories and extending already existing theories. WHY3 includes also a general purpose programming language WHYML used as an intermediate language for program verification. The semantics of the language is well defined and the development of the platform is strongly supported by both academic and industrial partners.

5.2.3 BlockLibrary translation

The *BlockLibrary* formalism has two main aspects: 1) structuring the specification data; 2) specifying the properties of interest. Both of these aspects need to be given a translation to a common logical data structure on which formal reasoning can be performed. We rely on the structure of the specification provided by our SPL approach and specifically the *Signature* calculus that extracts all the possible instances of the specification. *Annotations* expressed on *StructuralFeatures* are translated to axioms as they should be true at any time. The other *Annotations* are translated to predicates. The *signature* is then considered as a conjunction of those predicates.

Annotations are written using OCL or our custom simple action language. Each of these languages has been given a translational semantics. For OCL we relied on the semantics of the original specification [10]. We provided an axiomatisation for a large subset of the language operations through dedicated WHY3 theories [11]. An example of translation of an OCL constraint (Listing 3 lines 27-31) is provided in Listing 4. The logical specification of the *select* OCL operator is given in Listing 5. The translational semantics of our custom action language has been given by mapping the imperative constructs (conditionals, loops, variable declaration and variable assignment) to their equivalents in WHYML.

```
predicate multipleinput_modeInv_0
  (out0 : tOut0_Sum_Main_TArrayDouble)
  (in0 : tIn0_MultipleInput_TArrayDouble)
  (signs : tSigns_Sum_Main_TString)
  (sum_over : tSum_over_Sum_Main_TSum_over)
  (dimension : tDimension_Sum_Main_TInt) =
  length (
    select signs.value_pt
           (\ bind_i: tChar.
              bind_i={code=43} \/
              bind_i={code=45})) =
        length in0
```

Listing 4: OCL constraint in Why3

```
function select (l: list 'a)
                (p: HO.pred 'a): list 'a =
  match l with
  | Nil -> Nil
  | Cons hd tl -> if (p hd)
      then Cons hd (select tl p)
      else select tl p
  end

lemma Select_Selected:
  forall l: list 'a, p: HO.pred 'a.
  let res = select l p in
    forall i: int.
      0 <= i < length l -> p res[i]

lemma Select_NotSelected:
  forall l: list 'a, p: HO.pred 'a.
  let res = select l p in
    forall i: int.
      0 <= i < length l ->
      (not p l[i] -> not mem l[i] res)
```

Listing 5: OCL Select operator in Why3

5.2.4 Variability correctness properties

The mode constraint that the i^{th} *Signature* of a *BlockMode* m has to satisfy is formally given in (1). The **completeness** and **disjointness** of a *BlockType* are then respectively (2) and (3).

$$\forall m \in \{BM\}, \forall v_j \in \{BV\}_i, \exists k_j \in \mathbb{N},$$

$$\begin{aligned} Sig_{m,i} = \quad & mode\_inv(m)_1 \wedge \ldots \wedge mode\_inv(m)_j \wedge \\ & mode\_inv(v_1)_1 \wedge \ldots \wedge mode\_inv(v_1)_{k_1} \wedge \\ & mode\_inv(v_p)_1 \wedge \ldots \wedge mode\_inv(v_p)_{k_p} \end{aligned} \quad (1)$$

$$Sig_1 \wedge \ldots \wedge Sig_n \quad (2)$$

$$\forall i, j. i \neq j \Rightarrow \neg(Sig_i \wedge Sig_j) \quad (3)$$

5.2.5 Verification of properties

The *Signature* constraints are translated by the Why platform to the input formalism of SMT solvers. In our experiments the verification of completeness and disjointness of the specifications of all the blocks in our study succeeded fully automatically in very small time.

When a property cannot be proven using the Why platform, it is possible to debug the proof by splitting the properties and relaunching the generation for each sub-property. For the **Completeness** (2) property a simple split provides a goal for each *Signature*. Launching the proof on this set of goals points to the unproven goal(s) and

guides the user to finding the errors in the specification. For the **Disjointness** (3) property, an automatic script can extract and compose a goal for each pair of *Signatures*. With this, it is possible to gain better insight into the specification problem. Both methods can be refined also further. We plan to integrate this into our tooling to give the user useful feedback at the *BlockLibrary* level.

Additional technical details about the transformation of the *BlockLibrary* language, its formalisation using WHY3 theories, axiomatisation of the OCL operations using WHY3 theories and specification of its transformation to WHY3 can be found on the project's website [11] under the *BlockLibrary instance verification* sub-page.

5.3 Semantic correctness

The semantic specification of a block's behaviour is given for each *BlockMode*. An axiomatic semantics should be provided in the form of pre- and postconditions of the expected functional behaviour. Operational semantics in the form of function definitions can also be provided. There might be more than one *Signature* for each *BlockMode*. But, the number of different functional contracts for a *block* specification can be less. This is due to the fact that a set of *Signatures* might have exactly the same set of *BlockVariants*, but different *BlockModes*. Additionally, it is mandatory that the specified behaviours differ according to dynamic block values computed at each execution of the block (value of an input, memory variable...). In this case, only one functional contract is generated with multiple behaviour definitions. These behaviours are distinguished by the *mode invariant(s)* provided in each *BlockMode*. An example of such a specification is given for a block performing one-dimensional interpolation [11]. The function that the block computes depends on the run-time value of the block's input.

Hoare triples can only be assessed with respect to a provided behaviour (the computation between the pre and post conditions). In our case an operational semantics of a *BlockMode* plays this role. A translation of both axiomatic and operational semantics to WHY3 produces a function with its contract and body. The correctness of the operational semantics with respect to the axiomatic one is then verifiable using the Why tooling. Whereas simple functions might be easily proven correct, constructs implying loops and memory must need care, as they require more sophisticated mechanisms like **loop invariant** annotations. There is already a lot of work done in this field, e.g. [17]. We decided not to tackle this problem up to now. Verification can still be done, if the invariants are provided during the specification, in the generated WHY3 specification or when there is no need for such complementary invariants.

6. RELATED WORKS

In [18] the authors use FM for *Feature-Oriented Software Development*. Their approach is to structure features (packages, classes, methods and attributes) using FM. Their definition of a Feature: *a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option* is very close to the one we use in our work. Our addition is to explicitly define the semantics of the program in the FM via the *BlockMode* features. This allows to fully specify the program in a single data structure.

The nature of FM makes its analysis through SAT solving very convenient and efficient. This approach is developed and used in multiple works among which are [5], [19] and [6]. In these works, FM are translated to a SAT solver formalism for verification of the structural correctness of the FM and their conformance to the semantics. As features are not fully specified and are not given semantics, this verification remains focused on the FM and not on the meaning of its features. In our work, the selection of features is done according to the relations between features, but the correctness of this selection is assessed thanks to the properties specified for each feature. This adds semantic meaning to the feature selection, which is mandatory for our use.

7. CONCLUSIONS AND FUTURE WORK

This contribution presented a DSL and associated tools for the specification, validation and formal verification of block libraries, a key aspect in data flow modelling languages for safety critical embedded systems. The DSL relies on SPL principles in order to harness the huge variability in the structure and semantics of block libraries in languages like SIMULINK and SCICOS. We have shown how we rely on formal verification techniques and the WHY3 platform in order to verify semantic properties of the specification.

We plan to refine the whole formalisation, ac-

tion languages and improve feedback to the user when a proof cannot be performed. We plan to also improve the efficiency of the verification of the block's semantics by the introduction of loop invariant generation. And finally, further experimentation on industrial use cases from ProjectP and Hi-MoCo projects will be conducted to analyse the impact of the use of such formal specification in qualified software development.

8. REFERENCES

[1] A. Dieumegard, A. Toom, and M. Pantel, "Model-based formal specification of a DSL library for a qualified code generator," in *Proceedings of the 12th Workshop on OCL and Textual Modelling*. New York, NY, USA: ACM, 2012, pp. 61–62.

[2] A. Dieumegard, A. Toom, and M.Pantel, "Formal specification of block libraries in dataflow languages," in *Embedded Real-Time Software and Systems, ERTS2*, 2014. [Online]. Available: http://www.erts2014.org/Site/0R4UXE94/Fichier/erts2014_6D4.pdf

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, September 1991.

[4] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.

[5] J. Sun, H. Zhang, Y. Fang, and H. Wang, "Formal semantics and verification for feature modeling," in *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, June 2005, pp. 303–312.

[6] R. Gheyi, T. Massoni, and P. Borba, "A theory for feature models in Alloy," in *In: Proceedings of the 1st Alloy Workshop*, 2006, pp. 71–80.

[7] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Information Systems*, vol. 35, no. 6, pp. 615–636, 2010.

[8] O. Haugen, B. Moller-Pedersen, J. Oldevik, G. Olsen, and A. Svendsen, "Adding standardized variability to domain specific languages," in *Software Product Line Conference, 2008. SPLC '08. 12th International*, Sept 2008, pp. 139–148.

[9] M. Voelter and E. Visser, "Product line engineering using domain-specific languages," in *Software Product Line Conference (SPLC), 2011 15th International*, Aug 2011, pp. 70–79.

[10] OMG. OCL specification. [Online]. Available: http://www.omg.org/spec/OCL/

[11] Blocklibrary repository. [Online]. Available: http://dieumegard.perso.enseeiht.fr/bl/

[12] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps, "Generic semantics of feature diagrams," *Comput. Netw.*, vol. 51, no. 2, pp. 456–479, Feb. 2007.

[13] ACSL: ANSI/ISO C Specification Language. [Online]. Available: http://frama-c.com/download/acsl.pdf

[14] Spark Ada GPL edition. [Online]. Available: http://libre.adacore.com/tools/spark-gpl-edition/

[15] J. B. Almeida, M. J. Frade, and J. S. Pinto, *Rigorous software development : an introduction to program verification*. London: Springer, 2011.

[16] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, "Why3: Shepherd Your Herd of Provers," in *Boogie 2011: First International Workshop on Intermediate Verification Languages*, Wroclaw, Poland, 2011, pp. 53–64.

[17] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using gröbner bases," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04. New York, NY, USA: ACM, 2004, pp. 318–329. [Online]. Available: http://doi.acm.org/10.1145/964001.964028

[18] S. Apel, C. Lengauer, B. Möller, and C. Kästner, "An algebra for features and feature composition," in *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, ser. AMAST 2008. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 36–50.

[19] D. Batory, "Feature models, grammars, and propositional formulas," in *Proceedings of the 9th International Conference on Software Product Lines*, ser. SPLC'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 7–20.