Formal specification of block libraries in dataflow languages

Arnaud Dieumegard, Andres Toom, Marc Pantel

► **To cite this version:**

Arnaud Dieumegard, Andres Toom, Marc Pantel. Formal specification of block libraries in dataflow languages. Embedded Real Time Software and Systems (ERTS 2014), Feb 2014, Toulouse, France. hal-02272313

HAL Id: hal-02272313

https://hal.science/hal-02272313

Submitted on 27 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Formal specification of block libraries in dataflow languages ⋆

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

[1] IRIT - ENSEEIHT, Université de Toulouse, 2, rue Charles Camichel, 31071 Toulouse Cedex, France
FirstName.LastName@enseeiht.fr
[2] Institute of Cybernetics at Tallinn University of Technology, Akadeemia tee 21, EE-12618 Tallinn, Estonia
[3] IB Krates OU, Maealuse 4, EE-12618 Tallinn, Estonia
FirstName@krates.ee

Abstract. Graphical dataflow-style modeling languages like Simulink and Scicos are widely used in the development of embedded control systems as high-level engineering languages. A significant part of their modeling power is captured in function block libraries. In this paper we present an on-going work on the model-based formalisation of such libraries, which intends to bridge the gaps between the different parts of the development process: high-level requirements, design, implementation and verification. Our approach is based on a specification domain specific language (DSL), which captures the variability of blocks through a software product line approach. We have defined translations to other languages like the Why3 language for verification, different documentation formats, code generator configuration files, etc. These experiments have been carried out in the context of the GeneAuto embedded code generator project and are being extended and applied in its successor projects ProjectP and Hi-MoCo.

Keywords: model driven engineering, feature modeling, formal specification, software qualification, automatic code generation, Simulink, Scicos, Xcos, Why3

1 Introduction

Embedded software plays an essential role in high integrity safety critical systems, for human related activities such as transportation, healthcare or energy domains. Their use is of strategic importance for industrial actors with regard to the impacts on safety and performance. Overall, complexity of such systems is ever-increasing and so are the costs of development and verification. Manual coding of semi-formal specifications is error-prone and the usual verification by proofreading or testing is costly and often non-exhaustive. Those problems are addressed by the development of Automatic code generators (ACGs). Although such tools are complicated software, their correct operation and use need to be ensured. This may be hard to tackle, especially when the tool does not have formal specification and/or is closed source.

Many guidelines and standards are meant to ensure the quality and reliability of high integrity systems. One of the most advanced and stringent ones is DO-178 dedicated to software development in civil avionics. These guidelines provide a set of objectives to be reached in order to prevent flaws in the system that may lead to catastrophic consequences. One of the major

⋆ 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.

evolutions in the recent version of this guideline (DO-178C) is the account being taken of new development techniques like: i) model-based development and verification; ii) object-orientation; iii) formal methods; and iv) the development of tools used to facilitate some activities of the system design, development and verification process.

The current work was started in the context of the GeneAuto[4] project, where an open source embedded code generator for Simulink[5] and Scicos[6] like dataflow modeling languages was developed. The work is carried on and extended in GeneAuto successor projects ProjectP[7] and Hi-MoCo[8]. Since one of the intended objectives of these projects is to achieve qualification according to the DO-178C guideline, there is a significant focus on the specification and verification of all aspects of the development, including a rigorous specification of the code generator input languages.

Dataflow languages are widely used in the design of embedded control and command systems. These languages are mostly made up of computational nodes (blocks) and directed connection between them denoting *data flow* (signals). Some *elementary* blocks that are highly reused in systems design are stored in libraries. This eases the reuse of already specified blocks and stores industrial knowhow regarding the design of critical systems. It is common for large industrial organisations or service providers to develop their own complete block libraries that are tailored to the specific needs of their domain and/or customers.

In this paper we present a model-based formalisation of block libraries using as a *BlockLibrary* Domain specific Language (DSL) and its applications in the software development and verification process. An earlier version of the work has been presented in [4]. Here we shall develop the methodology further and show its relations and differences from a more common Software Product Line (SPL) approach. We shall also specify some formal correctness properties of *BlockLibrary* models and present an approach for verifying them. Finally, we shall present some applications of *BlockLibrary* specifications for the generation of different artefacts used in the software development lifecycle.

2 Block libraries in dataflow languages

Dataflow languages are widely used by systems engineers as they are convenient for modeling mathematical control laws. They allow to model both continuous and discrete time dynamic systems. As our work is closely related to embedded code generation we restrict our study to discrete time synchronous dataflow models only.

Block libraries are an important extension point of the basic dataflow languages. They enable software reuse and hence capture a lot of the actual modelling power. Most of the basic blocks are in fact quite complex as their semantics can vary according to: a) parameters; b) input/output types (double, int, . . .); c) input/output dimensionality (scalar, vector, . . .); d) number of inputs/outputs; e) memory management. In the following, we will refer to those elements as *structural features* of a block.

An example of such variability is given in Figure 1 describing three of the allowed configurations of the same block type *Sum* from the Simulink standard library. The block can do summation of inputs ("multi input mode"), summation of all the input elements ("single input-full summation mode") or summation of elements along specified dimension ("single input-dimension

[4] http://www.geneauto.org/
[5] http://www.mathworks.com/products/simulink/
[6] http://www.scicos.org/
[7] http://www.open-do.org/projects/p/
[8] http://www.eurekanetwork.org/project/-/id/6037

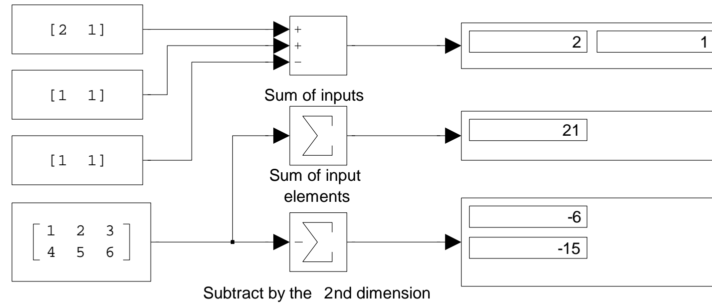

Fig. 1. Simulink model with different configurations of the Sum block

summation mode"). Furthermore, the signs at each port, rounding algorithm and other computational details can be tuned. This configuration is stored in the parameter values of each block instance.

Such polymorphic variability makes precise specification of the block's semantics as well as its implementation and verification challenging. In the following section we describe a formal model-based specification of block types that allows to manage such variability.

3 *BlockLibrary* specification language

3.1 Elements of the *BlockLibrary* language

We have defined a Domain Specific Language (DSL) named *BlockLibrary* for the specification of block libraries. This DSL has several commonalities with the *feature modelling* or FODA [6] approach from the software product line (SPL) engineering domain.

Our DSL is based on a custom metamodel that is tailored to formally specify the properties of interest. The metamodel is specified as an Ecore[9] (a variant of the OMG MOF[10] standard) model. We have followed the common practice to complement the metamodel with OCL[11] constraints in order to specify more detailed structural correctness properties. We show in figure 2 an overview of the *BlockLibrary* metamodel. To avoid visual clutter, most of the inheritance relations have been represented with colour-coding instead of links. Abstract classes (class name in *italics*) are superclasses of the similarly colored concrete classes (class name in normal style).

The main elements of this metamodel are:

- *BlockLibrary* (not drawn): A container of *BlockTypes*, but also common definitions like data types and reusable *BlockVariants*.
- *BlockType*: A container for the specification components of one block type. The specification is structured into *BlockVariants* and *BlockModes*.
- *BlockVariant*: A node that encapsulates some structural or semantic variation point of the *BlockType*. *BlockVariants* can inherit from other *BlockVariants*.
- *BlockMode*: A node that encapsulates a semantically distinct and complete configuration of a *BlockType*. A *BlockMode* always implements one or more *BlockVariants*. The computational semantics of a *BlockModes* is divided according to the common semantic operations in dataflow languages:
  - memory and output initialization

[9] http://www.eclipse.org/emf/
[10] http://www.omg.org/mof/
[11] http://www.omg.org/spec/OCL/

- computation of outputs based on the current inputs and block's memory
  - updating of the block's memory based on the current inputs and previous memory
- *StructuralFeature*: We model the following structural features
  - *ParameterType*: Specification of a configuration parameter, its data range and constraints
  - *PortGroup*: Specification of a group of input or output ports. In the actual block instance there can be one or more ports that correspond to a *PortGroup*, but they all perform similar function.
  - *MemoryVariable*: Specification of a unit of memory that the *BlockVariant* requires for one purpose. The *MemoryVariable* is sized according to the maximal length of history that it must maintain.
- *Annotation*: All specification elements can have formal annotations. We distinguish the following annotation kinds: *definition* (constant or function), *invariant*, *mode invariant*, *precondition* and *postconditon*. The mode invariants are of specific interest, since they are used in the *mode resolution* process. Annotations can be specified in different languages. The choice of the annotation language is application specific.

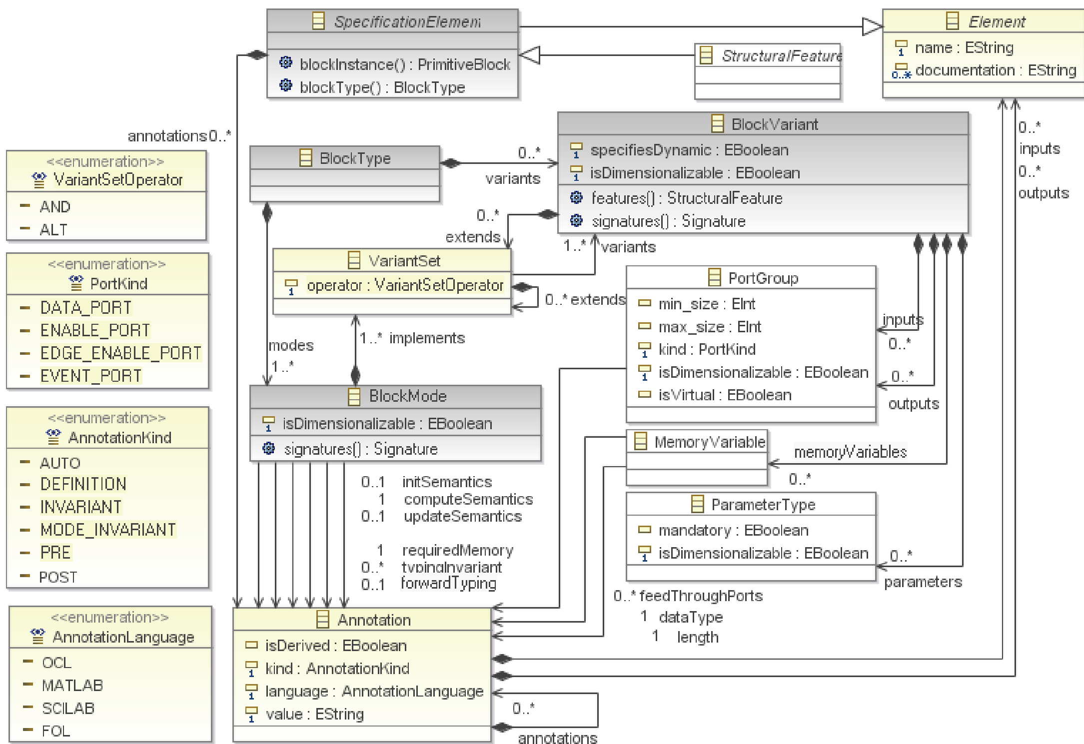

Fig. 2. Simplified version of the *BlockLibrary* metamodel

Specification of block library instances that conform to such a metamodel can be done in many ways. One can define either a concrete textual syntax or develop a graphical editor. We have chosen the first approach. In Listing 1.1 there is a fragment of such a textual specification. The example is based on another block type from the SIMULINK standard library called *Delay*.

This block delays the input signal by either a fixed or variable (bounded) amount of time. A *Delay* block can be configured to support variable delay, external reset and external initial value. Depending on that it has from one to four input ports. These are not the only configuration options, but splitting the block's semantics to *BlockVariants* and *BlockModes* along these structural parameters turned out to be a good compromise.

The inner structure of a *BlockType* (*BlockVariants* and *BlockModes*) forms a directed acyclic graph (DAG). The intermediate nodes are *BlockVariants*. *BlockModes* occur as leafs of the DAG and they contain the specification of the computational semantics of the *BlockType* in one valid configuration. Since this structure is an important part of the specification, we have generated a simple graphical representation from the textual specification. An example can be seen in Figure 3. *BlockVariants* are represented as ellipses, *BlockModes* as rectangles and relations between them as AND and ALT nodes. The variant graph for *Delay* has two root variants: *DelayRoot* that contains the parameters that exist in each configuration of *Delay* and a generic *Resettable* variant that provides a Reset port, associated memory and a TriggerReset function. This variant is included in the VarRstDelayMode. It can be seen from the figure that there are two *BlockVariant* configurations that the VarRstDelayMode implements. In one of those, the initial value X0 is supplied via a parameter (*BlockVariant* FixedX0) and in the other via an input port (*BlockVariant* VarX0). In our metamodel these two cases can be represented with one *BlockMode* by modelling the X0 parameter as a virtual port.

These alternative configurations of *BlockVariants* that a *BlockMode* implements are called *BlockVariant Signatures*. A block instance is matched with *BlockMode* by computing the *mode invariants* wrt. the parameter values of the block instance. We qualify a *BlockType* specification of:

- *Complete*, when for all valid block instances there exists a corresponding *BlockMode* and *BlockVariant Signature*
- *Consistent*, when for all valid block instances there exists exactly one corresponding *BlockMode* and *BlockVariant Signature*.

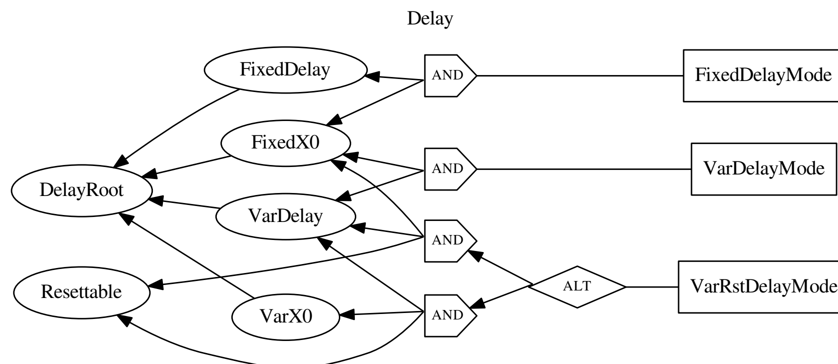

Fig. 3. Variation graph of the *Delay* block type

3.2 Relation to Feature Models

BlockLibrary instances represent a generic block specification as a graph structure closely related to the *Feature Models* (FM) [6] used in the Software Product Line (SPL) engineering approach.

Such models are made of features arranged in a hierarchical tree-like way. A FM allows to specify all the possible features that a product can implement, the hierarchy between features and conditions holding between features (some combinations of features may be exclusive). FM can be extended in order to include more informations for each feature of the model (attributed FM). This allows to be more specific and to define more precisely the features. Finally, in an extended FM, it is both possible to define constraints between features and in a more fine grained setting between attributes of features.

In the SPL terminology a *BlockType* corresponds to SPL (a software product line) and a *BlockMode* to a (valid) product. All other elements are generally *features*. Our approach is more related to the extended FM as we define *StructuralFeatures* (parameters, inputs/outputs and memory variables) that have a datatype (called the domain of the feature in attibuted FM) and formal *Annotations*. Most of the time, in FM, constraints specify conditional existence of features. We propose to use them as constraints on the value and behavior of the *StructuralFeatures* themselves. We also define different classes of *Elements* with different attributes, relations and roles in the *BlockLibrary* metamodel. We would need to add similar structure to a FM.

A *BlockType* graph can be nonetheless converted to a FM. Here is an informal specification of this transformation:

1. *BlockType* is transformed to a FM root feature (BT_F)
2. *BlockModes* are transformed to alternative subfeatures (BM_F) of the root feature
3. *BlockVariants* are transformed to features (BV_F)
4. *StructuralFeatures* are transformed as features (F_F)
5. BV_F and F_F are linked with mandatory relations
6. Mandatory or alternative relations are added to the FM between the BV_F-s and between the BV_F and BM_F to represent the relations in the *BlockLibrary*

However, such a derived FM isn't a typical feature model. It looks like top-down modeling, which is less natural than the bottom-up modeling promoted by the *BlockLibrary* DSL. In that case the elements (features) that the *BlockMode* semantic definitions require are specified incrementally and the specifier has clear visibility of the full set of features that can be used in a *BlockMode*. Conflicting and alternative parameter values need to be modeled as distinct *BlockVariants* and/or implemented in distinct *BlockModes*.

A number of analysis techniques for FM have been developed in the SPL comunity over the past 20 years. A summary of them has been presented in [1]. Not all those are directly applicable for the *BlockLibrary* DSL. We comment on some of the more relevant ones below:

- Compute the set of all products: Allows to find all the possible *BlockModes*. This analysis is used in the verification of completeness and consistency of a *BlockType* specification.
- Assess the validity of a product: Decide whether there is a *BlockMode* that corresponds to a given block instance. This is part of the process that we call *mode resolution*.
- Detect anomalies in the model: Find if some *BlockVariants* are not implemented (dead features). This check is implemented by OCL constraints on the *BlockLibrary* metamodel.
- Refactoring: Help to factorize the elements of a specification by finding possible sets of factorizable elements. This could be used to provide hints to the *BlockLibrary* specifier to simplify the specification.
- Filtering: Find the set of *StructuralFeatures* or *Annotations* (e.g. invariants) used in a configuration (e.g. *BlockMode* or *BlockVariant*).

3.3 Correctness of a BlockLibrary specification

Structural correctness of a *BlockLibrary* specifications can be verified by standard Ecore-MOF compliant tools by checking whether a *BlockLibrary* model conforms to the *BlockLibrary* metamodel and its additional OCL constraints.

More complex properties are related to the *completeness* and *consistency* of the specification. These concepts were informally introduced in Section 3. In order to specify and verify such properties we have implemented a model transformation taking as input a *BlockLibrary* instance and translating it to a Why3[2] theory: *StructuralFeatures* and their annotations are translated to axioms; *BlockVariants* are translated to predicates and finally the properties are translated to verification goals. This generated Why3 theory is then fed to an SMT solver that will solve and discharge it. Using this approach, we have successfully managed to verify these properties on several block instances. Technical details, examples and formalizations of this transformation can be found on our development page[12].

4 Applications of the *BlockLibrary* DSL

A *BlockLibrary* model contains information that can be used for different purposes. We will briefly explain here the motivation for different applications and the experiments we have done in the context of the development of automatic code generators and their verification.

Specification editor: In the state of the art MDE methodology, it is common to automatically generate editors for DSL-s based on metamodels, making it easy for the domain engineers to develop syntactically and to some extent also semantically sound specifications. We have developed a textual editor that integrates the *BlockLibrary* elements and action/constraint languages. The approach is similar to [5, 3]. These editors allow to verify static properties on the model and allow to perform a first verification of the specification while writing. The verification is performed according to the metamodel structure and associated OCL constraints. An extract of the *Delay* block type specification in a textual form is provided in Listing 1.1.

Documentation generation: From the formal *BlockLibrary* specification, it is possible to generate clear and rich user documentation. Automatic generation helps to maintain the consistency of the documentation, which is essential in high integrity systems. One of the main problems of documentation generation is that they suffer from a lack of automated verification techniques. It is therefore difficult to directly gain some qualification credit from its use in a qualified development chain.

Annotation generation: The constraints specified for the elements of the *BlockLibrary* are pre/post conditions and invariants providing both structural constraints and expected semantics. This approach, close to the design by contract ones [7], can be used to generate annotations in the generated code. The contract in the *BlockLibrary* can be related to a contract in the generated code template. This will help to verify the final generated code using proof checkers or other static analysis tool. An extension of GeneAuto is being experimented in that purpose by generating ACSL annotations along with the generated C code for Simulink or Scicos input models. A prototype already exists for generating annotated code for simple yet realistic systems (horizontal control for an helicopter [8]).

Generation of typing and code generation backends: The *BlockLibrary* specification contains detailed information about type inference rules for different block types and configurations, as well as the dynamic semantic functions related to block executions. These information can be used to generate parts of the code generator. It might be used for the generation of skeletons for functions that needs to be implemented manually by code generator developers. These skeletons can be

[12] `http://dieumegard.perso.enseeiht.fr/bl/Progress/BlockLibInstanceVerification.html`

augmented with automatically generated verification conditions taken from the *BlockLibrary* specification. Implementors must ensure that the implementation satisfies these properties. This together with generated documentation increases the extensibility and reliability of *BlockLibrary* specification.

Generation of test-cases: For each *Constraint Signature* that is associated to a *BlockMode* and *Variant Signature* we can extract a set of *StructuralFeatures* along with their specifications provided as structural annotations. Each of these pairs of elements specifies the definition domain of a *StructuralFeature*. This definition domain is potentially extended with additional constraints defined for this *StructuralFeature*, containing *BlockVariants* or *BlockMode*. We translate this definition domain as a constraint satisfaction problem (CSP) (we used the MiniZinc[13] language) that is solved by an automated solver. For example, we can apply this approach to the `InitialConditionSource` *StructuralFeature* defined in the *Delay* block specification in Listing 1.1. The corresponding generated CSP is a search for a value for an enumeration of type `TValueSource` according to the constraint: `value = INPUT`. Applying this approach to each *StructuralFeature* of the *BlockMode Signature* provides us with a set of specification-satisfying values for the inputs and parameters of an instance of this *BlockMode*. The values provided by this computation can be used as input and configuration data for the execution of generated code. Additionally, it is possible from the semantic specification expressed in a *BlockMode* to generate an EMBEDDED MATLAB or SCILAB function. This function will take some configuration values provided through parameters and inputs values. Once this function is fed with the expected values (either automatically generated or provided by the tester) it will give us the expected values for the computation of a block instance. In this way, we generate an oracle that provides reference data for testing a code generator that has been implemented according to the same specification. We have developed first prototypes of such generators and plan to extend the approach in the future.

5 Conclusions and future work

We have presented in this paper a DSL for the formal specification of function block libraries for dataflow modeling languages. Our *BlockLibrary* DSL intends to provide a structured, formal and multi-purpose specification method for the semantic definition of such blocks. The methodology is especially targeting blocks with a high degree of structural and hence also algorithmic polymorphism.

This specification methodology can be used: i) for extending the functionalities of existing languages like SIMULINK, SCICOS and XCOS by developing custom block sets for end-user specific applications; ii) for the generation of development support artifacts like documentation; iii) as a basis for formal verification of block libraries; iv) as a source for the generation of verification artifacts like test inputs and expected outputs.

We have developed some tools and transformations using *BlockLibrary* instances as input. Each transformation interprets the specification from its specific point of view and by applying a filter on it, will extract relevant data. We have shown that multiple languages can be used as formal annotation languages. One of the technical challenges of this work is to be able to transform these various languages to the target analysis-focused languages like the CSP languages or the Why3 language, or execution-focused languages like EMBEDDED MATLAB. We plan to address this problem in the future by relying on suitable pivot languages.

This approach has been currently validated on a small number of SIMULINK blocks that have distinctive polymorphic nature such as the *Delay* block and some other typical Simulink blocks.

[13] `http://www.minizinc.org/`

We plan to refine this language further and use it as a formal specification for the block library of the GeneAuto successor toolset developed in the projects ProjectP and Hi-MoCo.

References

1. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35(6), 615–636 (2010)
2. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64. Wroclaw, Poland (2011), http://hal.inria.fr/hal-00790310
3. Bräuer, M., Demuth, B.: In: Giese, H. (ed.) Models in Software Engineering, Lecture Notes in Computer Science, vol. 5002, chap. Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support, pp. 182–193. Springer (2008)
4. Dieumegard, A., Toom, A., Pantel, M.: Model-based formal specification of a DSL library for a qualified code generator. In: Proceedings of the 12th Workshop on OCL and Textual Modelling. pp. 61–62. ACM, New York, NY, USA (2012)
5. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Integrating ocl and textual modelling languages. In: Dingel, J., Solberg, A. (eds.) Models in Software Engineering, Lecture Notes in Computer Science, vol. 6627, pp. 349–363. Springer (2011)
6. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
7. Meyer, B.: Design by contract. In: Mandrioli, D., Meyer, B. (eds.) Advances in Object-Oriented Software Engineering. pp. 1–50. Prentice-Hall, Inc. (1992)
8. Wang, T.E., Dieumegard, A., Feron, E., Jobredeaux, R., Pantel, M., Garoche, P.L.: Autocoding of computer-controlled systems with semantics for formal code verification. In: S5 Symposium (2012), http://tinyurl.com/o4oskeu

```
type enum TValueSource { INPUT, DIALOG }
type enum TTriggerType  { NONE, RISING, FALLING, EITHER,
  LEVEL, LEVEL_HOLD, FUNCTION_CALL }

blocktype Delay {

  definition ocl U : Port {
    blockType().variants("DelayRoot").inputs("UPort")->portImpl.first()
  }
  variant DelayRoot {
    in   data UPort : allowedtypes  : TAny
    out data YPort : allowedtypes  : TAny
    parameter DelayLengthSource : TValueSource
    parameter InitialConditionSource  : TValueSource
    parameter ExternalReset : TTriggerType
    invariant ocl { ExternalReset.value <> FUNCTION_CALL }
  }
  variant VarDelay extends DelayRoot {
    modeinvariant ocl { DelayLengthSource.value = INPUT }
    in data DelayPort : TNumeric { invariant ocl { value > 0 }}
    parameter DelayUpperLimit : TNumeric {
      invariant ocl { value > 0 }
      invariant ocl { dataType.isScalar }
    }
    memory DelayBuffer {
      datatype ocl  { U.dataType }
      length ocl  { DelayUpperLimit.value / blockInstance().samplePeriod }
    }
  }
  variant FixedX0 extends DelayRoot {
    modeinvariant ocl { InitialConditionSource.value = DIALOG }
    in data virtual X0Port : TAny { implementedby InitialConditionParam }
    definition ocl X0 : TypedElement { InitialConditionParam.parameterImpl}
    invariant ocl {
      X0.dataType.isSubType ( U.dataType )
      && CompatibleDimensions( X0.dimensions , U.dimensions )
    }
  }
  variant VarX0 extends DelayRoot {
    modeinvariant ocl { InitialConditionSource.value = INPUT }
    in data X0Port: TAny
    definition ocl X0 : TypedElement { X0Port.portImpl->first() }
    invariant ocl {
      X0.dataType.isSubType( U.dataType )
      && CompatibleDimensions( X0.dimensions , U.dimensions )
    }
  }
  mode VarRstDelayMode implements  anyof (
        allof (VarDelay, FixedX0),
        allof (VarDelay, VarX0) ) {
    modeinvariant ocl { ExternalReset.value <> NONE }
    definition matlab [y, m, last] = Delay_Rst_Init {
      y = X0.value; m = X0.value; last = X0.value;
    }
    definition matlab y = Delay_Compute_Var_Rst_Delay (u, d, r, x0) {...}
    definition matlab [m, last] = Delay_Update_Var_Rst_Delay (u, d, r, x0)
        {...}
    initsemantics       : Delay_Rst_Init
    computesemantics    : Delay_Compute_Var_Rst_Delay
    updatesemantics     : Delay_Update_Var_Rst_Delay
  }

}
```

Listing 1.1. Extract of the *Delay* block specification