Block Library Driven Translation Validation for Dataflow Models in Safety Critical Systems

Arnaud Dieumegard[1], Andres Toom[2,3,4], and Marc Pantel[2]

[1] Institut de Recherche Technologique Antoine de Saint Exupéry,
118 route de Narbonne, CS 44248, 31432 Toulouse Cedex 4, France
[2] Institut de Recherche en Informatique de Toulouse, Université de Toulouse,
ENSEEIHT, 2 rue Charles Camichel, 31071 Toulouse Cedex, France
[3] Institute of Cybernetics at Tallinn University of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia
[4] IB Krates OÜ, Mäealuse 4, EE-12618 Tallinn, Estonia

Abstract. Model driven engineering is widely used in the development of complex and safety critical systems. Systems' designs are specified and validated in domain specific modeling languages and software code is often produced by autocoding. Thus the correctness of the final systems depend on the correctness of those tools. We propose an approach for the formal verification of code generation from dataflow languages, such as SIMULINK, based on translation validation. It relies on the BLOCKLIBRARY DSL for the formal specification and verification of the structure, semantics and variability of the complex block libraries found in these languages. These specifications are then used here for deriving model and block-specific semantic contracts that will be woven into the generated C code. We present two different approaches for performing the block matching and weaving step. Finally, we rely on the FRAMA-C toolset and state-of-the-art SMT solvers for verifying the annotated code.

Keywords: Translation validation, Deductive verification, Data flow languages, Block libraries, Why3 toolset, Frama-C toolset

1 Introduction

Automatic code generators (ACG) are nowadays used for the development of most safety-critical systems in order to avoid human-related programming errors, and ensure both quality standards conformance and efficient maintenance cycles. As these tools replace humans in a key software production step, their design or implementation flaws usually result in errors in the generated software. Safety critical software development must usually satisfy certification/qualification standards like the *DO-178C* for avionics which is one of the best known and most stringent one. One key requirement in its Model Based Software Engineering (MBSE) and Formal Methods (FM) supplements (*DO-331* & *DO-333*) is to provide a precise, complete and unambiguous specification of the input and output languages for ACG.

Overview of the approach Figure 1 gives an overview of the elements in our specification and verification process. In [3, 4] we proposed a Domain Specific Language (DSL) for writing and formally analyzing specifications for configurable function blocks in dataflow languages. We refer to this as the BlockLibrary specification language. In the current contribution it is used for the formal verification of automatically generated code from dataflow models. The Why3 toolset [2] is first used for the formal verification of the well-formedness and semantic consistency of the specification. Then these specifications are used for the verification of generated source code from dataflow models by weaving model-specific formal annotations into the generated source code. The annotated code is then verified using the Frama-C toolset[5]. Both verification steps are in turn relying on SMT solvers and proof assistants as a formal backbone.

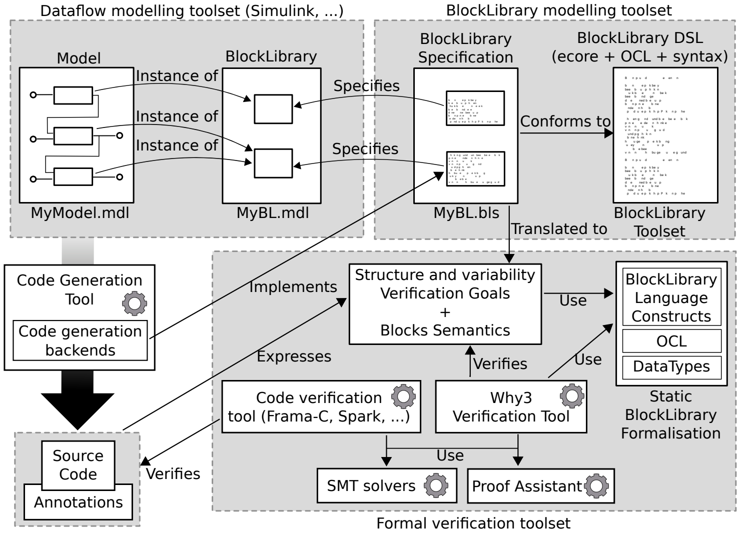
Dataflow modelling toolset (Simulink, ...)
BlockLibrary modelling toolset
Model
BlockLibrary
BlockLibrary Specification
BlockLibrary DSL (ecore + OCL + syntax)
Instance of
Instance of
Instance of
Specifies
Specifies
Conforms to
MyModel.mdl
MyBL.mdl
MyBL.bls
BlockLibrary Toolset
Translated to
Code Generation Tool
Code generation backends
Implements
Structure and variability Verification Goals + Blocks Semantics
Use
BlockLibrary Language Constructs
OCL
DataTypes
Static BlockLibrary Formalisation
Expresses
Verifies
Use
Code verification tool (Frama-C, Spark, ...)
Why3 Verification Tool
Source Code
Annotations
Verifies
Use
SMT solvers
Proof Assistant
Formal verification toolset

Fig. 1. The process for the formal specification of block libraries and the use of these specifications for the verification of automatically generated code

Dataflow languages Dataflow languages are widely used for specifying control and command algorithms. Simulink[6] is the most used in the industry but there exist other similar graphical formalisms, such as SCADE, Scicos, Xcos and textual languages such as Lustre and Signal. The main constructs in dataflow models are *blocks* (computation nodes) and *signals* (data connections). The concrete execution of dataflow models is divided into three phases: an *initialization* phase, where the memories of all the blocks in the model are initialized; followed

[5] http://frama-c.com

[6] http://www.mathworks.com/simulink

by a recurring cycle alternating the *compute* and *update* phases. During the former each enabled block produces data to its outputs according to data at its inputs, previous state(s) and configuration parameters. The computed data is immediately available for the following blocks. The order of these computations is defined statically by a sequencing algorithm such as [9], formalized in [7]. The *update* phase updates the memories of all sequential blocks according to the data at their inputs, previous values in memory and configuration parameters.

Use case Our final goal is the verification of the generated code for a complete dataflow model with respect to requirements for the system expressed as top level software contracts that are either manually written or generated from a specification model (e.g. dataflow system observer). This part is not directly addressed in the current work. However, the semantics of the model that implements this top level specification is given by the semantics and configuration of the blocks it relies on. We propose to weave the generated code with semantic contracts derived from the BlockLibrary specifications for two purposes: (a) verify that the generated code for each block matches its specification and (b) help the deductive verification tools in proving the top level contracts. We illustrate our proposal with the specification of the *IntegerDelay* block and the verification of its code as generated by the GeneAuto ACG [17].

Organization of the paper This paper is organized as follows: Section 2 provides an overview of the BlockLibrary language and associated verification technique. Section 3 focuses on the verification of the low-level requirements for an ACG relying on a translation validation approach. Section 4 applies this approach on a use case. Section 5 compares our approach to related works. Planned future extensions and a conclusion are given in Section 6.

2 Formal Specification of Blocks in Dataflow Languages

Besides the core principles of data flowing through signals from output ports to input ports, the semantics of dataflow languages is mainly determined by the semantics of elementary blocks that compute data on their outputs depending on data on their inputs, memories and parameters. Tools such as Simulink have large block libraries with highly configurable blocks. The BlockLibrary DSL proposed in [3, 4] relies on the core concepts of Software Product Line Engineering (SPLE) adapted to the domain of block libraries to handle this variability. This DSL is expressed as a metamodel in MOF (OMG Meta Object Facility standard) capturing the concepts required for the specification of both the structure (ports, parameters and memories) and the semantic phases (initialization, computation and memory update) of dataflow blocks. Each structural element is parameterized by data types and constraints expressed in the OMG standard Object Constraint Language (OCL) describing the set of allowed values for valid instances of each structural elements. A semantic specification element contains the specification of the behavior of one specific configuration of the block expressed either in an axiomatic style using pre/post-conditions or operationally by giving the function's definition. One can also provide both, since the two styles offer different possibilities for automatic verification. The pre/post-conditions are specified using OCL. Operational specifications are specified using the BlockLibrary Action Language (BAL).

The next section illustrates the specification for a block family with an emphasis on the variability management relying on the Simulink *IntegerDelay* block. The verification technique for the block specification is then summarized (the interested reader can find more detailed information in [3, 4] and on the project's website http://block-library.enseeiht.fr/html). Finally, we will elaborate on the verification of loop constructs.

2.1 Example of Block Specification: IntegerDelay

Listing 1.1 provides a partial specification of the *IntegerDelay* block that delays data flow by a given number N of clock cycles. Such blocks are often used in control and command algorithms for writing recurrent equations (the discrete equivalent of difference equations). This block is one of the simplest, but its Simulink version is nevertheless highly variable with multiple semantics variations. The number of delayed clock cycles is specified statically as a parameter. As the block delays its input values by N clock cycles, it is mandatory to provide the first N values to be used for the block's output. These initial values IV are either provided by a static parameter or via an input of the block. The block has other parameters and inputs making this block representative of the typical variability of Simulink blocks. In the complete specification also N can be provided via the block's input and there can be yet another input to dynamically reset the block's state (according to 4 different activation algorithm variants – rising edge, falling edge, zero crossing and level). Finally, the specification provided here only handles scalar and vector values of the double data type for the input and output ports, whereas the full specification also allows integer (signed, unsigned, 8, 16 or 32 bits) and boolean data types, as well as matrices of all of those types. The full specification of this block family has 144 distinct configurations.

```
library BlockLibrary {
  type signed realInt TInt32 of 32 bits
  type realDouble TDouble
  type array TArrayDouble of TDouble [-1]
  blocktype IntegerDelay {
    variant DelayParameter {
      parameter N : TInt32 { invariant ocl { N.value > 0 } }  }
    variant IOScalar {
      in data Input : TDouble
      out data Output : TDouble}
    variant IVScalarParam extends IOScalar {
      parameter IV isMandatory : TDouble}
    variant IVScalarInput extends IOScalar {
      in data IV : TDouble  }
    variant IOVector {
      in data Input : TArrayDouble
      out data Output : TArrayDouble}
```

```
        variant IVVectorParam extends IOVector {
          parameter IV isMandatory : TArrayDouble }
        variant IVVectorInput extends IOVector {
21        in data IV : TArrayDouble}
        variant ScalarValues extends ALT(IVScalarParam, IVScalarInput)
        variant VectorValues extends ALT(IVVectorParam, IVVectorInput)
        variant ScalarOrVectorValues extends ALT(ScalarValues, VectorValues)
        variant Mem extends AND(ScalarOrVectorValues, DelayParameter) {
26        invariant ocl { Input.size = Output.size }
          invariant ocl { IV.size = N.value }
          memory Mem {
            datatype auto ocl { Input.value }
            length auto ocl { N.value }}}
31      mode DelaySemantics implements Mem {
          definition bal = init_Delay {
            postcondition ocl { Mem.value = IV.value }
            Mem.value := IV.value;}
          definition bal = compute_Delay {
36          postcondition ocl { Output.value = Mem.value }
            Output.value := Mem.value[0];}
          definition bal = update_Delay {
            postcondition ocl { Mem.value->last() = Input.value }
            postcondition ocl {
41            Mem.value := (Mem.value@pre)->subList(2,N.value)
                                              ->append(Input.value)}
            Vector_Shift_Left(Mem.value, 1, Input.value);}
          init init_Delay
          compute compute_Delay
46        update update_Delay}}
    }
```

Listing 1.1. Partial specification of the *IntegerDelay* block

2.2 Specifying a Block Family

A block family is a set of possible configurations for a given block (a product line of blocks in the "feature modeling" or FODA methodology and terminology from Kang et al. in [8]). A block family specification in the BlockLibrary language starts with the definition of block's structure. The structure can be decomposed using *BlockVariant*s and *BlockMode*s. *BlockVariant* is a basic specification unit that defines a subset of structural or semantic elements. *BlockVariant*s can be reused in the specifications of different block types. They can be extended and combined into larger units by using inheritance. A *BlockMode*, on the other hand, is associated to a particular subset of the complete block configurations whose behavior can be captured with a similar algorithm. Each *BlockMode* implements one or more *BlockVariant*s and inherits the elements defined in those. The specification of a block type thus consists of a structural variation graph (DAG) whose roots and inner nodes are *BlockVariant*s and leafs are *BlockMode*s.

Two kinds of inheritance: AND and ALT, both n-ary operators, are available. For AND, respectively ALT, the inheriting node inherits the definitions and contracts from *all*, respectively exactly *one*, of the inherited nodes. AND is similar to multiple inheritance as found in many object oriented languages. In our case, we require that all the inherited elements are distinct (i.e., no overloading or overlap is allowed). This relation is thus unambiguous and commutative. Line 25 of Listing 1.1 is an example of AND, and line 22 an example of ALT.

This last one specifies that, in one configuration of the `Delay` block, the initial value is provided as a static parameter and, in the other, it is provided via an input.

From a BlockLibrary specification we can extract the set of all valid block configurations. Each configuration has exactly one *BlockMode* and one or more *BlockVariant*s. Figure 2 displays the block configurations extracted from the specification in Listing 1.1. In a well-formed dataflow model, all block instances must match to exactly one configuration in the respective block family.

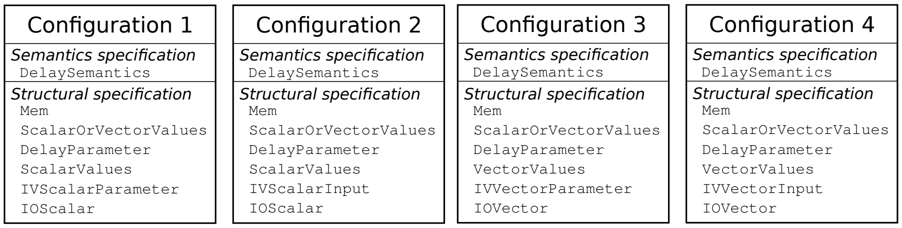

Fig. 2. Configurations extracted from the specification of the *IntegerDelay* block

2.3 Verification and Validation of Block Specifications

The block specifications written in the BlockLibrary language are formally assessed using a translation into the Why property and WhyML behavior specification languages [2]. These generated specifications can be verified using the dedicated Why3 toolset relying either on automated proofs by SMT solvers or manual ones using proof assistants such as Coq. The structural part of the BlockLibrary specifications, including the variability aspects are translated to Why. The blocks' semantics are translated to WhyML. This translation is currently written in Java and relies on the modeling support in EMF[7]. BlockLibrary models are translated to Why and WhyML models that are serialized using the XText toolset[8]. It is not the aim of this paper to assess the correctness of this translation. On the one hand, it defines the semantics of the BlockLibrary language. On the other hand, it enables writing accurate specifications for the blocks in the chosen languages (e.g., Simulink). This translation is being validated using various testing strategies.

Verification of structure and variability For each block configuration extracted from a BlockLibrary specification, we generate a Why theory containing the definitions for each structural element contained in its structural specification

[7] https://eclipse.org/modeling/emf/

[8] https://eclipse.org/Xtext/

altogether with a set of predicates expressing the: a) explicit constraints defined as invariants of the structural elements; b) implicit constraints related to the structural elements' data types; and c) global invariants constraining the *BlockVariant* and *BlockMode* elements.

Two properties, *completeness* and *disjointness*, are used to assess the well-formedness of the structure and variability of the specification. The first one states that for all the well-formed instantiations of the block there exists at least one suitable configuration in the specification. This covers both the fact that, all *BlockVariant*s are used and that no contradictions are present in the set of associated constraints. The second one ensures the unambiguous interpretation of any configuration of a block. Both properties are automatically converted to goals relying on the previously generated predicates.

The verification is performed using the Why3 platform and SMT solvers. In our experiments it was performed in a few seconds for most of the blocks on a common modern laptop. Some block specifications with more variability require more time (with a time factor of up to 100 in our experiments) for fully automated verification. This cost is generally not an issue as this verification must be done only once for each block specification.

Verification of semantics If the block's semantics is specified both axiomatically and operationally, then the consistency of these definitions can be verified. The specification of the three execution phases (in Listing 1.1 the resp. sections are following: initialization lines 32-34, computation lines 35-37 and update lines 38-43) are translated to distinct WhyML functions. The body of each function is a direct translation of the respective semantic function of the block in this mode. This function is given a contract consisting of the pre/post conditions specified in the axiomatic semantics and an additional pre-condition computed based on the structural properties that hold in this particular mode.

The verification of the WhyML functions is done as previously by relying on the Why3 platform and SMT solvers. While this verification can be straightforward for the specifications of simple blocks, it can require additional annotations for code with more complex data types (vectors, matrices) or algorithmically more complex blocks. In practice, fully automatic proofs are often only possible, when additional annotations are given in the form loop invariants, variants and even ghost code. The `BAL` language allows one to write such annotations and our transformation tool translates them into annotations in the generated WhyML function body. If a consistent specification is provided, the verification is usually handled automatically in a short time (a few tenths of a second) by the SMT solvers. As the number of configurations can grow exponentially, the verification of a complete block library specification can still take some time but, as previously stated, this verification is only done once for a given library.

Validation of specifications While it is important to verify the specifications for well-formedness and consistency, it is also necessary to validate that they adequately capture the intended semantics of the corresponding blocks, e.g., that is in the existing reference library implementations. The validation strategy that we have planned and are currently implementing for the specifications of SIMULINK blocks relies on the translation of all configurations of all blocks to corresponding MATLAB code. That code is executed and the execution result is compared to the result of the simulation of a correspondingly configured SIMULINK block. Test vectors for block inputs and parameters can be automatically generated based on the block specifications. This work is currently ongoing and the results will be reported in the future.

2.4 Handling Loop Constructs

Programs containing loops are the main difficulty in the automated deductive verification. Loops usually must be annotated with complex invariants and variants. Loops are very common programming constructs that occur especially often in mathematical computations, including the ones done in dataflow blocks. However, in this context, loops have often a very regular structure with similar annotations. Thus, the specification designer should be relieved from the writing of these annotations. This topic has been addressed by many authors and various solutions have been proposed. For example, Furia and Meyer [6] propose a technique to generate invariants for certain loop patterns automatically, when larger program units such as routines have explicit post-conditions which is also the case in our context. The technique in [6] was partly implemented in the Boogie tool. Unfortunately, in the general case such methods are also limited due to undecidability. Recent work by Wiik and Boström [19] targets contract-based verification of MATLAB-style matrix programs making this work very relevant to ours. The authors propose a solution for efficient encoding of a subset of MATLAB's built-in matrix functions for deductive verification. However, the specification of loop invariants is still required. According to the authors' experiments, k-induction can be used to remove some, but still not all of these invariants. The work currently addresses partial correctness only. Filliâtre and Pereira [5] have presented an approach that can be applied to different iteration paradigms including non-deterministic and infinite iterations as well as iterations with side-effects. It does not completely remove the need to specify loop invariants, but it allows encapsulating and hiding the implementation aspects of iterators. Thus, simplifying considerably the specification left to the "users".

In our context loop specifications occur in two places: specifications of the blocks operational semantics and annotations of generated code. Given the state of the art, we have chosen the following strategy to deal with them. As explained earlier, the operational specification of blocks semantics is optional and is not needed for the translation validation. However, it can be beneficial otherwise. The `BAL` language contains *for* and *while*-style loop statements with both *invariant* and *variant* annotations that the user can explicitly specify. However, explicit loop constructs can be in many cases avoided by using polymorphic operators with implicit support for non-scalars and/or using higher order iterators. For instance, element-wise operations on non-scalars can be specified in terms of an elementary operation and a higher-order *map* operator and collapsing operations using a *fold* operator (i.e., catamorphism). Other operations, such as

```
var iter = 0;
ghost { var i = 0; }
while (iter < (N.value - 1)){
  invariant ocl { 0 <= iter < N.value }
  invariant ocl { 0 <= i and i < iter and
     Mem.value->subList(1,i) = (Mem.value@pre)->subList(2,i+1) }
  variant ocl { N.value - iter }
  Mem.value[iter] := Mem.value[iter + 1];
  iter := iter + 1;}
Mem.value[N.value - 1] := Input.value;
```

Listing 1.2. Expanded version of the Vector_Shift_Left operation

matrix product, require more specific encoding. The *Vector_Shift_Left* binary operator used in line 43 of Listing 1.1 is another example. It is applied on a vector or a matrix. In case of a vector, it shifts all the values in the vector by one position downwards, starting from the second element, and inserts a specified new value as the very last element. If the first argument is a matrix, it applies the same operation to each row of the matrix. The last argument should be a vector in this case. Line 43 of Listing 1.1 is automatically expanded to Listing 1.2. The specification of these operators in BAL and their translation to Why and WhyML are currently in progress. This work relies largely on earlier work performed in GeneAuto [17] and its successor project QGen[9] on specifying the semantics of the embeddable subsets of Matlab and Simulink and can benefit also from the formalization presented in [19]. The difference in our work is that the invariants and variants for such operators will be generated based on the static parameters of concrete block instances.

The same loop annotations are required in the translation validation context for verifying the correctness of the generated code. The BAL language supports such annotations and it would be thus possible to specify them manually. However, this is rather laborious. We could try to weave these annotations into the generated code just like the pre- and post-conditions. However, this can be significantly more complex. First, it would make the existence of the operational specification mandatory for semantic functions containing loops. Secondly, it would require the generated low-level code to be structurally very close to the operational specification and prohibit the usage of more abstract operators in the specifications or the generation of optimized code. One solution is to let the ACG itself generate annotations for primitive operators involving loops. This might seem to jeopardize the intent of translation validation. However, if applied with care, it does not. It is a similar situation to the Assertion Inference Paradox stated in [6]. The pre- and post-conditions of the larger program units (here code sections generated from block instances) are not generated by the untrusted ACG whose output is under verification. The intent of the intermediate annotations derived from the primitive operators is to help the verification tool in proving the satisfaction of the main contracts. If these annotations do not introduce any axioms to the axiom base used by the deductive verification

[9] http://www.adacore.com/qgen

toolset, then they cannot contribute to a false claim of correctness. In our case no axioms would be generated. However, generally, it must be separately ensured.

The generated C code for the *Vector_Shift_Left* operator corresponding to Listing 1.1, together with the generated annotations, can be seen in lines 32-42 of Listing 1.3. Note that in case the block's semantics is not implemented using operators supporting automatic annotation generation and uses general loop constructs instead, the current deductive verification tools might not be able to automatically prove the correctness of the code. Thus, it would be necessary to use a proof assistant for proving these parts or add some annotations manually in the generated code.

3 Verification of the Correctness of Generated Code

Automatic code generation is one of the key benefits of MBSE for critical systems. The correctness of the code can be assessed by verifying the correctness of the ACG or by verifying that the code has the expected properties. For both approaches there are many techniques available. The assurance gained by classical methods like testing or code reviewing has well-known limits. Formal methods can provide a much higher level of confidence but they have a higher entry barrier that requires good tool support and well integrated development processes.

We will detail below our approach for the formal verification of automatically generated code relying on the Translation Validation (TV) methodology proposed by Pnueli et al. in [15]. In TV the transformation (code generation) workflow is complemented by an independent verification workflow. The latter relies on the common formalization of the input and output languages and must establish whether the input and output data of a given transformation run conform to the expected equivalence relation or not. In our case we rely on the formalization of dataflow blocks in the BlockLibrary language and use that to derive formal contracts for code generated from block instances in a given input model. In particular ACSL [1] annotations for C code. The same principle can also be applied for other languages providing Design By Contract or annotation facilities like SPARK/Ada 2012, Spec#, JML, B method, Eiffel, etc.

The annotated code is passed to the Frama-C toolset and its weakest precondition plugin generating proof obligations that are then verified using SMT solvers. This can be done either directly through Frama-C or through the Why3 toolset. In many cases the verification can be performed fully automatically. When this cannot be achieved, one can use proof assistants to complete the verification manually. Yet another option is to manually add additional annotations such as loop invariants and ghost code into the generated code to assist the verification tools. This option is, however, undesirable, since it is rather laborious and requires modifying the generated code, which impairs the whole process.

3.1 Semantic Annotation of the Generated Code

In order to derive appropriate semantic contracts for the block instances in the input model the model must be first analyzed and the block instances and their configurations must be identified. Secondly, the concrete (ACSL) contracts must be generated and woven into the (C) code generated by the ACG. This involves a number of technical steps. We have considered two alternatives for that.

White box approach. If the ACG source code is available, then a light-weight option is to encode the annotation generation directly into the ACG, e.g., by hand-writing the annotation generation functions for each block type. This is rather laborious. Alternatively, these functions could be automatically generated from the BlockLibrary specifications and integrated with the ACG source code. However, this approach still isn't true translation validation as it relies a lot on the ACG internals. Also, according to *DO-330*, this extra verification-oriented code must be qualified with the same stringency as the rest of the tool as it may introduce errors into the code generation. We have considered this option as a compromise allowing one to concentrate on the BlockLibrary specific aspects and paving the way for the more complete approach explained next.

Black box approach. In this case an external tool must parse the input model, generate the annotations and weave them into the generated code. Regarding *DO-330*, such a separate verification tool must be also qualified in order to gain certification credit for using it. However, the qualification process is much lighter as the tool cannot have any direct impact on the generated code.

3.2 Verification Using the Frama-C Toolset

We decided to target C code annotated with ACSL as: a) this language is widely used in the safety-critical systems industry, b) the GeneAuto ACG has mainly been used and evaluated in this context, and c) the ACSL annotation language is supported by formal analysis tools such as the Frama-C framework. Similar work could have also been done based on most of the other alternatives mentioned earlier.

The Frama-C framework targets the analysis of C code in order to extract information provided by various plugins. These plugins allow for the static analysis of the source code to extract information like variables' ranges and scope, code metrics, detect dead code, and many others[10]. Verification of ACSL annotations expressed on the source code is done through the WP plugin implementing a weakest precondition calculus generating proof obligations to be verified. Those are directly assessed using SMT solvers or sent to the previously presented Why3 platform to rely both on a wider range of SMT solvers or, when needed, on proof assistants. Proof assistants are used in order to manually tackle difficult proofs when the automatic SMT solvers fail to achieve the proof.

White box experiment using the open source GeneAuto ACG. We have extended GeneAuto with the support for annotation manipulation[11]. For this purpose, we developed a metamodel based on a subset of the ACSL specification

[10] Visit the Frama-C framework website for detailed information: http://frama-c.org.

[11] This work has been performed in partnership with Timothy Wang from the Georgia Institute of Technology and has been partly used in the context of the verification of automatically generated code presented in [18].

including annotations, ghost code and function contracts. We relied on the EMF framework to generate the corresponding Java classes that we integrated into the GeneAuto source code. We also implemented printing facilities for ACSL annotations. Finally, we implemented the annotation generation functions. As a possible future alternative, the current version of Frama-C also accepts annotations written directly in Why3 instead of ACSL.

Black box proposal through Frama-C. For the full translation validation we are considering a following approach. The Frama-C toolset provides C code manipulation facilities and is extensible using plugins. It is thus possible to define a new Frama-C plugin to conduct automated annotation of the generated code. This plugin must extract data from both the dataflow (Simulink) model, as well as from BlockLibrary specifications. Developing a parser for these two formalisms directly in the plugin would be very time-consuming. Instead, we plan to develop or reuse a separate parser for dataflow models and provide within the BlockLibrary toolbox a transformation exporting the derived contracts as a data structure that would be easier to use within the Frama-C plugin. The plugin must then weave the contracts into the generated code. It is reasonable to simplify that part by relying on minimal traceability information provided by the ACG relating blocks and corresponding code sections. ACGs used for the development of critical software must anyhow provide such links for traceability.

For instance, GeneAuto provides them in the form of code annotations.

4 Translation Validation of IntegerDelay

As example, an instance of the *IntegerDelay* block is embedded in a small model containing an *Inport* block `In1` and an *Outport* block `Out1`. Such models can also be generated automatically from BlockLibrary specifications. In this example, the block's input and initial conditions are scalars, and the delay length `N` is 2. Listing 1.3 shows the code and annotations generated for this example.

The code for the initialization phase is defined in the `system_init` function while the computation and update phases are together in the `system_compute` function. While the initialization code is straightforward, the code for the two other phases requires some explanation. The compute function starts with the assignment of the `Delay2` block's output, since it only depends on the value of the block's memory (lines 14-17). Then the output of the `In1` block is computed based on the current input to the system (lines 18-21), and the output of `Out1` is computed (lines 22-25) based on the output of `Delay2` computed earlier. Finally, the memory of `Delay2` is updated (lines 32-42).

Each code section generated for a block needs to be annotated with pre/post-conditions (in the example, only post-conditions are used) derived from its BlockLibrary specification. These are the *ensures* annotations surrounding each block code. These annotations are currently generated and inserted into the code using the ACG and the white box approach. Whereas in the black box approach they would be generated directly from the BlockLibrary toolset and woven into the code using the weaver plugin that relies just on the traceability annotations provided by the ACG.

While these annotations make the core for the semantic verification of the generated code, some additional annotations are required in order to ensure that the generated code is embedded in a suitable environment: these annotations specify properties like memory independence of the data structures (the **separated** annotations) and correct instantiation of the previously described data structures (the **valid** annotations). These annotations make explicit the implementation choices made by the ACG for the generated code. The annotated code in Listing 1.3 is verified automatically using the Frama-C toolset and the Alt-Ergo[12] and Spass[13] SMT solvers in a few seconds.

```
/*@ requires \valid(_state_->Delay2_memory+(0..1)); */
void system_init(t_system_state *_state_) {
  /*  START Block: <SequentialBlock: name=Delay2>  */
  /*@ ensures _state_->Delay2_memory[0] == 2;
      ensures _state_->Delay2_memory[1] == 3;*/
  { _state_->Delay2_memory[0] = 2;
    _state_->Delay2_memory[1] = 3;}
  /*  END Block: <SequentialBlock: name=Delay2>  */
}
/*@ requires \separated(_state_->Delay2_memory , _io_);
    requires \valid(_io_);
    requires \valid(_state_->Delay2_memory+(0..1)); */
void system_compute(t_system_io *_io_, t_system_state *_state_) {
  /*  START Block: <SequentialBlock: name=Delay2>  */
  /*@   ensures system_Delay2 == _state_->Delay2_memory[0]; */
  system_Delay2 = _state_->Delay2_memory[0];
  /*  END Block: <SequentialBlock: name=Delay2>  */
  /*  START Block: <SourceBlock: name=In1>  */
  /*@   ensures system_In1 == _io_->In1; */
  system_In1 = _io_->In1;
  /*  END Block: <SourceBlock: name=In1>  */
  /*  START Block: <SinkBlock: name=Out1>  */
  /*@   ensures _io_->Out1 == system_Delay2; */
  _io_->Out1 = system_Delay2;
  /*  END Block: <SinkBlock: name=Out1>  *//
  /*  START Block memory write: <SequentialBlock: name=Delay2>  */
  /*@ ensures append: _state_->Delay2_memory[1] == system_In1;
      ensures sublist: \forall integer i; 0 <= i < 1 ==>
        _state_->Delay2_memory[i] ==
        \old(_state_->Delay2_memory[i+1]);
  */
  {/*@ loop invariant \forall integer i; 0 <= i < iter ==>
         _state_->Delay2_memory[i] ==
         \at(_state_->Delay2_memory[i+1],LoopEntry);
       loop invariant 0 <= iter < 2;
       loop assigns iter, _state_->Delay2_memory[0];
       loop variant 1 - iter; */
  for (int iter = 0; iter < 1; iter++){
    _state_->Delay2_memory[iter] = _state_->Delay2_memory[iter+1];
  }
  _state_->Delay2_memory[1] = system_In1;}
  /*  END Block memory write: <SequentialBlock: name=Delay2>  */
}
```

Listing 1.3. Annotated code for a small subsystem containing the *IntegerDelay* block

[12] http://alt-ergo.ocamlpro.com/

[13] http://www.spass-prover.org/

5 Related Work

Our work relies on translation validation as introduced by Pnueli [15]. This technique has suffered in its early years from scalability issues as the verification was mostly done using model checking. This has improved with the use of modern automated provers (SMT) and proof assistants. However, for complex blocks handling complex data types, it still seems mandatory to rely on intermediate annotations to avoid relying on proof assistants too often.

Recent examples of translation validation are found in the work of Ryabtsev et al. [16] for the verification of the code generated from Simulink models and [12] for the proof of preservation of clock related properties on generated code. The BlockLibrary language allows for a simple specification of the blocks semantics and its formal verification that was a missing point in Ryabtsev approach. Our approach differs as we do not interpret the generated code in order to compare its semantics to the one of the input model, but we rather rely on code verification tools to check that the generated code complies with the input model semantics specified as code annotations.

O'Halloran [13] reports on the successful use of the ClawZ system for translation validation from a subset of Simulink to Ada independently from any concrete ACG. This work relies on a Simulink formalization effort using the Z notation performed over many years. The author reports that, in the last versions, the translation validation can be performed fully automatically using the ProofPower toolset which is a significant achievement. The Z notation is a very general formalism that requires a sophisticated methodology to reach fully automated proofs. Our BlockLibrary DSL is designed to model only block families. Thus, it permits meta-level analysis such as the completeness and consistency of specifications or automated generation of validation tests. Inspection and testing techniques for the verification of implementations could have been preformed also using a formal specification language such as SOFL CDFD [10], but the specification of the blocks variability would suffer from the same limitations.

Our proposal can also be related to proof-carrying code by Necula and Lee [11] as the generated code contains annotations required to verify safety properties. But, in their setting the purpose of the carried proof was to ensure the correctness of the executed code dynamically and to replay the proof during execution. We do not have such need in our work as the correctness proof can be performed once and for all prior to the compilation of the generated C code.

A similar approach was proposed by Pires et al. in [14] for the verification of hand-written code. The system specification is written using the UML and OCL languages. The implementation is hand written as the models are not low level enough to generate the whole software. The generated annotations are not sufficiently low level either to ensure the automatic verification of the user code and the user may need to provide intermediate annotations and conduct the proof. We believe and partly experimented that relying on formally verified specification such as provided by the BlockLibrary language is providing a higher level of automation and thus eases the verification task.

6 Conclusion and Future Work

To summarize, we have advocted the advantages brought by the formal specification of an existing complex language such as Simulink. We designed the BlockLibrary DSL for capturing the specific complex features of block libraries and have shown how to use that for verifying the correctness of generated code in a formal translation validation style approach. The verification technique has been demonstrated here on a simple block, but the *white box* strategy has been experimented on a representative subset of the Simulink blocks selected by the industrial partners of the GeneAuto project.

In the close future, we plan to prototype the *black box* annotation generation process including the generation of loop annotations to simplify writing specifications of complex blocks. The BlockLibrary specification language will be used as a source for the automatic generation of a reliable set of test cases for the verification of the ACG implementation. The test cases will be used for both simulation and code generation. The results will then be compared for the verification and validation of the ACG. Another application direction is using the block-level annotations in conjunction with top-level formal contracts expressed for the entire system to aid proving the functional correctness of the latter. Our current experiments have been performed mainly using the GeneAuto ACG. We are currently extending the whole work and, among other things, are adapting it for the industrial *DO-330* conformant qualification process of the QGen ACG that is a successor of GeneAuto.

Acknowledgements This work has been funded by the French and Estonian Ministries of Research, Industry and Defense through the Projet-P[14], Hi-MoCo[15] and Vorace[16] projects and through the Estonian Ministry of Education and Research institutional research grant no. IUT33-13. The authors wish to thank the members of these, the QGen project and the anonymous reviewers of this paper for providing valuable feedback for improving the work.

References

1. ANSI/ISO C Specification Language (ACSL). http://frama-c.com/acsl.html
2. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Proc. of 1st International Workshop on Intermediate Verification Languages (Boogie 2011). pp. 53–64. (2011)
3. Dieumegard, A., Toom, A., Pantel, M.: Model-based formal specification of a DSL library for a qualified code generator. In: Proc. of 12th Workshop on OCL and Textual Modelling. pp. 61–62. ACM, New York. (2012)
4. Dieumegard, A., Toom, A., Pantel, M.: A software product line approach for semantic specification of block libraries in dataflow languages. In: Proc. of the 18th

[14] http://www.open-do.org/projects/p/

[15] http://www.adacore.com/press/project-p-and-hi-moco/

[16] http://projects.laas.fr/vorace/

International Software Product Line Conference, SPLC '14. pp. 217–226. ACM, New York. (2014)

5. Filliâtre, J.C., Pereira, M.: A Modular Way to Reason About Iteration. In: Rayadurgam, S., Tkachuk, O. (eds.), NFM 2016, LNCS, vol 9690, pp. 322-336. Springer, Heidelberg (2016)
6. Furia, C.A., Meyer, B.: Inferring loop invariants using postconditions. In: Blass, A., Dershowitz, N., Reisig, W. (eds.) Fields of Logic and Computation, Essays Dedicated to Yuri Gurevich on the Occasion of His 70th Birthday. LNCS, vol. 6300, pp. 277–300. Springer, Heidelberg (2010)
7. Izerrouken, N., Pantel, M., Thirioux, X.: Machine-checked sequencer for critical embedded code generator. In: Breitman, K., Cavalcanti, A. (eds.) ICFEM 2009. LNCS, vol. 5885, pp. 521–540. Springer, Heidelberg (2009)
8. 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 (1990)
9. Lee, E., Messerschmitt, D.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Transactions on Computers, **C-36**(1), 24–35 (1987)
10. Liu, S., Offutt, A.J., Ho-Stuart, C., Sun, Y., Ohba, M.: SOFL: A Formal Engineering Methodology for Industrial Applications. IEEE Transactions on Software Engineering **24**(1), 24–45 (1998)
11. Necula, G.C., Lee, P.: Safe kernel extensions without run-time checking. SIGOPS Operating Systems Review **30**, 229–244 (1996)
12. Ngo, V., Talpin, J.P., Gautier, T., Le Guernic, P., Besnard, L.: Formal verification of synchronous data-flow program transformations toward certified compilers. Frontiers of Computer Science **7**(5), 598–616 (2013)
13. O'Halloran, C.: Automated verification of code automatically generated from Simulink. Automated Software Engineering **20**(2), 237–264 (2013)
14. Pires, A.F., Polacsek, T., Wiels, V., Duprat, S.: Behavioural verification in embedded software, from model to source code. In: Moreira, A., Schätz, B., Gray, J., Vallecillo, A., Clarke, P.J. (eds.) MODELS 2013, LNCS, vol. 8107, pp. 320–335. Springer, Heidelberg (2013)
15. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)
16. Ryabtsev, M., Strichman, O.: Translation validation: From Simulink to C. In: Bouajjani, A., Maler, O. (eds.) CAV 2009, LNCS, vol. 5643, pp. 696–701. Springer, Berlin Heidelberg (2009)
17. Toom, A., Naks, T., Pantel, M., Gandriau, M., Wati, I.: Gene-Auto - an automatic code generator for a safe subset of Simulink-Stateflow and Scicos. In: ERTS 2008. Société des Ingénieurs de l'Automobile, http://www.sia.fr (2008)
18. Wang, T.E., Ashari, A.E., Jobredeaux, R.J., Feron, E.M.: Credible autocoding of fault detection observers. In: 2014 American Control Conference. pp. 672–677 (June 2014)
19. Wiik, J., Boström, P.: Contract-based verification of MATLAB-style matrix programs. Formal Aspects of Computing **28**(1), 79–107 (2016)