IEC 61131-3 Model for Model-Driven Development

Monika Wenger and Alois Zoitl
Vienna University of Technology
Automation and Control Institute
Gußhausstraße 27-29, 1040 Vienna
wenger@tuwien.ac.at

***Abstract*—Existing industrial automation projects require automated methods for faster changes and updates as well as continuous improvement and maintenance. Software Engineering uses Model Driven Development to shorten the development time by keeping documentation, models and code consistent during the whole development time and by using code generators to create many similar code artifacts and code for different platforms.**
To make this concept also available for automation projects we present an IEC 61131 model based on the Extended Backus Naur From provided by the standard. For the proposed IEC 61131 model also related application areas are outlined.

I. Introduction

Steadily growing and changing industrial automation projects lead to code aging and increasing complexity. Maintenance work may degrade code quality, which makes further maintenance and system reconfiguration even more difficult. The improvement of code quality for industrial automation projects is still a manual and time consuming task since there are currently no code analysis or refactoring methods available for IEC 61131 languages.
Software Engineering provides Model Driven Development (MDD), a method to address code quality improvement, handling the increasing complexity, simpler integration of new functionality, minimization of repeats and cost reduction due to reusability for different platforms. Providing MDD also for industrial automation projects offers automatic code quality checking and improvement and therefore reduces engineering costs. There has already been some effort to establish MDD within the industrial automation domain.
In [1] Unified Modeling Language (UML) is used to bridge the gap between hardware and software engineering by automatically generating IEC 61131 Structured Text (ST) and Sequential Function Chart (SFC) out of an UML model. This process supports the entire life cycle by an appropriate view for each phase of the project.
The work of [2] concerns the question how additional description methods like SysML, UML or IEC 61499 can be connected to the IEC 61131 standard, especially Function Block Diagram (FBD), to build more abstract models, to display different aspects of a system in a proper manner and to overcome the early link to a target platform caused by the direct use of IEC 61131.
In [3] eXtensible Markup Language (XML) is used to improve the tool interoperability of industrial automation systems based on IEC 61131. They provide a method to close the gap between heterogenous tools used within different development phases by the use of a functional model consisting of a hardware model, a software model and their relationships. The entire application is then generated out of this functional model.

The work of [4] aims at automatically closing the gap between a logical model designed by Pipe and Instrumentation Diagram (P&ID) and its implementation model based on IEC 61131. For closing this gap and for unifying and improving the development process of control applications Systems Modeling Language (SysML) is used as intermediate model from which not only any procedural or Object Oriented (OO) IEC 61131model but also an IEC 61499 model or a Service-oriented Architecture (SoA) model is supposed to be generated. The SysML profile contains a logical model of the system which describes system reactions and their interrelationships. As output model a simplified IEC 61131 model is provided.

In [5] a code generator is presented which is able to produce modular Instruction List (IL) and ST code for different Programmable Logic Controllers (PLCs) out of a controller model. The controller is designed with Timed Net Condition/Event Systems (TNCES) which has a petri-net based structure.

Most of the presented work generates PLC code out of one or more abstract models. For PLC code generation template languages are used where no grammar or meta-model is needed for the target language. Within this work we present an IEC 61131 model not only covering the general elements of the IEC 61131 standard but also the different languages provided by the standard. This model is usable not only to improve tool interoperability but also for the transformation of ST based IEC 61131 projects into other languages and respectively documentation. It also makes code analysis methods available especially for ST which provides information on code quality and mechanisms for an automatic improvement.

At first the field of application for such an IEC 61131 model is described in more detail as well as the considered technologies and its integration into an MDD process. The IEC 61131 model is presented afterwards as central element of the process. Then this model is applied on an 8BitCounter example built in CoDeSys [6]. The work is summed up at the end and an outlook is provided.

II. Use of an IEC 61131 model

The use of an IEC 61131 model not only covering the general structure of IEC 61131 but also considering the languages provided by the standard, allows many application possibilities. Fig. 1 shows the IEC 61131 model as the central point of MDD applications, whereas Model-to-Code (M2C) technologies can be identified as general starting point.

A. IEC 61131-3 model application

The MDD applications supported by an IEC 61131 model can be divided into three types. The first type concerns the use of Code-to-Model (C2M) technologies. These C2M technologies read code into a model also known as Abstract Syntax Tree (AST). This AST can then be used for further processing like code analysis techniques or transformations. In Fig. 1 C2M technologies are used to parse a CoDeSys based PLC program into the IEC 61131 model.

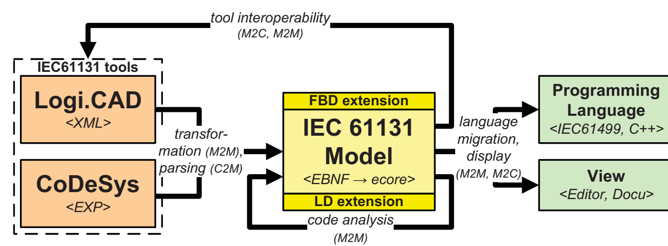

Fig. 1. IEC 61131 model application

The second MDD application type concerns the use of Model-to-Model (M2M) technologies. These technologies allow the conversion of an existing model into another model through proper transformation rules. M2M therefore does not only support interoperability between different tools but also model based language migration. The precondition for M2M is the existence of a grammar or meta-model for each input and output model. In Fig. 1 M2M technologies are used to read a PLC project designed by logi.CAD into the general IEC 61131 model, to convert this general IEC 61131 model into another IEC-61131 tool or to transform it into another language like IEC 61499.

The third MDD application type concerns the use of M2C technologies. M2C technologies allow the generation of source code out of a model. Since there exist template languages for code generation a meta-model is only needed for the source model. In Fig. 1 M2C technologies are used to, generate C++ code for a specific runtime environment or to generate a documentation for example in terms of Hypertext Markup Language (HTML) files.

For projects without a valid and accessible meta-model or grammar, like CoDeSys based projects, the use of C2M technologies is a precondition to get the desired data into a processable model supporting the possibilities shown in Fig. 1.

B. Code-to-Model technologies

There are several tools supporting MDD. Since Eclipse is mostly used at research facilities its technologies are of major interest. Eclipse provides two C2M technologies Xtext [7] and EMFText [8]. Both of them allow the definition of own textual languages and a corresponding infrastructure like an editor where the designed language is read into an model dependent on the defined grammar. We tried out both of them for generating an IEC 61131 editor with syntax highlighting and designing the desired IEC 61131 model. Fig. 2 illustrates our experiences with Xtext and EMFText.

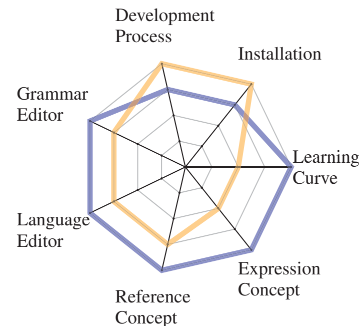

Fig. 2. Code-to-Model technologies: EMFText (blue) vs. Xtext (orange)

We divided our evaluation criteria into four scale units and seven categories. The scale units indicate how good we think the specific category is addressed by the technology.

The first category concerns the installation of the C2M technologies. Xtext can be integrated into existing Eclipse by the Eclipse update site whereas for EMFtext it has to be done by the project's update site. Therefore the installation of Xtext is slightly easier than those of EMFtext.

The second category concerns the development process used for those technologies. Both technologies need an abstract as well as a concrete syntax. For Xtext the abstract syntax is derived automatically from the user defined concrete syntax but there is also a possibility to provide a model for the abstract syntax whereas for EMFtext both the concrete and abstract syntax has to be defined. Xtext already preconfigures transformation technologies for M2M and M2C whereas in EMFtext this integration has to be done manually. This makes the development process if Xtext a bit more comfortable than those of EMFtext.

The third and fourth category concerns the editors provided for the specific C2M technology and generated by such a technology. The grammar editor where the concrete syntax of the language is designed, provides syntax highlighting, on the fly parsing with error/warnings and auto-completion in both cases. Additionally EMFtext provides color definitions for tokens and keywords within its concrete syntax definition file. It is also possible to restrict containment references on specific subclasses within EMFtext.

The language editor which is generated by both technologies also provides syntax highlighting, on the fly parsing with error/warnings and auto-completion in both cases. Both technologies also generate an extendable outline view whereas those of EMFtext are by default more detailed than those of Xtext. The outline generated by EMFtext also shows the

element type names defined within the abstract syntax which makes it easier to identify errors especially for large grammars.
The fifth category concerns the reference concepts provided by both technologies. Xtext as well as EMFtext provide containment and non-containment references. In EMFtext containment references can be restricted that they only use specific subclasses. For non-containment references Xtext allows cross-references across file boundaries, as long as the referenced models are on the classpath. EMFtext does not automatically provide this feature but the generated resolvers can be rewritten easily to resolve references to any desired file. The non-containment reference handling also seems to be easier if the abstract syntax behind the grammar rules is self-defined.
The penultimate concept concerns the definition of expressions. Both technologies provide similar mechanisms to describe expressions. Xtext uses implicit priority settings since they are derived from the hierarchy of the designed grammar. The later rule is defined within the hierarchy the higher the priority. Besides implicit priority settings which may lead to large syntax trees, EMFtext also provides explicit priority settings which keep the parser information separate of the entire grammar by the use of annotations.
The last category concerns the observed learning curve of both technologies. The user guide of Xtext is quite long compared to those of EMFtext. But even if the user guide of EMFtext is more compact both of them describe the use of the particular technology quite well. Xtext also provides some example grammars whereas EMFtext offers a whole syntax zoo including small examples for special concepts, like simplemath for expressions and lwc11 (Language Workbench Contest 2011) for resolver rewriting, but also large examples like a complete Java5 syntax. As beginner also the screencast provided by EMFtext is very helpful. For both technologies first results can be achieved quite easy. But when designing larger languages like IEC 61131 it seems to be easier when the abstract syntax stays under developer control. And also the existence of small examples showing specific technology concepts supports a faster progress.
Our experiences with both technologies are summed up in Fig. 2. Based on this results we decided to continue with EMFtext for creating the desired IEC 61131 model. The IEC 61131 model equates to the abstract syntax whereas the grammar rules equate the concrete syntax.

III. IEC 61131-3 MODEL

The proposed IEC 61131 model is based on the Extended Backus Naur Form (EBNF) description provided by the IEC 61131-3 standard [9, Annex B]. The EBNF description provides the basic programming model of IEC 61131-3 which defines possible library elements. It also contains definitions for letters, digits, identifiers and literals, all possible data types and variable types as well as definitions for Configurations and Program Organization Units (POUs). Also three of five IEC 61131-3 languages (ST, IL, SFC) are described.
The model designed within this work mainly reflects the provided EBNF description. The general structure and the challenges during the creation of the IEC 61131 model are outlined in the following sections.
Basically there have been three reasons for changing the provided EBNF descriptions. The first need for change is caused by the fact that the use of containment and non-containment references has to be gathered out of the context since EBNF does not explicitly provide them. The second need for change is due to the model-driven approach and the chosen technology. The last reason for change concerns redefinitions which had to be minimized to overcome parsing problems.

A. General structure

In general the structure provided by the standard has been adopted. But when having a look at [9, Annex B] it becomes quite clear that this EBNF description results in a large model. To structure the IEC 61131-3 model and make it better handable it has been divided into several smaller packages:

- IEC 61131 model root: basic programming model and extension for standard library containing IEC 61131 POUs
- IEC 61131 main model: *types* for data type names, *operators* containing all existing operators, *literals* for literal definitions, *variables* for variable types, *interfaces* for interface elements and *Global Variables*, *pous* for POU elements, *configurations* for a whole IEC 61131 project
- IEC 61131 programming languages: *il* for IL, *sfc* for SFC, *st* for ST, *ld* for Ladder Diagram (LD), *fbd* for FBD

The standard library part has been added since POUs often instantiate existing POUs or make use of them. Those library blocks are unknown during parse time. Usually no source code is available for such library blocks but nevertheless their names and their interfaces are needed for parsing. Therefore a library block model has been added which is extendable without compilation of the grammar and also accessible by an Integrated Development Environment (IDE). Used library blocks therefore point to an external library file containing the missing blocks and their interfaces. To simplify reference resolving and avoid repeated definitions existing descriptions have been reused to model the external library blocks. Standard library blocks are therefore modeled just as user defined blocks. Every existing POU type, like Program (PROG), Function Block (FB) or IEC 61131-3 Function (FUNC), can be part of that library model. Since the internal functionality of such blocks is unknown the body element is set to `other language`. In contrast to the user defined blocks, standard library blocks are not stored in textual manner but as XML Metadata Interchange (XMI) file which already represents the IEC 61131 model and therefore has not to be parsed into it any more. Some standard library blocks require the use of generic inputs and outputs. But the definitions provided by the standard only allow elementary types and simple types as input and output variable type. By adding generic types to the definition of `simple_specification` and the `function_declaration` element its is now possible to define blocks with generic types like the `LIMIT` function. This

library model made elements like standard_function_name and standard_function_block_name redundant so that they could be removed.
The data type structure provided by the standard has been adopted. But a similar concept than those of the standard library has been added to put all existing types into an external XMI file. In both cases the standard library as well as the data type the generated resolvers had to be changed that they allow a reference to an external file.
Operator names and symbols are used repeatedly within the EBNF description provided by [9, Annex B]. Since such reputations lead to an unclear model design, which can cause errors at parser generation time, an own operator model with name and symbol elements has been added. The name element represents the function name used within ST code whereas the symbol element mainly concerns the symbols allowed within ST code for arithmetic functions. The operator model combines existing operator definitions like the comparison_operator with some new ones to cover all existing operators. Some elements have been added to group the operators like the equUequ_operator element for the equal and unequal operators. Since the multiply_operator also includes the modulo and divide operator it has been changed to dot_operator. To avoid redefinitions within the IL definitions an arithmetic_name element and a comparison_name element has been added.
Some repeatedly used tokens in the standard are NIL and EOL. The former has been replaced by 0..* containment references instead of the suggested 1..* reference whereas the latter which is mainly used for IL definitions has been removed since its use caused errors in other language parts.
When designing the SFC model part it turned out that the IEC 61131 standard uses simple_intruction_list element for transition conditions of SFC. This element does actually not exist and therefore has been replaced by the il_simple_instruction element.
The standard does not provide an element which is able to handle comments this is why comments would be lost after the creation of the AST. Therefore a comment element was added to our IEC 61131 model. All elements which are supposed to have comments inherit this comment element.

B. *Challenges of IEC 61131 modeling*

Basically the challenges during the creation of the IEC 61131 model out of the EBNF definition of the standard can be divided into three types. These challenge types concern the left recursive nature of some model elements but also the differentiation between containment and non-containment references as well as some overlapping definitions.

1) Left recursion: The IEC 61131 model is supposed to be used for the generation of a parser. Usually parsers only accept left or right recursive grammars but not a mixture of both. According to [10] a rule of a grammar is left recursive if it contains a non-terminal which calls the rule itself either directly (direct left recursion) or in any derived non-terminal (indirect left recursion).

The definitions within [9, Annex B] are not exclusively right recursive but also contain left recursive ones. On the one hand this concerns Arrays and Structures and on the other hand the Expression definitions. To resolve such left recursive rules EMFtext provides Operator Precedence Annotations which allow explicit priority settings for specific grammar rules. But these Operator Precedence Annotations require some changes within the model since all involved elements need a common parent element.
The EBNF definitions for Arrays and Structures provided by [9, Annex B] result in a left recursive model which can not be handled by EMFtext's underlying parser generator technology. Listing 1 shows some examples for the left recursive EBNF definition of Arrays and Structures in the lines 2 to 6. EBNF defines that an Array initialization element contains not only other Array initializations but also Structure initializations whereas Structure initialization elements not only contain other Structure initializations but also Array initializations.

```
1  array_1 : ARRAY [0..1] OF INT;
2  array_2 : ARRAY [0..1] OF array_1 :=
3      [[1,2], [3,4]];
4  array_3 : ARRAY [0..1] OF struct_1 := [
5      (array_1 := [1,2], real_1 := 2.2),
6      (array_1 := [4,5], real_1 := 5.5)
7  ];...
8  TYPE struct_1 : STRUCT
9      array_1 : ARRAY [0..1] OF INT;
10     real_1 : REAL; ...
```

Listing 1. Left recursive Arrays and Structures

To resolve the left recursiveness of Arrays and Structures the common parts of array_initial_element and structure_element_initialization have been combined within one new element. Fig. 3 outlines the new structure.
Since operator classes can not be used outside the inheritance tree an additional layer has been added to break the inheritance tree. By inserting a containment reference it is possible to reference the enumerated_value, structure_initialization and array_initialization elements everywhere in the model.

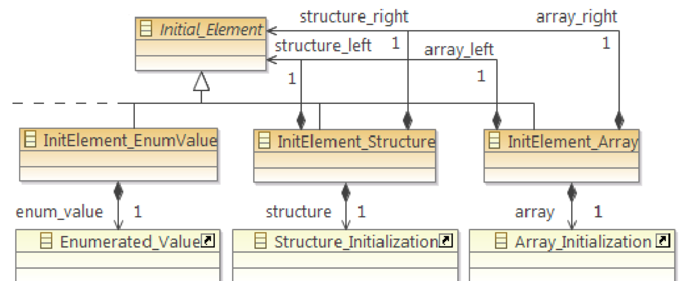

Fig. 3. Model changes for Arrays and Structures

The definitions of Expressions provided by the IEC 61131-3 standard can lead to huge syntax trees even for simple ST code. Models with huge syntax trees are difficult to handle for further processing especially code analysis. To avoid such

huge syntax trees and better support further model analysis methods the provided definitions have been redesigned.
Annex B of the IEC 61131-3 standard defines a hierarchical structure for Expressions. Changing this hierarchical structure to a flat model leads to the need of explicit priority settings to remain compliant to the original EBNF definition but also not resulting in an left recursive model.
The flattened Expressions model is structured in a similar way than those shown in Fig. 3. All elements of the former hierarchy are inherited of the new class expression_type. For each new expression element one left and one right side expression of the common parent class has been added. The former hierarchy could then be rebuild without running into large trees, by defining priority settings based on the former hierarchy. The inheritance tree had to be broken by containment references when the specific element where needed outside the operator class hierarchy. This mechanisms allowed the constructions of an optimized expression tree.

2) Overlapping elements: There are two parts of the standard affected by overlapping. On the one hand those variable definitions which refer to an existing FB and on the other hand parts of the constant description.
The first overlapping part concerns the interface descriptions since they are not able to resolve both elementary type variable declarations and instances of FBs at the same time. Listing 2 shows an example where variable_1 and block_1 are both considered to be a fb_name_decl or a var1_init_decl dependent on the definition order within the concrete syntax. The definition put at the second place within the concrete syntax definition could never be reached.

```
variable_1 : INT;
block_1 : F_TRIG;
```

Listing 2. Resolving variable and Function Block references

To resolve both types INT as elementary_type_name and F_TRIG as derived_function_block_type, the original simple_specification of [9, Annex B] has been extended by a function_block_type_name element and the generated resolver has been changed to refer to both files the standard library as well as the type definitions.
The second overlapping part concerns the constants of [9, Annex B] which contain all literal definitions. At some points these definitions overlap each other and also influence other definitions like CASE statements and ARRAY definitions especially their range definitions.
Both the real_literal and the fixed_point definition contain a number '.' number part as it is shown in Listing 3 line 1 and 3. The fixed_point definition also overlaps with the signed_integer definition since it also allows the use of a single number as it is shown in Listing 3 line 2 and 4.

```
real_1 : REAL := 1.3E-3;
int_1 : INT := 1;
time_1 : TIME := t#1.3ms;
time_2 : TIME := t#2ms;
```

Listing 3. Overlapping literal definitions

To avoid the overlapping of the fixed_point, integer and real_literal definition an unsigned_integer has been introduced. The fixed_point definition has been restricted to only allow number '.' number but inherits from a fixed_point_literal has been added which can also be an unsigned_integer. Then real_literal uses fixed_point instead of redefining the number '.' number construction.
The optional signs of all number definitions could neither made optional by adding it to a token definition nor by adding it to a rule. Therefore the signs have been represented as boolean which is set true for negative numbers.
The exponent definition contains ('E'—'e') which would be interpreted as keyword during parsing this would also mark a variable e as keyword. This is why an EXPONENT token had been defined which also included the sign of an exponent.

3) Containment and non-containment references: For Variables both containment and non-containment references are needed. A containment reference describes a variable declaration at the beginning whereas the non-containment reference describes all further variable uses. Especially for defining the primary_expression the standard's EBNF description mixes both types of references.

```
TYPE
  struct_type : STRUCT
    int_1 : INT;
    real_1 : REAL;
  END_STRUCT;
END_TYPE ...
VAR_INPUT
  variable_1 : INT;
  array_1 : ARRAY [0..1] OF INT;
  struct_1 : struct_type;
END_VAR
... := variable_1;
... := %IB1;
... := array_1[1];
... := struct_1.int_1;
```

Listing 4. Containment and non-containment references

The variable element within the EBNF definition provided by [9, Annex B], contains four types. The different types are direct_variable, variable_name, array_variable and structured_variable as they are outlined in Listing 4. Not all of these variable types can be displayed as non-containment reference since at least array_variable and structured_variable contain elements which have to be instantiated.
Fig. 4 shows a part of the revised Expression model where a expression_variable element has been added to represent the four variable types and their corresponding reference types. The expression_variable element has been inserted to break the inheritance hierarchy and allow its use also for non operator classes outside the Expressions definitions.

IV. Code-to-Model Test Case

The presented IEC 61131 model has been defined as Ecore model. Based on this model the concrete syntax of the

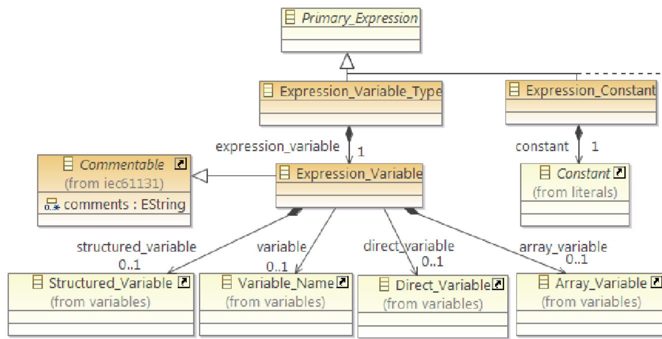

Fig. 4. Variables within expressions

IEC 61131 has been rebuilt. Out of the abstract and the concrete model Java code for an IEC 61131 editor has been generated. To verify the correctness of these models an 8BitCounter has been chosen as representative example since it is still manageable but also offers features of an real automation project like statefulness, reactiveness on signal changes, and timing-behavior.

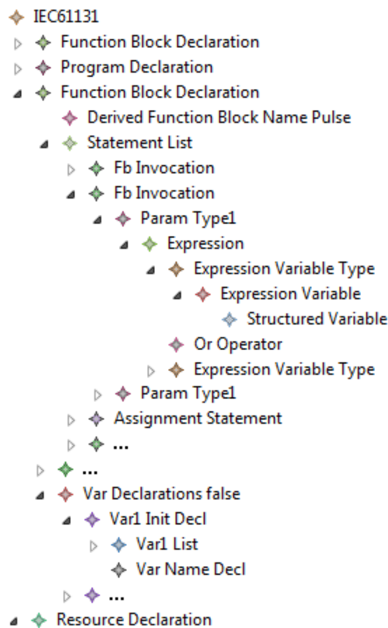

Fig. 5. Resulting syntax tree of 8BitCounter

The 8BitCounter increments an eight bit number dependent on the given pulse settings. It provides three modes, Single, Permanent and a both modes active. The Single mode only increments once. The Permanent mode increments at every rising edge of the pulse generator. Setting both modes active prevents incrementing.

The original 8BitCounter has been provided by an Austrian IEC 61131 tool provider, logi.cals [11]. Their 8BitCounter was exported as ST code and adapted for its use within CoDeSys. The CoDeSys project export (EXP) has then be used as input for the generated IEC 61131 editor.

The resulting syntax tree is outlined within Fig. 5. All references to variables, external types and POUs could be resolved correctly. The tree hierarchy for the ST code inside every POU also remained manageable. The resulting syntax tree therefore provides a proper starting point for further processing like code analysis methods.

V. Conclusion

In this work an IEC 61131 model based on the EBNF definition in [9, Annex B] has been introduced. Also its application area to establish MDD within the industrial automation domain has been described. The abstract and concrete IEC 61131 model has been verified by a representative example. The results showed its usability for further processing methods like code analysis or language migration.

In a further step the IEC 61131 model will be tested with more complex examples to integrate missing parts like an extended FBD model and to solve identified difficulties. One challenge concerns timing constants like T#1s1ms which can currently only be used with blanks T#1s 1ms. Also further work on the initialization values is needed to guarantee that they fit to the defined data type. Some of the generated resolvers for types and external POUs have to be rewritten as well.

The refined model will then be used to create a first version of a ST to IEC 61499 transformation and as a further step also a combined transformation with FBD. The provided model also provides a possibility to use POUs from different different vendors within one project.

References

[1] B. Vogel-Heuser, D. Witsch, and U. Katzke, "Automatic code generation from a UML model to IEC 61131-3 and system configuration tools," in *IEEE International Conference on Control and Automation (ICCA)*, 2005, pp. 1034–1039.

[2] G. Frey and K. Thramboulidis, "Integration of IEC 61131 into model driven development processes," in *Proceedings of the Kongress Automation, VDI-Berichte 2143*, June 2011, pp. 21–24, Baden-Baden, Germany.

[3] E. Estévez and M. Marcos, "An approach to use model driven design in industrial automation," in *13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2008, Hamburg, Germany.

[4] K. Thramboulidis and G. Frey, "An MDD process for IEC 61131-based industrial automation systems," in *15th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Septemper 2011, Toulouse, France.

[5] J. Thieme and H.-M. Hanisch, "Model-based generation of modular PLC code using IEC 61131 function blocks," in *IEEE International Symposium on Industrial Electronics (ISIE)*, vol. 1. Department of Engineering Science, Martin-Luther-Universität, Halle-Wittenberg, 2002, pp. 199–204.

[6] 3S-Smart Software Solutions GmbH, "Codesys," Online, June 2012. [Online]. Available: http://www.3s-software.com

[7] H. Behrens, M. Clay, S. Efftinge, M. Eysholdt, P. Friese, J. Köhnlein, K. Wannheden, S. Zarnekow, and contributors, *Xtext User Guide*, 2010. [Online]. Available: http://www.eclipse.org/Xtext

[8] Core Developer Team, *EMFtext User Guide*, 2011. [Online]. Available: http://www.emftext.org

[9] IEC SC65B, "IEC 61131-3 – programmable controllers – part 3: Programming languages," International Electrotechnical Commission, 2003, Geneva.

[10] K. D. Cooper and L. Torczon, *Engineering a Compiler*, 2nd ed. Elsevier, 2012, iSBN: 978-0-12-088478-0.

[11] logi.cals, "logi.CAD - Online-Help - example: 8-bit-counter," June 2012. [Online]. Available: http://www.logicals.com/downloads/manuals/