CERN-ACC-NOTE-2013-0040

2013-11-14

{ddarvas,bfernand,eblanco}@cern.ch

Transforming PLC programs into formal models for verification purposes

Daniel Darvas, Borja Fernandez Adiego, Enrique Blanco
EN/ICE/PLC, CERN

Keywords: PLC, Formal methods, Program modelling, Verification, UNICOS

Abstract

Most of CERN's industrial installations rely on PLC-based (Programmable Logic Controller) control systems developed using the UNICOS framework. This framework contains common, reusable program modules and their correctness is a high priority. Testing is already applied to find errors, but this method has limitations. In this work an approach is proposed to transform automatically PLC programs into formal models, with the goal of applying formal verification to ensure their correctness. We target model checking which is a precise, mathematical-based method to check formalized requirements automatically against the system.

Contents

1 Introduction

This chapter overviews the background and the motivation of the current work. To support the motivation, CERN and the UNICOS framework are introduced first.

1.1 CERN

CERN is the European Organization for Nuclear Research, founded in 1954. The goal of this particle physics laboratory is to study and understand the fundamental structure of the universe, as well as application and transfer of new technologies. For this reason, they use the world's largest and most complex scientific instruments [1]. These complex instruments, the particle accelerators and detectors need accurate and reliable auxiliary systems, like cooling and ventilation, cryogenics, gas systems, etc.

Many of these systems need conventional industrial controllers and for that reason, PLCs (Programmable Logic Controllers) are widely used. In order to increase the efficiency of the development of process control systems based on PLCs, the engineers at CERN reinforced the concept of standardization of the control systems. They developed a framework called UNICOS (Unified Industrial Control System) that makes possible to reuse common components of the control system by using standard libraries.

1.2 UNICOS

UNICOS[1] is a CERN in-house framework to develop industrial control applications [2]. It mainly covers the upper two layers of the classical industrial process control systems: the supervision and the control layers.

This framework consists of a library of generic objects, a development methodology, and finally, a code generation tool.

UNICOS contains about 20 generic objects, called **baseline objects**. These objects are representing I/O objects (e.g. digital and analogue inputs and outputs), interface objects (parameters and statuses exchanged between the supervision and control layers), field objects (representations of physical real equipment, e.g. valves, motors, heaters, etc.) and control objects (e.g. alarms, PID control) [2]. These are the base components of every UNICOS program. Beside the baseline objects, the UNICOS projects consist of an application-specific **specification files** and the implementation of the **custom** (application-specific) **logic**. The specification details the instantiation of the baseline objects and the connections between them. The custom logic contains the application-specific PLC code. Based on the baseline objects and the application-specific data, the source code for the supervision and control layers can be generated automatically. This makes the application development easier and faster, and also reduces the risk of faults by reducing the amount of handwritten source code.

1.2.1 UNICOS applications in real life

UNICOS-based applications are used widely at CERN in large installations as the LHC (Large Hadron Collider) , particle physics detectors (e.g., ALICE, CMS, ATLAS) and other experimental facilities (e.g. ISOLDE) mainly for the industrial control of the auxiliary systems as cooling, gas , cryogenics[2] and vacuum systems [3]. UNICOS has also been applied in other areas as interlock based applications (LHC collimators) or motion systems (e.g. winding machines, ATLAS Big wheels…).

[1] It has no connection to the operating system developed by Cray Research, Inc.for the Cray supercomputers which has the same name UNICOS.
[2] Cryogenics is the study of the very low temperature. In the LHC, superconducting magnets are used which are cooled below 2 K.

1.3 Motivation

As it was described earlier, every UNICOS application is based on the same baseline objects. The source code of these objects has been created manually based on a non-formal specification. As it is a common component of every UNICOS application, it is crucial to ensure that the code of these components is correct. Indeed, manual and automated testing is already applied for the baseline objects, however, no formal verification was used since their development.

Therefore a project started in the EN-ICE-PLC section of CERN to verify formally the baseline objects and the generated PLC code. This work is part of this project and **the current goal was to create a formal model representation for the baseline objects for verification purposes**.

The source code generated by UNICOS is a manufacturer-dependent ST (Structured Text) code, currently producing code for Siemens and Schneider and therefore multiple variants exist. Similarly, multiple model checkers exist with different model formats. Because of the similarities and the limited resources, this work focuses only on one input and one output language: namely the Siemens SCL format as input language and the NuSMV model format as output language.

1.4 NuSMV model format

NuSMV is a symbolic model checker developed by FBK-IRST, Carnegie Mellon University, University of Genova and University of Trento [4]. For this work, NuSMV is considered as a black box; only the input model format is of our interest. This input language is briefly introduced here.

The input language of NuSMV describes the input model as a Finite State Machine (FSM) whose states are determined by the current values of variables. The definition of a model consists of **variable declarations** and the **transition rules between states**. There are two possible definition syntaxes for the transition rules: with the *TRANS* keyword, *(from_state; to_state)* pairs can be declared, with the *ASSIGN* keyword, the next states of each variable can be declared. If no next state is defined for a variable, then it will non-deterministically get a value from its range.

This is a very short overview of the NuSMV language. NuSMV is much richer as it supports the usage of modules and module instances, invariants, initialization statements, special variables, constants, fairness constraints, etc. For a detailed overview, please refer to [4].

2 Overview of the workflow

The first part of the work was to propose a workflow for the transformation between the PLC ST source code (Siemens SCL format) and the formal model (NuSMV format).

The first step of the workflow is to create an easy-to-handle **representation of the input language**. It is not needed to create a parser from scratch, because several already existing tools can help this work. To parse the ST code, more precisely to build the abstract syntax tree from a textual ST code representation, Xtext [5] is used. Xtext is a well-known Eclipse-based framework for Domain Specific Languages. It creates an object model (abstract syntax tree, AST), a parser and an editor based on a given grammar. Therefore a grammar for the Siemens SCL language had to be developed.

After the abstract syntax tree is given, it has to be translated to the NuSMV format. The first idea was to directly translate the parsed ST code (more precisely, the abstract syntax tree representation of the ST code) to the output language. A proof-of-concept translator was developed during the first days, which worked well for the simple constructions (like variable assignments, conditional statements), but as soon more complex constructions were added (like function calls), it became complicated and unmaintainable. The problem is that two different transformations are needed: the program flow has to be represented by automata, and also the automata representation has to be translated to NuSMV. The first one is a “semantic” transformation, while the second is a

"syntactic" transformation. A related problem is that during the output generation, the order of the operations is highly influenced by the format of the output, which makes the transformation even more difficult.

For this reason, a new approach is proposed, which decouples the two translations by introducing an **intermediate model**. This intermediate model represents a network of automata, which is close to the theory of the automata, in this way it can be relatively easily represented in NuSMV, but in other modelling languages as well. It also contains some features that are useful for the representation of ST codes (like transition synchronization, which is missing from the NuSMV language). Furthermore, this intermediate model can be built in an order which is appropriate for the program flow, the output syntax does not influence it. The intermediate model can make the approach more flexible, as it will be discussed later.

In this way, the workflow of the transformation between ST code and NuSMV model can be drawn up, as it can be seen on Fig. 1.

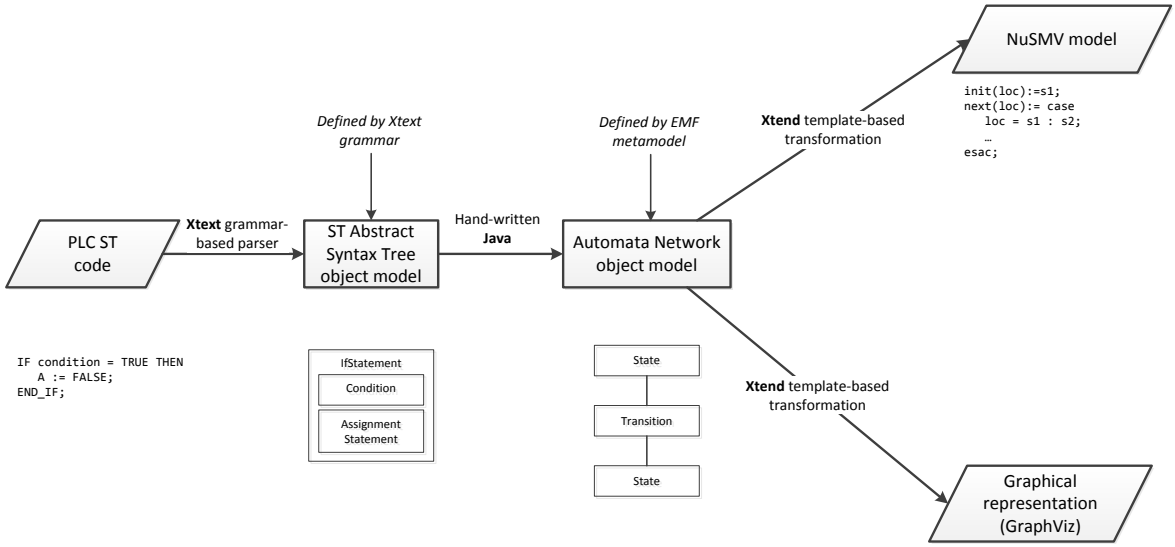


Fig. 1: Overview of the workflow (with examples)

The workflow consists of three main phases:

1. **Parsing** PLC ST source code (creating in-memory object representation of the source code).
2. **Transforming** the parsed ST source code to an intermediate automata model.
3. **Creating** NuSMV formal **models** (and visualization) from the intermediate automata model.

The following sections overview each phase in details.

3 Parsing ST code

3.1 Xtext

Xtext [5] is an open-source Eclipse-based framework for developing domain specific languages. It covers all the main needs: it generates an editor, a parser and an abstract syntax tree model from the given grammar.

As input, Xtext uses a custom grammar format. This format is similar to the widely used BNF (Backus–Naur Form), but it has some extensions which enables Xtext to create easy-to-use object models. For example, let's see the next small grammar in BNF:

```
<Variable> ::= <ID> ':' ('INT' | 'BOOL') ';'
<VariableAssignment> ::= <ID> ':=' (<IntLiteral> | <BoolLiteral>) ';'
```

The same grammar would be represented in Xtext format like this:

```
Variable: name=ID ':' type=('INT'|'BOOL') ';' ;
VariableAssignment: variable=[Variable] ':=' value=(IntLiteral|BoolLiteral) ';';
```

The first difference between the BNF and the Xtext form is that the elements of the rules can be named. From these named parts, fields will be generated in the corresponding (generated) classes. For example, the *Variable* class will have a *name* and a *type* field, as it can be seen on Fig. 2 which shows the corresponding part of the generated EMF metamodel. The *name* property is a special property in Xtext, it is used to identify objects.

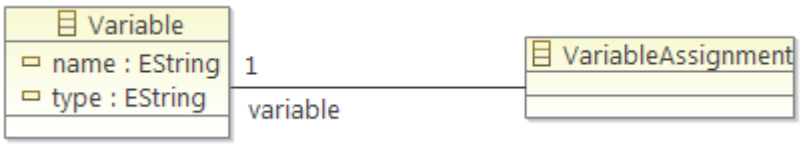


Fig. 2: Example EMF metamodel generated from an Xtext grammar

Further difference is that object references can be expressed with the grammar. In the *VariableAssignment* rule, the type of the *variable* field will be *Variable*, which is identified by the name value of the *Variable*. These references will be parsed by Xtext automatically.

The following example input can be successfully parsed with the given example grammar (assuming that the missing referenced parts are implemented too):

```
v1 : INT;
v2 : BOOL;
v1 := 123;
```

This is just a short introduction to Xtext. For a detailed introduction, see the Xtext documentation [6].

3.2 Representing the ST grammar in Xtext

To use Xtext, the grammar of ST language has to be created in Xtext format. As it was mentioned before, the standard ST language has many "dialects". Slightly different ST language is used for Siemens, Schneider and Beckhoff PLCs. Because it seems to be the most wide-spread and most complex language, we used the Siemens dialect of the ST language (Siemens SCL) as the base of the grammar.

To build this grammar, there are two possible approaches:

– Create a full grammar first. The IEC 61131-3 standard [7] contains a grammar for the full ST language. However, even if this grammar is given, creating an Xtext grammar is non-trivial because Xtext has some additional needs (naming fields, object references) and limitations (the grammar cannot be left-recursive, the terminal symbols have to be "deterministic", the object references have to be unambiguous, etc.). Therefore creating a full grammar would be a huge effort. Also, Siemens SCL is slightly different from the standard ST format, these differences have to be handled too.

– Build the grammar incrementally. In this case, a new grammar can be built based on the real needs. The advantage of this approach that it will produce a proof-of-concept tool and results earlier, then the grammar can be extended by the real needs. The disadvantage is that the incremental grammar building can impose need for code restructuration and this approach makes the planning more difficult.

To be able to produce early results, the second approach was chosen and the grammar was developed incrementally.

3.3 Current state of the ST grammar

Currently a simplified ST grammar is implemented. It covers a big part of the ST grammar declared in the IEC 61131-3 standard. It supports simple variables (with Boolean, integer or real types), arrays and structures (but structures can only contains simple variables); input, output and input-output variables; variable assignments, logic and arithmetic expressions, conditional statements, FOR and WHILE loops, function block and function definition, function calls. Some advanced parts are also covered, among others, the timed expressions, time value types, pointers and nested structures. However, some other parts are missing and their implementation is foreseen to be done in the near future.

4 Intermediate model

This chapter introduces the intermediate model that is used for representing the formal model on an abstract level.

4.1 Advantages of using an intermediate model

The first important question is whether it is appropriate to use a new model and then introduce more transformation steps. This section overviews the reasons.

– By introducing a new abstraction level, the intermediate model **hides some difficulties** from the developer of the transformation tool. The intermediate model decouples the ST specialities and the NuSMV specialities. For example, NuSMV supports using modules, but there is no built-in support for synchronizing transitions in two different modules[3] (which means two transitions can fire together only).

– The transformation is **easily extendable**. Adding a new input language (for example the Schneider syntax of the ST code) or extending the input language (for example adding a new type of loop) does not modify the generation of the formal models, if there is no need to extend the intermediate model. Similarly, to add a new output (for example UPPAAL model), there is no need to modify the input parsing and the automata representation part. In this way, the intermediate model provides independence between the inputs and the outputs.

– This approach makes possible to perform other operations during the transformation. For example, reduction and abstraction techniques can be applied on the intermediate model. As it is an in-memory object model, these techniques can be easily applied (e.g., there is no need to parse the model). Also, if the reductions are performed on the intermediate model, all the outputs can benefit from this operation.

4.2 Development of the intermediate model

The intermediate model should comply with three different requirements:

[3] It can be expressed with TRANS blocks, but it cannot be expressed with ASSIGN blocks. The usage of these two constructions cannot be mixed, and ASSIGN blocks provide a better model which is easier to understand and validate.

- The intermediate model should be close to the theory of automata. This guarantees that the models represented by the intermediate model can be transformed easily to any automaton-based language. As many model checking tools use an automaton-like input language, this will allow us to use different model checkers.
- The intermediate model should be easy-to-use, it should hide the difficulties of representation of the common tasks. For example, it should hide that the synchronization between two automata cannot be expressed easily in NuSMV.
- The intermediate model should be able to represent all the features of the PLC control systems, that we want to represent.

As a compromise, an intermediate model close to the automata network model used in UPPAAL [11] was developed, but without its logic clock representation. Its main components are the following:

- Automata system: represents the whole system, which may contain several (1 or more) automata.
- Automaton (automaton template): represents an automaton template (the same as module in NuSMV or template in UPPAAL). It can have multiple instances and it contains states and variables.
- Automaton instance: instance of an automaton template.
- State: a state (location) in an automaton. (It means just the location of the represented automaton, without the variable valuations.)
- Transition: state transition between two states (locations) in an automaton. Interactions and variable assignments can be attached to it.
- Interaction: synchronization constraint between two transitions in two automaton instance. (Similar to the synchronization in UPPAAL or connector in BIP.)
- Variable assignment: assigns a given value to an instance of a variable.
- Automata Expression: an expression used as value or condition. Can be a constant, a variable instance or an arithmetic or logic operation.

Important to notice, that this automata model is deterministic, as the execution of the subset of PLC programs aim to be modelled is deterministic too. Therefore it is not enabled to have two different interactions enabled at the same time.

The intermediate model was created using EMF (Eclipse Modelling Framework). The diagram of the metamodel can be seen on Fig. 3.

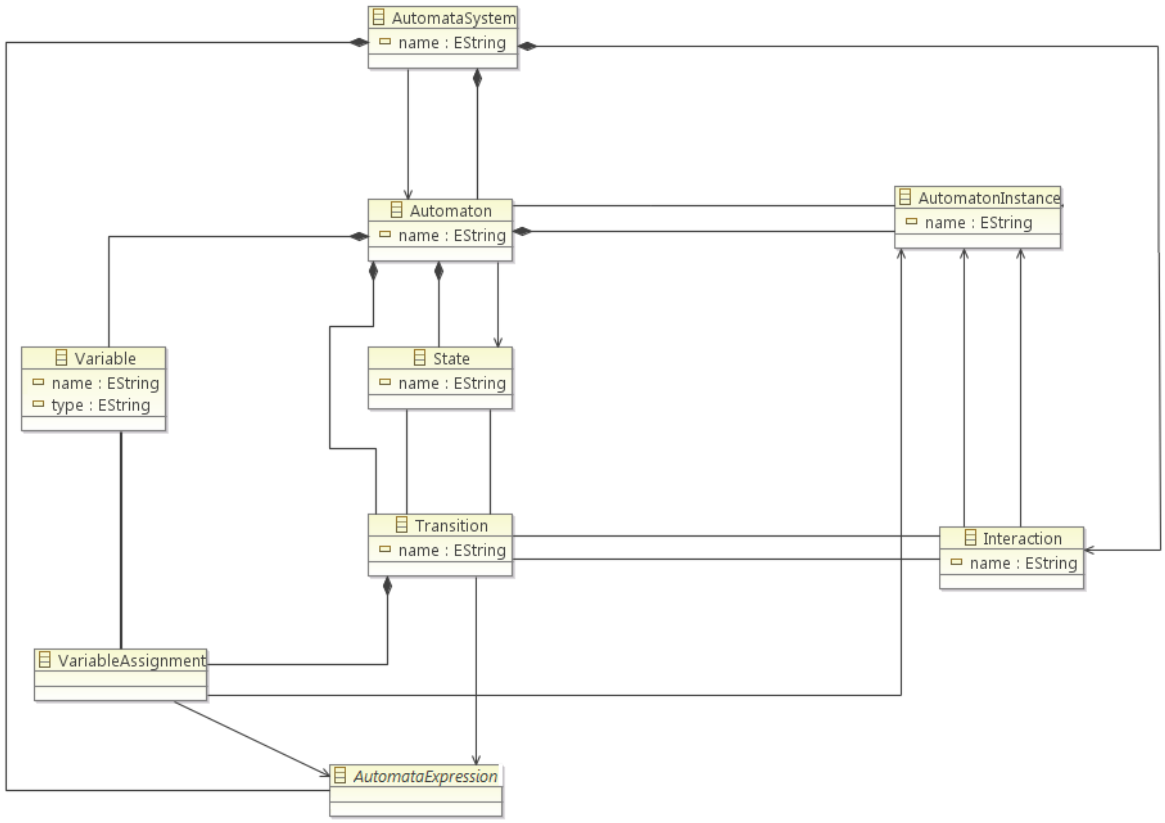


Fig. 3: Structure of intermediate model (extract)

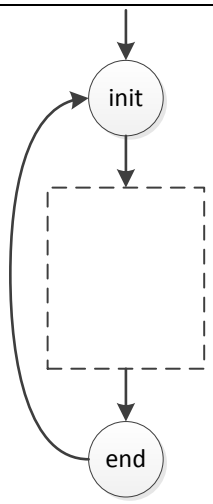
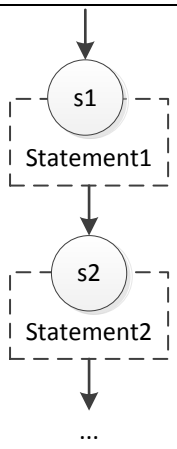
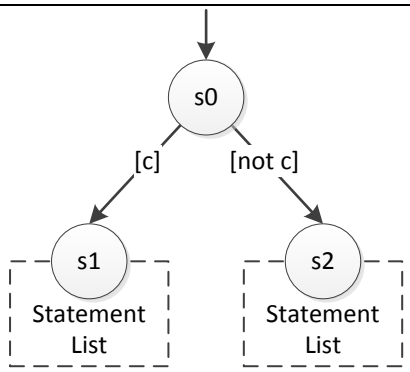
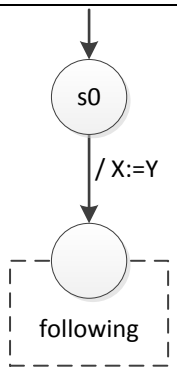
4.3 Transformation between parsed ST code and intermediate model

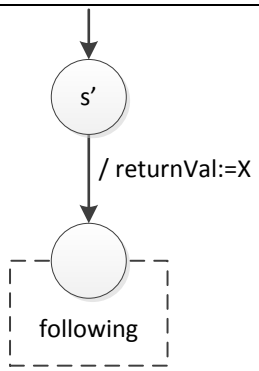
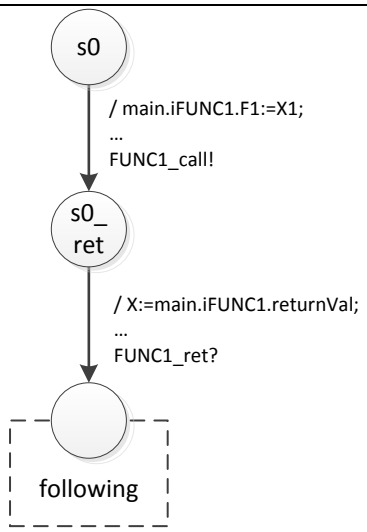
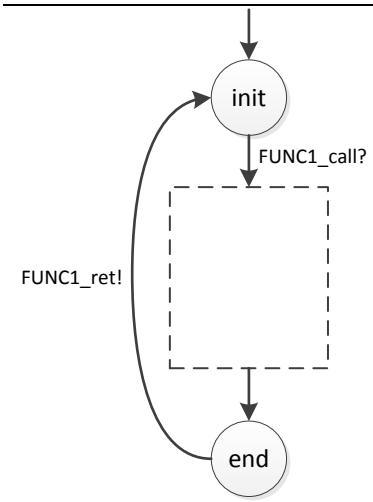
The basic idea of this transformation is that we assign “labels” to every location where the program counter (instruction pointer) can point. In the automaton for every location there is a corresponding state. The transitions correspond to the possible modifications of the program counter. It can be conditional (for example in the case of the IF statement). The variable assignments are assigned also to the transitions.

The PLC programs have a cyclic behaviour. In every execution cycle (called PLC cycle or scan cycle), they (1) read the input values from the real inputs and store them, (2) execute the program and calculates the output values, (3) write the computed output values to the real outputs. This cyclic behaviour should be represented in the automata too. Therefore every automaton will be a loop, starting from the state initial, which is followed by the defined computation path. After the computation, the state end will be active, followed by the state initial again.

The transformation between the parsed ST code and the intermediate model is implemented in pure Java.

The main transformation rules are given in an intuitive format. (The “labels” (used as states in the automata) are shown in parentheses with italic font, like this: *(s0)*. The transformation rules are recursive, the rule names between < > signs means that this rule should be applied recursively.) Interactions are marked using the UPPAAL notation, i.e. with ‘?’ and ‘!’ signs. In this intermediate model, the two parties of the interaction are equivalent, therefore the ‘?’ and ‘!’ signs can be exchanged without the modification of the meaning.

Rule name	**ST structure**	**Automata model**
Program	*(init)* <StatementList> *(end)*	
StatementList	*(s1)* <Statement*1*>**;** *(s2)* <Statement*2*>**;** … *(sn)* <Statement*n*>**;**	
IfStmt	*(s0)* **IF** <c : condition> **THEN** *(s1)* <StatementList> **ELSE** *(s2)* <StatementList> **END_IF**	
AssignmentStmt	*(s0)* X **:=** <Y : constant or variable>;	

FuncDeclaration	FUNCTION name : type … END_FUNCTION	–
FuncReturnVal	<in the declaration of function *func*> … *(s')* func := X; …	
FunctionStmt	*(s0)* X **:=** FUNC(F1 := X1, F2 := X2, …);	
		

It has to be noticed, that the current representation only supports finite function call chains, i.e. recursion is not allowed. However, it is not a real limitation, as PLC programs should not use recursion according to the standard [7].

A basic example can be seen in Appendix A.

5 Creating formal models

This chapter introduces the transformation between the intermediate model and the outputs. In the frame of this work, the intermediate model was converted to two different output formats: the NuSMV model format and a format needed for visualization.

As both output formats are textual, the Xtend technology [8] was used for this transformation. Basically, Xtend is a Java-based, higher level programming language (a "Java dialect"). Its main advantage is the support of templates which enables the programmer to create maintainable output generators easily. To illustrate this feature, a small example is shown on Table 1. The example Java and Xtend code is equivalent, i.e. the value of the variable *str* will be the same.

Table 1: Example of differences between Java and Xtend syntax

Java	*str = "MODULE " + name + "\nVAR " + vname + " : " + vtype + ";\nEND";*	
Xtend	*str =*	*''' MODULE «name»* *VAR «vname» : «vtype»;* *END''';*
Example	*str :*	*MODULE modulename* *VAR v1 : integer;* *END''';*

5.1 NuSMV model

Creating NuSMV model from the intermediate state is relatively easy as for most of the elements there is a straightforward mapping.

For every automaton template, a new NuSMV module will be created. Each NuSMV module has a location variable which stores the current state (location) of the corresponding automaton. The domain of this variable is the set of possible states of the automaton. The NuSMV modules have also variables corresponding to the variables declared in the source automaton in the intermediate model.

The following table summarizes the transformation from the intermediate model to the NuSMV model.

Intermediate model element	NuSMV model element
AutomataSystem	The whole NuSMV model, including a main module. The main module contains a variable interaction, which is described in Section 5.1.1. *MODULE main* *VAR* *interaction : {NONE, i1, i2, …};*
Automaton (template)	Module (MODULE) + *location* variable that represents the current state of the automaton. *MODULE automaton_name(main, interaction)* *VAR* *location : {s0, s1, s2, …};*

Intermediate model element	**NuSMV model element**

AutomatonInstance	Instance of the corresponding module in module main
	`MODULE main`
	`VAR`
	`  inst_automaton : automaton(self, interaction);`

Variable	Variable (VAR) in the corresponding module. The variable will have a next-state statement in this module, and if it is not modified by an assignment explicitly, it will keep its value.
	`MODULE automaton_name`
	`VAR`
	`  var_name : var_type;`
	`  …`
	`ASSIGN`
	`  next(var_name) := case`
	`  …`
	`  TRUE: var_name;`
	`  esac;`

State	A value of the corresponding *location* variable

Transition	ASSIGN rule for the corresponding *location* variable.
	
	If transition *t* goes from *s1* to *s2* with a guard *g*, the location variable of the corresponding automaton will be extended as follows:
	`next(loc) := case`
	`  …`
	`  loc = s1 & g : s2;`
	`  …`
	`esac;`
	If an interaction is also connected to the transition, it is also added as a condition, as it is described in Section 5.1.1.

Variable assignment	ASSIGN rule for the corresponding variable
	
	If transition *t* goes from *s1* to s2 with a guard *g*, and assigns *Expr* to the variable *v*, the next-state definition of the variable v will be extended as follows:
	`next(v) := case`
	`  …`
	`  loc = s1 & g : Expr;`
	`  …`
	`esac;`
	Note that currently it is not allowed to have two transitions from the same location with the same guards.

Interaction	Complex representation, see Section 5.1.1

5.1.1 Representation of the interactions

As NuSMV does not support synchronization, the interactions have to be represented in other way.[4] The main transformation rules are the following:

- The main module contains an *interaction* variable. Its possible values are the possible interactions and the "NONE" value.
- Every module instance is instantiated in the main module. Every module has two parameters: the interaction variable and the main variable. The latter provides a reference for the main module to every instance. In this way, every module can access every variable.
- The model is deterministic, therefore if an interaction is enabled, it has to fire. This necessity is expressed by invariants (in INVAR block) in the main module. The invariant contains the source locations of the connected transitions and their guards.
- If a transition has an attached interaction, this is added as an additional condition in the corresponding part of the ASSIGN block of the location variable.

  The following example shows a model with one interaction defined.

```
MODULE M1(interaction, main)
  VAR
        location : {initial, s0, s1, end};
  ASSIGN
        init(location):=initial;
        next(location):=case
              location = initial : s0;
              location = s0 & interaction = i1 : s1;
              location = s1 : end;
              location = end : initial;
              TRUE : location;
        esac;
END;
MODULE M2(interaction, main)
  VAR
        location : {initial, s0, end};
  ASSIGN
        init(location):=initial;
        next(location):=case
              location = initial : s0;
              location = s0 & interaction = i1 : end;
              location = end : initial;
              TRUE : location;
        esac;
END;
MODULE main
  VAR
        interaction : {NONE, i1};
        instM1 : M1(interaction, self);
        instM2 : M2(interaction, self);
  INVAR
        (instM1.location = s0 & instM2.location = s0
```

[4] The idea is influenced by the BIP translation of Wang Qiang.

```
                    <-> interaction = i1);
END;
```

5.2 Visualization

It was not a primary goal, but for demonstration and debugging purposes it was useful to create a graphical representation for the intermediate automata model. For this subtask, we used Graphviz [9], as it provides visualization for graph-like models. The input of GraphViz is a simple text file which describes the nodes and the edges of the graph. Also, there is a possibility to create clusters in the graph.

The mapping between the intermediate model and the Graphviz model is straightforward. Every automaton instance became a cluster in Graphviz. The states are nodes and the transitions are edges between the corresponding nodes. The variable assignments and the interactions are expressed by labels attached to edges.

A simple visual output example (the graphical output and the corresponding source) can be seen in Appendix A.3.

6 Further challenges

This section overviews some additional challenges about the transformation.

6.1 Data type representation

Every data type defined in ST has to be represented in NuSMV too. In this work, only a reduced set of base types were used: Boolean types, integer types and real types. (Time types, like TIME, TIME_OF_DAY, etc. and the type CHAR are also implemented, but they are not discussed here.)

Boolean types are easy to represent in NuSMV, as NuSMV have Boolean type too.

– For the integers, NuSMV have two possibilities: intervals (like 0..255) or words (which are fixed length bit arrays). First, intervals seemed to be better, because it is easier to handle later. However, when we tried to represent larger types, like INT which is a 16-bit signed integer type, a dramatic performance dropdown was experienced. Inspecting the pre-processed NuSMV model (so-called flatted model that can be written to a separate file using NuSMV) it was observed that the interval types are converted to enumerations internally, which is not practical for large types. Therefore we represented the integer values as words. The consequences are discussed in the next chapter.

– NuSMV does not support floating-point values, but in the UNICOS baseline objects, there are real values. Simply rounding (or truncation) them to integers could produce an inacceptable inaccuracy. For example, the expression X<0.05 could mean “a value is less than 5%”, while its truncated equivalent X<0 would mean “a value is negative”. If X represents a percentage, the first is a valid expression, while the second is impossible. Therefore we used another “abstraction”: all the floating-point values are represented as fixed-point values with preconfigured precision. For example, if the precision is configured to 1000, the floating-point value 3.14159265 will be represented as integer 3141 in NuSMV. This provides better accuracy than the simple rounding, but there are still two problems:

  • The operations in fixed-point arithmetic are different from the simple integer arithmetic. The addition is the same, but the multiplication and division is slightly different because the precision constant. If the original real values are X and Y, their NuSMV representation is $X' = X \cdot PREC$ and $Y' = Y \cdot PREC$. Their product $X \cdot Y$ should be $X \cdot Y \cdot PREC$ in NuSMV, but $X' \cdot Y' = X \cdot Y \cdot PREC^2$. Therefore at each

multiplication, the result has to be divided by the precision constant. Similar applies to the division too.

- The floating-point representation does not have as strict value limits as the integer types. (For example, the value range of the 32-bit floating-point type used in SCL is approximately $-3.4 \cdot 10^{38} .. 3.4 \cdot 10^{38}$, while the 32-bit integer value should be between –2147483648 and 2147483647.) As the fixed-point values are stored as integers, these limits apply to them also. Therefore the size of variables storing the fixed-point values have to be chosen carefully, and the user has to be informed about the restrictions imposed to the possible values, who can decide if it is an acceptable restriction or not.

6.2 Value type handling

NuSMV uses a strong typing system meaning that every variable has a type and every constant is also typed. There is no implicit type conversion, that means the X:=Y statement is forbidden, if X and Y have different types. For variables, the same applies to the ST language, but there is no such restriction in ST for the constants. Therefore it has to be handled during the transformation.

For example, the variable assignment in ST "X:=1" should be transformed to NuSMV in the following way:

- if X is BOOL in ST: X:=TRUE;
- if X is INT in ST: X:=0sd16_1; (it means signed decimal 1 represented on 16 bits)
- if X is UDINT in ST: X:=0ud32_1; (it means unsigned decimal 1 represented on 32 bits)
- if X is REAL in ST: X:=0sd32_1000; (it means signed decimal 1000 represented on 32 bits, if the precision constant is 1000, see Section 6.1)
- …

Therefore it is not enough to simply transform the ST expressions into NuSMV, first the type of every expression element has to be identified.

6.3 Uninitialized and input variables

According to the standard [7], the value of the **uninitialized** variables is implementation-specific; therefore we cannot have assumptions about that. The same applies to the **input** variables: we does not have assumptions about the input values, therefore we have to handle them as random values. For this reason, every uninitialized and input variable are initialized to a random value at the beginning of every PLC cycle. This is easy for the Boolean types:

```
next(B) := case
  loc = initial : {TRUE, FALSE};
  …
esac;
```

But the same approach does not work for the integer types, as it is syntactically forbidden to create a non-deterministic value assignment. However, NuSMV assigns random value to each variable, if no assignment was made for the next value. But in the assignment block, the case should cover all possibilities (practically it has to have a last statement which is always true). Therefore for every integer values without explicit initial value, we added a random variable in the main module, whose next state is not restricted; therefore it always contains a random value. This value can be used for the explicitly not initialized variables. It has to be noticed, that it can have negative impact on the performance of the model checking, as it largely increases the reachable state space.

In the UNICOS framework, a special kind of variable exists, called **parameter**. The parameters are used to specialize the general objects for their specific role. For example, the same object is used to control a valve, a heater or a motor, and the parameters make possible to adapt the object to the particularities. In the ST code, parameter is an input variable, therefore in the model its value is unknown (can be chosen non-deterministically), but it is known, that during the execution its value is not changed. These variables can be handled easily in NuSMV: they do not have initial assignment, but after they will keep their values.

7 Additional work

To have a usable editor produced by Xtext, some additional work was needed, as the default scoping rules had to be modified. By default, Xtext uses fully qualified names internally to identify the named elements (in this context it means variables, functions, structures, etc.). This allows for example to have variables in different functions with the same name. If it is not modified, every variable is resolved in the scope of its container. But consider the following function call: *FOO(X1 := F1, X2 := F2);*. In this case, the scope of variables *F1* and *F2* are the same, as the scope of the function call itself. But the *X1* and *X2* variables are typically non-existing in this scope, as they are defined in the definition if the function *FOO*. Therefore we modified the default scoping method to provide the appropriate scope for the function call parameters. The same applies the variables defined inside structures.

Also, some constraints existing in SCL cannot be expressed by the grammar. The editor generated by Xtext can be extended by additional validators. Some new validators were created, for example for validating that no return value is given in a void function, variable names are unique in a function or function block, no value is assigned to a constant value, etc. A screenshot of the editor can be seen on Fig. 4**.**

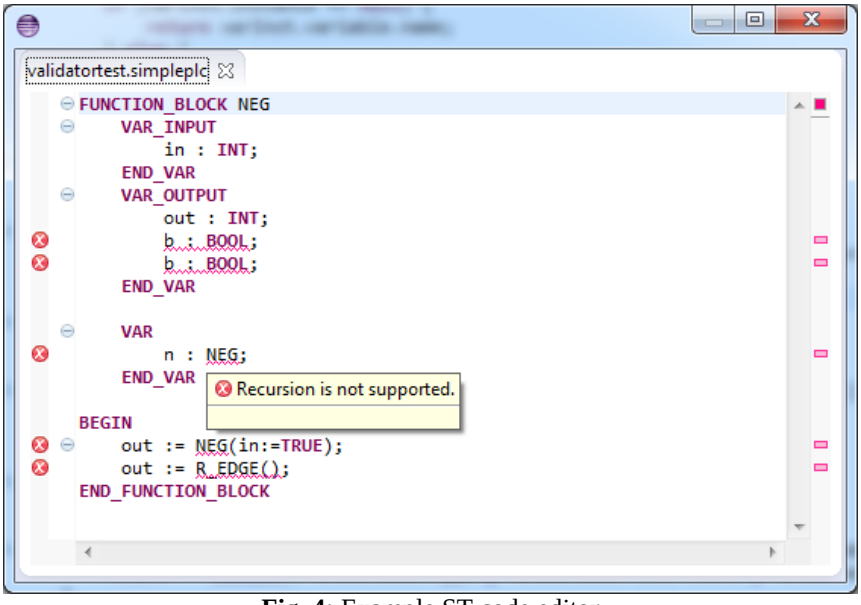


Fig. 4: Example ST code editor

8 Future work

The work presented in this report will be continued. The first goal is to extend the grammar to cover all features necessary to parse and translate the UNICOS baseline objects. Later, the translation of more complex PLC programs is planned (e.g. analysis of the TSPP – Time Stamp Push Protocol).

As the last stage, it will be possible to formalize the requirements and verify them on the formalized models. If necessary, translation to other modelling languages will be developed, e.g. to the input language of UPPAAL [12], BIP [13] or PetriDotNet [10]. Similarly, new input languages, like SFC or IL can be developed.

9 References

[1] CERN website, About CERN section.
http://home.web.cern.ch/about

[2] UNICOS website.
https://j2eeps.cern.ch/wikis/display/EN/UNICOS

[3] UNICOS website, applications subpage.
https://j2eeps.cern.ch/wikis/display/EN/UNICOS+Applications

[4] NuSMV website.
http://nusmv.fbk.eu/

[5] Xtext website.
http://www.eclipse.org/Xtext/

[6] Xtext documentation.
http://www.eclipse.org/Xtext/documentation/2.4.0/Documentation.pdf

[7] IEC 61131-3:2013 standard. Programmable controllers - Part 3: Programming languages

[8] Xtend website.
http://www.eclipse.org/xtend/

[9] Graphviz website.
http://www.graphviz.org/

[10] PetriDotNet website
http://petridotnet.inf.mit.bme.hu/

[11] G. Behrmann, A. David, and K. G. Larsen, “A tutorial on UPPAAL,” in International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures, ser. LNCS, M. Bernardo and F. Corradini, Eds., vol. 3185. Springer Verlag, 2004, pp. 200–237.

[12] UPPAAL website
http://www.uppaal.org/

[13] BIP website
http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html

Appendix A: Simple transformation example

In this section, a simple example is given: the source ST code, the automatically generated NuSMV code and the visualization of the intermediate model.

A.1 ST code

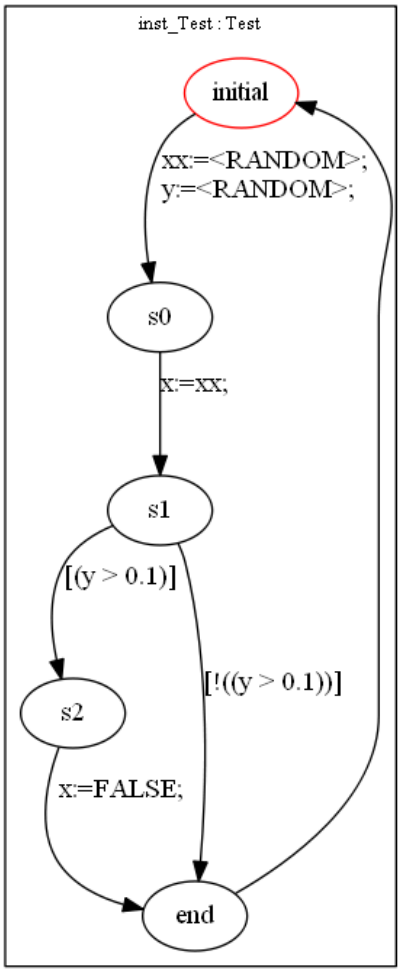
```
FUNCTION_BLOCK Test
VAR_INPUT
	x : BOOL;
	y : REAL;
END_VAR

VAR_OUTPUT
	out : BOOL;
```

```
END_VAR

BEGIN
	out := x;
	IF y > 0.1 THEN
		out := FALSE;
	END_IF;
END_FUNCTION_BLOCK
```

A.2 Transformed NuSMV code

```
MODULE TEST(interaction, main, instNo)
	VAR
		loc : {initial, end, s0, s1, s2};
		X# : boolean;
		Y# : signed word[32];
		OUT : boolean;

	ASSIGN
		init(loc) := initial;
		next(loc) := case
			loc = end : initial;
			loc = initial : s0;
			loc = s0 : s1;
			loc = s1 & ((Y# > 0sd32_10)) : s2;
			loc = s1 & (!((Y# > 0sd32_10))) : end;
			loc = s2 : end;
			TRUE: loc;
		esac;

		next(X#) := case
			loc = initial : {TRUE, FALSE};
			TRUE : X#;
		esac;
		next(Y#) := case
			loc = initial : main.random_r2;
			TRUE : Y#;
		esac;
		next(OUT) := case
			loc = s0 : X#;
			loc = s2 : FALSE;
			TRUE : OUT;
		esac;

MODULE main
	VAR
		interaction : {NONE };
		inst_Test : TEST(interaction, self, 0);
		-- Randoms
		random_r2 : signed word[32];

	-- INVAR block omitted. There is no interaction.
```

A.3 Visualized intermediate model



A.4 Source of the visualized intermediate model

```
digraph G {
  fontsize=11;
  subgraph clusterinst_Test {
            node [shape=ellipse, style=filled];
            color = black;
            fontsize=10;
            ranksep = 0.4;
            label = "inst_Test : Test";
            inst_Test_initial [label = "initial", color=red, fillcolor="white"];
            inst_Test_end [label = "end", color=black, fillcolor="white"];
            inst_Test_s0 [label = "s0", color=black, fillcolor="white"];
            inst_Test_s1 [label = "s1", color=black, fillcolor="white"];
            inst_Test_s2 [label = "s2", color=black, fillcolor="white"];

            inst_Test_end -> inst_Test_initial [label = ""];
            inst_Test_initial -> inst_Test_s0
            [label = "xx:=<RANDOM>;\ly:=<RANDOM>;\l"];
            inst_Test_s0 -> inst_Test_s1 [label = "x:=xx;\l"];
            inst_Test_s1 -> inst_Test_s2 [label = "[(y > 0.1)] \l"];
            inst_Test_s1 -> inst_Test_end [label = "[!((y > 0.1))] \l  "];
            inst_Test_s2 -> inst_Test_end [label = "x:=FALSE;\l"];
  }
}
```