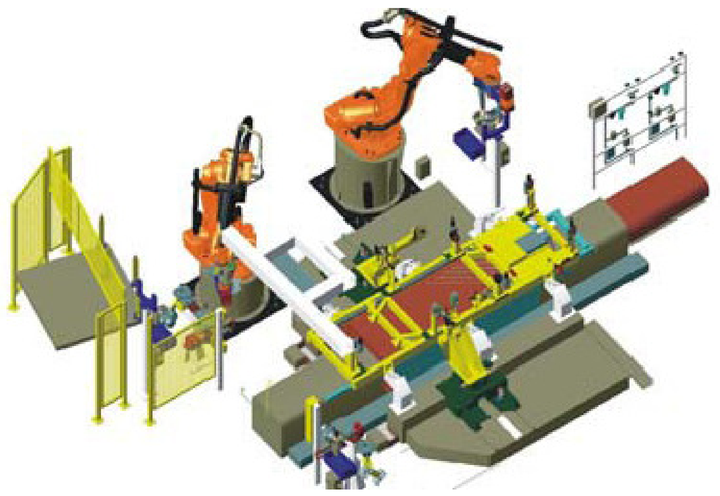
MASTER'S THESIS 2010

IMPLEMENTATION OF A SOFTWARE TOOL FOR DEVELOPMENT, SPECIFICATION AND VERIFICATION OF LOGIC CONTROL PROGRAMS



Massoud Alahdini

Division of Control, Automation and Mechatronics
Department of Signal and Systems
Chalmers University of Technology
Göteborg, Sweden 2010

IMPLEMENTATION OF A SOFTWARE TOOL FOR DEVELOPMENT, SPECIFICATION AND VERIFICATION OF LOGIC CONTROL PROGRAMS

Master's Thesis 2010

Department of Signal and Systems

Division *of* Control, Automation and Mechatronics

Automation Research Group

Chalmers University of Technology

SE-41296 Göteborg, Sweden

Telephone: +46-(0)31 772 1000

www.chalmer.se

Department of Signal and Systems

Göteborg, Sweden 2010

Abstract

This master thesis presents a practical work for implementing a software tool that is used for automating formal verification tasks for PLC-programs.

PLC-programs being used in manufacturing industries are on the demand of new requirements such as correct usage and behavior of the components. However, it is required to facilitate quick and correct modification of the programs as much as possible. One possible solution is to reuse the programs code.

The code that is structured into reusable components may speed up the development process. Furthermore it may cause the control program to have fewer bugs. For this aim, the previous research suggested to apply Reusable Automation Components (RACs). The RACs contain the implementation and formal specification.

To enhance reusability, it is necessary to identify and specify the requirements for the RACs. Additionally, the complete RAC including the specification must be translated into input to a tool for formal verification, to determine whether the implementation of the components fulfils the specification or not.

Performing all of these tasks manually is time consuming for developers, and even it is impossible to test all different cases in which the components can be used. Therefore, building a software tool to assist developers was a part of this project. The tool automatically generates inputs to Cadence SMV tool for formal verification, and finally the result of formal verification is presented as a feedback to developer.
To build the above mentioned tool, PLCOpenEditor that is an open source PLC programming environment has been extended to support RAC and formal verification using Cadence SMV.

The tool has successfully been used for non-trivial case study; an industrial example of translating the specification for a scan cycle based component is presented.

Acknowledgements

This work would not have been completed without help and support of many individuals. I would like to thank everyone who has helped me along the way. Particularly: Dr. Knut Åkesson for providing me an opportunity to conduct my master's thesis with new research area and for his guidance and support over the course of my thesis, Oscar Ljungkrantz for his supervision and for helping me with various technical details, helpful comments on the text and his conversations during the development of the ideas in this thesis, special thanks goes to my brother Ali and my family without whose support none of this would have been possible.

Contents

List of abbreviations

FB: Function Block

FBD: Function Block Diagram

LD: Ladder Diagram.

PLC: Programmable Logic Controller.

RAC: Reusable Automation Components.

SMV: Symbol Model Verification.

XML: Extensible Markup Language

CTL: Computation Tree Logic

ST: Structured Text

LTL: Linear Temporal Logic

ST-LTL: Structured Text-LTL

CHAPTER 1 - Theory

This chapter gives a brief introduction, an overview and background theory to my master thesis project. Furthermore, it describes a case study through an industrial example, which has been applied to the implemented tool in this thesis work.

1.1 Introduction

This master thesis presents a practical work for implementing a software tool which is useful in computer software development, in particular for automating the formal verification task for PLC-programs. The software tool will be used for testing and formal verification, which are important issues for developing and building Reusable Automation Components (RACs).

PLC (Programmable Logic controller) is a typical embedded system to instrument and control systems. The PLC-programs that are generally referred to as logic control programs are used to control and coordinate machines and robots in automated manufacturing system.

Traditional PLC-programs have been used successfully in industry so far, but they would be very difficult to be modified, and to be extended to meet the new requirements. Moreover, PLC-programs tend to be time consuming to change when the manufacturing systems have to be changed (see [1]). Since, they often are tested to work first on the real equipments. Moreover, the same equipments cannot be used for production if error should be solved. Furthermore, it will be costly, since sometimes the traditional PLC-programs must first be tested to check if they work on real equipment or not. Hence, fixing the errors before production line would be expensive.

Developing PLC program with reusable components is a possible solution to enable fast and correct modification of the control logic. The code can be encapsulated and reused as Function Blocks (FBs). However, just encapsulating the control code into FBs is not enough [1]. The FBs should also be specified and be verified in order to work properly. For instance, it is required to determine how the efficient FBs (The FBs that are reused) shall be used and what the components guarantee.

The RACs mainly consist of two parts: implementation and specification. The implementation is designed in Ladder Diagram (LD) which is the most common language in Programmable Logic Controller (PLC) programs. The specification is written in Structured Text- Linear Temporal Logic (STL-LTL) which is a text based specification language for PLC program components, see [1]. Both the implementation and the specification are translated into inputs to a tool (SMV, Symbolic Model Verification, see [1, 2]) for performing formal verification, in order to determine whether the implementation fulfils the specification or not.

The SMV tool allows the developer to verify the behavior of the PLC program over all possible operating conditions, and if the verification is not successful a *counter example* will be shown to the user.

In order to enhance the development process, it is required to utilize an appropriate software tool that can automate the mentioned tasks (translation and formal verification).

The tool is applied by the developers who require implementing, specifying and verifying the re-usable software components of PLC programming. So a RAC can be implemented, specified and verified in the developed software tool.

A former java based prototype at Chalmers University of Technology (CTH) has been implemented based on PLCOpenEditor. PLCOpenEditor is a free and open source IEC 61131-3 automation IDE. In this project, the PLCOpenEditor graphical user interface was extended for building the software tool. The editor can be used for writing the specification data and saving the specification data in XML file.

The java based prototype was extended for translating of the specification data into SMV format. This part of software tool creates the automatic generation of inputs, which they will be input to Cadence SMV tool. Some parts of the developed code launch the SMV tool properties for the formal verification; in consequence, the results of the verification process are stored to different output files. The output files can be observed by the user to see the faults, warnings and errors which may exist.

This report starts by introducing the main issues and a general overview of thesis work, and a background theory is described in the first Chapter, and then by presenting a case study shows an application perspective of such a software tool, below is a short guide to the rest of the report.

In Chapter 2 developing the python code for building specification editor is explained. In Chapter 3, developing the java program for building translator is described. In Chapter 4, integration of programs with SMV tool is described. Chapter 5 includes the conclusion, and the future work that is suggested for the improvement of tool.

1.2. Overview

This chapter will state the problem which is going to be solved, and describes the problem and requirements on developing the program.

1.2.1 Problem Description

A main part of this master thesis work was dedicated to develop and implement a software package. The aim has been to provide a tool for the developers in the automation group of signal and system department who want to implement, specify and verify the RAC components.

To make such software tool, the following main tasks has been carried out:

- Developing a graphical tool for specifying the FBs
- Developing the automatic generating of inputs to Cadence SMV tool for formal verification.
- Integrating the Editor, Translators, and SMV tool along with directing the information to the paths, running and starting the program from command lines.

1.2.2 Information Flow of Program

A general view of the whole process or the information flow of program has been depicted in Fig.1, a developer builds the FB diagrams and writes the corresponding specification text into the editor, the developer also is able to edit the previous information and FBs within this editor, and after completion can store the information to a file with ".xml" extension.

Whenever developer clicks on verify button in the editor, the translation will be started first, the translation of FBs or LDs into the SMV module along with Translation of ST-LTL specification into the SMV form. The results of these two translation processes are stored in a file with ".smv" extension. Other commands in the program launch the SMV and set this ".smv" file, this text file is as an input to the SMV tool. After that, SMV is started to perform the verification process**.** In consequence, the verification result is stored in a text file. A part of the program checks the fulfilments of the verification, mainly it checks if the verification is true or false, if it is false means that the requirements have not been met and it gives a message or show up the *counter example* to the developer.

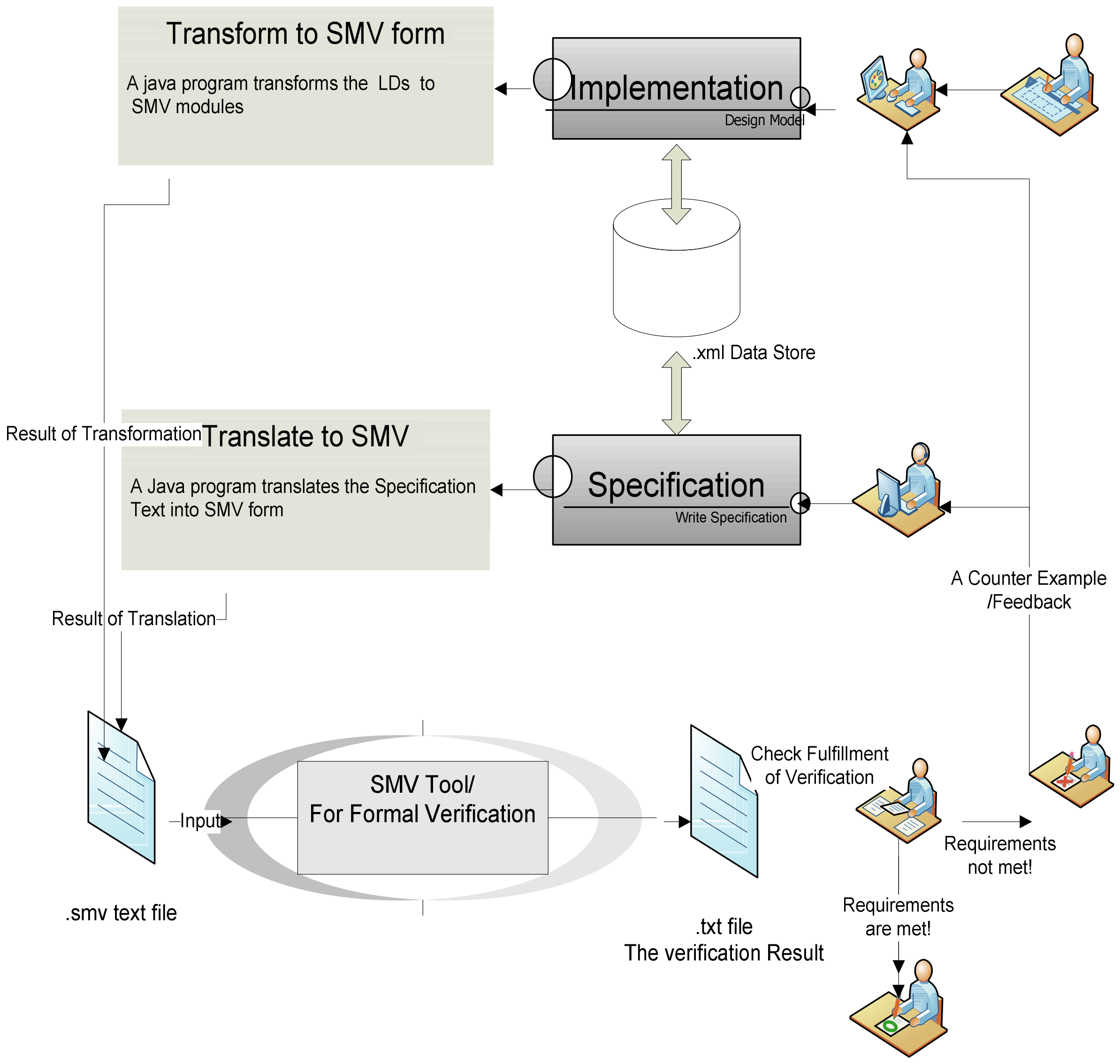

Transform to SMV form
A java program transforms the LDs to SMV modules
Implementation
Design Model
.xml Data Store
Result of Transformation
Translate to SMV
A Java program translates the Specification Text into SMV form
Specification
Write Specification
A Counter Example /Feedback
Result of Translation
Input
SMV Tool/ For Formal Verification
Check Fulfillment of Verification
Requirements not met!
Requirements are met!
.smv text file
.txt file The verification Result

Fig.1 - An overview of the complete processes in this project.

The later chapters describe the development of Translator program which has been used for building the software tool, the overview of the processes with software tool is shown in Fig.1.

1.3. Background Theory

This section attempts to give a background to the thesis work and argue why there is a need for such a tool for specification and verification of PLC components.

In industrial application, there are number of manufacturing systems for mass production which use machines and robots to automate tasks. Moreover, specific hardware and software components are required for the automation of machines and robots.

PLC-programs are kind of programs, which coordinates the machines and the robots in industry. However, the PLCs that have been used so far in industry shown to be successful, in order to meet the new requirements in developing of the PLC-programs.

As in previous years, the challenge of shorter life-cycles of many mass-produced products has been increased. This competition places new requirements on the PLC programs too. For instance, it is required that the PLC programs to be easily modifiable or extendable. The modification or extension is a difficult task, in which it needs to be tested on the real equipments, and it might be difficult to be changed. Moreover, the cost of the production will be increased if any error is detected. Since the error must be resolved in the previous equipment that caused an error, finding a new solution becomes expensive when the equipment cannot be reused. Moreover, changing the PLC-programs is time consuming task [1, 3, 4, and 5].

In some manufacturing companies, it is common to run the control logic against simulated manufacturing systems. Moreover it has been possible to test and troubleshoot the control logic in the earlier stages of development process. On the other hand, the control logic codes also can be encapsulated and be reused as Function Blocks (FBs). Since, reusing components is known as a beneficial design methodology in the development of embedded software products. One reason is due to the fact that it alleviates the cost of designing new solutions. Other reason, the development is faster if the designer can take the benefit of previous works and reduce the repetitive testing or verification task, hence, creating the PLC-programs from reusable components is also a way to expedite the correct modifications of the control logic [3, 4].

In order to create more effective software, a designer must be able to implement a new solution and adapt that to fit the requirements of the design of PLC components. In the following sections a RAC and its related issues are described, RACs are proposed for the components which are used for developing the control programs. It is followed by an overview of formal specifications and formal verification, the formal verification is to verify the reusability of software components.

1.3.1 RAC/RAC Framework

A Reusable Automation Component (RAC) is a component and specification structure that is useful for building PLC-programs, the term “component” that often has been used in this project refers to RAC too[1]. A RAC prototype development tool has been implemented in Chalmers University of Technology within Automation group in the Department of Signals and Systems. In Fig.2, a simple schema of a RAC is displayed.

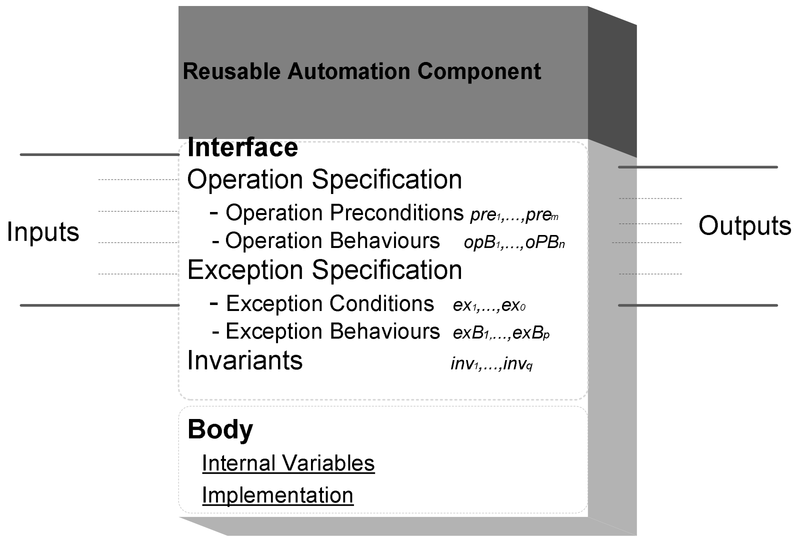

Reusable Automation Component
Interface
Operation Specification
- Operation Preconditions $pre_1,...,pre_m$
- Operation Behaviours $opB_1,...,oPB_n$
Exception Specification
- Exception Conditions $ex_1,...,ex_o$
- Exception Behaviours $exB_1,...,exB_p$
Invariants $inv_1,...,inv_q$
Body
Internal Variables
Implementation
Inputs
Outputs

Fig.2. The main part of the Reusable Automation Component, RAC.[1]

The RAC (shown in Fig.2) consists of an *interface* and a *body*. The *interface* is the part which is visible to the end users. However, the *body* of the RAC is only accessible to the developers of the component [1].

The *interface* consists of three parts: Inputs, Outputs and Specification. The *body* includes the implementation part of the components function along with the declaration of any internal variables.

1.3.2 Function Block

The Function Block (FB) is the basic functional software unit that is the smallest element of a distributed control system, which has its own data structure and a set of algorithms. These abilities make it possible to access and modify the data. The models and the concepts introduced by the PLC standards (such as: IEC 61131-3, see [7] and Appendix A) provide the possibility to define such a distributed application to be hardware independent, the applications can be implemented by modular components, and the components can be reused too [12]. The most PLC-programs are implemented according to the IEC 61131-3 standard [7]. Hence in this project, only the IEC 61131-3 standard is applied.

The RAC can be implemented as a FB; the FB concept is a kind of typical offered solution by automation companies which was previously introduced in the PLC language IEC 61131-3 standard. FBs can be considered as small reusable pieces of software that prepare a software solution in advance to a small problem. The RACs can be developed as IEC 61131-3 function blocks which is extended with specification [2]. However, the FB might not be formally specified before, it should be specified formally, even formal verification requires a formal specification. Hence, the RAC is complemented with a formal specification [1, 3]. Moreover, the end-user will be able to integrate the pre-made software block without further knowledge about the complex contents such as internal components and functions etc. One advantage of using the IEC 61131-3 in contrast to traditional PLC programming is the easy reusability of its software and easy integration into the user program [8].

1.3.3 Scan Cycle

There is a scheduling function that is provided to ensure the correct execution sequence and priority. Moreover, a PLC does not handle many tasks in the same time, but there is a parallel computing that is simulated by an operation cycle. This operation cycle which will be run several times per second is referred as the scan cycle of a PLC [12].The scan cycle is composed of four basic stages:

1) Self-checking for finding the hardware and software faults
2) Input scan is the stage when all inputs are copied into the memory.
3) Logic scan is the stage which is using only the memory copy of the inputs while the program is executed.
4) Output scan is the stages which are using the temporary values in memory while all the outputs are updated. The output value can be changed only in the temporary memory.

Each FB may have a set of Input variables, outputs variables and enclosed algorithms in its structure. This structure will be reviewed by an example that will be described in the later parts of the case study.

1.3.4 Formal Specification

In this part, an overview and the description of the formal specification in the current project is presented.

By convenient, “Formal specification in computer systems is defined as a mathematical description of software or hardware”. The specification and implementation (or designs) are two mixed processes. The specification describes what the system should do and how the system should perform an action.

Informal descriptions of a RAC are not suitable for formal verification, since they might be ambiguous and could be misinterpreted [3]. Nevertheless, formal specification is necessary and suitable for formal verification. Formal specification handles with more effort in the early stages of software development. With this way, the user will be able to specify the requirements of the implementation.

Formal specification along with formal verification, reduce the requirements, errors and faults. In consequence, any inconsistency or any unstable situation can be discovered and resolved in the earliest phases of implementation. It is also expected to reduce any ambiguity or errors in the requirements, which will result in saving the costs (time, expenses) of implementation and validation in the developing of the system.

In this project, the formal specification must be expressed in a temporal logic notation with precisely defined the grammar and syntax and semantics. Moreover, temporal logics are used to describe Boolean logic and its relation over time. Two most common temporal logics are *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL). They differ in how they handle branching in the underlying computation tree. In CTL temporal operators it is possible to quantify over the paths departing from a given state. In LTL operators are intended to describe properties of all possible computation paths. [2, 15]

1.3.5 Specification Text

There is no former support of formal specification in developing and reusing FB's, the ongoing research in department[1,2] has proposed solution to use a text based specification language instead of other existing language for PLC programming.

A text based specification language, is similar to the structured text (ST). Furthermore, it contains some extensions. The aim of adding these extensions is to handle temporal/ time relations, since it is mixed with simplified and clarified temporal logic[1,2], such that the specification apply the temporal operators and functions.

The suggested specification language in [2] for PLC program components is based on the IEC 61131-3 language and LTL (Linear Temporal Logic). For instance, if "a rising edge of a *Boolean* input I will always guarantee that at least one of the two outputs O1 and O2 will eventually be *true*" be an example that should be specified (when the input transits from *false* to *true),* this in LTL can be expressed as:

$$G\left(\sim I \bigwedge \quad (X\, I) \Rightarrow FX\left(O1 \bigvee \quad O2\right)\right)$$

This property can be specified using a text-based variant such as what written bellow.

```
Spec1: = ALWAYS (NOT I_previous & I -> EVETUALLY (O1 OR O2));   or
Spec1: = ALWAYS (I_risingEdge -> EVENTUALLY (O1 or O2));
```

The IEC 61131-3 standard contains Boolean and comparison functions such as AND (), NOT (), OR(), GT() and EQ()[2]. These functions might be applied in both graphical languages (LD and FBD) and text-based languages (Structured Text and Instruction List) of IEC 61131-3. Moreover, the logic functions to be applied in text based languages have corresponding *operators* [2]. For example, AND in IEC 61131-3 is both an operator and a function.

In structured Text the Boolean “and” relation between two variables p and q can be written such as: *p AND q, p & q, or AND (p,q). AND (p,q)* is the function type of the other two forms. However, there is a possibility for writing specifications graphically in Ladder Diagram (LD) or Function Block Diagrams (FBD) as well. In this project, the specifications are written in the text based language.

1.3.6 Formal Verification

The implementation of PLC program has been designed in LD. The developer can write the specification for this implementation. Then after writing the formal specification, it is possible to apply formal verification techniques in order to demonstrate whether a system design is correct or not. A developer is interested in testing if the implementation is correct with respect to the given specification. Otherwise the result might be shown to the user and check what has been wrong within implementation process.

1.3.7 Symbolic Model Verification (SMV Model Checking)

By definition, model checking (on a given model of system) refers to automatically testing whether the model meets a given specification [9]. SMV (Symbolic Model Verifier) is a tool for checking the finite-state systems to meet the requirements (or specifications), the specifications has given in LTL. The model is required to be translated into the input language of a model checking tool. The tool that has been used in this project is “Cadence SMV” tool [2, 11, and 14].

In the process of model checking, it checks if whether a given model satisfies the specification. The most valuable feature of SMV tool is its *counter example* feedback. If a given model does not fulfil the requirements of its specifications, then SMV presents a trace identifying why the specification is false.

1.4. Case Study

This section provides a case study which briefly studies the usage of a software tool that has been implemented in this project.

1.4.1 Introduction

The software tool has been applied for the purpose of the implementation, specification and verification of Logic Control Programs. In this section, the case study represents an example showing how this integrated program can be an appropriate tool for the developers who require implementing, specifying and verifying the re-usable software components of PLC programming. Fig.3 depicts a whole process of implementation, specification and translation of both into a SMV tool for the purpose of verification.

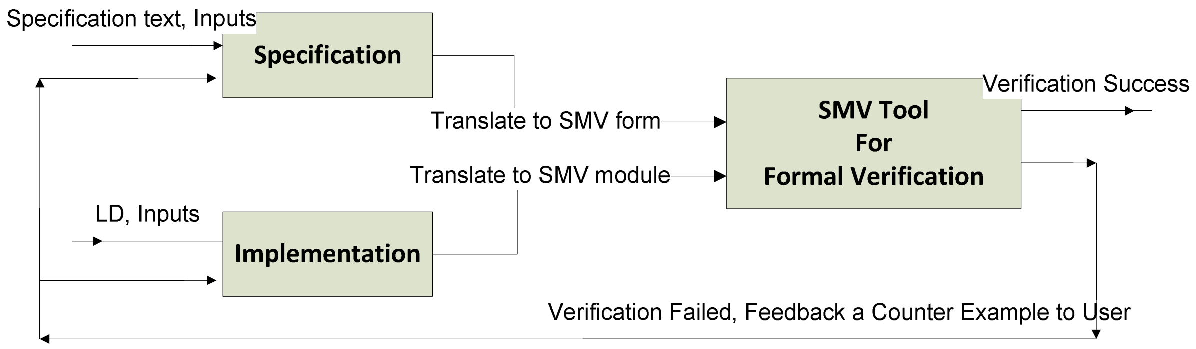

Specification text, Inputs
Specification
Translate to SMV form
Translate to SMV module
SMV Tool For Formal Verification
Verification Success
LD, Inputs
Implementation
Verification Failed, Feedback a Counter Example to User

Fig.3. depicts the whole schema of Implementation, specification, translation and verification stages.

As it is shown in Fig.3, the implementation (i.e. Implemented in a LD) is transformed into a set of SMV modules; moreover the transformation retains the structure of block diagram model which has been designed. The specification text also will be translated into the SMV language form. The transformation and translation will be an input to SMV tool for the formal verification process.

In this case study, the design of PLC Ladder Diagram, the specification text and SMV approach to PLC program verification is illustrated with an example. The design and implementation of this software tool will be described in details in later chapters.

1.4.2 Previous Work at University

A RAC prototype development tool has been developed [3]. The Implementation part of designing a model of PLC software components has been developed previously by Oscar Ljungkrantz in automation group of the department of Signals and Systems in CTH, using Java programming language to translate the control logic components into SMV. For the specification part, it was suggested to be translated into the text format of Cadence SMV. PLC Open has defined an open interface for PLC-programs, with an XML schema. Furthermore, The RACs should be saved to a XML file according to that XML format [3].

1.4.3 Aims and Objectives of utilizing the tool

- To view the implementation of PLC software component, and to transform it into SMV modules.
- To write a specification for the component, and translate it into SMV format
- To verify the result of SMV files, and convey a feedback to the developer

1.4.4 The Prepared Inputs of Example

This example, "*BufferCounter*" is presented to show how a RAC can be implemented, specified and verified in the developed software tool. The "BufferCounter" is a RAC for collecting some information from a buffer, the information that can be used while controlling the buffer. The inputs and their types in this case study are depicted in Fig.4.

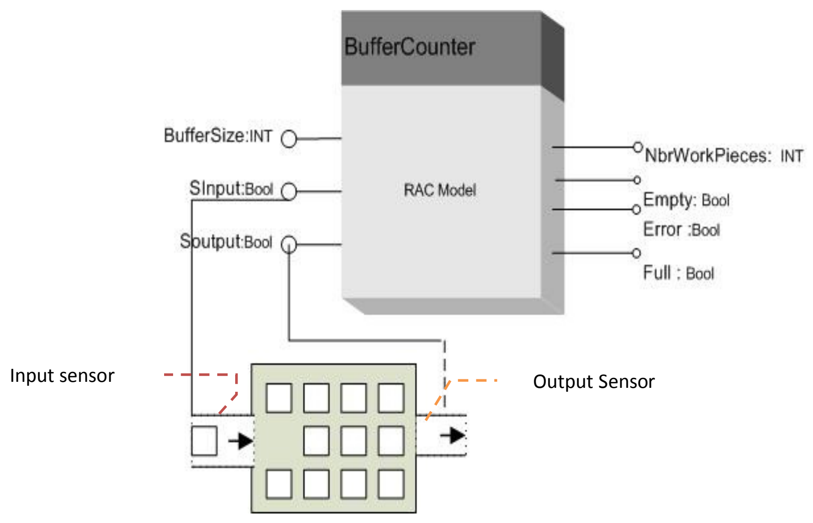


Fig. 4- The variable declaration and code icon for a Buffer Counter

The RAC model has been proposed and provided by my supervisors (Oscar Ljungkrantz, Knut Åkesson) at Chalmers University of Technology. This is an example of a FB that counts the number of work pieces in a buffer (NbrWorkpieces). “Buffer Size” and “NbrWorkpieces” are defined as integer, the other Inputs and Outputs are Boolean.

By conducting an inquiry on input and output sensors of the buffer, it will be possible to count the NbrWorkpieces. Passing the respective sensor into the high or passing it out of the buffer is considered for changing of the NbrWorkpieces. If the NbrWorkpieces is over the capacity of the buffer size, it will give an error. Moreover, such information will be useful for controlling the buffer on next steps.

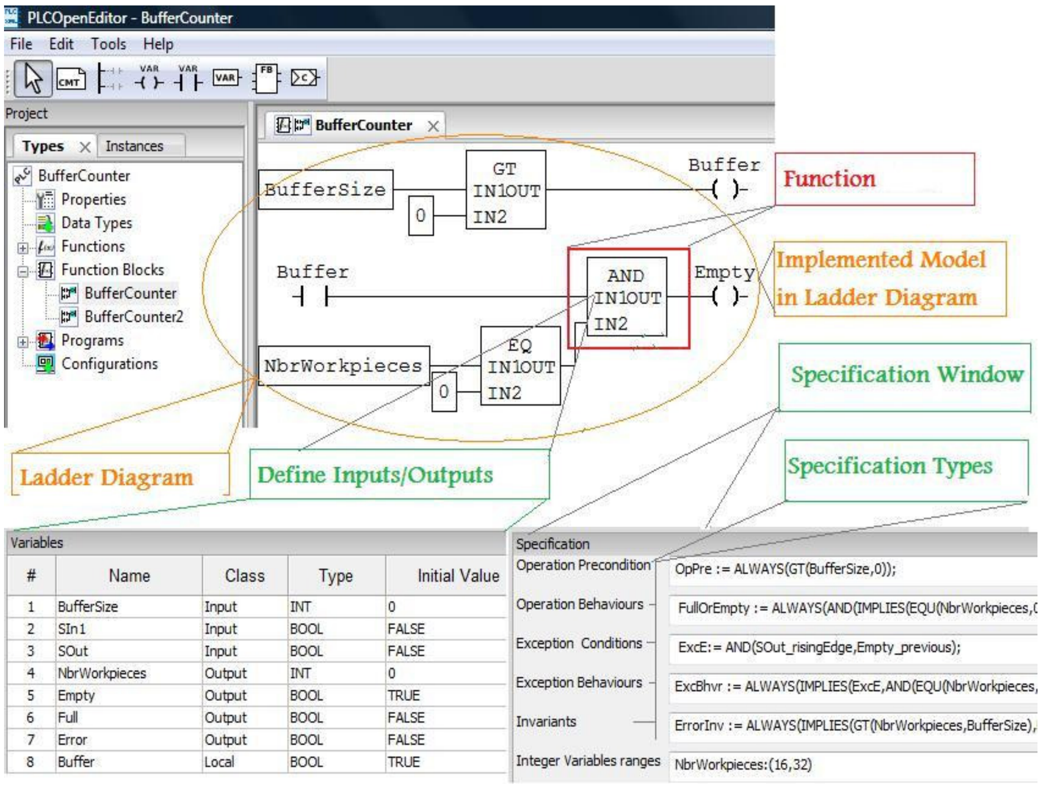
The input file is a RAC example which should be opened in the editor. The user will be able to build the LDs, FBs, and all the corresponding elements along with writing the specification, based on the example stated in this study. The implementation will be transformed into the SMV module. The files related to this case study along with software tool can be viewed in the attached CD to this report, thus it is possible to build the RAC of this case study.

1.4.4.1 Implementation

The example demonstrates the development of a RAC BufferCounter, which has been implemented as a FB in LD. LD is the most common language for developing PLC-programs (for instance in car industries).

In the Fig.5, there are different panels in a PLCOpenEditor. Some of these panels are *implementation* panel, *specification* panel and *variables* panel which have been highlighted in

Fig.5. The developer can implement the FB example within the implementation panel (window) in LD. For each FB might have an implementation window.



#	Name	Class	Type	Initial Value
1	BufferSize	Input	INT	0
2	SIn1	Input	BOOL	FALSE
3	SOut	Input	BOOL	FALSE
4	NbrWorkpieces	Output	INT	0
5	Empty	Output	BOOL	TRUE
6	Full	Output	BOOL	FALSE
7	Error	Output	BOOL	FALSE
8	Buffer	Local	BOOL	TRUE

Operation Precondition	OpPre := ALWAYS(GT(BufferSize,0));
Operation Behaviours	FullOrEmpty := ALWAYS(AND(IMPLIES(EQU(NbrWorkpieces,
Exception Conditions	ExcE:= AND(SOut_risingEdge,Empty_previous);
Exception Behaviours	ExcBhvr := ALWAYS(IMPLIES(ExcE,AND(EQU(NbrWorkpieces,
Invariants	ErrorInv := ALWAYS(IMPLIES(GT(NbrWorkpieces,BufferSize),
Integer Variables ranges	NbrWorkpieces:(16,32)

Fig. 5 - The input/output variables, implementation, and specification window are shown in this figure.

The developer can choose a XML file from the menu "File/Open" on menu bar of the editor. It is also possible to choose the FBs on the left side of the editor (in Project panel). The developer can design and implement the LD and the elements which are the basis for reusing software. The implementation later will be translated into inputs to the SMV tool for performing the formal verification task.

1.4.4.2 Specification

The specification is written in specification windows within the separated text fields. The inputs and outputs variables should be defined on the *variables* panel in advance whenever the user implements the FBs.

As mentioned previously in the report, the specification "*BufferCounter*" example can be specified in the specification language (ST-LTL), which is based on LTL and ST. Moreover, it is assumed that the input sequences strings of specification be expressed in "prefix" form, which is one way of writing expressions, these strings should not be in the form of "postfix" or "infix", because the program does not support the possibility of translating “postfix” and “infix”. This specification text includes all five specification types which are stated in Table.

Specification type		***Specification texts(specified in ST-LTL)***
Operation Specification	***Operation preconditions***	*OpPre := ALWAYS(GT(BufferSize,0));*
	Operation behaviours	*CountUp := ALWAYS(IMPLIES(AND(SIn1_risingEdge,NOT(SOut_risingEdge)),(EQ(NbrWorkpieces,ADD(NbrWorkpieces_previous,1)))));*
		CountDown := ALWAYS(IMPLIES(AND(SOut_risingEdge,NOT(SIn1_risingEdge)),(EQ(NbrWorkpieces,SUB(NbrWorkpieces_previous,1)))));
		UpNDown := ALWAYS(IMPLIES(AND(SIn1_risingEdge,SOut_risingEdge),(EQ(NbrWorkpieces,NbrWorkpieces_previous))));
		FullOrEmpty := ALWAYS(AND(IMPLIES(EQU(NbrWorkpieces,0),Empty),IMPLIES(GT(NbrWorkpieces,BufferSize),Full)));
Exception Specification	***Exception conditions***	*ExcE:= AND(SOut_risingEdge,Empty_previous);*
	Exception behaviours	ExcBhvr := ALWAYS(IMPLIES(ExcE,AND(EQ(NbrWorkpieces,0),Empty)));
Invariants		*ErrorInv := ALWAYS(IMPLIES(GT(NbrWorkpieces,BufferSize),Error));*

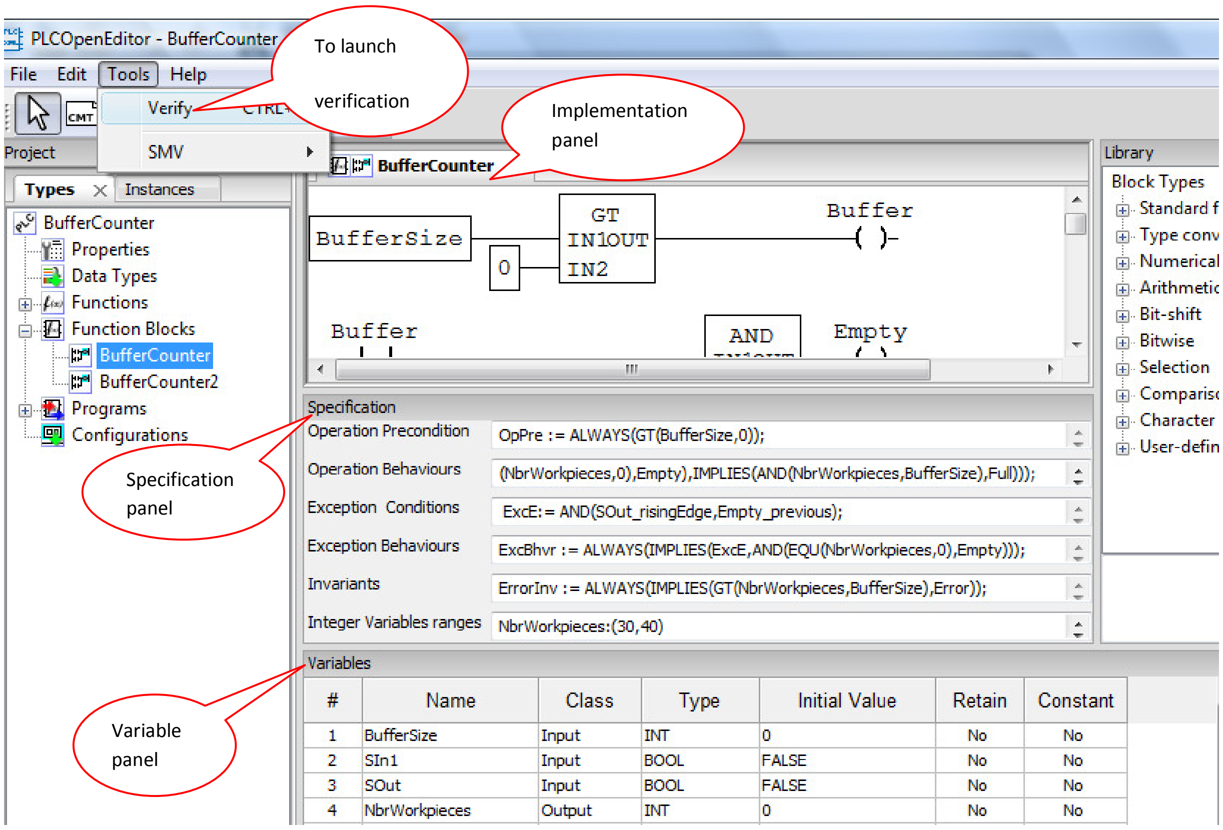
Table1. Specification contents that are written within separated fields

Besides the writing of these eight specifications that were described in Table1, the developer can specify the *minimum* and *maximum* values of the integer variables within the editor, which is required later for verification stage, these amounts should be written in the last text field of specification window(within the editor). For instance, “BufferSize:(40, 50)” shows the initial value of size that is dedicated to Buffer, these values are used in formal verification. In this example, the *minimum* value of *BufferSize* is considered to be *“40”* and the maximum is given by *“50”*. “NbrWorkpieces:(-1, 70)” shows the initial value for the number of work pieces, so the minimum given value of the number of workpieces(i.e. NbrWorkpieces) is “-1” and the maximum given value of this value is “70” in the first attempt of running the program.

After implementing FBs, designing in LDs language and writing the specification in the form of ST language, the user can save both implementation and specification in an XML file by clicking on “File/Save”.

1.4.5 The Output Result of Example

All the specification and information should be edited in the PLCOpenEditor and be saved into the BufferCounter.xml file. There is a program which has been developed in Java, and it acts as a translator which automatically does the translation procedure. So the next step is the translation from ST language to SMV code. The whole process of translation will be triggered when user Click on "*verify*" on the sub-menu under the *Tools* menu of main editor in the software tool. (See in Fig.6).



Specification

Operation Precondition	OpPre := ALWAYS(GT(BufferSize,0));
Operation Behaviours	(NbrWorkpieces,0),Empty),IMPLIES(AND(NbrWorkpieces,BufferSize),Full));
Exception Conditions	ExcE:= AND(SOut_risingEdge,Empty_previous);
Exception Behaviours	ExcBhvr := ALWAYS(IMPLIES(ExcE,AND(EQU(NbrWorkpieces,0),Empty)));
Invariants	ErrorInv := ALWAYS(IMPLIES(GT(NbrWorkpieces,BufferSize),Error));
Integer Variables ranges	NbrWorkpieces:(30,40)

Variables

#	Name	Class	Type	Initial Value	Retain	Constant
1	BufferSize	Input	INT	0	No	No
2	SIn1	Input	BOOL	FALSE	No	No
3	SOut	Input	BOOL	FALSE	No	No
4	NbrWorkpieces	Output	INT	0	No	No

Fig. 6- Launching the verification in the editor

1.4.5.1 Translate to SMV Code

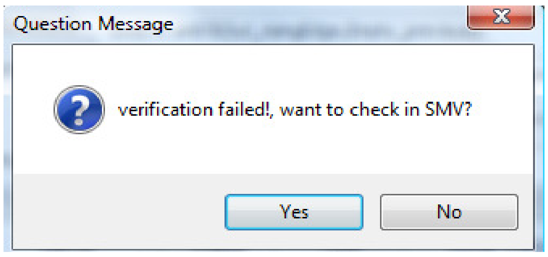
In translation process the implementation will be translated into SMV modules. The specification will be translated to SMV language. Thus the translator first parses the specification text, and then converts it to an SMV code. In consequence of the translation process, the result of translation as input is fed to SMV tool for formal verification, the code will be verified in the SMV tool and the output of the verification will be saved within different files. For instance, the SMV trace will be placed in BufferCounter.smv.

The "BufferCounter.smv" file contains one record for each specification that has been checked. If it is true, states that specifying the truth value of the specification. And if it is false, it presents a *counter example* showing why the specification is false. The result of this checking is also stored in the file "BufferCounter.out" , another output file is "BufferCounter.warn", which contains the errors and warning that has been made by verification process (i.e. running "verify all" within tool bar menu of SMV). In the following sections the verification on the BufferCounter example will be described.

1.4.5.2 Verification

This section takes a closer look at the process of formal verification of specification text that was described in the previous sections, by going through an example.

As it has been explained in previous sections, the result of translation to SMV format and the result of translating the implementation will be the inputs into SMV tool for performing formal verification, to determine whether the implementation fulfills the specification or not. If the Implementation does not fulfill the specification, then the result might be shown to user, moreover the user can check what has been wrong within Implementation process. After completing the verification process, a verification message will be pop up to developer as shown in the Fig.7.

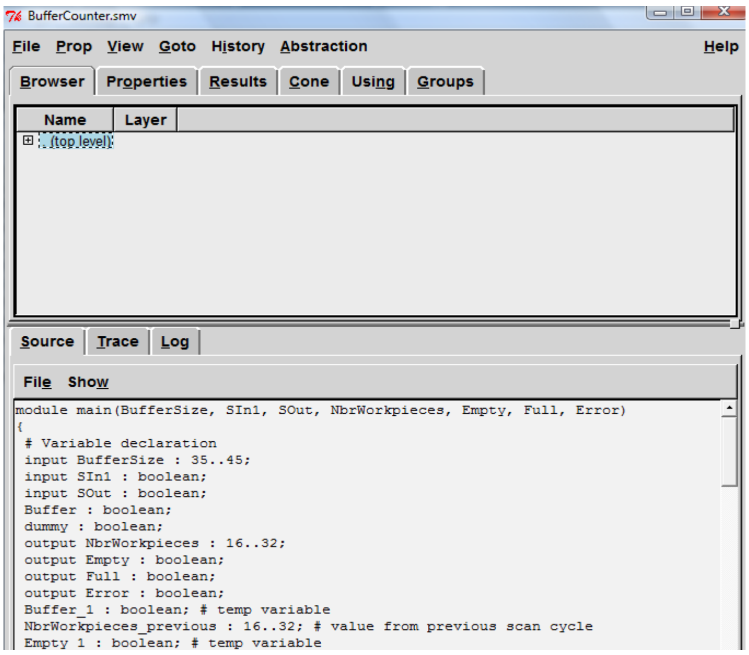


verification failed!, want to check in SMV?

Fig. 7- Informing message when requirements

If "yes" option is chosen, a graphical user interface for "SMV" will be shown. Then the SMV result can be viewed by the developer. However, the SMV graphical interface itself provides the ability of source browsing, *counter example* browsing, abstraction editing, etc [2].

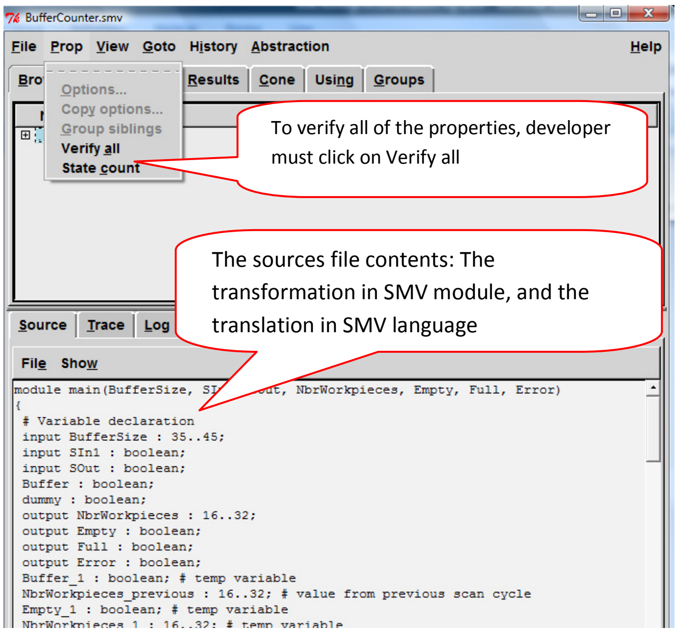
The SMV graphical interface is formed from a main menu bar, and a collection of "panes", each one can be visible by clicking on its tab. These pans are depicted in Fig.8. The first pane that is viewed is called "Source pane", showing the currently loaded source-file (BufferCounter.smv)


BufferCounter.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
Name Layer
(top level)
Source Trace Log
File Show

```
module main(BufferSize, SIn1, SOut, NbrWorkpieces, Empty, Full, Error)
{
 # Variable declaration
 input BufferSize : 35..45;
 input SIn1 : boolean;
 input SOut : boolean;
 Buffer : boolean;
 dummy : boolean;
 output NbrWorkpieces : 16..32;
 output Empty : boolean;
 output Full : boolean;
 output Error : boolean;
 Buffer_1 : boolean; # temp variable
 NbrWorkpieces_previous : 16..32; # value from previous scan cycle
 Empty_1 : boolean; # temp variable
```

Fig. 8- The SMV graphical interface shows source file.

All of the properties in the file can be verified by selecting the "Verify all" option from the "Prop" menu in the program (shown in Fig.9). One particular property (i.e. property name) can be verified by choosing "Verify property" option from the “Prop" menu. This action is supposed to run the SMV model checker in background, and to give a notification to the user whenever the verification is completed.



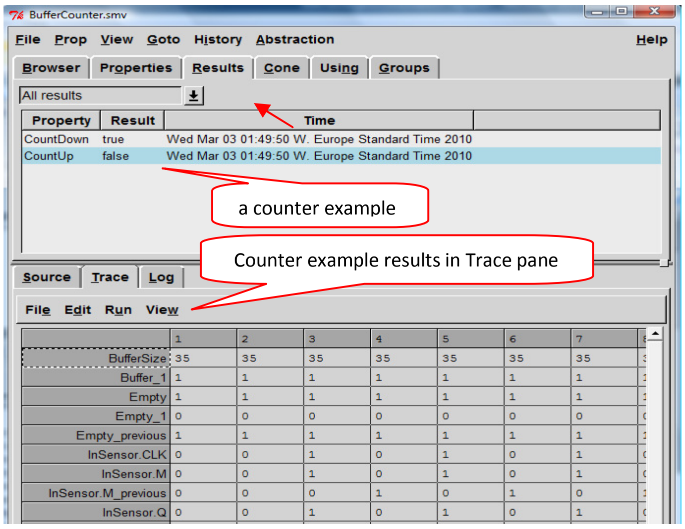
To verify all of the properties, developer must click on Verify all

The sources file contents: The transformation in SMV module, and the translation in SMV language

```
module main(BufferSize, SI      Out, NbrWorkpieces, Empty, Full, Error)
{
 # Variable declaration
 input BufferSize : 35..45;
 input SIn1 : boolean;
 input SOut : boolean;
 Buffer : boolean;
 dummy : boolean;
 output NbrWorkpieces : 16..32;
 output Empty : boolean;
 output Full : boolean;
 output Error : boolean;
 Buffer_1 : boolean; # temp variable
 NbrWorkpieces_previous : 16..32; # value from previous scan cycle
 Empty_1 : boolean; # temp variable
 NbrWorkpieces_1 : 16..32; # temp variable
```

Fig. 9 – The SMV graphical interface for verifying the Source file

The results of the most recent verification run will be displayed in “Results” pane. The properties that have been checked are listed in this pane. The results, either True or False, will also be shown. It has been highlighted in Fig.10.


BufferCounter.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
All results
a counter example
Counter example results in Trace pane
Source Trace Log
File Edit Run View

Property	Result	Time
CountDown	true	Wed Mar 03 01:49:50 W. Europe Standard Time 2010
CountUp	false	Wed Mar 03 01:49:50 W. Europe Standard Time 2010

	1	2	3	4	5	6	7
BufferSize	35	35	35	35	35	35	35
Buffer_1	1	1	1	1	1	1	1
Empty	1	1	1	1	1	1	1
Empty_1	0	0	0	0	0	0	0
Empty_previous	1	1	1	1	1	1	1
InSensor.CLK	0	0	1	0	1	0	1
InSensor.M	0	0	1	0	1	0	1
InSensor.M_previous	0	0	0	1	0	1	0
InSensor.Q	0	0	1	0	1	0	1

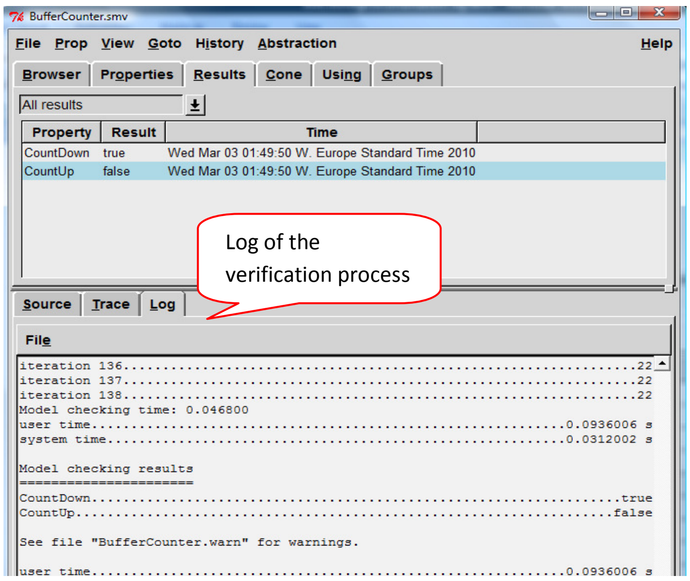
Fig. 10 – The SMV graphical interface for viewing the result in Trace pane

1.4.5.3 TRACE PANE

The *counter examples* and simulation traces can be displayed in the trace pane view. It is similar to a spreadsheet (i.e. shown in Fig.10), where the rows represent the variables and the columns represent time [2].

1.4.5.4 Viewing Log File

A log of the verification process will appear in another pane which is called log pane. This file is created as SMV is progressing to complete the process of model checking.


BufferCounter.smv
File Prop View Goto History Abstraction Help
Browser Properties Results Cone Using Groups
All results
Log of the verification process
Source Trace Log
File

Property	Result	Time
CountDown	true	Wed Mar 03 01:49:50 W. Europe Standard Time 2010
CountUp	false	Wed Mar 03 01:49:50 W. Europe Standard Time 2010

```
iteration 136...............................................................22
iteration 137...............................................................22
iteration 138...............................................................22
Model checking time: 0.046800
user time.....................................................0.0936006 s
system time...................................................0.0312002 s

Model checking results
======================
CountDown...................................................................true
CountUp....................................................................false

See file "BufferCounter.warn" for warnings.

user time.....................................................0.0936006 s
```

Fig. 11 – The SMV graphical interface for observing the log file

The verification process is launched by internal commands which save the results in different files. For instance *"log file"* is one of those files that a user can observe the result of verification. There are some other output files that will be created such as "*BufferCounter.warn*" showing the warnings, and "BufferCounter.out*"* that shows the errors if any exists.

However, if verification failed after launching the verification process, the user is informed before showing the contents of the output files. If any of specifications is false, the SMV model checker tool will produce a *counter example*, the *counter example* contents can prove that the specification is false [11].

1.4.6 Benefits

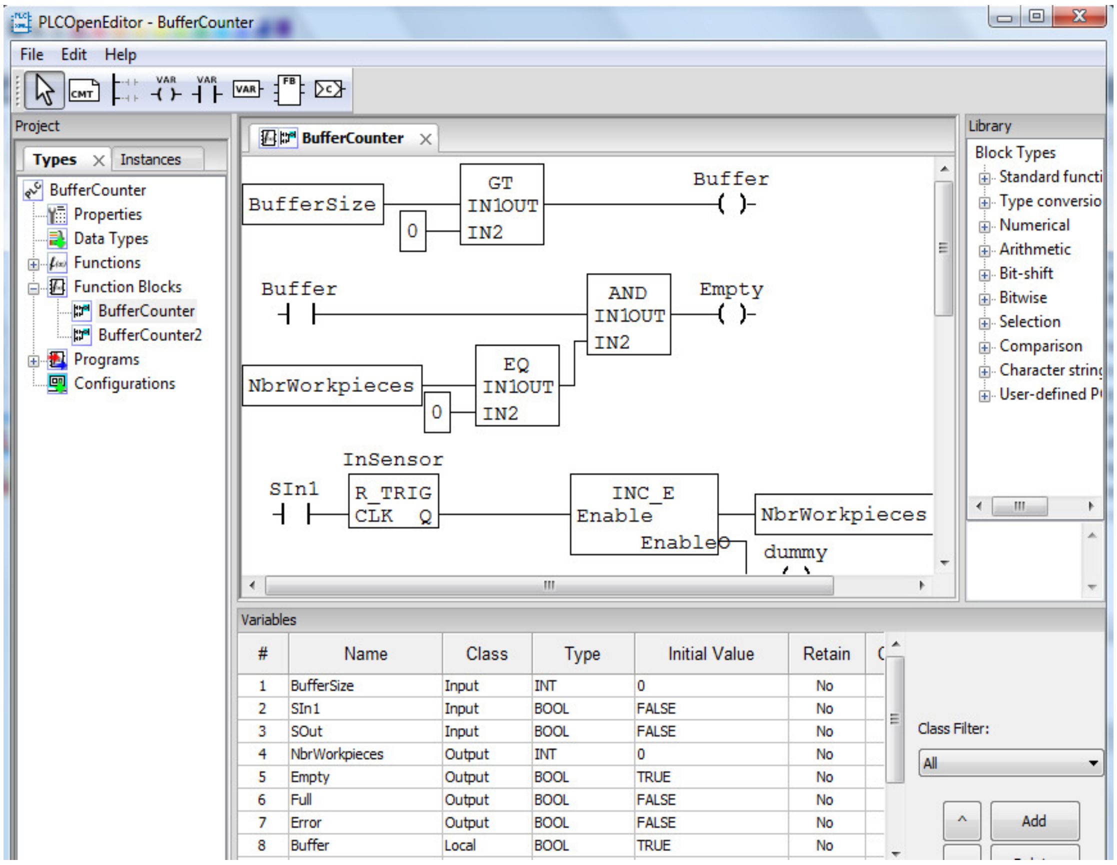
This case study presented an industrial example that contains a formal verification of a PLC system, which shows how the specification can be automatically translated into Cadence SMV input language, and how it is formally verified. It shows how the formal verification process can be useful in real applications. The Software tool automatically verifies that the implementation fulfils the specification. Otherwise, if the implementation does not fulfil the specification, it produces a *counter example* to state the reason that the specification is not fulfilled. In consequence, the RAC can be corrected and verified again until the complete specification is fulfilled [3]. In next chapters will describe how the program has been implemented.

CHAPTER 2 – Specification Tool

In this chapter, the implementation of the specification editor is described. This editor window is integrated to PLCOpenEditor that was implemented by LOLITech group in Beremiz PLCOpen project. The PLCOpen Editor is the one of Beremiz's sub-projects; Beremiz is an open source for automation IDE that is based on IEC 61131-3, which is a multi-platform IDE for automation [6].

2.1 Building Graphical User Interface

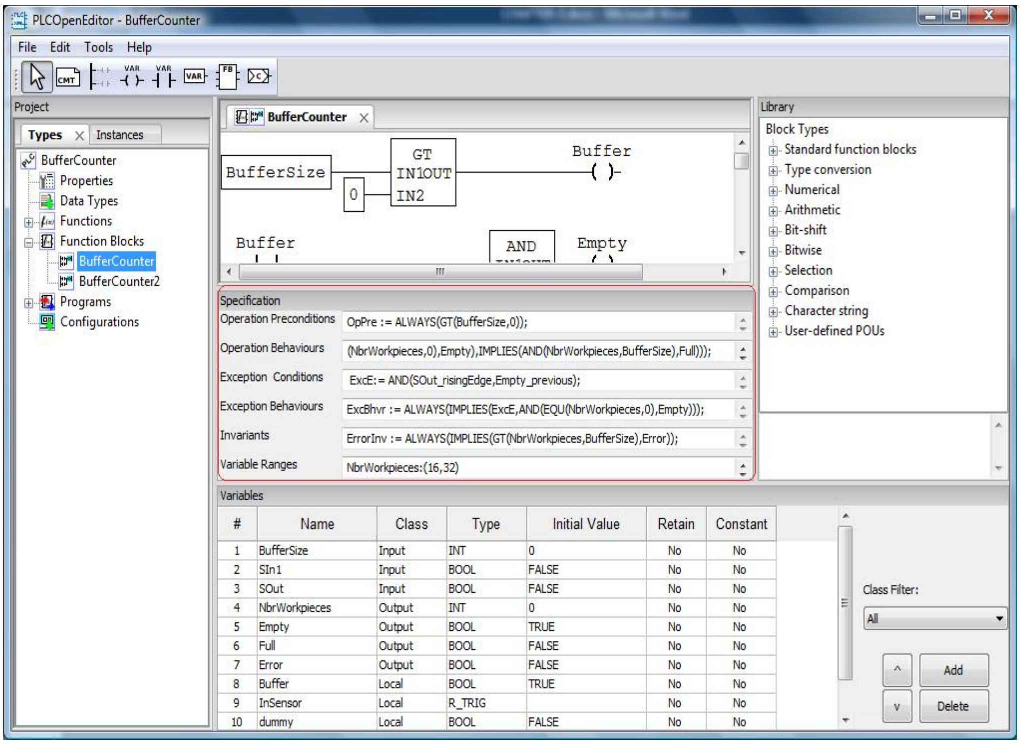
A Graphical User Interface (GUI) is required to be used for creating the example components in it. In this project, we decided to develop the same PLCOpenEditor GUI that was implemented in python by LOLITech group; this PLCOpenEditor is shown in this Fig.11.



#	Name	Class	Type	Initial Value	Retain	C
1	BufferSize	Input	INT	0	No	
2	SIn1	Input	BOOL	FALSE	No	
3	SOut	Input	BOOL	FALSE	No	
4	NbrWorkpieces	Output	INT	0	No	
5	Empty	Output	BOOL	TRUE	No	
6	Full	Output	BOOL	FALSE	No	
7	Error	Output	BOOL	FALSE	No	
8	Buffer	Local	BOOL	TRUE	No	

Fig.11- PLCOpenEditor before integrating specification window

The PLCOpenEditor before integrating the specification window is similar to what is shown in Fig.11, in this editor the developer can make Functions and Function Blocks, and set the input and output variables, etc. After integrating another panel for specification, it looks like Fig.12 that is shown in next page.



Specification

Operation Preconditions	OpPre := ALWAYS(GT(BufferSize,0));
Operation Behaviours	(NbrWorkpieces,0),Empty),IMPLIES(AND(NbrWorkpieces,BufferSize),Full)));
Exception Conditions	ExcE:= AND(SOut_risingEdge,Empty_previous);
Exception Behaviours	ExcBhvr := ALWAYS(IMPLIES(ExcE,AND(EQU(NbrWorkpieces,0),Empty)));
Invariants	ErrorInv := ALWAYS(IMPLIES(GT(NbrWorkpieces,BufferSize),Error));
Variable Ranges	NbrWorkpieces:(16,32)

Variables

#	Name	Class	Type	Initial Value	Retain	Constant
1	BufferSize	Input	INT	0	No	No
2	SIn1	Input	BOOL	FALSE	No	No
3	SOut	Input	BOOL	FALSE	No	No
4	NbrWorkpieces	Output	INT	0	No	No
5	Empty	Output	BOOL	TRUE	No	No
6	Full	Output	BOOL	FALSE	No	No
7	Error	Output	BOOL	FALSE	No	No
8	Buffer	Local	BOOL	TRUE	No	No
9	InSensor	Local	R_TRIG		No	No
10	dummy	Local	BOOL	FALSE	No	No

Fig.12- PLCOpenEditor after integrating specification panel

The aim of the first part of this chapter is to show how the specification editor has been developed within framework of this project, the specification editor (shown in Fig.12) is one editor which has been integrated to the previous editor (shown in Fig.11) that basically was made by Beremiz group. In the following sections, this development has been described. It is not possible to describe all details of programming and codes in this report; however the attempt was to give a main view from what has been done.

2.1.1 Implementation and Integration of Specification Panel

To Implement and integrate the specification window into the PLCOpenEditor, the following lines has been written in the PLCOpenEditor to build the specification panel.

```
self.panel = SpecificationPanelIndexer(self, self, self.Controler)
 self.AUIManager.AddPane(self.panel,
wx.aui.AuiPaneInfo().Caption("Specification").Center().Layer(0).BestSize(wx.Size(50,200)).CloseButton(False))
```

Table 2. Part of PLCOpenEditor code which implements the specification panel

The following Fig.13 shows the main files of editor that have been extended in this project. Those classes that have been imported in “PLCOpenEditor.py” file have .py extension.

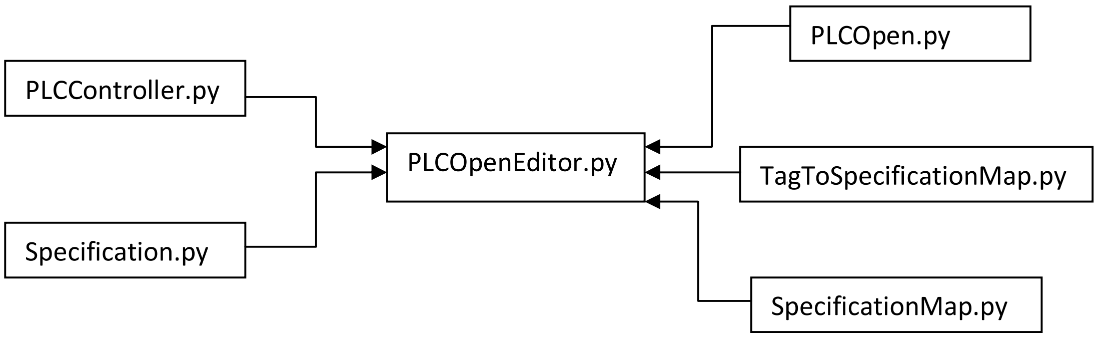


Fig.13- The files that have been changed or built in this project

The *"Specification.py", "SpecificationMap", and "TagToSpecificationMap*" have been built, and the other files just have been changed.

2.1.2 Dividing Specification Panel

A specification panel has been divided to six parts. Each part has a label and a text field (shown in Fig.14), so it is more convenient for developer to enter the input texts into these text fields. The specification text should be saved in and be loaded from the file with .xml extension.

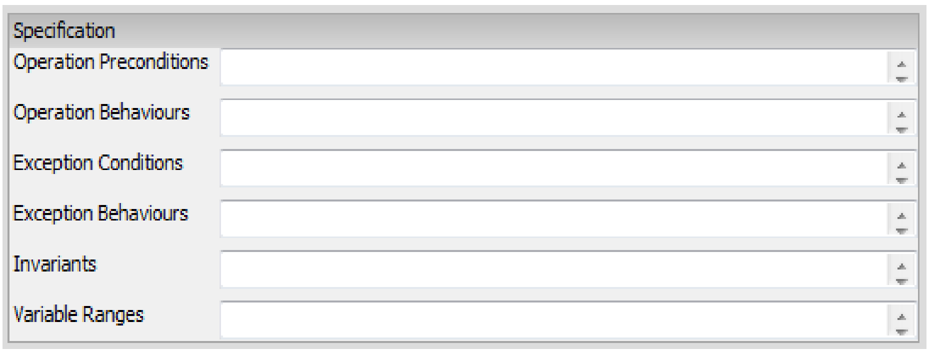


Fig.14- specification panel view that has been integrated to PLCOpenEditor

2.1.3 Resizing Specification Window

At this stage, a specification panel (i.e. Fig.14)within PLCOpen Editor(i.e. main editor shown in Fig.12) has been created. However, PLCOPen Editor has many panels such as *variables* panel, library, project. Whenever the developer clicks on the *maximize* button(on the right corner of window), all of these panels will be resized to the maximum. The specification panel should also be maximized whenever the panels are maximized.

The code (i.e. shown in Table 3) was extended within "PLCOpenEditor.py" to provide a possibility for automatically resizing of the specification window. When a new file is opening to the main editor or clicking on maximize button that is available on the right corner of the main editor.

```
sizer = wx.FlexGridSizer(cols=2, hgap=6, vgap=6)
sizer.Hide(self)
sizer.AddMany([Ope_PreLbl, (Ope_PreTxt, 1, wx.EXPAND), Ope_BehLbl, (Ope_BehTxt, 1, wx.EXPAND), Exc_ConLbl,
(Exc_ConTxt, 1, wx.EXPAND),Exc_BehLbl, (Exc_BehTxt, 1, wx.EXPAND),InvariantsLbl, (InvariantsTxt, 1,
wx.EXPAND),VariablesLbl, (VariablesTxt, 1, wx.EXPAND)]])
self.panel.SetSizer(sizer)
sizer.AddGrowableCol(1,0)
```

Table 3. Part of PLCOpenEditor code which extends the size of specification panel

The aim of the part of code that is shown in Table 3 is to solve the problem of resizing ability, to resize the specification panel whenever the size of main editor changes. The panel has put in a "sizer", the proportion is set to "1", and the wx.EXPAND style flag is used for each "widget" in order to resize each of the text fields within the panel. Here, a widget is an element of editor such as: text field, buttons, text area, etc [6]. The widget displays information arrangement changeable by user. All eight rows and the second column have been made growable. By this way, the text controls is allowed to be extended (i.e. growable), when the window is resized. The eight text controls will grow in horizontal direction. "wx.EXPAND" has been applied to make the widgets expandable.

2.1.4 Switching Between Tabs

Inside the PLCOpenEditor (shown in Fig.12), the user can open different FBs. Whenever the user chooses a FB, one different tab is opened (for each FB). By clicking on tabs within editor different implementation and specification are shown. Since each tab in the PLCOpenEditor has its specific information, the user should visit different information regarding to different tabs when switches between various tabs. So it is expected that information change when the user switch to another tab. For instance, the information within each specification editor panel belongs to only one tab that is opened; hence the corresponding information should be updated by switching to another tab. hereafter this possibility of switching between different FB's is called Tab-Switching.

During the development of specification panel, there was a problem when user was switching to the next tab (Tab-Switching). The contents of specification within the text editor were the same as the information in the previous tab. while the contents should be changed over different tabs.

The quality of Tab-Switching for specification panel resembles from the previous implementation of Tab-Switching for Variable panel. In this example, each "Tab" has a "tag name", and each "Variable Panel" instance is associated to a tag name.

The class “TagToSpecificationMap ()” has been created in “TagToSpecificationMap.py” to be used for adding, updating the contents of specification panel to a hash map or removing those contents from the hash map. The classes within this file will be imported into PLCOpenEditor.py file.

The hash map data structure that keeps all specification information over different tabs was named: mySpecificationMap = {}, It is more efficient to find / search for items in a hash map than in a list.

For instance, if there are three tabs which are opened, myspecificationMap has all three part of information that loaded in the editor. Each one having the list of its own displayed specification. Whenever the user switches to a different tab, the list corresponding to that tab will be filled.

The “storeSpecification(self, tabName)” method is implemented such that it has the ability of storing data in the file when Tab-Switching will happen, the contents of each specification window will be stored before Tab-Switching , when “self.storeSpecification(oldTab)” has been called. The last information within the tab will be removed just after switching a “Tab”.

2.2 Saving and Loading Specification Data

It was important to know which part of previous code in PLCOpenEditor.py has been written for opening and loading the data such as inputs to *variable* panel window. This learning from the previous code hinted us to apply the similar implementation for loading the information from .xml file to the specification window. Also the implementation must be extended for saving the information to the .xml file. Two methods have been written in PLCControler.py file for this purpose. “SetEditedElementSpecification()” and “GetEditedElementSpecification()”.

2.2.1 Saving Specification to a XML file

The entire project is saved when the developer clicks on the File/Save on the main menu of PLCOpen Editor, then the contents will be saved in a XML file based on PLCOpen standard (See [6]). In this standard, there is not such a pre-provided tag for storing the content of new window editor, but it is required to store the contents only in this file. According to the PLCOpen standard, each tag of the .xml file can have a ‘documentation’ tag. Within this tag, a user is allowed to write any context. Hence, the only appropriate place for storing the specification data was suggested to be in documentation tag within that “.xml file”, the method for set specification function is written as follow:

```
def SetEditedElementSpecification(self, tagname, specification, debug=False):

  words = tagname.split("::")
  if words[0] == "P":
    pou = self.GetEditedElement(tagname, debug)
    if pou is not None:
      if(pou.interface is None):
        pou.interface = plcopen.pou_interface()
      if (pou.interface.getdocumentation() is None):
            pou.interface.adddocumentation()

      pou.interface.getdocumentation().settext(specification)
      self.BufferProject()
      self.ProjectBuffer.CurrentSaved()
```

Table 4. Part of PLCOpenEditor code which implements storing from *specification* panel into XML file

It was needed to call *"self.BufferProject() "* in *"SetEditedElementSpecification"* after setting specification for POU. The calling was done from PLCOpenEditor code. It must also refresh PLCOpenEditor title and EditMenu.

So by calling the method that is written in Table 4, the specification contents can be stored into the *"documentation"* tag of the interface of each "POU" in the XML file.

2.2.2 Loading Information from XML-File to Editor

The method "GetEditedElementSpecification"is to get the specification data from .xml file, and loading those data into text fields. Transferring data into the specification Editor of PLCOpenEditor is done when the user opens the FB, one of the important classes for this task was "xmlClass" File.

```
def GetEditedElementSpecification(self, tagname, debug=False):
  words = tagname.split("::")  # words is a list of ['p','temp']
  if words[0] == "P":
    pou = self.GetEditedElement(tagname,debug)
    if (pou is not None):
      if(pou.interface is not None):
        if(pou.interface.getdocumentation() is not None):
          return pou.interface.getdocumentation().gettext()

   return ""
```

Table 5. Part of PLCOpenEditor code which implements the loading of information to the editor

Hence, when the GetEditedElementSpecification() is called, it is expected to load all of the information and transfer them into the text fields of the editor screen , the information included all the specification and initial variable inputs which have been saved within documentation tag within XML file. The documentation is a single text element.

2.3 Creating “Tools” Menu

On the “*PLCOpenEditor”* menu bar, a “*Tools” menu* has been created for running the Java program. Under this “*Tools”* menu, a sub menu named “Verify” has been provided too, the “Verify” sub menu is created to launch the program. When developer clicks on the “Verify”, both the specification and the implementation will be transferred into SMV format, the SMV tool will be run and the result of verification process will be checked to see if the requirements are met. For running this process, several commands have been run with in background on command line while program is running. All of the commands has been called from “OnVerifyMenu(self, event)” method within PLCOpenEditor.

The methods“ ShowVerificationFulfilledMessage(self,event)”,“ ShowWarningMessage(self,event) ” and “ ShowVerificationFulfillmentFalseMessage(self,event)”, are those methods that have been implemented to demonstrate the results messages to the developer.

CHAPTER 3- Java Implementation

This chapter emphasize on the implementation of translator program which has been developed in Java.

3.1 Former Work at Department- (Translation of Control Logic into SMV)

The folder structure of "*translator*" implementation has been shown in Fig.15.

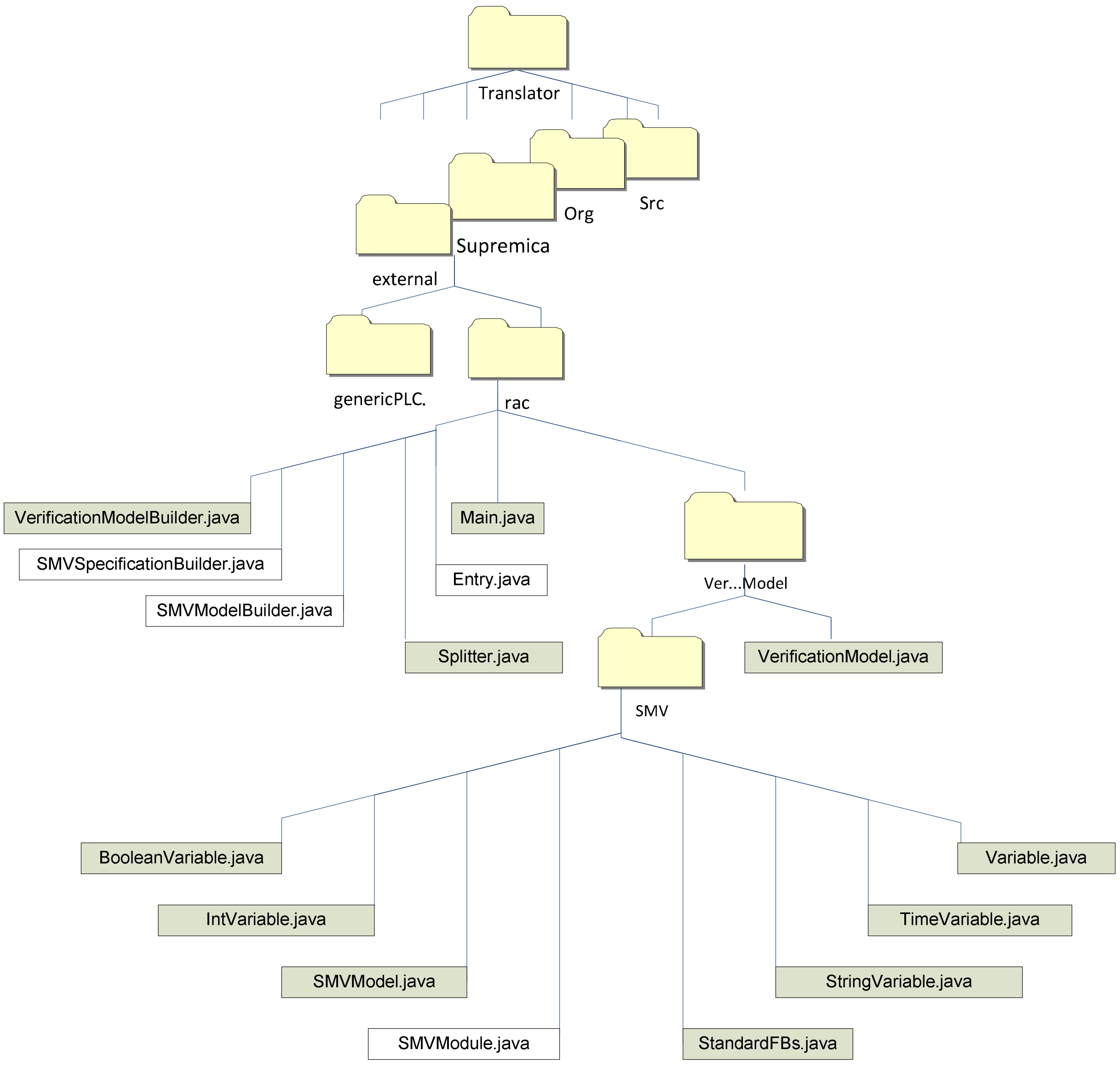


Fig.15 - the folder structure of Translator

3.2 Implement Translator (Translation of Specification Text into SMV Form)

In Fig 15, under the *"rac"* folder, the name of the java files which have been developed is depicted. The white boxes including names such as: "SMVSpecificationBuilder.java", "SMVModule.java" and "SMVModelBuilder.java", are the files that have been developed or changed in this project.

The remaining java files, those files that are marked in gray colour boxes in Fig.15, have been implemented previously to this work. Mainly, they belong to the Implementation part of designing a model of PLC software components, which has been developed by Oscar Ljungkrantz.

Furthermore, the Translation of Control Logic to verification tool (SMV) has been described in paper [3] p.8. On this Translation the *body* of the RAC (shown in chapter1-section3, and Fig.2) is transformed into an input into Cadence SMV.

3.2.1 Loading Original Specification for Converting into SMV

The original specification string which has been saved in the documentation part of .xml file should be read by translator program. The Translator will convert the specification string into the SMV format.

The "Public boolean createSpecification(String originalSpec) " is the the most important method of the "SMVSpecificationBuilder" class, which translates the original specification (taken from the editor) to the corresponding specification in SMV.

3.2.2 Parsing Original Specification String and Identifying Type of Words

The original specification (originalSpec), which has been stored in ".xml" file, should be separated from tag elements. The tag element is the word inside "<" and ">". The words that are tag elements are considered as delimiters. And the method "specSeperator (String strSep, String originalSpec)" separates the texts between tags(saved in "strSep" list). Then the texts are retuned and are stored into "strSplited" string list.

3.2.3 Build a Parse Tree Structure for Keeping Specification Texts.

An ordered tree data structure in the prefix form is used to store the array of strings, where the keys are the separated specification strings. The tree will be a general tree, none of nodes store the key associated to that node, and while the position of node in the tree indicates what key a node is associated with. All the descendants of a node have a common prefix of the associated string to that node. Moreover, the root of tree is associated with an empty string. The *values* are associated with leaves and some inner nodes.

In order to traverse this tree (tree T), the preorder traversal is applied. In a preorder traversal of the tree T, the root of T is visited first, and consequently the subtrees of the root are traversed one by one. However, if the tree has a specific order then the subtrees are traversed according to the order of the children. The subtrees are traversed recursively.

The algorithm which is used for performing the preorder traversal of the subtrees of the tree T, is useful to make a linear ordering of the nodes of the T such as string. by this way, the parent must always come before their children in the ordering.

In this part, the string will initially be set in a tree structure by "generateTree(String input)" method. By calling "generateTree" method, another method "evaluate (tree, tree.root (), tokens)" is called to check if whether the tree structure string is correct or not! The methods traverse the tree and the subtrees recursively until it reaches to the last nodes (the left and the right nodes of subtrees) and then evaluate the nodes and translate the root of that subtree, the result will be return to the one level before the current level. This process will be continued recursively until the result of each level will be return to the one level before, in the root we will have the evaluations from the left and right sides of root along with the translation of the node.

3.2.4 Build "_previous" for Each Variable

The "Boolean addPreviousVariableForSpecification(string variableName,int previousLevel) " is a method that creates the variable , it is used to add *"_previous"* suffix as many times as is required, it depends to the level of variable, *"_previous"* is the value of variable for the last time execution of RAC .
The method counts the level (i.e. number) of *"_previous"* suffix that is used and is stored as n. then the method will add the *"_previous"* to the name of variable, the number of "_previous" that will be added is equal to the number of variable level. For instance, if the "*variableLevel*" of "*X*" is equal to n, then method will add n-1 X prefix's around every specification, to avoid problems with pre-initial values. For n=3, the format of variable will be *X _previous_previous_previous*. The parentheses are added to the resulting SMV specification to preserve the order of precedence.

The static structure of *translator* program is described in following pages. Fig.17 and Fig.18 show the form of class diagrams that is applied.

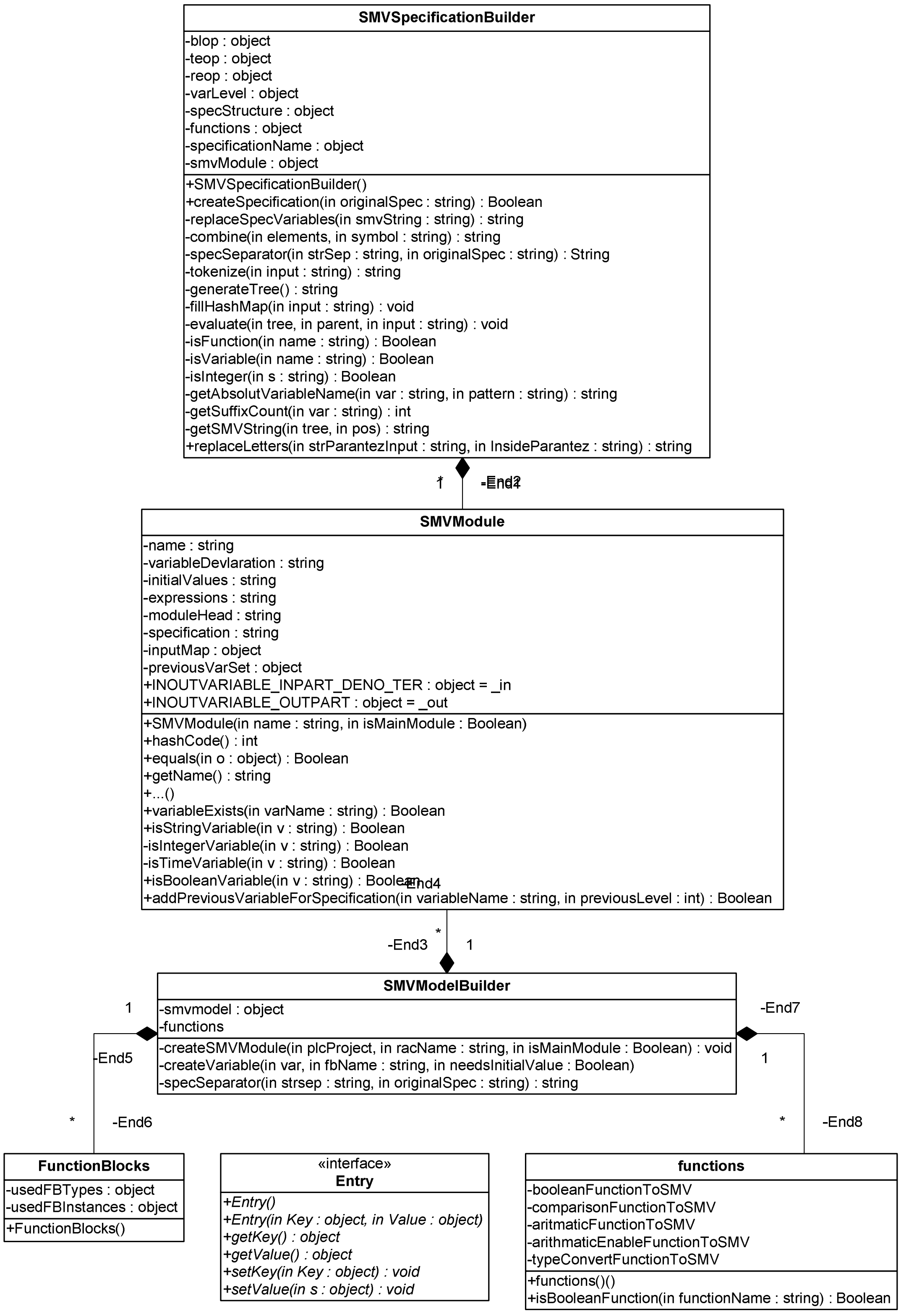

SMVSpecificationBuilder
-blop : object
-teop : object
-reop : object
-varLevel : object
-specStructure : object
-functions : object
-specificationName : object
-smvModule : object
+SMVSpecificationBuilder()
+createSpecification(in originalSpec : string) : Boolean
-replaceSpecVariables(in smvString : string) : string
-combine(in elements, in symbol : string) : string
-specSeparator(in strSep : string, in originalSpec : string) : String
-tokenize(in input : string) : string
-generateTree() : string
-fillHashMap(in input : string) : void
-evaluate(in tree, in parent, in input : string) : void
-isFunction(in name : string) : Boolean
-isVariable(in name : string) : Boolean
-isInteger(in s : string) : Boolean
-getAbsolutVariableName(in var : string, in pattern : string) : string
-getSuffixCount(in var : string) : int
-getSMVString(in tree, in pos) : string
+replaceLetters(in strParantezInput : string, in InsideParantez : string) : string
1
-End2
SMVModule
-name : string
-variableDevlaration : string
-initialValues : string
-expressions : string
-moduleHead : string
-specification : string
-inputMap : object
-previousVarSet : object
+INOUTVARIABLE_INPART_DENO_TER : object = _in
+INOUTVARIABLE_OUTPART : object = _out
+SMVModule(in name : string, in isMainModule : Boolean)
+hashCode() : int
+equals(in o : object) : Boolean
+getName() : string
+...()
+variableExists(in varName : string) : Boolean
+isStringVariable(in v : string) : Boolean
-isIntegerVariable(in v : string) : Boolean
-isTimeVariable(in v : string) : Boolean
+isBooleanVariable(in v : string) : Boolean
-End4
+addPreviousVariableForSpecification(in variableName : string, in previousLevel : int) : Boolean
*
-End3
1
SMVModelBuilder
-smvmodel : object
-functions
-createSMVModule(in plcProject, in racName : string, in isMainModule : Boolean) : void
-createVariable(in var, in fbName : string, in needsInitialValue : Boolean)
-specSeparator(in strsep : string, in originalSpec : string) : string
1
-End5
*
-End6
-End7
1
*
-End8
FunctionBlocks
-usedFBTypes : object
-usedFBInstances : object
+FunctionBlocks()
«interface»
Entry
+Entry()
+Entry(in Key : object, in Value : object)
+getKey() : object
+getValue() : object
+setKey(in Key : object) : void
+setValue(in s : object) : void
functions
-booleanFunctionToSMV
-comparisonFunctionToSMV
-aritmaticFunctionToSMV
-arithmaticEnableFunctionToSMV
-typeConvertFunctionToSMV
+functions()()
+isBooleanFunction(in functionName : string) : Boolean

Fig.17- Part (I) of Class Diagram for Translator program

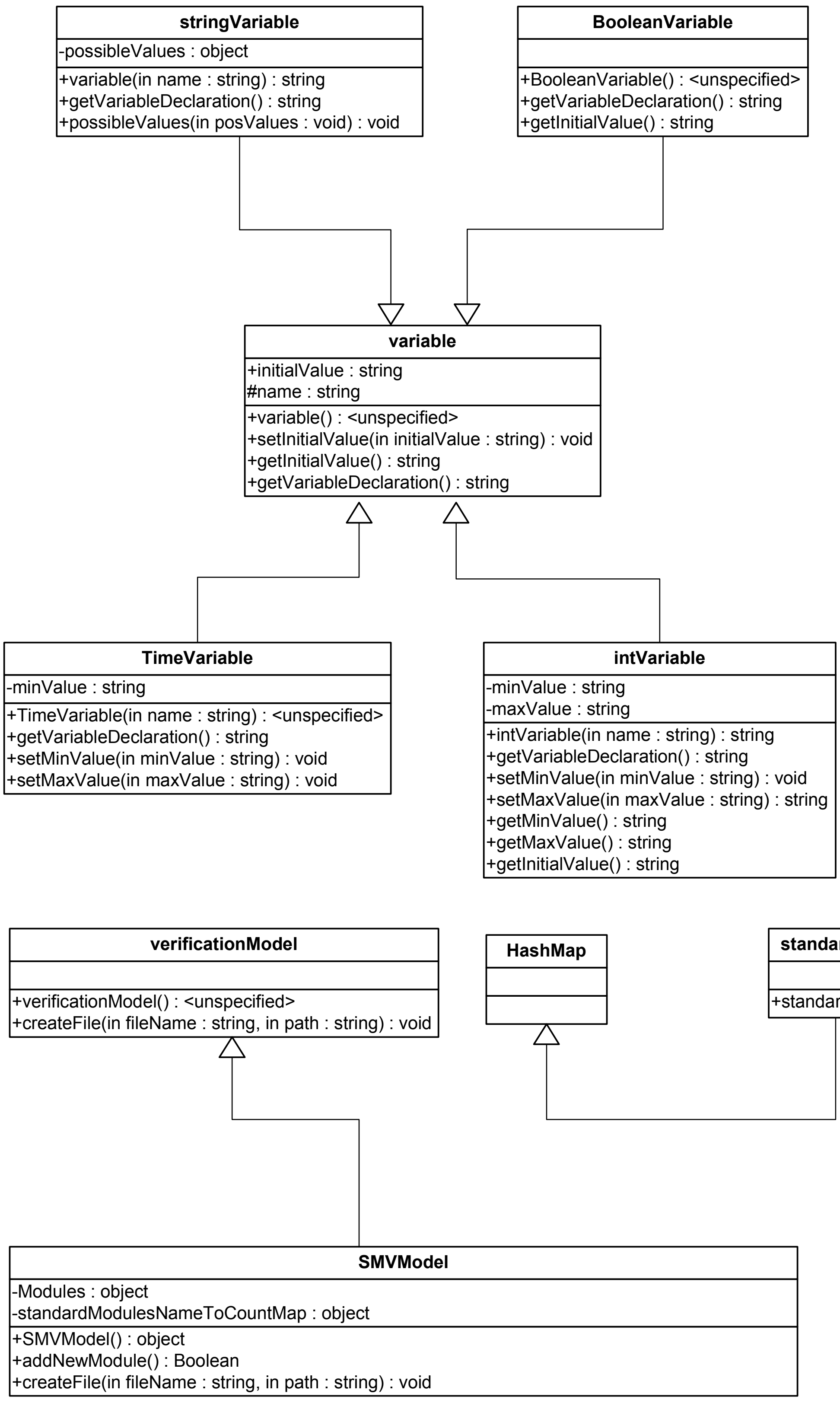

stringVariable
-possibleValues : object
+variable(in name : string) : string
+getVariableDeclaration() : string
+possibleValues(in posValues : void) : void
BooleanVariable
+BooleanVariable() : <unspecified>
+getVariableDeclaration() : string
+getInitialValue() : string
variable
+initialValue : string
#name : string
+variable() : <unspecified>
+setInitialValue(in initialValue : string) : void
+getInitialValue() : string
+getVariableDeclaration() : string
TimeVariable
-minValue : string
+TimeVariable(in name : string) : <unspecified>
+getVariableDeclaration() : string
+setMinValue(in minValue : string) : void
+setMaxValue(in maxValue : string) : void
intVariable
-minValue : string
-maxValue : string
+intVariable(in name : string) : string
+getVariableDeclaration() : string
+setMinValue(in minValue : string) : void
+setMaxValue(in maxValue : string) : string
+getMinValue() : string
+getMaxValue() : string
+getInitialValue() : string
verificationModel
+verificationModel() : <unspecified>
+createFile(in fileName : string, in path : string) : void
HashMap
standardFBs
+standardFBs()
SMVModel
-Modules : object
-standardModulesNameToCountMap : object
+SMVModel() : object
+addNewModule() : Boolean
+createFile(in fileName : string, in path : string) : void

Fig.18- Part (II) of Class Diagram for Translator program

3.2.5 Converting Tree to SMV Expression

In converting the tree into the SMV expression each operator acts as a parent and the operands as children in tree in which each operand could be an expression by itself (i.e. represented as a sub tree, containing operator and operands). Generally, the translation is done recursively; the operator (the parent) will be translated and get the evaluated SMV expression of both children (recursively) which results in two strings.

Each element of the string (i.e. a tree structure) will be translated to SMV by another methods which are called "getSMVString(NodeTree tree, Position pos)"and "replaceSpecVariables(String smvString)". Moreover, in this project the translating process is done without considering the grammar issues.

The translation is based on the table in appendix B (similar table exist in [3]). An example of original specification string and its translation will be shown in following tables as well.

For simplicity, the input string made in the form of prefix; prefix is one way of writing expressions. For instance, an expression such as "A * (B + C) / D" form will be changed to the form that operands are written before their *operators*, then it looks like this form: / (* (A, + (B, C)), D).

```
<![CDATA[<OpPre>OpPre1:=ALWAYS( (MaxMoveTime>0)& ( (DesiredState=Forward) OR
(DesiredState=Backward)));</OpPre>
<OpPre>OpPre2:=ALWAYS( MoveIn & MoveIn_Previous->(DesiredState=
DesiredState_previous));</OpPre>
<OpBhvr>MoveOrAlarm:=ALWAYS((ALWAYS MoveIn)->EVENTUALLY((State = DesiredState) OR
TimeOutActFwds  OR TimeOutActBwds));</OpBhvr>
<ExCond>Reset:= ResetAlarms;</ExCond>
<ExBhvr>ResetBhvr:= ALWAYS(ResetAlarms->(Not TimeOutActFwds &  NOT TimeOutActBwds & NOT
AlarmUnauthMove)); </ExBhvr>
<Inv>NotIllegalMove:=NEVER(ActuatorFwds & ActuatorBwds);</Inv>
<Inv>Stop:= ALWAYS((NOT MoveIN)->(NOT ActuatorFwds & NOT ActuatorBwds));</Inv> ]]>
```

Table 6: showing the specification string which has been loaded from the .xml file

The translated string is similar to following:

```
OpPre1:  assert G ( (MaxMoveTime>0)& ( (DesiredState=Forward)  | (DesiredState=Backward)));
OpPre2 : assert G (( MoveIn & MoveIn_Previous)->(DesiredState= DesiredState_previous
MoveOrAlarm : assertG (((G  MoveIn))-> F ((State = DesiredState)  | TimeOutActFwds   |
TimeOutActBwds));
Reset : assert ResetAlarms;ResetBhvr : assert G ((ResetAlarms)->( ~ TimeOutActFwds &   ~
TimeOutActBwds &  ~ AlarmUnauthMove)); ~IllegalMove : assert G ~(ActuatorFwds &
ActuatorBwds);
Stop : assert G ((( ~ MoveIN))->( ~ ActuatorFwds &  ~ ActuatorBwds));
```

Table 7: showing the translation of the specification string

As it is shown in the above tables, for instance the string from Table 7 is:

“OpPre1:=ALWAYS ((MaxMoveTime>0) & ((DesiredState=Forward) OR(DesiredState=Backward)));” Is translated based on the Table (I) into following string:

“OpPre1: assert G ((MaxMoveTime>0) & ((DesiredState=Forward) | (DesiredState=Backward)));”

3.2.6 Call Methods

In order to check the validity of a string evaluate() method is called. This method checks if the input string is valid word, if it is not valid then it prints warnings and it would return false. The “smvModule.variableExists(varName) ” method is called to check if whether such a word is a variable name. This method is called from isVariable(String name) method. This function is implemented in another file which is called SMVModel.java.

If the word does not exist among the variables, then it will be checked if that is available in any of other Hash maps. If it is available then the key word should be replaced with its equivalent value that is available in hash map. Otherwise if that word doesn’t not exist inside any of the maps, it will be considered as an unknown variable, operator or functions either suffix. However, the program should warn the user for correcting the mistakes.

Inside the “isVariable” method, also “isInteger” method is called to check if the input is not variable, then it would be a valid integer or not. There are different maps for boolean *operators* and *functions*, for temporal *operators* and *functions,* and for suffix.

“isFunction(String name)” also will be called inside the evaluate method to check if the taken name would be a name of the functions which has been defined previously.

getSuffixCount(String var), getAbsolutVarName(String var, String pattern) are the others methods which will be called too. “setSpecification(String spec) ” method is used to set/add the finished SMV- Specification to the SMV module.

Different words have been substituted with the dedicated words that are available in different maps. The program has the ability to split the main string to tokenize the words and then compare each word (i.e. token) with its pair which is available in the map (i.e. shown in below). If a word is available in any map then swap that words with its equivalent, otherwise it must give a warning to the user for correcting that word.

In order to translate this string we have put the different words in the separated ‘Hash Map’, the schema of these Hash maps is shown in Appendix (B).

3.3 The Structure of Specification

Whenever the translation all of the specification strings is finished, it is required to merge the all of the strings together to build one string.

3.3.1 Merging Strings Together

The formal specification is a structured approach which allows us to break a system into subparts and rearrange them later by putting the subparts together again, hence this formal way is suitable for describing the structure of reusable component system such as RAC, and moreover it should also describe the main functionality of the component.

In paper [1], it has been proposed to apply a formal specification for complementing the description of RAC. The structure of the proposed specification has been inspired from by the concept and terminology of "Design by Contract".

Applying "Design by Contract" has particular attention to reliability aspects of object-oriented software, and discusses how to reduce bugs by building software components on the basis of carefully designed contracts [13].

Model checking in SMV tool can be performed automatically and gives the possibility to produce *counter examples*; the *counter examples* demonstrate the reason that specification was not fulfilled.

The five part of specification structure for scan cycle based RACs have been described in [2]. The RAC is consisted of interface, and implementation. Specification is an important part of the interface and it consists of the following five parts:

- Operation Preconditions
- Operation Behaviors
- Exception Conditions
- Exception Behaviors
- Invariants

All invariants and all exception behaviors should hold after every update of the RAC. Each exception behavior should include at least one exception condition (and each exception condition must be included in at least one exception behavior). Each operation behavior must be fulfilled after every update if all operation preconditions and no exception conditions were fulfilled at the start of and during the update. Mathematically, this is expressed as:

$$opBhvr \bigvee \sim\left(\bigwedge_{opPre\ \in OpPre} opPre\right) \bigvee F\left(\bigvee_{exc\ \in Exc} exc\right) \quad (1)$$

In formula (1), where F is the temporal "future" operator, should hold for each $opPhvr \in OpBhvr$. If for instance the $|OpPre| = 0$, then $\wedge_{opPre\ \in OpPre}\, opPre$ should be replaced by "*true"*, however if $|Exc| = 0$, then $F(\vee_{ecx\, \in\, Exc}\, exc)$ should be replaced by false [2].

The function of the RAC indicated by formula (1) has been implanted in "*Translator*" part of this project. Inside the method "createSpecification(String originalSpec)" in Java program, the "OpBhvr", "opPre" and "exc" will be taken from "specStructure " Hash map and will be combined based on the formula(1) , with the "& " either "|" operator to construct the final specification structure.

CHAPTER 4 – Integrating all programs in one tool

This part describes how the integrating of the implemented programs is performed. The aim of integrating the programs is to build a software tool, and the integrated software tool is a combination of three programs: the implementation of translator, the implementation of GUI, and SMV tool.

4.1 Calling Translator

The java program should be called from the editor, the "OnVerifyMenu(self, event)" method is the method which has been developed in "PLCOPenEditor.py "file for this purpose.

The translation is started with the following command line that will run the java program.

cmd_line1= 'java -cp build;lib/jdsl.jar -enableassertions org.supremica.external.rac.Main -SMV ' + middle + ' . %5 %6 %7 %8 %9'
Table 8. Command line for running the translator program

4.2 Starting SMV

The command line (cmd_line2) run the SMV on the input file, and the result will be saved in "output.txt" file.

cmd_line2="smv -all "+tagname+".smv > output.txt"
Table 9. Command line for starting the verification tool

The content of *"output.txt"* will be checked to see if any of model checking results is false or not, in any condition it will give a feedback to the user.

4.3 Applying Tool Feedback

A single automated test is to assure either the implementation meet the specification or not. The output of executing the software tool would be one of three results: success, failure, or error. Success indicates that all specifications were true, which means that the implementation fulfills the specification and no errors were triggered. That is, of course, the desirable outcome. Failure and error indicates different problems with the implementation of completion of verification task. A failure result, for instance, means that one of the specifications returns false; Indication that the verification runs successfully, but it is not what the user expect. An error result means that an error was triggered somewhere in the verification stage, showing that the user verification was not running successfully. Since the errors will be saved in a log file, the user can review that text file to see which errors have been detected. And this also shows which specification is false and which one is true. So the developer is able to correct the mistakes and move forward on to the next.

The feedback to the user will be presented by showing view messages, such that at the last stages of running verification process, some messages windows will popup. These messages are based on analysing the output files. For instance if the requirements are not met, a popup message as shown in Fig.7 will be displayed to inform the user about the verification result. Following methods have been developed for displaying different popup messages.

1- “ShowVerificationFulfilledMessage(self, event)” is the method that will be called for showing a window to the user, the information in the window imply that the verification has been fulfilled.

2- “ShowSpecFulfillmentFalseMessage(self, event)” is the method that will be called for showing a window to the user if the verification has not been fulfilled. The user can also view the SMV file to check the errors.

3- “ ShowWarningMessage(self,event)” is the method that will be called for showing the warning messages to the user, if there is a warning inside of a file with “.warn” extension, it will be shown to user.

4.4 Automating Tool

The possibility of automatic verification provided in this project, a user only click on verify within the editor program, it calls the different programs in the background, the user will not be involved to run one by one programs that has been developed. Even the programs that may exist inside of different paths, they will be run automatically, moreover the output results also will be sent to specific files and folders.

The tool which has been built should automatically do all the tasks and produce the results. The result of verification process should be announced anyway to the user who has written the specification into the PlcOpenEditor.

In this automated tool, it is expected that after entering and editing the specification into the specification panel of the Editor, the developer clicks on Verify button under *Tools* menu of Editor. This action runs both the conversion and the verification process automatically. First, the translator is called for converting the specification to SMV format. Then after, the result of conversion set as input to the SMV verification tool, for checking and verifying if the verification is fulfilled or not. The result of verification is announced to the user. Moreover, if there is some error it is showed to the user too.

CHAPTER 5 – Conclusion

The following two sections describe the conclusion and the future works that is foreseen from doing this project.

5.1 Conclusion

The overall goal of this master thesis is to implement a software tool for specification and automatic verification of Programmable Logic Controller (PLC) program components. The tool can be used by control logic engineers who develop the implementation and write the corresponding specification for PLC components. In this report the frame work, the related work for specifying and verifying control logic component in industrial applications have been introduced. The proposed solution which emphasizes on Reusable Automation Components (RACs) has been described.

An industrial example showed how this implemented software tool can be applied on the application of RAC framework. In this case study, the implementation and the specification of RAC were stored in an .XML file. The RAC implementation and specification automatically translated into inputs to cadence SMV tool. In consequence, the tool automatically verified whether specification requirements are met or not. Finally, the notified test result was displayed.

The implementation of specification editor and implementation of translator part of tool have been explained, the integration of three programs: editor, translator and SMV tool was described.

Utilizing a software tool for designing RAC components, formal specifications and formal verification of PLC systems gives a possibility to analyze faster the problems that arise in such systems.

As in many conditions it is too time-consuming to test or simulate all different scenarios in which the components could be used [1]. Hence, the automated test and verification by utilizing the mentioned tool will be a great importance in developing the software components. The software tool assists the developers to find errors and inconsistencies within the components, making it easier to do modifications of the code.

5.2 Future Work

The lack of time caused to not develop all the required functionality that may facilitate the usage of software tool. Following describes those that have been left for future work.

There might be some errors when developer is writing specification. Checking spelling error and checking grammar errors at the earlier stages of writing specification will help developer to have a certain test. One example of spelling error would be a misspelled word like “Albays” instead of word “Always”. If the grammar is incorrect, it will be more convenient if programme warns the developer about the errors at the time of writing specification. Furthermore, developing the specification assistance and guidelines for better PLC programming has been suggested in [3].

In this project work, the input strings sequence of specification was expressed in “prefix” form. We think it will be a great potential to handle the complete specification language, including operators and not only functions.

The order of precedence for operators should be preserved. For instance, operators of equal precedence associate to the left, or parentheses might be applied for group expressions.

References

[1] O. Ljungkrantz, K. Åkesson, M. Fabian och C. Yuan. Formal Specification and Verification of Industrial Control Logic Components. IEEE Transactions on Automation Science and Engineering, 2010.

[2] K.L. McMillan. SMV Manual, Cadence Berkeley Labs, 25 April 2001.

[3] O. Ljungkrantz, K. Åkesson, M. Fabian. Formal Specification and Verification of components For Industrial Control Logic Programming. In Proceedings of the IEEE International Conference. On Automation Science and Engineering, Washington DC, USA, 2008, pp. 935-940.

[4] O. Ljungkrantz, and K. Åkesson. A study of industrial logic control programming using library Components. *In proceeding of the 3rd annual IEEE conference on automation science and Engineering, Scottsdale, AZ, USA, 2007.*

[5] O.Ljungkrantz. On Industerial Automation Software Components. Reusable Software Componenets for Logic Control Program Development. Automation research group in CTH, Technical report number: R006/2008, ISSN 1403-266X, 2008.

[6] E.Tisserant, L.Bessard, and M.de Sousa. An Open Source IEC 61131-3 Integrated Development Environment. Industrial Informatics, 2007 5th IEEE International Conference, volume: 1, Digital Object Identifier: 10.1109/INDIN.2007.4384753. Publication Year: 2007, Page(s): 183 – 187.

[7] International Standard IEC 61131-3. Programmable controllers, Part 1: general information. *International Electrotechnical Comission, 2003.*

[8] IEC 61131-3. 1993 Retrieved Dec. 2009. *Accessible at: www.plcopen.org.*

[9] K.L.McMillan.Getting Started with SMV. Cadence Berkeley Labs. March 23, 1999.

[10] G.Dunning. Introduction to programmable Logic Controllers-*Part I, EBook, 2002.*

[11] LOLITECH, Beremiz User Manual, *2008.*

[12] J. L. Martinez Lastra, L.Godinho, A.Lobov, and R. Tuokko. An IEC 61499 Application Generator for Scan-Based Industrial Controllers. *Industrial Informatics, 2005. INDIN '05. 2005-3rd IEEE International Conference on. 10/09/2005.*

[13] Bertrand Meyer, Applying “Design by Contract”, IEEE October 2009.

[14] K.L. McMillan, The SMV system, for SMV version. *2.5.4 -November 6, 2000.*

[15] B. Berard , M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen. System and software verification: Model checking techniques and tools Chapter 2. Springer-Verlag (2001).

[16] R.W.Lewis, Programming industrial control systems using IEC 1131-3*, Book.*

Appendixes

Appendix A: (Concepts, Terms and Definitions)

ST: stands for “structured text”. ST is a high level textual language, similar to Pascal and C, designed for programming automation processes and industrial control application. This language is mainly used to implement complex procedures that cannot be easily expressed with graphical languages, for instance it can be used to express the behaviour of FBs and programs [15].

CTL: stands for “computation tree logic”, it is branching-time logic, such that its model of time is a tree-like structure in which the future is not determined [14]. The logic CTL used by model checking tools, it serves to formally state properties concerned with the executions of a system [15]. For instance, it might be used in formal verification of software or hardware artifacts. [14]

LTL: stands for “Linear Temporal Logic”, it is one type of temporal logic, LTL is well known in the area of formal verification.

POU: stands for Program Organization Unit; a POU consists of a header and a body. The header declares the variables, the body contains the instructions. POU’s for instance can provide the possibility for re-use of software from Program (Macro level) to FB and Functions.

Overview of IEC 61131

IEC 61131 is the only global standard for industrial control programming. It harmonizes the way people design and operate industrial controls by standardizing the programming interface [8]. Such a standard programming interface, allows different people with different backgrounds and skills be able to create different element of a program during the stages of software lifecycle (specification, design, implementation, testing, installation and maintenance). The standard is used to build the internal organization of a program, via decomposition into logical elements, modularization and so on; each program is structured such a way to increase its re-Usability, reducing errors, increasing programming and user efficiency. [6, 8]

IEC 61499 is a standard developed by the International Electro technical Commission (IEC) for helping to achieve the requirements of present and future manufacturing world. [6]

PLC Open: PLC Open is the name of an organization in the field of Industrial Control, it was founded in 1992 and it is headquartered in Germany and Netherlands. The aim was to support the IEC 61131-3. The PLC Open activities are in creating a higher efficiency in application software development and also in minimizing the life-cycle costs of software. However PLC Open is based on standard available tools to which extensions are defined and later to which extensions will be defined. Motion Control Library, Safety, XML specification, Reusability Level and Conformity Level are sample result that the PLCopen made such solid contributions to the community, such that extending the hardware to be independence from the software code.
In order to support the use of international standard in the control programming, PLCopen association has developed a vendor independent format for resolving issues related to this field [8]. For this purpose, this association has several technical and promotional committees; one of

the activities of these communities in PLCopen is focused around IEC 61131-3(explain in following).
In order to have a valid xml document, the xml tag elements has been described in an external Schema. This XML schema has been dedicated to the PLCOpenEditor by Beremiz developing group. So each PLC program can be saved as an XML file.

PLCOPenEditor: is a Multi-platform automation IDE developed by Beremiz comany, LOLITech group. The PLCOpenEditor saves and loads PLC projects accordingly to PLCOpen TC6-XML Schema [11]. The view of PLCOpenEditor after opening a file (e.g. BufferCounter.xml) looks like to Fig. 6 (shown earlier in this report).

PLC Scan cycle: when a PLC is executing a ladder logic program, it tests the input modules first; also it stores the status of the input devices (e.g. High or Low) of the input devices. Consequently, the PLC scans the ladder logic program rung-by-rung from the top of the program to the bottom. Through scanning process, the PLC updates the statues of the instructions in each rung. After scanning the entire ladder logic program, the PLC copies the output instruction's memory statues (i.e. high or low) to update the output terminals. This period from the beginning of the input terminal examination to the end of the output terminal updates is called a Scan Cycle [10].

XML and XML Schemas: XML stands for Extensible Markup Language which is a general-purpose Specification for creating custom markup languages, that specifies lexical grammar and parsing requirements. An XML schema is a description of a type of XML document for the purpose of defining the legal building blocks of an XML document, it describes how should be the structure of an XML document. For instant, it defines elements and attributes that can appear in a document, defining data types for the elements and the attributes, defining their default and fixed values for the elements and the attributes, it defines how many child elements exist, which elements are the child elements.

Temporal logic: It is a form of logic tailored for statement and reasoning which involved the notation of order in time [15].

Appendix B: (TABLES)

The following table is taken from [ref.3, page5]. It is a Translation table that is used for expressing Linear Temporal Logic (LTL) properties.

Type of Operators	The specification term	The Translated term
teop	ALWAYS *p*	G *p*
functions	ALWAYS *(p)*	G *p*
teop	NEVER p	G ~p (= ~F p)
functions	NEVER (p)	G ~p (= ~F p)
teop	EVENTUALLY p	F p
functions	EVENTUALLY (p)	F p
teop	p WHILE_NOT q	(p U q) I G p
Functions	WHILE_NOT (p,q)	(p U q) I G p
blop	p -> q	p -> q
blop	P IMPLIES q	p -> q
blop	p ONLY_ IF q	p -> q
functions	IMPLIES(p,q)	p -> q
blop	p <-> q	p <-> q
functions	IF_AND_ONLY_IF(p,q)	p <-> q
blop	AND	&
blop	&	&
blop	OR	|
blop	NOT	~
blop	XOR	^
reop	>=	>=
reop	>	>
reop	<=	<=
reop	<	<
reop	< >	~ =
reop	->	->
reop	=	=
	AND D(A,B)	
	GE(A,B)	
	Variable suffixes	***SMV Expression***
	v_previous	v_previous: “same type as v”; Next(v_previous):=v;
	V_risingEdge	v&~v_previous
	V_fallingEdge	~v& v_previous
	Inoutvar_in	Input inoutvar_in
	Inoutvar_out	Output inoutvar_out
Boolean Operator(blop), Relational Operators(reop), Temporal Operator(teop)		