ST to MISRA-C Translator and Proposed Changes in IEC 61131-3 Standard

Ashutosh Kabra, *Member, IACSIT*, Gopinath Karmakar, *Member, IACSIT*, Jose Joseph, and R. K. patil

***Abstract*—Programmable Controllers (PLC) are being increasingly used in control systems including safety-critical systems like control application for Nuclear Power Plants. The reason lies in their ease of programming and configurability. They are programmed in languages complying to IEC 61131-3 standard. This work provides a necessary tool, ST to MISRA-C translator, for developing application program using textual language Structured Text (ST). During the development of the translator, the authors found some issues with ST language syntax and semantics given in IEC-61131-3 standard (second edition). These issues are discussed in this paper along with the other deficiencies identified by the other authors. We also proposed some changes in grammar of ST language for its unambiguous parsing.**

***Index Terms*—IEC 61131-3, PLC programming language, ST, POU, function block, program, translator.**

I. Introduction

Programmable Controllers (PLC) are used for control applications in various industries including the Nuclear Power plants for their ease of programming and configurability. In the past many vendors for PLC systems used their own programming languages, which were incompatible with others. To enhance reusing of components, compatibility and interoperability among different products, the IEC 61131 standard was introduced to unify the main different approaches. Third part of IEC 61131 standard [1] specifies the syntax and semantics of a unified suite of programming languages for PLC.

Structure Text (ST) is one of the textual languages defined in IEC 61131-3 standard, which closely resembles standard procedural languages and has syntax similar to Pascal. Unlike the graphical PLC programming language FBD (Function Block Diagram), ST gives more flexibility to the programmer in writing his/her own FB (Function Blocks) requiring complex logics and in writing applications involving loops, branches, state machines etc. In order to give the support of ST language and make the PLC suitable for safety critical applications, this translator translates the ST code into MISRA-C code. MISRA-C is a safe subset of C for use in safety-critical applications as per the guidelines of Motor Industry Software Reliability Association, UK [2], which has also been adopted in safety-critical applications other than automobile industries.

During the development of the translator, we found some and a modified grammar has been used for this development. Some deficiencies, which have already been identified by other authors, are also discussed in context while presenting our own findings. This translator also takes care of program execution and/or run-time issues like task creation, associating a program to task and communication between different programs. Translated MISRA-C code can be compiled using Open Source GNU C Compiler (gcc) for binary code generation, which will be subsequently incorporated on target hardware. Additionally, an RTOS interface has been developed to run this translated code on any particular RTOS.

The paper is organized as follows. Section II gives brief introduction to IEC 61131-3. In section III, the issues in the ST syntax and semantics have been analyzed and the proposed changes in grammar have been discussed. Section IV elaborates the translation scheme along with RTOS interface to make this translation adaptable to any custom RTOS. Section V discuses our conclusion and future direction for this work.

II. Introduction to IEC 61131-3

IEC 61131-3 specifies syntax and semantics of programming languages for Programmable Controllers. It defines not only the PLC programming languages but also a set of common elements viz. data types, type declarations, variable declarations, program organization units (POUs), configuration elements etc. between all languages. The IEC 61131-3 software model consists of configuration elements viz. configuration, resource, task, global variables and instance-specific initializations, which support the installation of PLC programs into Programmable Controller systems.

At the highest level, the configuration contains global variables and resources. A resource is a kind of interface between the PLC hardware and a virtual machine that is able to execute IEC programs. A resource can be associated with a processor. The resource contains tasks and associated programs. The programs and/or function blocks are associated with tasks for their execution. There are three POUs namely functions, function blocks and programs. Functions have similar semantics to those in traditional procedural languages, which when executed returns a single output value. Function blocks are like classes in object-oriented languages with the limitation of having a single public member function. Function blocks are instantiated as variables in a program or in another function block. All the values of output variables and internal variables of function blocks persist from one execution to the

Manuscript received April 15, 2012; revised June 3, 2012.

The Authors are with the Bhabha Atomic Research Centre, Mumbai, 400085 India (e-mail: {kabra,gkarma,jjoseph,rkpati}@ barc.gov.in).

next. Programs are similar to function blocks but can be instantiated only inside a configuration and not inside other programs or function blocks. All the POUs may be programmed in any one of the PLC languages. Directly represented variables are located at specific addresses in physical I/O or memory and are used to access field inputs and outputs.

III. Issues with ST Language and Proposed Changes

Several authors, over the years, have published works commenting on the deficiencies in the syntax and semantics of IEC 61131-3 languages mainly textual languages ST and IL (Instruction List). De Sousa and Carvalho [3] identified the deficiencies like non-use of name–spaces, which does not allow using variable name identical with any derived function name or using global variable name identical with any program name. They also pointed out the conflict where several constructs that include 'NOT' or 'MOD' keyword may be interpreted either as a function call or as an execution of an operator. I. Plaza and C. Medrano [4] pointed out the fact that although the standard allows the declaration of global variable in programs, the formal syntax definition in standard does not allow the same. It was also identified that there are two possible ways of parsing the '1' and '0' literals either as a boolean literal or as an integer. De Sousa [5] pointed out that there is an error when the standard describes VAR_EXTERN as an additional feature of program declarations implying that it is not available in function block declaration. He also pointed out the issues related to the possibility of program running inside one task and a function block instantiated within the same program but associated with another task, which is allowed in the IEC 61131-3 standard. Further, he identified that there are the two possible ways of parsing the variable declaration of STRING or WSTRING type; either as a simple type declaration or as a string type declaration resulting in conflict during parsing.

During the development of translator, we have found a good number of unresolved issues with ST language syntax and semantics given in IEC-61131-3 standard second edition [1], not pointed out by any other author. In this paper, only the new found issues are discussed in following sub-sections along with the proposed changes to resolve these issues.

A. Specification of Bit String Literal

Under section B.1.2.1 of the IEC-61131-3 standard, the formal specification of *integer_literal* and *bit_string_literal* are defined as follows.

integer_literal: = [integer_type_name '#'] (signed_integer | binary_integer | octal_integer | hex_integer)

signed_integer: = [+|-] integer

bit_string_literal:=**[**('BYTE' | 'WORD' | 'DWORD' | 'LWORD') '#'] **(** unsigned_integer | binary_integer | octal_integer | hex_integer)

There must be 'BYTE#', 'WORD#', 'DWORD#' or 'LWORD#' as prefix for a *bit_string_literal*. Otherwise, any integer may be reduced to *integer_literal* or *bit_string_literal* and that will cause a reduce-reduce conflict during parsing. E.g. 107 may be reduced in two ways as following.

107 → integer → bit_string_literal

107 → integer → signed_integer → integer_literal

To resolve this issue, we have made these prefix ('BYTE#', 'WORD#', 'DWORD#' or 'LWORD#') compulsory in the rule of *bit_string_literal*. Also, in the above specification of *bit_string_literal*, a symbol *unsigned_integer* has been used, which is not defined anywhere else. In ST grammar, *integer* represents the unsigned integer definition. Therefore it should be *integer* instead of *unsigned_integer* in above rule as suggested in [4]. So above rule should become

bit_string_literal: = ('BYTE' | 'WORD' | 'DWORD' | 'LWORD') '#' (integer | binary_integer | octal_integer | hex_integer)

B. Optional Data Type for Global Variable Declaration

The formal specification of *global_var_decl* makes it optional to declare the data type of global variable at the time of declaration as defined in section B.1.4.3 of the IEC-61131-3 standard.

global_var_decl ::= global_var_spec ':'
[located_var_spec_init | function_block_type_name]

Therefore there is a possibility of global variable declaration without declaring its data type. Instead of optional, it must be compulsory to declare the data type of global variable at the time of declaration. It will also make global variable declaration in consistence with all other variable declaration. So above rule should become

global_var_decl: = global_var_spec ':'
(located_var_spec_init | function_block_type_name)

C. Interval of Periodic Task

Under section B.1.7 of the IEC-61131-3 standard, the formal specification of *task_initialization* is defined as follows.

task_initialization: = '(' ['SINGLE' ':=' data_source ','] ['INTERVAL' ':=' data_source ','] 'PRIORITY' ':=' integer ')'

data_source: = constant | global_var_reference | program_output_reference | direct_variable

According to formal specification of *task_initialization*, interval of a periodic task can be the value of program output variable. Therefore, in a valid configuration, there may be a task having program output variable as its interval and this program is associated with the same task. In this case, program may be executed by this task only and task can be initialized by program output variable. However, program output variable cannot have a meaningful value unless the program has been executed at least once. This will be a deadlock situation. To resolve this issue, we changed the formal specification of *task_initialization* as follows. According to this changed formal specification, a task can have *duration* as its interval.

task_initialization: = '(' ['SINGLE' ':=' data_source ','] ['INTERVAL' ':=' duration ','] 'PRIORITY' ':=' integer ')'

It was suggested in [5] to change the formal specification of *task_initialization* as follows.

task_initialization: = '(' ['SINGLE' ':=' data_source ','] ['INTERVAL' ':=' time_literal ','] 'PRIORITY' ':='integer ')'

But, using *time_literal* as interval of task can cause a problem, where TIME OF DAY or DATE may also be accepted as a valid interval of a task in ST configuration.

D. Specification of Primary Expression

Under section B.3.1 of the IEC-61131-3 standard, the formal specification of *primary_expression, enumerated_value* and *variable* are defined as follows.

primary_expression: = constant | enumerated_value | variable | '(' expression ')' | function_name '(' param_assignment {',' param_assignment} ')'

enumerated_value: = [enumerated_type_name '#'] identifier

variable: = direct_variable | symbolic_variable

symbolic_variable:= variable_name | multi_element_variable

variable_name: = identifier

Any identifier can be parsed as *primary_expression* in following two ways and it will cause reduce-reduce conflict.

identifier → variable_name → symbolic_variable → variable → primary_expression

identifier → enumerated_value → primary_expression

To resolve this issue, we have changed the above specification as follows.

primary_expression:= constant | enumerated_value_only | variable | '(' expression ')' | function_name '(' param_assignment {',' param_assignment} ')'

enumerated_value_only:= enumerated_type_name '#' identifier

enumerated_value:= enumerated_value_only | identifier

E. Declaration of Temporary Variables in Function Blocks

According to the declaration and usage of *function blocks* and *programs*, it is not clear from IEC 61131-3 standard that whether a function block type declaration may include VAR_TEMP construct (for temporary variable). In section 2.5.3, VAR_TEMP are described as additional features of program in comparison of function block. It implies that function block declaration does not allow using this construct. But, in section 2.5.2.2, this construct is described as a feature of function block declaration and usage. Further, formal specification of *function_block_declaration* is defined in section B.1.5.2 of IEC 61131-3 standard as follows.

function_block_declaration: = 'FUNCTION_BLOCK' derived_function_block_name {io_var_declarations | other_var_declarations } function_block_body 'END_FUNCTION_BLOCK'

other_var_declarations: = external_var_declarations | var_declarations | retentive_var_declarations | non_retentive_var_declarations | temp_var_decls | incompl_located_var_declarations

temp_var_decls: = 'VAR_TEMP' temp_var_decl ';' {temp_var_decl ';'} 'END_VAR'

From the above formal specifications, it is clear that the VAR_TEMP construct may be used inside a function block declaration. To resolve this issue, we suggested that the specification needs to be modified to state that VAR_TEMP construct may also be used inside a function block declaration.

F. Miscellaneous Issues

1) Issue 1

Under section B.1.3.3 of the IEC-61131-3 standard, the formal specification of *simple_type_name, subrange_type_name, enumerated_type_name, array_type_name, structure_type_name and string_type_name are defined as follows.*

derived_type_name: = single_element_type_name | array_type_name | structure_type_name | string_type_name

single_element_type_name: = simple_type_name | subrange_type_name | enumerated_type_name

simple_type_name: = identifier subrange_type_name: = identifier

enumerated_type_name: = identifier array_type_name: = identifier

structure_type_name: = identifier string_type_name: = identifier

So there will be reduce-reduce conflict on identifier. To resolve this issue, we have deleted the formal specification of *simple_type_name, subrange_type_name, enumerated_type_name, array_type_name, structure_type_name* and *string_type_name* and. changed the formal specification of *derived_type_name* as follows.

```
derived_type_name: = identifier
```

Also, we have replaced all references of *simple_type_name, subrange_type_name, enumerated_type_name, array_type_name, structure_type_name* and *string_type_name* in other specification rules by *derived_type_name*.

2) Issue 2

Under section B.1.7 of the IEC-61131-3 standard, the formal specification of *configuration_name, resource_name, task_name, program_name* and *fb_name* are defined as follows.

configuration_name: = identifier resource_name:= identifier

task_name:= identifier

program_name:= identifier

fb_name := identifier

So there will be reduce-reduce conflict on identifier. To resolve this issue, we have deleted the formal specification of *configuration_name, resource_name, task_name, program_name* and *fb_name* and introduced the formal specification of *element_name* as follows.

```
element_name:= identifier
```

Also, we have replaced all references of *configuration_name, resource_name, task_name, program_name* and *fb_name* in allother specification rules by *element_name.*

G. Changes in ST Grammar for LALR Parsing

LALR parser is most commonly used in practice. LALR parser looks ahead for only one input symbol in making parsing decision. But in case of some rules in ST grammar, it may be required to look ahead for more than one input symbol during parsing of valid ST code. A set of rules, which is defined in section B.3.2.3 of IEC 61131-3 standard is given below, which cannot be handled by any LALR parser.

case_statement: = 'CASE' expression 'OF' case_element {case_element} ['ELSE' statement_list] 'END_CASE'

case_element: = case_list ':'statement_list

case_list: = case_list_element {',' case_list_element}

case_list_element: = subrange | signed_integer | enumerated_value

enumerated_value_only: = enumerated_type_name '#'

identifier

enumerated_value:= enumerated_value_only | identifier

For example, parsing of following case statements may lead to a shift-reduce conflict on MAX identifier.

CASE XYZ OF 2: *A*: = *B* + *C*; MAX: *D*: = *E* + *F*; END_CASE;

CASE XYZ OF 2: *A*: = *B* + *C*; MAX: = *E* + *F*; 5: *F*: = *G* + *H*; END_CASE;

In this case, there will be shift-reduce conflict on identifier MAX because on the basis of single token, parser cannot decide whether it is starting of *case_list_element* (as *enumerated_value*) or *assignment_statement* (*variable* as first token of rule). To make it possible to parse these types of rules with LALR parser, we have to change the grammar. To resolve the above issue, there are two options.

First option is to introduce BREAK as a keyword, which will be used as an end-mark of a case_element. In that case, formal specification of *case_element* shall be changed as follows.

case_element: = case_list ':' statement_list 'BREAK'

Second option is to remove *enumerated_value* from the specification rule of *case_list_element.* In that case, formal specification of *case_list_element* shall be changed as follows.

case_list_element: = subrange|signed_integer

We preferred the second option of limiting the syntax rule instead of introducing a new syntax definition.

IV. Translation Scheme

This Translator executes in four stages: lexical analyzer, parser (syntax analyzer), semantics analyzer and code generator [6]. Open source tools LEX [7] and YACC [8] have been used for implementation of parser for ST language. Semantic analyzer has been developed with limited functionality (type checking, array bound checking and range checking only) as other semantic errors in the ST source code will also result in semantic errors in the translated MISRA-C code that will be caught by the gcc compiler (e.g., use of undefined variable, number of parameters mismatch in function call etc.). Translation scheme (in terms of Syntax Directed Translation Rules) has been designed and implemented in C. The highlights of this implementation are as follows.

A. Data Types not Supported by MISRA-C

Some elementary data types of IEC 61131-3 standard such as TIME_OF_DAY, DATE are not supported directly by MISRA-C. These data types have been implemented as specific C structure. E.g. translation of DATE data type is shown in Fig. 1.

ST code	MISRA-C Code

DATE D1;	typedef struct DATE{
	UI__16 year;
	UI__8 month;
	UI__8 day;
	}
	C__DATE;
	C__DATE D1;

Fig. 1. Translation of DATE data type.

B. Straightforward Translation

Translation of some ST constructs like IF-ELSE, FOR, WHILE etc. is straightforward as these constructs are also available in MISRA-C. Translation of FOR construct is shown in Fig. 2, as an example.

ST code	MISRA-C Code

FOR I:=1 TO 5 DO
FOR J:=1 TO 10 BY 2
DO
A := -B+37*(-2);
END_FOR;
SUM:=SUM+I;
END_FOR;
	for(I=1;I<=5;I++)
	{
	for(J=1;J<=10;J=J+ 2)
	{
	A = -B+37*(-2);
	}
	SUM = SUM + I;
	}

Fig. 2. Translation of FOR construct.

C. Mapping of Directly Represented Variables (for field I/O Access) in ST to C code

Arrays for field I/O are declared explicitly in resource declaration. Size and type of these arrays shall be according to number and type of Input-output cards. These arrays are updated periodically (By invoking a separate task having highest priority and period equal to scan time requirement of I/O). Mapping of directly represented variables in ST program to the elements of these arrays are shown using some examples in Fig. 3.

ST constructs	Mapping in MISRA-C
%IX1.1	DV__I_X[1][1]
%IX1.4	DV__I_X[1][4]
%QW2.5	DV__Q_W[2][5]
%QX5.2	DV__Q_X[5][2]

Fig. 3. Mapping of directly represented variables.

D. Function Block and Program Type Constructs

Both of these POUs are mapped onto a combination of structure of POU type, initialization function and definition function. Output variable and internal variable of POU are made member elements of structure data type. Initialization function initializes all member elements of structure data type. POU body (program or function block body) has been mapped to definition function. Mapping of a ST program example is shown in Fig. 4. Standard function blocks have been incorporated in translated program by including corresponding data structure, initialization function and definition function for every standard function block explicitly.

E. Configuration Construct

Task initialization has been mapped on to corresponding task creation, while program association with task has been translated by invoking program declaration function in that task, with which this program is associated.

F. RTOS Interface:

The runtime programs are associated to tasks, which run under a RTOS. We have developed a configurable RTOS interface for POSIX compliant RTOS and μC/OS-II, to run

the translated code on any hardware platform supporting these RTOSes. For any custom RTOS, this interface provides support for changes in RTOS specific task management utilities, task synchronization utilities and time-tick utility etc., which can be done easily.

V. Conclusion and Future Work

Present work resolved the conflicting issues in the grammar of ST language and a Context Free Grammar (CFG) for ST has been produced to facilitate the translation of application program (in ST) to MISRA-C. Formal verification of this tool is being carried out for its field-worthiness in safety critical applications. A tool that will verify the application program itself against its safety requirement properties will complement this work towards a total development solution for safety-critical PLC. For this purpose, a translator that will translate ST (application program) to a synchronous language like LUSTRE or ESTEREL needs to be developed.

ST code

```
PROGRAM PUMP_CONTROL
  VAR_INPUT
    P_START_C : BOOL;
    P_STOP_C : BOOL;
    P_DIS_PR : INT;
  END_VAR
  VAR_OUTPUT
    P_ON_OFF : BOOL;
  END_VAR
  VAR_EXTERNAL
    P_PSH : INT;
  END_VAR
  VAR
    RS1 : RS;
    GE7_OUT : BOOL;
    OR8_OUT : BOOL;
  END_VAR

  GE7_OUT:= GE(P_DIS_PR, P_PSH);
  OR8_OUT:= OR(P_STOP_C, GE7_OUT);
  RS1(P_START_C, OR8_OUT);
  P_ON_OFF := RS1.Q1;
END_PROGRAM
```

MISRA-C Code

```
typedef struct
{
    U__1 P_ON_OFF;
    /*typedef uint8_t  U__1;*/
    RS RS1;
    /*RS is data structure for RS standard FB*/
    U__1 GE7_OUT;
    U__1 OR8_OUT;
} PUMP_CONTROL;

PUMP_CONTROL PUMP_CONTROL__INIT (PUMP_CONTROL
PUMP_CONTROL__INIT )
{
    PUMP_CONTROL__INIT.P_ON_OFF = 0 ;
    PUMP_CONTROL__INIT.RS1 = RS__INIT(PUMP_CONTROL__INIT.RS1);
    PUMP_CONTROL__INIT.GE7_OUT = 0 ;
    PUMP_CONTROL__INIT.OR8_OUT = 0 ;
    return PUMP_CONTROL__INIT;
}

PUMP_CONTROL PUMP_CONTROL__BODY(U__1 P_START_C ,U__1
P_STOP_C,SI__16 P_DIS_PR,PUMP_CONTROL PUMP_CONTROL__DATA)
{
    extern SI__16 P_PSH ;
    /*typedef uint16_t SI__16;*/
    PUMP_CONTROL__DATA.GE7_OUT = ((P_DIS_PR>= P_PSH));
    PUMP_CONTROL__DATA.OR8_OUT =
              (P_STOP_C||PUMP_CONTROL__DATA.GE7_OUT);
    PUMP_CONTROL__DATA.RS1 =
              RS__BODY(P_START_C,PUMP_CONTROL__DATA.OR8_OUT,
              PUMP_CONTROL__DATA.RS1);
    PUMP_CONTROL__DATA.P_ON_OFF=
                  PUMP_CONTROL__DATA.RS1.Q1;
    return PUMP_CONTROL__DATA;
}
```

Fig. 4. Mapping of program.

Acknowledgment

The authors thankfully acknowledge Shri G.P. Srivastava, Director, Electronics and Instrumentation Group, BARC, Shri B.B.Biswas, Head, RCnD, BARC for giving us the opportunity to work on the ST to MISRA-C Translator development. The authors also thankfully acknowledge Shri Anup Bhattacharjee, RCnD, BARC for his thoughtful comments on this paper.

References

[1] *International Standard IEC 61131-3, Programmable Controllers-Part 3: Programming Languages*, 2nd ed., International Electro technical Commission, Geneva, 2003.

[2] *MISRA-C Guidelines for the Use of the C Language in Critical Systems*, Motor Industry Software Reliability Association, UK, 2004.

[3] M. de Sousa and A. Carvalho, "An IEC 61131-3 Compiler for the MatPLC," in *Proc. 9th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 03)*, Portugal, 2003, pp. 485 – 490.

[4] I. Plaza, C. Medrano, and A. Blesa, "Analysis and implementation of the IEC 61131-3 software model under POSIX real-time operating systems," *Microprocessor and Microsystems*, vol. 30, pp. 497–508, Dec. 2006.

[5] M. de Sousa, "Proposed corrections to IEC 61131-3 standard," *Computer Standard and Interface*, vol. 32, pp. 312-320, Oct. 2010.

[6] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and tools*, Pearson Education Pte. Ltd., 2003, ch. 1, pp. 4-15.

[7] M. E. Lesk and E. Schmidt, "Lex: A Lexical Analyzer Generator," *Comp. Sci. Tech. Rep. No. 39*, Bell Laboratories, Murray Hill, New Jercy, 1975.

[8] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," *CS TR 32*, Bell Labs, 1978.

Ashutosh Kabra received his Bachelor's degree in Computer Science and Engineering from Rajasthan University, India in 2005 and Master's degree in Computer Engineering with specialization in Nuclear Engineering from Homi Bhabha National Institute, India in 2009. He joined Bhabha Atomic Research Centre (BARC), Mumbai, India in 2007 as a Scientific Officer. He has been working in field of Embedded System Development for Safety/Safety-related applications since last four years. His research interest is compiler development, hard real-time systems and embedded software development for safety systems of Nuclear Power Plants.

Kabra is a member of IACSIT (International Association of Computer Science and Information Technology)

Gopinath Karmakar received his bachelor's degree in Electrical Engineering from Jalpaiguri Government Engineering College, India in 1985. He joined Bhabha Atomic Research Centre (BARC), Mumbai, India after graduating from BARC Training School in 1987 and at present holding the position of Scientific Officer-G. He has been engaged in various embedded system development projects for Indian nuclear plants since last 25 years. His research interest is hard real-time systems and embedded software development for safety critical systems, especially of Nuclear Power Plants.

Karmakar is a professional member of ACM, IEEE and IACSIT.

Jose Joseph did his BSc. Engineering (Electrical) from Kerala University, India in 1977. He joined Bhabha Atomic Research Centre (BARC), Mumbai, India, after graduating from BARC Training School in 1978. Presently, he is the Head, Control System Engineering Section, RCnD, BARC. His expertise is in the field of control system and control room design for Nuclear Reactor as well as design safety review of nuclear reactors.

R. K. Patil received his bachelor's degree in Electrical Engineering from Bhopal University, India in 1971. He joined Bhabha Atomic Research Centre (BARC), Mumbai, India, after graduating from BARC Training School in 1972. At present, he is holding the position of Associate Director (Control), of Electronics and Instrumentation Group, BARC, India. He has 30 years of RandD experience in the field of development of control and monitoring system for power and research reactors, man-machine interaction design and simulators as well as development of radiation and process related sensors and detectors.