Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science
Formal System Analysis Research Group

2IM91 – Master's Thesis

# Verification of PLC code used at CERN

Petra van den Helder, BSc.

*Supervisors:*
dr.ir. T.A.C. Willemse
dr. D. Bošnački

May 18, 2016

## Abstract

At CERN over 1000 PLCs (Programmable Logic Controllers) are used for automation in, among others, the Large Hadron Collider experiments. To ensure safe and correct functionality of the PLCs, model checking is used. In general model checking is applied to both hardware and software. For different applications, different model checking tools are used. However, there are not yet any model checking tools specifically for verification of PLCs. In this thesis a number of verification tools have been considered for the verification of PLC programs written in SCL (Structured Control Language). We explored verification tools that work on models - Spin, NuSMV, and nuXmv, as well as verification tools that work on code - CMBC, K-Inductor, CBMC Incremental, 2LS, CPA-checker, and SATABS. For all tools a translation from the SCL code and the PLC semantics to the input language has been made and some reductions are considered to increase the efficiency of the verifications. We have experimented with example programs and we have done a realistic case study. The results show that some of the verification tools on models gave better results on the example programs, but for the case study the results from the verification tools on code were better. Despite the fact that they were not able to verify all given properties, the tools CBMC Incremental and 2LS gave the most promising results.

# Preface

As a graduation project for my the master program Computer Science and Engineering at Eindhoven University of Technology I have done research on verification of PLC code used at CERN. This master thesis describes the results of this research. The project was performed internally at the Formal System Analysis group of the Mathematics and Computer Science department of Eindhoven University of Technology.

First of all I would like to thank my supervisors Tim Willemse and Dragan Bošnački for helping me getting started with my graduation project and for all the feedback and the meetings, which were most of the time very useful. I also want to thank Daniel Darvas and Borja Fernández Adiego for providing me with information from CERN and for helping me understand it. Working with all sorts of tools never comes without problem. Therefore I would like to thank Peter Schrammel for helping me with the tools CBMC Incremental and 2LS. Furthermore I would like to thank Rianne Broere for the last feedback on my master thesis, mainly on the English language.

I would like to thank Tessa, Hugo, Roel, Femke, Myrthe, and Sanne for making me feel at home at the university, I always had a nice time during the lunch breaks and the tea breaks. I would also like to thank Mahmoud, Sarmen, and Fei for making me feel comfortable in the office. Lastly I would like to thank my boyfriend Peter and my parents for always supporting me during my project, especially when I had a hard time.

Petra van den Helder
Eindhoven, May 2016

# Contents

# 1 Introduction

Automation is used for many reasons; not only to reduce the human work or to decrease the process time, but also because machines can do things that humans cannot and at places that are not reachable by humans.

To control machines, Programmable Logic Controllers (PLCs) [9] can be used. A PLC is a robust computer that can work in extreme environments. It reads input from sensors and writes output to machines such that they can be controlled. At CERN over 1000 PLCs are, among others, used for the following systems:

- The LHC cryogenic control systems, which uses around 100 PLCs,
- Many cooling and ventilation control systems,
- Gas control systems for the LHC particle detectors,
- Vacuum control systems for the ISOLDE particle accelerator.

To make sure that the PLCs control the machine correctly and to guarantee safety, model checking [14, 30, 2, 16] can be used. Verification tools use model checking to check a certain property of a program. With model checking the given properties are checked for a given program. If a verification tool refutes a safety property for a given program it gives a counterexample. Usually this is an execution trace of the program that results in a state where the property does not hold. The verification tools can also prove properties for a given program by checking the property for every possible execution or by proving it mathematically.

Problem description This thesis discusses the problem of verification of PLC programs written in SCL with existing verification tools. This includes a translation of the SCL code and the PLC semantics into the input language of existing tools. For determining which tools can be used for this purpose the verification results, verification times, and the counterexamples given by the tools will be taken into account.

Approach In this thesis we will look at existing model checking tools which can be used to verify PLC code written with the programming language SCL (Structured Control Language). A translation from the SCL code and the semantics of the PLC to the input languages of the model checking tools will be made. Our research focuses on two groups of tools: verification tools on models and verification tools on C code. The first group consists of Spin, NuSMV, and nuXmv. For this group we have to make a model from the SCL code before we can run the verifications. The second group consists of CBMC, K-Inductor, CBMC Incremental, 2LS, CPA-checker, and SATABS. For this group such a model is not needed, but a translation to C that covers both the SCL code and the PLC semantics is. A number of experiments will be done using the tools from both groups and the results will be compared. A large case study involving an SCL program of a PLC that is used at CERN [1] will be done with the most promising tools.

Contributions With this thesis we have contributed with the following aspects:

- Different model checking tools have been compared for verification of PLC code. The comparisons are done on ease of translation, verification results, running times, and giving counterexamples. To the best of our knowledge, a comparison with this many tools for PLCs and SCL has not been done before;

- Translations are made from the PLC language SCL to different input languages. The input languages are PROMELA, SMV, and C;
- Parallel assignments are used to optimize the SMV translation;
- Limitations of different tools that use k-induction were identified, by creating an example that is provable by k-induction but could no be proved by the tools.

Results We have found that for the large case study, the verification tools on code gave faster results than the verification tools on models. CBMC Incremental is a fast and reliable verification tool that can handle the semantics of the PLCs at CERN. A small disappointment is the fact that none of our tools are able to verify or refute all of the given properties. We have seen that multiple tools are unable to verify the same properties. This shows that further research and further development in model checking is needed for verification of all properties.

Related work The topic of PLC program verification has been investigated before. Pavlovic et al. [29] made an automated verifier for the PLC programming language IL. This verification uses the model checking tool NuSMV that uses SMV as input language. Rausch and Krogh [31] also used the SMV language, but they used PLC programs that use Relay Ladder Logic. Park et al. [28] used simulation to verify PLC code. One of the differences with our work is the input language. We will be looking at PLC programs written in SCL (Structured Control Language). Some research has been done on verification of an input language similar to SCL. Meulen [27] used propositional logic to verify PLC programs in STL language and Barbosa and Déharbe [3] also verified programs in the STL language, they used the B method. Finally Fernández Adiego [23] has used an intermediate model for verification of STL programs at CERN with, among others, NuSMV. We will discuss this technique later in this thesis.

Outline First the PLC semantics and the different aspects of SCL code are described in Section 2. Then we look at the problem description as well as previous work from CERN and the approach for our research in Section 3. In Section 4 and 5 the different tools, the translation of the code, and the experiments as well as the results for the example programs are described. In Section 6 we describe a case study on which we have done experiments with the most promising tools. The thesis concludes in Section 7, where additionally a list with ideas for future research is presented. The appendix contains all used programs.

# 2 Preliminaries

This section contains a short introduction about PLCs and the language SCL which is used to program PLCs.

## 2.1 Programmable Logic Controller (PLC)

A PLC is an industrial computer control system that continuously monitors the state of input devices and makes decisions based upon a custom program to control the state of output devices. The first PLCs were made with logical ports, hence the name. Nowadays PLCs are made with microprocessors, which makes them easier to program and to use. PLCs are mainly used for automation.

A PLC can be designed with modules that have analog and digital input and output ports. For analog input and output ports the module transforms the signal from an analog signal to a digital signal and vice versa. The input ports can be connected to sensors and switches to control the system. The output ports can be connected to machinery or screens.

A PLC executes a program in a cyclic manner. A cycle starts with the PLC reading all values from the input ports, after which it executes the whole program code. After the execution it writes all values to the output ports before beginning the cycle again. By execution in this way, all outputs are written at the end of the program, ensuring that output ports can only be changed once in a cycle.

We are interested in PLCs used at CERN, which are Siemens Step 7 PLCs. Step 7 is the software for programming these PLCs.

## 2.2 Structured Control Language (SCL)

The Step 7 software can be used for programming with various programming languages. We will focus on SCL [33], since this is the language used at CERN. SCL is a high level language that it is based on PASCAL, which makes the language suitable for programming complex problems. A program in SCL can call programs in other PLC languages and programs in other PLC languages can call programs in SCL.

The Step 7 software allows structuring of a program by using blocks. We will give a short description of the supported blocks. Examples of these blocks can be found later in this section.

- Organization block (OB) determines the structure of the program. OBs are predefined in the Step 7 software. The organization block for normal program execution on PLCs is determined in OB1. This block determines the cyclic semantics of the PLCs as explained in the previous section. We will only look at programs that use this organization block;
- Functions correspond to functions we know from programming;
- Function blocks are functions which can also store data between function calls;
- Data blocks are used for storing and sharing data;
- User-defined data types are used to define complex data types.

Additionally, there are some functions integrated in the Step 7 software. These are typically functions that are widely used in SCL programming. They are part of the operating system and are not loaded as part of the program.

The programs we use in the experiments start with a function block, to which we will refer as the main function block. The programs can have calls to other function blocks, functions, data blocks and data types.

All functions and function blocks in SCL can have variables of different types. Input variables get values from the calling block. For the topmost function block, the input variables get values from the input ports. Output variables are used to return values to the calling block. For the topmost function block, the output variables contain the values that are sent to the output ports. In-output variables are a combination of input variables and output variables, these variables get values from the calling block, or input ports, and return values to the calling block, or sent them to the output ports. Static variables can be used within the blocks. Function blocks have memory, therefore they can keep the values of static variables after the program has returned to the calling block. This also makes it possible for these variables to have an initial value. A function has no memory, therefore static variables in a function have no initial values and do not keep their values after the program has returned to the calling block.

SCL uses control statements to take care of selective instructions and repetition instructions. The control statements we use are: `IF`, `ELSEIF`, `ELSE`, and `WHILE`. SCL also supports case distinction, loops, and jump statements. For conditional expressions the standard boolean operators can be used.

The predefined data types we use are: `BOOL`, `INT`, `UINT`, `WORD`, `ARRAY`, `STRUCT`, `TIME`, and `REAL`. Other predefined data types are dates, chars, timers and doubles. The size and value ranges of the data types we use are shown in the Table 1. The data types `ARRAY` and `STRUCT` do not have a specified size, because the size varies per specification.

| Type | Bits | Values |
|---|---|---|
| `BOOL` | 1 | `true`, `false` |
| `INT` | 16 | Signed Integer |
| `UINT` | 16 | Unsigned Integer |
| `WORD` | 16 | Bit combinations |
| `TIME` | 32 | -24d 20h 31m 23s 647ms to |
| | | 24d 20h 31m 23s 647ms |
| `REAL` | 32 | Floating Point |

Tab. 1: Data types in SCL

The SCL code is executed cyclically by the PLC. The execution consists of three phases and is part of a non-terminating loop. In the first phase the input is read: all input variables will be read from the input ports of the PLC. The next phase consists of the execution of the code. The last phase consists of writing the output: all output variables will now be sent to the output ports of the PLC. After the last phase the execution continues again with the first phase.

There are a couple of rules for the structure of an SCL program. Called blocks must precede the calling blocks. In a block, the variables must be defined first. Each variable type gets a subsection, which should contain all variables of that type. There is no fixed order for the subsections.

Line comments in SCL are introduced by '`//`' and block comments are introduced by '`(*`' and terminated by '`*)`'. The language is not case-sensitive. For clarity, capital letters will be used for all reserved words.

**Examples** SCL code of a running example is given in Listing 2.1. The example was taken from [20].

```
// The main program
FUNCTION_BLOCK SimpleExample
VAR_INPUT
    error : BOOL;            // not used
    toMode1 : BOOL;          // request to switch to mode1
    toMode2 : BOOL;          // request to switch to mode2
    toMode3 : BOOL;          // request to switch to mode3
    mode3Forbidden : BOOL; // if it is true, it is forbidden to be in
        mode3
END_VAR

VAR
    mode1 : BOOL;            // true if the block is in mode1
    mode2 : BOOL;            // true if the block is in mode2
    mode3 : BOOL;            // true if the block is in mode3
END_VAR
VAR_OUTPUT
    mode : INT;
END_VAR

BEGIN
    // Operation mode handling
    IF NOT mode1 AND NOT mode2 AND NOT mode3 THEN
        mode1 := TRUE;
    END_IF;

    IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
        mode1 := TRUE;
    END_IF;
    IF toMode2 THEN
        mode2 := TRUE;
    ELSIF toMode3 THEN
        mode3 := TRUE;
    END_IF;

    IF mode1 THEN
        mode := 1;
    ELSIF mode2 THEN
        mode := 2;
    ELSIF mode3 THEN
        mode := 3;
    ELSE
        mode := 0;
    END_IF;
END_FUNCTION_BLOCK
```

Listing 2.1: Running Example in SCL

The program starts by defining the main function block at line 2. The input variables are created in the section `VAR_INPUT` at lines 3-9, followed by the static variables `VAR` at lines 11-15 and an output variable `VAR_OUTPUT` at lines 16-18. Next the program section begins with `BEGIN` at line 20, it consists of assignments and `IF`, `ELSEIF`, and `ELSE` statements. At the end the function block is closed by the `END_FUNCTION_BLOCK` statement at line 44.

The execution of this program will be as follows: In the first phase the input variables are read from the input ports (these are the variables from lines 4 to 8). In the second phase the code from lines 21 up to 43 will be executed. In the third phase the value of the output variable `mode` will be written to the output port. After this last phase the execution will go back to the first phase and will repeat this indefinitely.

An example for a function in SCL is given in Listing 2.2.

```
1  FUNCTION R_EDGE : BOOL
2          VAR_INPUT
3                  new : BOOL;
4          END_VAR
5          VAR_IN_OUT
6                  old : BOOL;
7          END_VAR
8
9          BEGIN
10                 IF (new = true AND old = false) THEN
11                         R_EDGE := true;
12                         old := true;
13                 ELSE R_EDGE := false;
14                         old := new;
15                 END_IF;
16 END_FUNCTION
```

Listing 2.2: Function in SCL

The function starts at line 1 with the specification of the function by using the keyword `FUNCTION` followed by the name of the function and the type. Next we have the variables specification at lines 2 up to 7 that is similar to the variable specifications of the function block, except that we now also have an in-output variable. The body of the function at lines 10 up to 15 consists of assignments and statements as in the function block. Note that the function name itself is also a variable which is assigned the same way as any other variable. The function is closed with `END_FUNCTION` at line 16. We can call this function in a program in the following way: if we have boolean variables `edge_signal`, `signal` and `signal_old`, we call the function with: `edge_signal := R_EDGE(new := signal, old := signal_old);`

The values of `signal` and `signal_old` will be used in the function. The variables `edge_signal` in the main function block and `R_EDGE` in the function and the variables `signal_old` in the main function block and `old` in the function always have the same value.

Listing 2.3 shows an example for a User-defined data type in SCL. For this example consider a variable `out3` of type `ComplexSignal`. This data type is a complex data type which consists of four variables. In the program we can now use the following variables `out3.out1`, `out3.out2`, `out3.remaining`, and `out3.elapsed`.

```
TYPE ComplexSignal
STRUCT
        out1 : BOOL;
        out2 : BOOL;
        remaining : INT;
        elapsed : INT;
END_STRUCT
END_TYPE
```

Listing 2.3: User-defined data type in SCL

An example of a Data block in SCL is shown in Listing 2.4.

```
DATA_BLOCK ModeDB
STRUCT
        mode : INT;
END_STRUCT
BEGIN
        mode := −1;
END_DATA_BLOCK
```

Listing 2.4: Data block in SCL

This data block consist of one integer variable. In the code this variable can be used with `ModeDB.mode`. The initial value of this variable is $-1$.

# 3 Problem Description

The correctness of the behavior of the PLC programs is a major concern. A defect in the program can cause severe damage and dangerous situations because PLCs are often used at critical points. Verification can be used to prevent these problems.

There are a lot of model checking tools available for verification, but none of these tools supports the SCL language. SCL programs can have a large number of input variables with a many possible values. Combined with the cyclic manner of the PLC programs, this can cause a state space explosion. We want to find a model checking tool that can cope with this state space explosion and with the PLC semantics. First we will take a look at the verification method that is currently used at CERN.

## 3.1 Verification at CERN

Borja Fernández Adiego, automation engineer at CERN, has described a method for verification of PLC code in his PhD Thesis [23]. The method translates PLC code into an intermediate model (IM). After the translation reduction techniques are applied to this model. Reductions are used to improve the running time of the verification by keeping the state space as small as possible. After the reductions the model can be translated into the input language of the verification tool, such as SMV.

Intermediate Model The IM is a Control Flow Graph (CFG) based on an automata network model, which consists of synchronized automata.

An automaton is a tuple $a = (L, T, l_0, V_a, \mathit{Val}_0)$ where $L = \{l_0, l_1, \dots\}$ is a set of locations, $T$ is a set of guarded transitions, $l_0$ is the initial location, $V_a$ is a set of variables, and $\mathit{Val}_0$ is a vector with the initial values of the variables in a fixed order. A transition consists of the source location and the target location. It can also have a guard, variable assignments, and a synchronization. In Figures 1-3 we show the locations as circles and the transitions as arrows.

The IM is created as follows:

- Input variables are assigned non-deterministically at the beginning of the cycle. In the automaton of the OB this is modeled in the transition from $l0$ to $l1$;
- For each Function and Function Block in the SCL code, there is an automaton in the IM;
- For each assignment there is a transition in the automaton;
- Function calls in the SCL code are synchronization steps in the automata of both the callee and the caller function. An example is shown in Figure 1. On the left we see the automaton of the organization block and on the right the automaton of the function. At the organization block, two synchronization transitions are needed for the function call. In the example these are the transition in the OB block from `l2` to `l3` and from `l3` to `l4`. The transition from `l2` to `l3` first assigns the parameter of the function. In this case it assigns the value `TRUE` to variable `a` in the function. It also has a synchronization `i1!`. This ensures that this transition is synchronized with the transition with `i1?` in the Function block. The other transition (from `l3` to `l4`) with synchronization `i2?` has

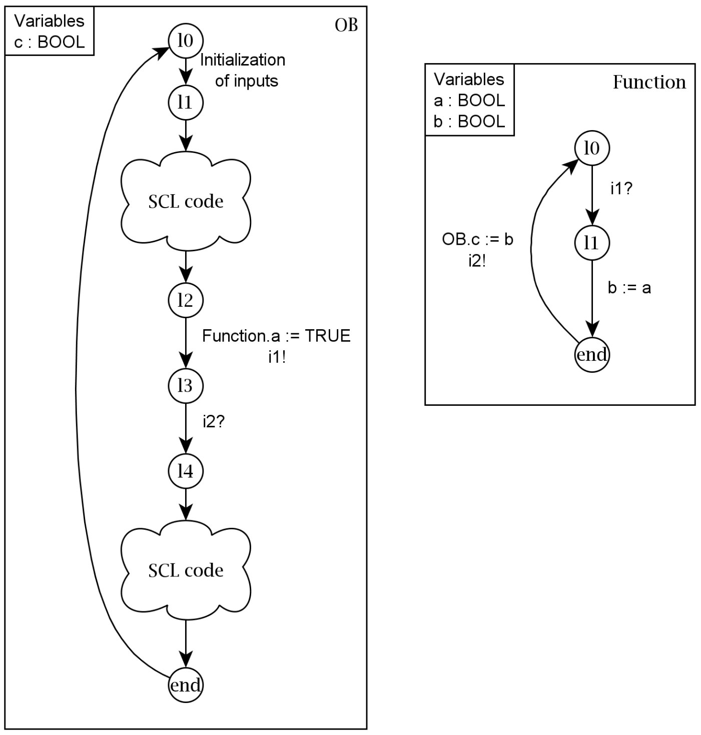

Fig. 1: Example of a function call in the IM

to wait until the function has reached the transition with `i2!`. The transition with `i2!` at the function also assigns a value to the variable `c` in the OB block. This is the return value of the function. The function has now returned to `l0` where it can be used again;

- An `IF`-statement in the SCL code is modeled by multiple branches in the automaton. Figure 2 shows SCL code with the IM of this code.

```
IF i < 10 THEN
    b := TRUE;
ELSEIF i > 10 THEN
    b := FALSE;
END_IF
```

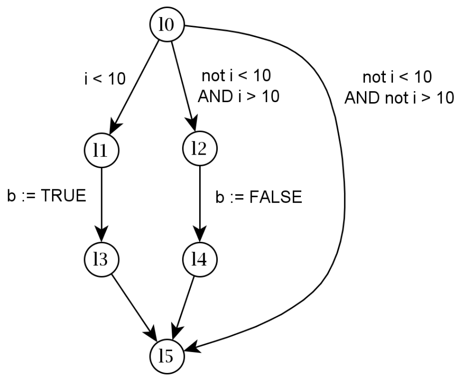

Fig. 2: Example of an `IF`-statement in the IM

We can see that there are three transitions added from `l0`. These transitions are the three branches of the control statement. One transition has the condition from the

`IF`-statement as guard (from `l0` to `l1`), one has the negation of this condition and the condition from the `ELSEIF`-statement as guard (from `l0` to `l2`), and one transition has the negation of both conditions as guard (from `l0` to `l5`). We can see that all three branches will end up in location `l5`. The first two branches will go through other locations to execute the assignments in the body of the conditional statement. Note that if the SCL code has assignments in the ELSE branch, then the ELSE branch in the IM would go through at least two more locations to execute this assignment;

- A `WHILE`-statement in the SCL code is modeled in a similar way as the `IF`-statement. In addition to this a guarded transition is added from the end-location of the statement to the starting location of the statement;
- To model the main cycle of the program there is a transition from the last location of the automaton corresponding to the main function block to the initial location.

Figure 3 shows the IM of the running example, see Listing 2.1. The locations are named `l0` up to `l12` and the last location is named `end`.

Reduction techniques The method describes four reduction techniques:

- *Cone of Influence(COI).* With this reduction technique all variables, assignments, and guards that are not relevant to the requirement are removed from the IM;
- *Rule-based reduction.* This reduction technique simplifies the CFG by removing empty branches, eliminating states and variables, and merging transitions and variables;
- *Mode selection.* With mode selection, function parameters that have a fixed value can be replaced by a constant value. This reduction is done before all other reductions;
- *Variable abstraction.* Variable abstraction is used to make an over-approximation of the model. However, this can cause spurious counterexamples, so all counterexamples have to be checked on the complete model.

With these reduction techniques, the state space of the programs is often drastically reduced, which results in a large improvement on the verification time.

SMV The IM can be translated into the input language of the verification tool, for instance the language SMV for the NuSMV and nuXmv tools. In the translation into SMV a dedicated variable is used to model the location in the IM. A translation from SCL into SMV will be explained in Section 4.2.

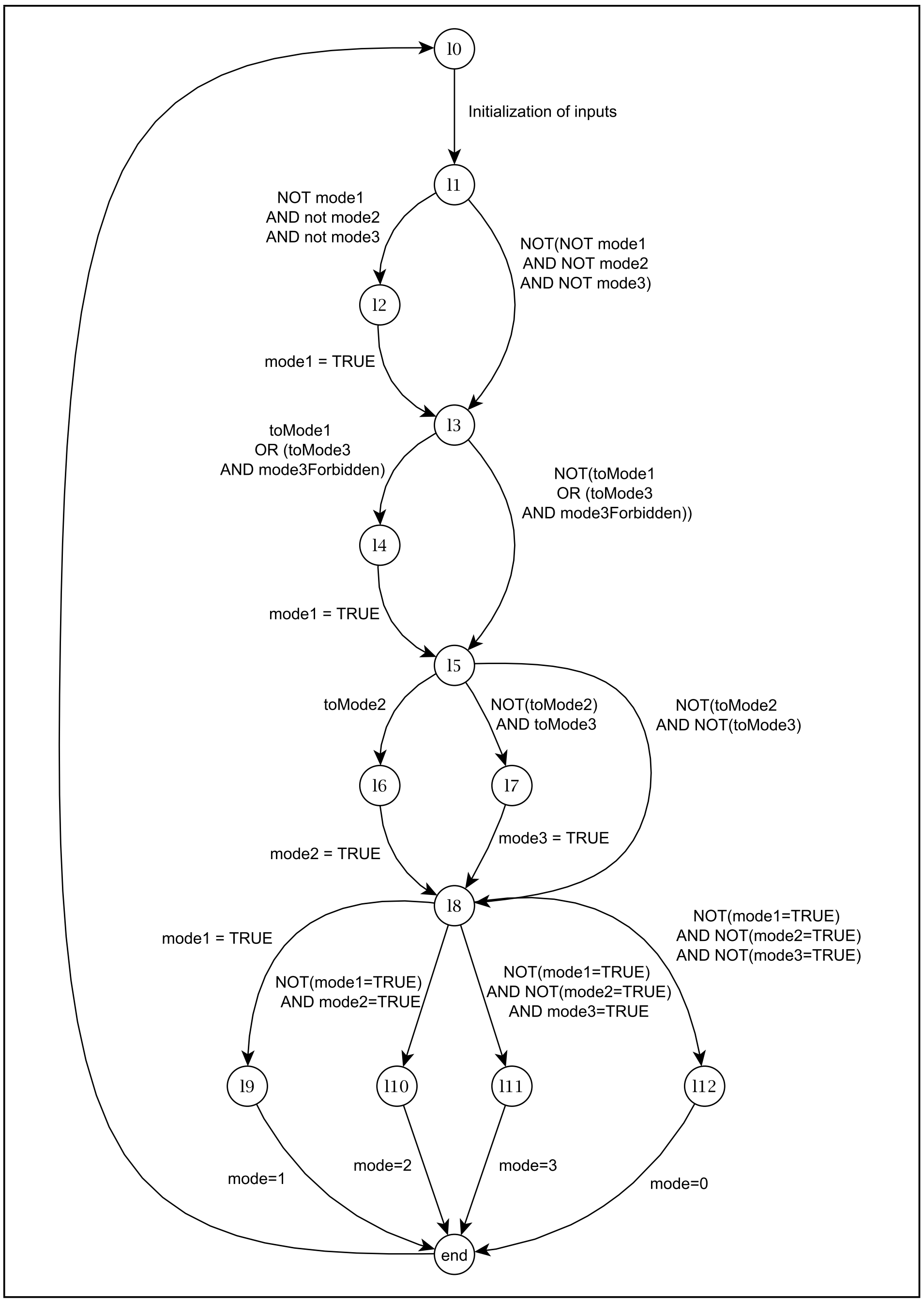
l0
Initialization of inputs
l1
NOT mode1 AND not mode2 AND not mode3
NOT(NOT mode1 AND NOT mode2 AND NOT mode3)
l2
mode1 = TRUE
l3
toMode1 OR (toMode3 AND mode3Forbidden)
NOT(toMode1 OR (toMode3 AND mode3Forbidden))
l4
mode1 = TRUE
l5
toMode2
NOT(toMode2) AND toMode3
NOT(toMode2 AND NOT(toMode3)
l6
l7
mode2 = TRUE
mode3 = TRUE
l8
mode1 = TRUE
NOT(mode1=TRUE) AND mode2=TRUE
NOT(mode1=TRUE) AND NOT(mode2=TRUE) AND mode3=TRUE
NOT(mode1=TRUE) AND NOT(mode2=TRUE) AND NOT(mode3=TRUE)
l9
l10
l11
l12
mode=1
mode=2
mode=3
mode=0
end

Fig. 3: Intermediate Model of the running example

## 3.2 Approach

We will be looking for languages that can capture the semantics of a PLC and for tools that can analyze a translation of PLC code to such a language. To do this, we will be looking at multiple verification tools. Performing experiments for all tools on a real PLC program would take a lot of time, mostly because of the time it takes to manually translate the SCL code to the input languages of these tools. To get a first impression of the tools, we will first conduct a number of experiments with example programs.

CERN provided us with three example programs of SCL code in increasing complexity, which can be found in Appendix A. The first program `Example` has 14 boolean variables and four integer variables. This includes six boolean input variables, four boolean output variables and two integer output variables. The program uses `IF`, `ELSE`, and `ELSEIF` control statements. The second program `Example_int` has the same variables as `Example` and one additional integer variable. It uses the same control statements as the first program with one additional `IF`-statement. The third program `Example_while` has the same variables as `Example_int`. Here, the new `IF`-statement in `Example_int` is replaced by a `WHILE`-statement.

There are four properties that we want to check for the example programs. For all three programs the properties are the same. We have two properties that are known to hold (henceforth referred to as TRUE-properties) and two properties that do not hold (henceforth referred to as FALSE-properties). All properties will be verified at the end of the main cycle, that covers the whole program code. This is the only moment that the values will be written to the output ports.

The properties are given in natural language. We have made a translation for all properties to assertions. Assertions are simple and we can use them for multiple tools. We have not used an implication because this is not supported by all tools. The variable types and the data types of the variables used in the properties can be found in Table 2.

- If `out2` is true then `out1` should be true too.
  Assertion: `(!out2 || out1)`
  Expected result: `true`

- If `signal` is false then `out2` should be false too.
  Assertion: `(signal || !out2)`
  Expected result: `true`

- `out3.out1` equals `out1`.
  Assertion: `(out3.out1 == out1)`
  Expected result: `false`

- `out3.elapsed` is 0 when `out1` is false.
  Assertion: `((out1 || (out3.elapsed == 0))`
  Expected result: `false`

| Variable | Data type | Variable type |
|---|---|---|
| *signal* | Boolean | Input variable |
| *out*1 | Boolean | Output variable |
| *out*2 | Boolean | Output variable |
| *out*3.*out*1 | Boolean | Output variable |
| *out*3.*elapsed* | Integer | Output variable |

Tab. 2: Data types and variable types in the properties

We will look at a number of verification tools. For each tool a translation will be made from SCL code into the input language of the tool. To make a correct translation, the tools should be able to model the semantics of the PLCs. Non-deterministic assignments will be used to model the input variables and we will use an unconditional loop to model the cyclic execution of the SCL code. The properties will also be translated to the tools specification language.

The tools are divided into two groups. The first group consists of software model checking tools that perform model checking on a model. The second group consist of software model checking tools that perform model checking on C code.

For each tool we will make two translations of each example program. The first translation is a full translation of the code. The second translation is a reduced version. Since we do not have access to the reduction techniques CERN used, we will do some reductions by hand. In contrast to the method of CERN, we will reduce the model only once. With this we only remove variables, assignments, and guards that are not relevant to any of the four properties. The reductions are comparable to the COI reductions and the rule-based reduction from CERN as discussed in section 3.1.

The following aspects will be taken into account for the comparison of the tools.

- Ease of translation
  If the SCL code differs a lot from the input language of the tool, a lot of choices will have to be made about the translation. This can cause errors and multiple ways to model the code. A tool will be preferred if it supports a language that does not differ much from SCL;

- Results and Running time
  We will look at the results and the running time for each property. In a large program, there will be a lot of properties to be checked. To make this feasible we would like for the running time to be at most 10 seconds, but preferably much less. We would also like the tools to correctly prove/refute as many properties as possible;

- Counterexamples
  For FALSE-properties, we would like to get a counterexample. It is preferred to get a counterexample that can be mapped back to the original SCL code.

We will compare the tools in each group on the example programs. CERN has also provided us with a larger SCL program with real code [1]. We will do a case study on this code using the most promising tools. The results of this case study will be compared to each other as well as to the results from CERN. For the case study we will not do any reduction as we have no reduction tools and to do this by hand would take a to much time and could give errors.

# 4 Verification tools on Models

This section describes a number of verification tools on Models. We will look at three different tools: Spin, NuSMV, and nuXmv. For all three tools the code of a program should be translated into a model before the verification can be done. Another tool that could be in this group of tools is mCLR2 [19]. However, after experimenting with this tool, we found that the results were not very promising. Therefore and due to time pressure, we will not discuss this tool in detail. NuSMV and nuXmv are similar tools from the same developers. They use the same input language and all functionalities from NuSMV are inherited in nuXmv. Because these tools are so similar we describe them in the same subsection.

For both Spin and NuSMV/nuXmv we will first describe which techniques are used. We will then look at the input language of the tools and show how a translation from the SCl code with the PLC semantics can be made. Next we will describe how the properties can be translated and added, and how to run a verification. Both subsections end with some techniques to improve the running times of the verifications.

We finish this section with the experiments and results of the tools in this section.

## 4.1 Spin

Spin [25] is a software verification tool that can also be used as a simulator and as a proof approximation system. The tool was developed at Bell Labs in the Unix group of the Computing Sciences Research Center in 1980 and it continues to evolve. For our experiments we have used version 6.4.3, which was released in 2014.
Spin can be run from the command line as well as from a graphical user interface, iSpin.

**PROMELA** The input language of Spin, PROMELA (Process Meta Language) is based on C. In addition it has guarded commands to capture non-determinism and send and receive communication statements for interaction between different processes.

PLC semantics To model the semantics an unconditional loop and non-deterministic choices are needed, as discussed in Section 3.2. For the program itself deterministic choices and conditional repetition are needed, as well as a translation of all the used types.

For the unconditional loop and the conditional repetition we use the repetition construct of PROMELA, which is as follows:
`do`
`::` *option1*
`::` *option2*
`od`

For the non-deterministic choices and the deterministic choices we use the select construct of PROMELA, which is as follows:
`if`
`::` *option1*
`::` *option2*
`fi;`

A construct can contain any number of options. An option consists of '::' followed by a sequence of statements. A statement can be an assignment, an assertion, a print statement, a communication statement, or a condition in the form of a logical expression. The first statement of an option is called the guard. An option is executable if the guard is executable. If the guard is a logical expression it is executable if the expression is satisfied. A guard `else` can be used that is only satisfied when all other options are not executable. If no options are executable the program will block. The repetition construct chooses one of the executable options non-deterministically and executes it. This is repeated until no executable options are available.

To make an unconditional loop we use the repetition construct with one option without a condition: namely the program.

```
do
:: program
od
```

Spin will repeat this option as long as it is executable.

To get the conditional repetition from SCL, we add a condition $b$ to the existing option and we add an option `break` with a guard `else`. With the `break` statement the program jumps to the end of the repetition loop. Spin will now repeat the option until the guard $b$ is no longer satisfied. Later in this section we will explain this `break` statement by means of an example.

To assign a non-deterministic value to a variable we use the select construct with options without conditions. For a boolean variable $b$ this is as follows:

```
if
   ::b = 0
   ::b = 1
fi;
```

To make the select construct deterministic, we add conditions ensuring that exactly one option is executable at a time.

The types `bool` and `short` are used for PROMELA.

When we compare PROMELA to SCL we see that both are structured programming languages. This makes the translation easy and straightforward, which makes it more corresponding to the original code.

**Translation** We are now going to look at the translation of SCL code and the semantics of this code to PROMELA. The structure of the PROMELA program is shown in Listing 4.1.

```
Variable declaration
active proctype go()
{
    do
        :: Non-deterministic input
        Program body
        Properties
    od
}
```

Listing 4.1: PROMELA structure

The translation to PROMELA starts with the declaration and initialization of all variables. The main function block from the SCL code is represented by an active proctype named go. A model in Spin consists of a main process that starts other processes; processes that start independently can also exist. Those processes are provided by the keyword active. We now have the unconditional loop (lines 4-8) with inside this loop the assignments of the non-deterministic input variables, the program body, and the assertions. The unconditional loop and the non-deterministic assignments of the input variables are used to model the semantics of the SCL code.

We will now take a look at the different aspects of the program.

Declaration and initialization Listing 4.2 shows the declaration and initialization of the variables of the running example.

```
bool error, toMode1, toMode2, toMode3, mode3Forbidden;
bool mode1 = 0;
bool mode2 = 0;
bool mode3 = 0;
short mode = 0;
```

Listing 4.2: PROMELA variable declaration

We can see that a variable is declared with the type followed by its name. Multiple variables of the same type can be declared together. The initialization of variables can be combined with the declaration with `= 0`, if we want it to be `0`. Every statement ends with a semicolon. User defined types in SCL can be translated into `typedef` in PROMELA as shown in the example below. On the left side we have the user defined type in SCL and on the right side the translation in PROMELA.

```
TYPE ComplexSignal
STRUCT
   out1 :  BOOL;
   out2 :  BOOL;
   remaining :  INT;
   elapsed :  INT;
END_STRUCT
END_TYPE
```

```
typedef ComplexSignal{
   bool out1;
   bool out2;
   short remaining;
   short elapsed;
};
```

Non-deterministic assignments The non-deterministic assignments to the input variables of the running example are shown in Listing 4.3. We have already seen how non-determinism is used earlier in this section. We see that the options do not end with a semicolon but the selective constructs do.

```
    if
        ::error = 0
        ::error = 1
    fi;
    if
        ::toMode1 = 0
        ::toMode1 = 1
    fi;
    if
        ::toMode2 = 0
        ::toMode2 = 1
    fi;
    if
        ::toMode3 = 0
        ::toMode3 = 1
    fi;
    if
        ::mode3Forbidden = 0
        ::mode3Forbidden = 1
    fi;
```

Listing 4.3: PROMELA non-deterministic assignments

Program body The following aspects of the SCL code will be discussed: `IF`-statements, `WHILE`-loops and function calls.
First we will look at the translation of an `IF`-statement. The select construct will be used as mentioned before. Consider the example below with on the left side the SCL code and on the right side the translation in PROMELA.

```
IF b1 THEN
   s1
ELSEIF b2 THEN
   s2
END_IF;
```

```
if
   ::b1 -> s1
   ::!b1 && b2 -> s2
   ::else
fi;
```

Note that to model an `ELSEIF` statement we have chosen to negate the preceding conditions. Another choice would be to use nested select statements. Because an option to be executable is necessary, we will always have an option with `else` as guard, even if there is no `ELSE` statement in the original code. When this is the case, we use the guard `else` without a sequence.

For the `WHILE`-loops we use the repetition construct as mentioned before. Consider the following example with SCL code on the left and PROMELA code on the right.

```
WHILE b DO
   s
END_WHILE;
```

```
do
   ::b -> s
   ::else -> break
od;
```

In PROMELA there is no difference between an `->` and an `;`. We use an `->` to emphasize that the first statement is a guard.

Because the function in the programs we use is only called once we decided to translate this without a function. We have treated the code as if the program of the function were at the location of the function call. Normal functions are not supported by PROMELA, but they can be represented by other features of PROMELA.

Properties For the properties Spin supports LTL requirements and assertions. We will use assertions because they are conceptually simpler and we supported by the majority of the tools. We have already seen the translation of the properties into assertion in Section 3.2. If we want to check a property, we add the following line: `assert(`*property*`)`.

This completes the translation from SCL to PROMELA. All programs used in the experiments are given in Appendix B.

**Verification** To perform verification with Spin from the command line, we use a number of commands. With `spin -a Example.pml` Spin makes an exhaustive state space searching program for the model, which results in five files named `pan.[bchmt]`. We can compile this program with `gcc pan pan.c`, which gives us an executable `pan`. Executing this executable completes the verification. With this last step we can use the option `-m N` to set maximal search depth to `N` steps, that is needed for larger programs. If there are multiple assertions in the code, the verification will terminate as soon as it has found a violation for one of the assertions. It will show which assertion this is and it will make a file with the trail of the counterexample. To get a counterexample from this trail we run Spin with `spin -t -p Example.pml`.

With the graphical user interface iSpin we can also do the verification, as well as simulations. With the trail file iSpin can be used to simulate and rerun a counterexample.

Improvements To improve the verification time, we use `d_step` in the PROMELA language. This term introduces a deterministic code fragment that is executed indivisibly, which works as follows: Consider the following sequence of statements: `s1; s2; s3`. This gives us 4 states: one at the beginning and one after each statement. Now if we use a `d_step` we get `d_step{ s1; s2; s3}`, which gives us only 2 states: one at the beginning and one at the end. In this way the statements will always be executed after each other without any interruption. This option is often used for mutual exclusion, however in our case it is used to reduce the state space. By reducing the state space the verification time is also reduced.

The `d_step` sequence can only contain deterministic code. We add the `d_step` after the non-deterministic assignments of the input variables. The structure of the program with the `d_step` is shown in Listing 4.4.

```
Variable declaration
active proctype go()
{
        do
        :: Non-deterministic input
        d_step{
        Program body
        }
        Properties
        od
}
```

Listing 4.4: PROMELA structure with d_step

## 4.2 NuSMV and nuXmv

NuSMV [13] and nuXmv [12] are symbolic model checkers developed as a joint project between the Embedded Systems Unit in the Center for Information Technology at FBK-IRST, the Model Checking group at Carnegie Mellon University, the Mechanized Reasoning Group at University of Genova, and the Mechanized Reasoning Group at University of Trento. The tools support multiple model checking techniques, including BDD-based symbolic model checking [11], SAT-based model checking [7], and bounded model checking [8]. For our experiments we have used version 2.5.4 of NuSMV and version 1.0.1 of nuXmv.

NuXmv inherits all the functionalities of NuSMV. In addition it has a few new types and constructs. Also a number of new model checking algorithms are added. One of these algorithms is based on IC3. IC3 (Incremental Construction of Inductive Clauses for Indubitable Correctness) [34] is an algorithm that produces lemmas in a similar way to how humans would produce lemmas. This is done by generating lemmas that are inductive relative to previous lemmas. These lemmas are used to prove properties.

**SMV** NuSMV and nuXmv both use the SMV language. This language is an automata-based programming language. It is used to write programs that describe a finite state machine (FSM). The states in this FSM are defined by the values of all variables in the program. To define the values of the variables in all states, two types of expressions are used: one for the initial value of a variable and the other for the value of a variable in the next state. A variable may or may not have an initial value. If a variable has no initial value, it gets an arbitrary value in the initial state. Every variable in the program should have a specification to get the value for the next state. An example of a specification for the value of a next state for a variable `a` is given below.

```
next(a) :=
   case
      b1 :  s1;
      b2 :  s2;
      TRUE: a;
   esac;
```

In this example, if in the boolean condition `b1` holds in the current state, the value of variable `a` in the next state will be `s1`. If `b1` does not hold and `b2` does, the value of `a` in the next state will be `s2`. If neither `b1` nor `b2` holds, the value of `a` will not change in the next state.

To keep track of the location of the FSM we introduce an SMV variable `loc`. Every transition in the FSM changes the location. Later we will merge some of those locations.

PLC semantics An unconditional loop and non-deterministic assignments are needed to model the semantics of a PLC program, as discussed in Section 3.2. Conditional choices and conditional repetition are needed for the program itself.

To construct the unconditional loop we use the `loc` variable. In the initial state of the FSM the variable `loc` has the value `start`. All possible paths from this location will reach the location where `loc` = `end`. To make the unconditional loop we add a transition from `loc` = `end` to the initial location. Note that when the FSM reaches the initial location again, it does not necessarily correspond to the initial state since variables other than the `loc` variable could have different values.

SMV has non-determinism built in the assignments of the state of a variable. To give a boolean variable `a` a non-deterministic value taken from the set {TRUE, FALSE} we give this set to a variable as we can see in the following statement:

```
next(a) := {TRUE, FALSE};
```

With this construction the variable `a` will get a non-deterministic value in every step, but we only want it to get a new value when the model is at a certain location. We use case distinction to give `a` a non-deterministic value at location `l`. At any other location its value will remain the same, as we can see in the following construction:

```
next(a) :=
   case
      (loc = l) :  {TRUE, FALSE};
      TRUE : a;
   esac;
```

For the conditional choices the location variable is used. This is shown in a small example with on the left side the SCL code and on the right side the SMV code. For the SCL code the value of the `loc` variable at the corresponding SMV code is shown.

```
(l1)    IF a THEN
(l2)       b := TRUE
(l1)    ELSE
(l3)       b := FALSE
(l4)    END_IF
```

```
init(loc) := l1;
next(loc) :=
   case
      (loc = l1) & (a) :  l2;
      (loc = l1) :  l3;
      (loc = l2) :  l4;
      (loc = l3) :  l4;
   esac;

next(b) :=
   case
      (loc = l2) :  TRUE;
      (loc = l3) :  FALSE;
      TRUE : b;
   esac;
```

The `loc` variable starts at location `l1`. For the conditional choice we use the condition as a guard in the case distinction of the `loc` variable. If the condition holds we go to location `l2` and if it does not we go to location `l3`. Note that we do not have to use the negation of the condition here, since the SMV code will check the cases from the top down. At the specification of the next state of variable `b`, we can see that `b` will be assigned at location `l2` and `l3`. These are the locations in the branches of the conditional statement. After the conditional statement, when in the SCL code the statement `END_IF` is reached, the branches go to a shared location; in this example this is location `l4`. If there are more branches in the conditional choice then there will be more case distinctions for this location. The location variable may get multiple different values in a branch before it reaches the shared location.

To make a conditional repetition we make a small loop in the locations. The location at the end of the loop goes back to the location at the beginning of the loop. When at the beginning of the loop the condition is satisfied, the program continues in the loop, otherwise the program continues after the loop.

A small example with on the left side the SCL code and on the right side the SMV code is shown below.

```
(l1)    WHILE i < 10 DO
(l2)    i := i + 1;
(l1)    END_WHILE;
(l3)
```

```
init(loc) := l1;
next(loc) :=
   case
      (loc = l1) & (i<10) :  l2;
      (loc = l1) :  l3;
      (loc = l2) :  l1;
   esac;

next(i) :=
   case
      (loc = l2) :  i+1;
      TRUE : i;
   esac;
```

In the example we can see a while loop that increases the value of $i$ with 1 if it is less than 10. The program starts with `loc = l1`. The condition $i < 10$ is checked at the location variable. If $i < 10$, the program will go to the body of the loop (location `l2`). Otherwise the program will continue after the loop (location `l3`). At the end of the body of the loop the location returns to the beginning of the loop (location `l1`). When we are at location `l2`, $i$ will be increased by 1. Otherwise the value of $i$ will remain the same.

The types used in the experiments are `boolean` and `signed word[16]`. A `signed word[16]` represents an array of 16 bits. This array represents the values of a 16 bits signed integer. The syntax of the representation of these values is of the form `(-)0sd16_`*`value`*. For example the values 0, 10, and $-1$ are denoted by `0sd16_0`, `0sd16_10`, and `-0sd16_1` respectively. Note that in the examples above integers are used to improve the readability of the examples. For the location variable we use an enumeration type which has all locations as possible values. An example will be shown later in this section.

Regarding the functions, we have chosen to eliminate all functions by substituting the function calls with the code of the function. Another option (which is used by CERN) is to make a new module for each function.

**Translation** We will now look at the translation of SCL code and its semantics to SMV code. The structure of the SMV program is shown in Listing 4.5.
The translation starts with declaring the module `main`, which represents the main function block from the SCL code. We first declare all variables from the SCL code, as well as the location variable `loc`. Subsequently we get the specification of the initial state of `loc` and the specification of the next state of `loc`. We will then get the initialization of the other variables followed by the non-deterministic input variables assignments and the program body. At the end of the program we add the properties we want to verify.

```
MODULE main
        Variable declaration
        Location specification
        Variable initialization
        Non-deterministic input
        Program body
Properties
```

Listing 4.5: SMV structure

We will now take a look at the different aspects of the program.

Declaration of variables   Listing 4.6 shows the declaration of the variables for the running example.

```
VAR
        error : boolean;
        toMode1 : boolean;
        toMode2 : boolean;
        toMode3 : boolean;
        mode3Forbidden : boolean;
        mode1 : boolean;
        mode2 : boolean;
        mode3 : boolean;
        mode : signed word[16];
        loc : {start, step1, step2, step3, end};
```

Listing 4.6: SMV variable declaration

We can see that a variable is declared with the name followed by a colon and the type, every statement in SMV ends with a semicolon. The type of the added variable for the location is a set of values.

SMV does not support user defined types. Therefore any variable in a user defined type from the SCL code should be added as a separate variable in the SMV model. Below we show an example for the user defined type in the Example program. On the left we have the SCL code of the user defined type and on the right side the translation in SMV of a variable `out3` of this type.

```
TYPE ComplexSignal
STRUCT
   out1 : BOOL;
   out2 : BOOL;
   remaining : INT;
   elapsed : INT;
END_STRUCT
END_TYPE
```

```
out3.out1 : boolean;
out3.out2 : boolean;
out3.remaining : signed word[16];
out3.elapsed : signed word[16];
```

Location specification   For the running example there are 14 different locations. Listing 4.7 shows the location specification for the running example.

```
init(loc) := start;
next(loc) :=
    case
```

```
        (loc = start)                                                   : step1;
        (loc = step1) & (!mode1 & !mode2 & !mode3)                      : step2;
        (loc = step1)                                                   : step3;
        (loc = step2)                                                   : step3;
        (loc = step3) & (toMode1) | (toMode3 & mode3Forbidden)          : step4;
        (loc = step3)                                                   : step5;
        (loc = step4)                                                   : step5;
        (loc = step5) & (toMode2)                                       : step6;
        (loc = step5) & (!toMode2) & (toMode3)                          : step7;
        (loc = step5)                                                   : step8;
        (loc = step6)                                                   : step8;
        (loc = step7)                                                   : step8;
        (loc = step8) & (mode1)                                         : step9;
        (loc = step8) & (!mode1) & (mode2)                              : step10;
        (loc = step8) & (!mode1) & (!mode2) & (mode3)                   : step11;
        (loc = step8)                                                   : step12;
        (loc = step9)                                                   : end;
        (loc = step10)                                                  : end;
        (loc = step11)                                                  : end;
        (loc = step12)                                                  : end;
        (loc = end)                                                     : start;
    esac;
```

Listing 4.7: SMV location specification

We can see that the program starts with location `start`. In this location we assign the non-deterministic values to the input variables. After the start location we go to `step1` where we start executing the program body. The number of locations depend on the program body, which we will see later in this section. The last location is `end`. In this location the properties can be checked. When the location is `end` it will go back to `start`.

Variable Initialization  Note that an initial state is not required for all variables. Listing 4.8 shows the initialization of all variables from the SCL program for the running example.

```
init(mode1) := FALSE;
init(mode2) := FALSE;
init(mode3) := FALSE;
init(mode) := 0;
```

Listing 4.8: SMV Variable initialization

Non-deterministic assignments  We have already seen how non-determinism works earlier in this section. Listing 4.9 shows the non-deterministic assignments of the input variables. We see that for each variable we get a specification for the value in the next state. If the location is `start` we get a non-deterministic value, otherwise we keep the value. These variables do not need an initial state, because they get a non-deterministic value at the beginning of the program.

```
next(error):=
    case
        (loc = start) : {TRUE, FALSE};
        TRUE : error;
    esac;
next(toMode1):=
    case
        (loc = start) : {TRUE, FALSE};
        TRUE : toMode1;
    esac;
next(toMode2):=
    case
        (loc = start) : {TRUE, FALSE};
        TRUE : toMode2;
    esac;
next(toMode3):=
    case
        (loc = start) : {TRUE, FALSE};
        TRUE : toMode3;
    esac;
next(mode3Forbidden):=
    case
        (loc = start) : {TRUE, FALSE};
        TRUE : mode3Forbidden;
    esac;
```

Listing 4.9: SMV non-deterministic assignments

Listing 4.9 shows the non-deterministic assignments of the input variables. We see that for each variable we get a specification for the value in the next state. If the location is `start` we get a non-deterministic value, otherwise we keep the value. These variables do not need an initial state, because they get a non-deterministic value at the beginning of the program.

Program body For each variable that is not an input variable we get a specification for the value in the next state. This specification always consists of a case distinction with the different possible values for the next state. The conditions of these case distinctions are values of the variable `loc`. The last case has condition `TRUE` and does not change the value. Note that this `TRUE` condition is not needed for the location variable, since the cases cover all possibilities.

The translation of the program follows the structure of the SCL code. We keep track of this translation with the variable `loc`, which divides the program into steps. At the start of the program body the variable `loc` has value `step1`.

A new step is created after every statement in the code. We have already seen an example for conditional choices and the conditional repetition. In addition every assignment will be in a different step.

Parallel Assignments To reduce the state space we will reduce the number of steps by executing some assignments in parallel. To do this we first take all sets of contiguous assignments. We have used the algorithm described by Stokely et al. [35]. This algorithm is used to determine which sets of contiguous assignments can be executed in parallel without changing the program. The algorithm does not change the order of the assignments but it can change the right hand side of the assignment if it does not alter the result of the block. With this algorithm, the least amount of parallel executions in these sets is acquired. All sets of assignments that can be executed in parallel will be in the same step in the code.

An example to show the main idea of the algorithm is shown below.

**Example** Consider the following block that consists of four assignments that should be executed in sequence.

```
x := 1;
u := 2;
y := x;
v := u;
```

We can see that the first two assignments can be executed in parallel since they do not affect each other. The third assignment, `y := x;`, depends on `x := 1;` so it cannot be executed in parallel with the first two assignments. The third and fourth assignments can be executed in parallel since they do not affect each other. We can replace the assignments `y := x;` with `y := 1;` and `v := u;` with `v := 2;` without changing the outcome of this block. With these replacements we can execute all four assignments in parallel.

Note that in the running example all assignment blocks consist of a single assignment, therefore this algorithm does not give improvements to the SMV code.

We have experimented with merging conditional statements with the steps of assignments. While for the program `Example` this reduced the number of locations from 24 to 10, the improvements on the verification times are minimal. Further research is required to find out how these reductions affect the verification times for SCL programs generally.

Properties The properties are located at the end of the program structure. A number of specifications are supported, which includes CTL, LTL, and invariant specifications. With NuSMV we have used CTL specifications as well as invariant specifications for the properties. For nuXmv we have only used invariant specifications.

Both CTL as invariant specifications in NuSMV and nuXmv use logical expressions. These expressions use the following Logical operators: negation, disjunction, conjunction, implication, and equivalence. In the SMV language these operators are represented by: `!`, `|`, `&`, `->`, and `<->` respectively.

An Invariant specification is a logical expression that should hold in every reachable state.

A CTL specification is a logical expression preceded by a pair of temporal operators that specifies when the expression should hold. We will only use the pair `AG` which specifies that the expression should hold in every reachable state.

We can see that the specification of the properties use the same logical expression for both CTL and Invariants.

This results in the following logical expressions in SMV for the properties:

- If `out2` is true then `out1` should be true too.
  Logical expression: `(loc = end -> (out2 -> out1))`

- If `signal` is false then `out2` should be false too.
  Logical expression: `(loc = end -> (!signal -> !out2))`

- `out3.out1` equals `out1`.
  Logical expression: `(loc = end -> (out3.out1 = out1))`

- `out3.elapsed` is 0 when `out1` is false.
  Logical expression: `(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))`

For all expressions an implication with `loc = end` is used because we want to check the property at the end of the program. CTL specifications are added with `SPEC` *property* and invariant specifications are added with `INVARSPEC` *property*.

This completes the translation from SCL to SMV. All programs used in the experiments are given in Appendix C.

Differences with the translation of CERN The main difference between our translation and the translation of CERN is that we do not use extra modules for functions. CERN also uses extra modules to model the data blocks in the SCL code, while we add these variables to the other variables. We have chosen not to use extra modules because the extra modules also need extra variables. Another difference is that we use parallel assignments, where at CERN the problem is solved with their reduction techniques on the IM.

**Verification** There are two ways to perform a verification with NuSMV; the interactive mode or the command line. When using the command line, the command `NuSMV Example.smv` is used to verify the properties in Example.smv. This works for both the CTL properties as the invariant properties. When the interactive mode is used, use the following sequence of commands for the CTL properties:

```
NuSMV -int Example.smv
go
check_ctlspec
```

For the invariant properties we change the last command to `check_invar`.
With nuXmv we use the new algorithm that uses ic3 engines. To perform this verification, we use the interactive mode of nuXmv with the following sequence of commands:

```
nuXmv -int Example.smv
go
build_boolean_model
check_invar_ic3
```

NuSMV and nuXmv give a counterexample when one of the given properties is violated. The counterexample is easy to read when you have the SMV model, it is harder to trace back to the original SCL code. There is an option to simulate a model which gives us the possibility to rerun the counterexample.

Improvements To improve the verification time we have used the following options in the command line.

- **-df** Disable the computation of the set of reachable states. This reduces the computation time because not all reachable states have to be created

- **-dynamic** Enables dynamic reordering of variables, this can reduce the size of the BDDs

- **-coi** Enables cone of influence reduction, this is similar to the reduction CERN implemented. A difference is that this reduction is on the SMV code. This reduction is less effective than the reduction by CERN. A more detailed description can be found in Section 3.1.

## 4.3 Experiments and results

We have seen how to make a translation of the SCL code for the tools above. For each input language we have made a full translation and a translation with reductions, as explained in Section 3.2. We have run the verification of the example files with the tools. The running times can be found in Table 3. These are the user time + the system time.

| **Program** | **Property** | **Spin** | **NuSMV** | | **nuXmv** |
|---|---|---|---|---|---|
| | | | **CTL** | **Invariants** | |
| `Example` | 1(true) | 3m0.03s | 44.85s | - | **10.59s** |
| | 2(true) | 2m52.44s | **0.34s** | - | 0.77s |
| | 3(false) | 35.28s | 0.31s | **0.26s** | 4.99s |
| | 4(false) | 11.43s | **0.30s** | 0.41s | 5.28s |
| `Example` reduced | 1(true) | 12.15s | 20.47s | - | **1.77s** |
| | 2(true) | 12.41s | **0.07s** | - | 0.15s |
| | 3(false) | 11.61s | **0.07s** | 0.09s | 0.14s |
| | 4(false) | 13.17s | 0.18s | **0.16s** | 0.62s |
| `Example_int` | 1(true) | 22m1.45s | 58m31.38s | - | **8.36s** |
| | 2(true) | 21m57.84s | **0.28s** | - | 1.35s |
| | 3(false) | 13.39s | **0.35s** | 0.47s | 5.02s |
| | 4(false) | 12.13s | 0.64s | **0.52s** | 3.51s |
| `Example_int` reduced | 1(true) | 17.19s | 69m36.24s | - | **1.59s** |
| | 2(true) | 17.93s | **0.14s** | - | 0.30s |
| | 3(false) | 11.40s | 0.18s | **0.15s** | 0.42s |
| | 4(false) | 12.20s | **0.19s** | **0.19s** | 0.96s |
| `Example_while` | 1(true) | 14m24.09s | 22m56.11s | - | **33.38s** |
| | 2(true) | 14m1.97s | **0.31s** | - | 2.33s |
| | 3(false) | 12.23s | 30.07s | **0.33s** | 5.62s |
| | 4(false) | 13.22s | 10.73s | **0.54s** | 11.09s |
| `Example_while` reduced | 1(true) | **17.26s** | 7m25.84s | - | 1m4.67s |
| | 2(true) | 16.71s | **0.11s** | - | 0.18s |
| | 3(false) | 11.50s | 3.62s | **0.12s** | 1.03s |
| | 4(false) | 11.97s | 3.40s | **0.16s** | 1.20s |

Tab. 3: Running times of the verification of the tools on models on the example programs

The NuSMV verification with invariants was not able to prove the TRUE-requirements. All other verifications have succesfully verified and refuted the properties. We can see that out of these tools the verification of Spin took the longest for most of the cases. For TRUE-property 1 nuXmv is the fastest except for the reduced version of `Example_while`. For TRUE-property 2 NuSMV with CTL is faster. For the FALSE-property, NuSMV with invariants has the best performances of almost all programs. We can also see that the reductions on the model improve the running time for most programs. Strangely it made the running time for nuXmv of `Example_while` with TRUE-property 1 worse.

All of the tools above can give a counterexample which is understandable with the translation of the program. The counterexample given by Spin can also be mapped back to the original SCL code. The counterexamples given by NuSMV and nuXmv cannot easily be mapped back to the original code. Knowledge about SMV is needed to understand the counterexamples given by these tools.

For the translation of the program code, Spin was the easiest language to translate into. This is due to the fact that both Spin and SCL are structured programming languages. SMV is an automata-based programming language that differs significantly from SCL, which made the translation harder.

# 5 Verification tools on Code

This section describes a number of verification tools on code. All of these tools use the C language as input language and assertions for the properties. Therefore we will first look at the aspects of C, the translation of the SCL code with the PLC semantics into C, and the translation of the properties. We will then look at six different tools: CBMC, Kinductor, CBMC Incremental, 2LS, CPA-checker, and SATABS. Another tool that could be in this group is JPF [26]. This is a verification tool for JAVA code. However, after experimenting with JPF, the results were not very promising. Therefore and considering the time constraints we will not discuss this tool in detail.

For each tool we give a description of the used techniques. We will then describe how the properties should be added and how to run a verification.

We finish this section with the experiments and results of the tools in this section and a small comparison to the other tools.

## 5.1 C

C is a structured programming language, which is widely used, for instance for programming operating systems and embedded system applications. The C language is known to have a good stability and speed. C is an extensive language of which we will use only some features. One of these features is the support of pointers for addressing locations in the memory. We will explain this feature further on in this section.

**PLC semantics** As discussed in Section 3.2 we need an unconditional loop and non-deterministic choices to model the semantics of the SCL program. For the translation of the program code we need a translation for the used types, deterministic choices, and conditional repetition. We will continue to discuss these aspects.

The unconditional loop and the conditional repetition can be constructed with a `while` loop as shown below.

```
while(b){
   s
}
```

In this loop, sequence `s` will be executed as long as the boolean condition `b` holds. The `while` loop in C always has a condition. To make this loop unconditional we use `true` as a condition.

The C language has no support for non-determinism. To introduce a limited way of non-determinism to C, all tools that we have used for verification of C programs have added the same construction in the language. This is done by the use of a function with prefix `nondet_`. If we want to give a non-deterministic value to a boolean variable *b* we use a function `b = nondet_bool();`. We have to declare this function in the code before we use it. The same construction can be used for all types in C. Note that the body of the function is undefined. The return type defines the type of the variable. This makes the name of the function irrelevant.

For the deterministic choices the `if` statement is used. An example is shown below.

```
if(b){
   s1;
}else{
   s2;
}
```

Note that the `if`-statement in C differs from the `if`-statement in PROMELA. In C it has the usual semantics common for the most programming languages. If the boolean expression `b` holds, statement `s1` will be executed and if `b` does not hold, statement `s2` will be executed.

For the experiments we have used the types `bool` and `short`.

**Translation** We are now going to look at the translation of SCL code into C. In this translation we also take care of the semantics of the SCL program. To be able to use boolean values in C we have to include `stdbool.h`. The structure of the C program is shown in Listing 5.1. The translation starts with the declaration of the variables. Subsequently the main function starts. The declaration of the variables could also be done inside this function if no global variables are needed. The main function contains the unconditional loop that is needed for the semantics of the SCL code. Inside this loop, the non-deterministic assignment of the input variables, the program body, and the properties are present.

```
#include <stdbool.h>
function declarations
int main(){
        Variable declaration and initialization
        while(true){
                Non-deterministic input
                program body
                properties
        }
}
```

Listing 5.1: C structure

We will now look at the different aspects of the program.

Function declarations There are different reasons to declare a function. These will be explained together with the other aspects of the program. Firstly, we will consider the declaration and initialization of the variables.

Variable declaration and initialization Listing 5.2 shows the declaration and initialization on the variable for the running example.

```
bool error, toMode1, toMode2, toMode3, mode3Forbidden;
bool mode1 = false;
bool mode2 = false;
bool mode3 = false;
short mode = 0;
```

Listing 5.2: C variable declaration

We see that multiple variables can be declared together and the initialization can be combined with the declaration. A user defined type in SCL can be translated to a `struct` in C as in the example below. On the left side the SCL code is given and on the right side the translation in C code.

```
TYPE ComplexSignal
STRUCT
   out1 :  BOOL;
   out2 :  BOOL;
   remaining :  INT;
   elapsed :  INT;
END_STRUCT
END_TYPE
```

```
struct ComplexSignal{
   bool out1;
   bool out2;
   short remaining;
   short elapsed;
};
```

Non-deterministic assignments The non-deterministic assignments of the input variables for the running example are shown in Listing 5.3. To be able to use the non-deterministic function, we have declared the function with `bool nondet_bool();`. This is placed at the top of the program code.

```
error = nondet_bool();
toMode1 = nondet_bool();
toMode2 = nondet_bool();
toMode3 = nondet_bool();
mode3Forbidden = nondet_bool();
signal = nondet_bool();
```

Listing 5.3: C non-deterministic assignments

Program body The program body of the programs used in the experiments consists of `IF`-statements, `WHILE`-loops, and function calls. The structure of the `while`- and `if`-statements is the same as in PLC code. Therefore we can translate these statements one to one to C code. The usage of functions in C differs from functions in SCL. A function in SCL can change its parameters, but functions in C cannot. To simulate this aspect in C we use pointers. For the parameters that are changed in the function in SCL we will use the address of this variable as parameter in C. To get the address of a variable in C we put an '&' in front of the variable. When we use variables in the function in SCL we have to use pointers in C to change the value of the variable and not the address. To do this we put an '*' in front of the variable. We can see this in the example below. On the left side the function in SCL is given and on the right side the translation in C.

```
FUNCTION R_EDGE : BOOL
   VAR_INPUT
      new :  BOOL;
   END_VAR
   VAR_IN_OUT
      old :  BOOL;
   END_VAR

   BEGIN
      IF (new = true AND old = false)
THEN
         R_EDGE := true;
         old := true;
      ELSE R_EDGE := false;
         old := new;
      END_IF;
END_FUNCTION
```

```
bool R_EDGE(bool new, bool *old){
   if(new && !*old){
      *old = true;
      return true;
   } else{
      *old = new;
      return false;
   }
}
```

All added functions will be placed in the part of the function declarations at the top of the program structure. The function call is placed in the program body. The function call in SCL code for the example above is as follows:
`edge_signal := R_EDGE(new := signal, old := signal_old);`
In C we will get the following function call:
`edge_signal = R_EDGE(signal, &signal_old);`
Note that we only put an '*' in front of the parameter in the function and an '&' in front of the parameter in the function call if the function can change this parameter, i.e. if it is an output variable or an in-output variable.

Properties For the properties all tools that we have used for verification of C programs use assertions. We have already seen the translation to assertions in Section 3.2. There are differences in the different tools in the way in which we add the assertion. We will discuss this in the subsections of the specific tools.

This completes the translation to C. All programs used in the experiments are given in Appendix D.

## 5.2 CBMC

CBMC [17] is a symbolic model checker that uses bounded model checking. For the experiments we have used CBMC version 5.1. This tool has no support for proving properties; it can only refute them. With bounded model checking the program will be checked for a given number of loop-iterations $k$. This value should be given by the user. CBMC unfolds the loop $k$ times and then checks the properties. A violation is reported if it is found within $k$ iterations. If the tool does not find a violation of the property, it will state that the verification is successful. When this occurs we still do not know if the property is true for the whole program. A violation of the property could still occur in further iterations of the loop. When a property is violated CBMC will give a counterexample. The counterexample is easy to read with the C model and because the C code is very similar to the SCL code, the counterexample can also be read with the SCL model.

The CBMC tool is a basic tool. There are multiple tools that are built on CBMC or use some aspects of CBMC. We will look at some of these tools later.

To check a property with CBMC we add an assertion to the code. We will place the assertion at the location where we want to check the property, which in our experiments is at the end of the `while`-loop. The assertion is added as follows: `assert(`*property*`);`

**Verification** To perform verification with CBMC we use the following command in the command line:
`cbmc Example.c --no-unwinding-assertions -unwind k` where $k$ is the bound on the number of iterations of the loop in the program. When there are multiple loops in the program, the bound applies to all loops. It is possible to give a different bound to different loops, which is done with the option `--unwindset l:k` where $l$ is the name of a loop and $k$ is the bound on this loop. The names of the loops can be checked with the option `--show-loops`. The option `--no-unwinding-assertions` prevents CBMC from generating unwinding assertions. Unwinding assertions check whether the loops are fully unwound. All our programs have an unconditional loop to model the cyclic manner of the PLC program. For this main cycle we do not want the unwinding assertions. It is not possible to generate unwinding assertions for only some loops, so we have not used the unwinding assertions.

For the experiments we have used $k = 3$. For all properties that are violated in the programs, a violation is found within this bound.

## 5.3 K-Inductor

K-Inductor [22] is a verification tool that is built on top of the CBMC tool. This tool uses k-induction to prove properties. For the experiments we have used K-Inductor version 1.0.

We will take a short look at k-induction, a more complete description can be found in the paper by De Moura et al. [21].
First we will look at the traditional induction. To prove a property $p$ with induction we have to prove the base case and the step case. Let us say that if a property $p$ holds in iteration $i$ then $p_i$ holds. We have to prove the following:

- **base case** $p_0$
- **step case** $p_n \implies p_{n+1}$

For the base case we have to prove that $p$ holds in the first iteration. For the step case we can assume that $p$ holds for a loop iteration and we have to prove that $p$ holds for the next loop iteration.

For k-induction we get the following base case and step cases for a given value of $k$.

- **base case** $p_0 \dots p_{k-1}$
- **step case** $p_n \dots p_{n+k-1} \implies p_{n+k}$

For the base case we now have to prove that $p$ holds for the first $k$ iterations of the loop. For the step case we assume that $p$ holds for $k$ consecutive loop iterations and we have to prove that $p$ holds for the next loop iteration. Note that when we take $k = 1$ we get the traditional induction.

K-induction is more powerful than normal induction. We show this with another example, taken from [21].

**Example 5.1**

```
int main(){
    int a = 1;
    int b = 2;
    int c = 3;
    int temp = 0;
    while(true){
        temp = a;
        a = b;
        b = c;
        c = temp;
        assert(a != b);
    }
}
```

Listing 5.4: Example k-induction

Consider the C code in Listing 5.4. Additionally, let us say that the values of $a$, $b$, and $c$ in iteration $i$ have values $a_i$, $b_i$, and $c_i$ respectively. When we execute the program for one loop iteration we get: $a_{i+1} = b_i$, $b_{i+1} = c_i$, and $c_{i+1} = a_i$.

We will first try to prove this with the traditional induction. For the base case we have to prove that $a_0 \neq b_0$. We know that $a_0 = 1$ and $b_0 = 2$ so $a_0 \neq b_0$.

For the step case we have to prove $a_i \neq b_i \implies a_{i+1} \neq b_{i+1}$. This is not the case. Consider $a_i = 1$, $b_i = 2$, $c_i = 2$. Here $a_i \neq b_i$, but if we execute one iteration of the loop we get $a_{i+1} = 2$ and $b_{i+1} = 2$, thus $a_{i+1} \neq b_{i+1}$ does not hold.

We will now prove this property with k-induction. Consider $k = 3$. For the base cases we have to prove that $a_0 \neq b_0$, $a_1 \neq b_1$, and $a_2 \neq b_2$. We know that $a_0 = 1$, $b_0 = 2$, and $c_0 = 3$. With this we can already see that $a_0 \neq b_0$ holds. When we go through the loop for one iteration we get $a_1 = b_0 = 2$ and $b_1 = c_0 = 3$, which gives us $a_1 \neq b_1$. With the next iteration we get $a_2 = b_1 = 3$ and $b_2 = c_1 = a_0 = 1$, which gives us $a_2 \neq b_2$.

For the step case we have to prove that $(a_i \neq b_i) \wedge (a_{i+1} \neq b_{i+1}) \wedge (a_{i+2} \neq b_{i+2}) \implies a_{i+3} \neq b_{i+3}$. We can prove $a_{i+3} \neq b_{i+3}$ as follows: We have $a_{i+3} = b_{i+2} = c_{i+1} = a_i$ and $b_{i+3} = c_{i+2} = a_{i+1} = b_i$ this gives us $a_i \neq b_i$. Because this negation is already in the left hand side of the implication, we have now proven the property.

While k-induction is stronger than normal induction, still not all properties can be proven with k-induction. We show this with an example.

**Example 5.2** Consider a cyclic program with an integer variable $x$. The initial value of $x$ is 0 and in each iteration $x$ is increased by 2. We want to prove the property $x \neq 3$ in every iteration.

For the step case we have to prove $(x_i \neq 3 \wedge \cdots \wedge x_{i+k-1} \neq 3) \implies x_{i+k} \neq 3$. We can rewrite this to $(x_i \neq 3 \wedge \cdots \wedge x_i + 2(k-1) \neq 3) \implies x_i + 2k \neq 3$. A counterexample can be found for this implication for any value of $k$. Consider $x_i = 3 - 2k$. This will give us $x_{i+k} = x_i + 2k = 3$, which is a violation of the property.

Note that if we would make the property stronger by adding that $x \geq 0$ we would be able to prove the property. However, most tools, K-inductor included, cannot automatically strengthen properties and thus require human intelligence to prove such properties.

**Verification** Properties are added in the same way as with the BMC tool of CBMC. To perform verification we use the following command in the command line:
`kinductor --max-k k Example.c` where $k$ is the maximum $k$ for the k-induction.

If K-Inductor cannot prove the given property with k-induction for values for $k$ up to the given value, it will say: "Result is bad". This result is given when the property is refuted within $k$ iterations as well as when the property holds for the model but the tool is unable to prove this. If a violation of the property is found within $k$ iterations, a counterexample can be given with the option `--show-step-case-fails`. This is the same counterexample as for the BMC tool of CBMC. Note that also when this option is used and the tool is unable to prove the property, a trace of the program will be given. Because the difference in these situations is not very clear, we have only used K-Inductor to prove properties.

## 5.4 CBMC Incremental

This CBMC tool combines the bounded model checking from CBMC with the k-induction from K-Inductor. In addition it also adds incremental loop unwinding [32], so that the user does not have to give a bound for the bounded model checking or a value $k$ for the k-induction. For the experiments we have used CBMC version 5.2 with incremental loop unwinding. Note that this is a different tool than the CBMC tool we discussed before.

Adding a property is more difficult for this tool than for the other tools in this group. The code has to be instrumented in two places: code must be added for the base case and code must be added to accommodate the step case. An example is given in Listing 5.5.

```
int nondet_int();

int main(){
    int a, b, c, temp;
#ifdef BASE
        a = 1;
        b = 2;
        c = 3;
        temp = 0;
#endif

    while(true){

#ifdef STEP
            a = nondet_int();
            b = nondet_int();
            c = nondet_int();
            temp = nondet_int();
19                      __CPROVER_assume(a != b);
20  #endif
21
22          temp = a;
23          a = b;
24          b = c;
25          c = temp;
26          assert(a != b);
27      }
28  }
```

Listing 5.5: Base case and Step case in CBMC Incremental

- **Base case** The code for the base case is placed between `#ifdef BASE` and `#endif`, see lines 3-8 in the example above. In this part we place the initialization of all variables in the code, which results in a separate declaration and initialization of the variable. This part of the code is placed directly after the declaration of the variables;

- **Step case** The code for the step case is placed between `#ifdef STEP` and `#endif`, see lines 12-18 in the example above. In this part we assign every variable in the code a non-deterministic value. Next we create an assumption using the property with the following statement: `__CPROVER_assume(`*property*`);`. This part of the code is placed immediately after the beginning of the main loop of the program;

- **Assertion** The property is also added as assertion at the end of the code, in the same way as with the BMC tool and the K-Inductor tool of CBMC, see line 24 in the example above.

**Verification** The verification with this tool is done in two steps. We have to perform verification on the base case and on the step case. This is done with the following commands in the command line:

```
cbmc Example.c --incremental -DBASE
cbmc Example.c --incremental --stop-when-unsat -DSTEP
```

The base case verification terminates as soon as it finds a violation of the property, it gives a counterexample in the same way as the CBMC tool without Incremental unwinding. The step case verification terminates as soon as it proves the property. When both processes are run in parallel and one process terminates; we know whether the given property holds in the model. If the base case process terminated; we know that the property is violated. If the step case process terminated; we know that the property is proven. If the tool cannot prove or refute the property; both processes will not stop.

When a program has multiple loops incremental unwinding is used for all loops. It is also possible to check only one loop. When this is done the other loops should be given a bound. This is done by the options `--incremental check` *loopid* and `--unwind` *k*. Where *loopid* is the id of the loop which we want to check with incremental unwinding and *k* is the bound on the other loops.

## 5.5 2LS

2LS (2nd order Logic Solving) [10] is a tool for program analysis. 2LS uses the CPROVER infrastructure provided by CBMC. Like CBMC Incremental, 2LS uses bounded model checking and k-induction; for a description of k-induction see Section 5.3. The tool also supports an algorithm called *k*I*k*I. For the experiments we have used 2LS version 0.3.4.

**kIkI** The *k*I*k*I algorithm [10] combines k-induction, bounded model checking, and abstract interpretation. First the property is checked for the initial states. If there is no error a k-inductive invariant is generated. The algorithm attempts to prove the property with k-induction and the k-invariant. If there is a possible error state, bounded model checking is used to check if it this state can be reached. If it can be reached it finds a counterexample. If the error state cannot be reached in $k$ iterations, $k$ is incremented and a stronger k-invariant can be found. The algorithm loops until the invariant proves safety or until a counterexample is found.

**Verification** Properties are added in the same way as with CBMC and K-Inductor. This is done with `assert(`*property*`);`.
To run a verification with 2LS we use the following command in the command line.
```
2ls Example.c --k-induction --havoc
```
The option `--havoc` removes the loops and function calls from the model. To run the verification with the *k*I*k*I algorithm only the option `--k-induction` is used. The tool gives a counterexample if a property is refuted and the option `-show-trace` is used. Counterexamples are given in the same way as with the CBMC tool and the CBMC Incremental tool.

## 5.6 CPA-checker

CPA-checker [5] is a configurable software verification tool. In the last five years, the tool has won multiple prizes in the Competition on Software Verification held at the TACAS conference. For the experiments we have used CPA-checker version 1.5.

CPA-checker uses Configurable Process Analysis (CPA) [4]. This technique combines model checking with program analysis that automatically makes abstractions of a program. It is also used to analyze these abstraction. With CPA-checker different verification techniques can be expressed in the same formal setting. This can be very useful for experiments and comparisons.

Predicate Analysis For the experiments we have used the configuration `predicateAnalysis`. Adjustable-block encoding (ABE) [6] is used to make an abstraction of the model. ABE combines single-block encoding (SBE) with large-block encoding (LBE). In SBE abstractions are computed after every single program operation, where in LBE abstractions are only computed after a large number of operations. The abstraction is sound, i.e. it is done such that if a property holds on the abstract model, it also holds on the original program. The abstraction is checked with an SMT solver; in this case SMTInterpol. Because we use this abstraction a counterexample could be found that is not a counterexample on the original program. This is called a spurious counterexample. If a spurious counterexample is found, counterexample-guided abstraction refinement (CEGAR) [15] is used to refine the abstraction such that the counterexample is eliminated. This continues until the given property is proven or refuted.

In other configurations, other SMT solvers can be used, as well as other types of verifiers such as SAT based verification and BDD based verification. We have experimented with some more configurations. The configuration `predicateAnalysis` gave us the best results. Some other configurations that also used predicate analysis gave similar results and a few other configurations were significant slower especially on TRUE-properties. The configuration `CBMC` did not give us any results and the configuration `bddAnalysis` gave us wrong results. The configuration `bmc` gave similar running times as `predicateAnalysis` for the FALSE-properties, but could not prove the TRUE-properties.

**Verification** Properties in CPA-checker are added with an `if`-statement. The condition on the statement is the negation of the property. If the condition is satisfied, and thus the property results in false, the program uses a `goto`-statement to reach an `ERROR` state. This results in the following piece of code for the properties:

```
if(! property ){
   goto ERROR;
}
```

In the `ERROR` state we return −1. The structure of the program is shown in Listing 5.6.

```
#include <stdbool.h>
function declarations
int main(){
        Variable declaration
        while(true){
                Non-deterministic input
                program body
                properties
        }
        ERROR:
        return (-1);
}
```

Listing 5.6: Structure of a program in CPA-checker

Before we can run a verification with CPA-checker, the program should be preprocessed. This is done with the command `cpp` in the command line. We can now run the verification with `cpa.sh -predicateAnalysis Example.c`. The verification will run with the given configuration. The configuration files are given in the download of CPA-checker. To run the verification with other configurations, `predicateAnalysis` should be replaced by the other configuration.

The verification of CPA-checker creates multiple output files. If the property is violated there will be a file with the error trace. Other files include a visualization of the control flow automaton, coverage information, and time statistics. To prevent CPA-checker from creating these output files we add the following line to the configuration file: `output.disable = true`.

## 5.7 SATABS

SATABS [18] is a verification tool that works similarly to CPA-checker. For the experiments we have used SATABS version 3.2.

Like CPA-checker, SATABS makes an abstraction of the model, checks the abstraction with a model checker, and uses CEGAR to find spurious counterexamples and to refine the abstraction. Unlike CPA-checker, SATABS uses a SAT solver instead of a theorem prover to make a boolean program, which is the abstraction of the model. It also uses a SAT solver to refine the abstraction.

For the model checking part a number of model checkers are supported. For the experiments we have used NuSMV.

**Verification** Properties are added in the same way as with CBMC, K-Inductor, and 2LS. To do this we add the following line of code to the program: `assert(`*property*`);`. To run a verification with SATABS, we use the following commands in the command line.
`SATABS Example.c --modelchecker nusmv`
The SATABS tool uses NuSMV with the option -dynamic. The running times of SATABS might improve if it would use NuSMV with the options -df -dynamic -coi, but it seems unable to use these options for NuSMV with SATABS.

SATABS gives counterexamples in the same way as CBMC, CBMC Incremental, and 2LS.

## 5.8 Experiments and results

We have verified the example programs with each of the tools. For each example we have made two translations, a full translation of the program and a reduced version, as explained in Section 3.2. The running times of the experiments can be found in Table 5. All running times are given as the user time + the system time. The only tool that automatically uses multiple cores is CPA-checker, but to make a fair comparison we have prevented this by putting `taskset -c 0` in front of the command to run the verification with CPA-checker. The results of CPA-checker with and without usage of multiple cores can be seen in Table 4. Here the real times are given because the user times of the verifications with multiple cores include an summation of the running times of every used core. We can see that for most programs the running time was faster when using only a single core. When using multiple cores, CPA-checker did not manage to prove property one for `Example_int` and `Example_while` within the CPU-time limit of 900 seconds. This is the default limit of CPA-checker.

For K-Inductor we only have a result for the second property. This is because the tool cannot refute properties and it was unable to prove the first property. For CBMC we have not included the running times for the TRUE-properties, since this tool can only refute properties. For CBMC Incremental and 2LS we have stopped the verification process after 900 seconds, which is the same limit as the default limit of CPA-checker. We have experimented with longer running times to make sure that these tools are not able to verify the given properties.

When we look at the running times we can see that CBMC, CBMC Incremental and 2LS have similar running times which mostly are under a second. CPA-checker and SATABS have longer running times, but are able to prove all TRUE-properties. We can see that for our properties, the performances of 2LS did not improve when the $kIk$I algorithm is used.

The counterexamples of all of the tools in this section can be mapped back to the original SCL code.

The translation from SCL into C did not give many problems and almost all statements could be translated straightforwardly.

| Program | Property | CPA-checker multi-core | CPA-checker single-core |
|---|---|---|---|
| Example | 1(true) | **35.14s** | 52.23s |
| | 2(true) | 13.55s | **9.65s** |
| | 3(false) | 14.75s | **12.25s** |
| | 4(false) | **17.73s** | 19.78s |
| Example reduced | 1(true) | **23.59s** | 33.47s |
| | 2(true) | 11.21s | **9.31s** |
| | 3(false) | 13.57s | **10.60s** |
| | 4(false) | 14.49s | **11.99s** |
| Example Int | 1(true) | - | **12m52.55s** |
| | 2(true) | 12.70s | **9.04s** |
| | 3(false) | 14.50s | **12.51s** |
| | 4(false) | **18.09s** | 22.97s |
| Example Int reduced | 1(true) | 15.87s | **13.622s** |
| | 2(true) | 11.95s | **9.20s** |
| | 3(false) | 12.52s | **10.85s** |
| | 4(false) | 15.31s | **12.55s** |
| Example While | 1(true) | - | **2m34.81s** |
| | 2(true) | 11.51s | **9.15s** |
| | 3(false) | 15.16s | **12.73s** |
| | 4(false) | 14.95s | **13.38s** |
| Example While reduced | 1(true) | **54.25s** | 1m19.46s |
| | 2(true) | 12.29s | **8.48s** |
| | 3(false) | 13.77s | **11.14s** |
| | 4(false) | 14.05s | **11.26s** |

Tab. 4: Running times of CPA-checker with and without usage of multi-cores

When we look at all tools we can see that nuXmv has the best running times on the TRUE-properties. For the FALSE-properties we can see that the running times of NuSMV, nuXmv, CBMC, CBMC Incremental, and 2LS are all within a couple of seconds and mostly under a second.

| Program | Property | CBMC | K-Inductor | CBMC Incremental | 2LS | | CPA-checker | SATABS |
|---|---|---|---|---|---|---|---|---|
| | | | | | k-induction | *k*I*k*I | | |
| Example | 1(true) | - | - | - | - | - | **52.23s** | 1m9.76s |
| | 2(true) | - | 0.52s | **0.42s** | 0.48s | 0.43s | 9.65s | 0.61s |
| | 3(false) | **0.42s** | - | 0.43s | 0.51s | 0.64s | 12.25s | 41.71s |
| | 4(false) | **0.45s** | - | 0.47s | 0.51s | 0.83s | 19.78s | 1m8.33s |
| Example reduced | 1(true) | - | - | - | - | - | 33.47s | **23.26s** |
| | 2(true) | - | 0.50s | **0.41s** | 0.56s | 0.43s | 9.31s | 0.67s |
| | 3(false) | 0.43s | - | **0.41s** | 0.45s | 0.53s | 10.60s | 8.78s |
| | 4(false) | 0.41s | - | **0.40s** | 0.51s | 0.65s | 11.99s | 10.80s |
| Example Int | 1(true) | - | - | - | - | - | 12m52.55s | **1m21.85s** |
| | 2(true) | - | 0.51s | 0.46s | 0.50s | **0.45s** | 9.04s | 0.59s |
| | 3(false) | **0.43s** | - | **0.43s** | 0.49s | 0.66s | 12.51s | 44.80s |
| | 4(false) | **0.44s** | - | 0.46s | 0.53s | 0.88s | 22.97s | 1m22.14s |
| Example Int reduced | 1(true) | - | - | - | - | - | **13.622s** | 27.51s |
| | 2(true) | - | 0.47s | 0.41s | 0.45s | **0.35s** | 9.20s | 0.70s |
| | 3(false) | 0.43s | - | **0.42s** | 0.48s | 0.51s | 10.85s | 7.92s |
| | 4(false) | 0.42s | - | **0.41s** | 0.49s | 0.67s | 12.55s | 13.87s |
| Example While | 1(true) | - | - | - | - | - | 2m34.81s | **1m24.40s** |
| | 2(true) | - | 0.54s | - | 0.45s | **0.37s** | 9.15s | 0.63s |
| | 3(false) | **0.44s** | - | 0.45s | 0.51s | 0.75s | 12.73s | 43.89s |
| | 4(false) | 0.45s | - | **0.41s** | 0.55s | 1.14s | 13.38s | 1m16.11s |
| Example While reduced | 1(true) | - | - | - | - | - | 1m19.46s | **30.14s** |
| | 2(true) | - | 0.49s | - | 0.44s | **0.43s** | 8.48s | 0.59s |
| | 3(false) | 0.43s | - | **0.41s** | 0.48s | 0.65s | 11.14s | 8.21s |
| | 4(false) | 0.44s | - | **0.43s** | 0.51s | 0.95s | 11.26s | 12.76s |

Tab. 5: Running times of the verification of the tools on code of the example programs

**Additional experiments** We have done some more experiments to figure out when k-induction is able to prove properties. We have reduced the TRUE-property from the examples to get a small example. The program is shown in Listing 5.7. In Listing 5.8 we have made a small change to this program by removing the variable $z$ and the `if`-statement at lines 22-26 and replacing these lines with `a = true;`. The property holds for both programs.

```
1  #include <stdbool.h>
2  bool nondet_bool();
3
4  int main(){
5       bool a = false;
6       bool b = false;
7       bool x = false;
8       bool y = false;
9       bool z = false;
10
11      while(true){
12
13          x = nondet_bool();
14          z = nondet_bool();
15
16          if(!x){
17              a = false;
18              b = false;
19          } else if(!y){
20              b = !b;
21          } else{
22              if(z){
23                  a = true;
24              } else{
25                  a = false;
26              }
27          }
28          y = x;
29
30          assert(!a || b);
31      }
32 }
```

Listing 5.7: small example

```
1  #include <stdbool.h>
2  bool nondet_bool();
3
4  int main(){
5       bool a = false;
6       bool b = false;
7       bool x = false;
8       bool y = false;
9
10      while(true){
11
12          x = nondet_bool();
13
14          if(!x){
15              a = false;
16              b = false;
17          } else if(!y){
18              b = !b;
19          } else{
20              a = true;
21          }
22          y = x;
23
24          assert(!a || b);
25      }
26 }
```

Listing 5.8: the same example with a few changes

K-Inductor, CBMC Incremental, and 2LS could not prove the property in Listing 5.7. 2LS is able to prove the property in Listing 5.8, but K-Inductor and CMBC Incremental still are not. However in Example 5.3 we show a proof for this property with k-induction. With some additions this proof can also be used for the program in Listing 5.7. This shows that the restrictions of the tools are not in k-induction but in the implementation.

**Example 5.3** For the program in Listing 5.8 we give a proof for $k = 3$. Note that for the property we want to prove, $k = 3$ is also the smallest $k$ for which a successful k-induction proof can be given. Let us say that the values of $a$, $b$, $x$, and $y$ in iteration $i$ have values $a_i$, $b_i$, $x_i$, and $y_i$ respectively. For k-induction we have to prove the following cases:

- **base case** $(\neg a_0 \vee b_0) \wedge (\neg a_1 \vee b_1) \wedge (\neg a_2 \vee b_2)$

- **step case** $(\neg a_i \vee b_i) \wedge (\neg a_{i+1} \vee b_{i+1}) \wedge (\neg a_{i+2} \vee b_{i+2}) \implies (\neg a_{i+3} \vee b_{i+3})$

When we look at the program we get the following equations for the values of $a_{i+1}$, $b_{i+1}$, and $y_{i+1}$. The value of $x_{i+1}$ is always a non-deterministic value.

$$a_{i+1} = \begin{cases} \text{if } \neg x_{i+1} & false \\ \text{if } x_{i+1} \wedge \neg y_i & a_i \\ \text{if } x_{i+1} \wedge y_i & true \end{cases} \tag{1}$$

$$b_{i+1} = \begin{cases} \text{if } \neg x_{i+1} & false \\ \text{if } x_{i+1} \wedge \neg y_i & \neg b_i \\ \text{if } x_{i+1} \wedge y_i & b_i \end{cases} \tag{2}$$

$$y_{i+1} = x_{i+1} \tag{3}$$

We also know the initial values: $a_0 = false$, $b_0 = false$, $x_0 = false$, and $y_0 = false$.

From the first and second equations we can obtain the following formulas.

$a_{i+1} \iff (x_{i+1} \wedge \neg y_i \wedge a_i) \vee (x_{i+1} \wedge y_i)$
$b_{i+1} \iff (x_{i+1} \wedge \neg y_i \wedge \neg b_i) \vee (x_{i+1} \wedge y_i \wedge b_i)$

Base case We have three cases within the base case. The first case is $(\neg a_0 \vee b_0)$. When we fill in these values we get $(\neg false \vee false)$ which is *true*.

The second case is $(\neg a_1 \vee b_1)$. For this case we use the equations and we fill in the initial values. This gives us the following equations for $a_1$ and $b_1$.

$$a_1 = \begin{cases} \text{if } \neg x_1 & false \\ \text{if } x_1 & false \end{cases} \tag{4}$$

$$b_1 = \begin{cases} \text{if } \neg x_1 & false \\ \text{if } x_1 & true \end{cases} \tag{5}$$

As we can see the value of $a_1$ is $false$ and the value of $b_1$ is unknown. If we fill in the value of $a_1$ in the second case property we get $(\neg false \vee b_1)$ which is *true* regardless of the value of $b_1$.

The third case is $(\neg a_2 \vee b_2)$. We will again use the equations. We can fill in the initial values and the value of $a_1$. We can also use the equations for $y_{i+1}$ and $b_1$ to get only the unknown values of $x$ in the equations. We get the following equations for $a_2$ and $b_2$.

$$a_2 = \begin{cases} \text{if } \neg x_2 & false \\ \text{if } x_2 \wedge \neg x_1 & false \\ \text{if } x_2 \wedge x_i & true \end{cases} \tag{6}$$

$$b_2 = \begin{cases} \text{if } \neg x_2 & false \\ \text{if } x_2 \wedge \neg x_1 & true \\ \text{if } x_2 \wedge x_i & true \end{cases} \tag{7}$$

This gives us three possibilities for the values of $a_2$ and $b_2$. They can both be $false$, both be *true*, or $a_2 = false$ and $b_2 = true$. For all three possibilities $(\neg a_2 \vee b_2)$ is *true*.

**Step case** For the step case we will make equations for the values of $a_{i+2}$, $b_{i+2}$, $y_{i+2}$, $a_{i+3}$, $b_{i+3}$, and $y_{i+3}$.

$$a_{i+2} = \begin{cases} \text{if } \neg x_{i+2} & false \\ \text{if } x_{i+2} \wedge \neg x_{i+1} & false \\ \text{if } x_{i+2} \wedge x_{i+1} & true \end{cases} \tag{8}$$

$$b_{i+2} = \begin{cases} \text{if } \neg x_{i+2} & false \\ \text{if } x_{i+2} \wedge \neg x_{i+1} & true \\ \text{if } x_{i+2} \wedge x_{i+1} \wedge \neg y_i & \neg b_i \\ \text{if } x_{i+2} \wedge x_{i+1} \wedge y_i & b_i \end{cases} \tag{9}$$

$$y_{i+2} = x_{i+2} \tag{10}$$

$$a_{i+3} = \begin{cases} \text{if } \neg x_{i+3} & false \\ \text{if } x_{i+3} \wedge \neg x_{i+2} & false \\ \text{if } x_{i+3} \wedge x_{i+2} & true \end{cases} \tag{11}$$

$$b_{i+3} = \begin{cases} \text{if } \neg x_{i+3} & false \\ \text{if } x_{i+3} \wedge \neg x_{i+2} & true \\ \text{if } x_{i+3} \wedge x_{i+2} \wedge \neg x_{i+1} & true \\ \text{if } x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge \neg y_i & \neg b_i \\ \text{if } x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge y_i & b_i \end{cases} \tag{12}$$

$$y_{i+3} = x_{i+3} \tag{13}$$

We can write the equations for values of $a$ and $b$ in formulas.
From equations 8, 9, 11, and 12 we get the following formulas.
$a_{i+2} \iff (x_{i+2} \wedge x_{i+1})$
$b_{i+2} \iff (x_{i+2} \wedge \neg x_{i+1}) \vee (x_{i+2} \wedge x_{i+1} \wedge \neg y_i \wedge \neg b_i) \vee (x_{i+2} \wedge x_{i+1} \wedge y_i \wedge b_i)$
$a_{i+3} \iff (x_{i+3} \wedge x_{i+2})$
$b_{i+3} \iff (x_{i+3} \wedge \neg x_{i+2}) \vee (x_{i+3} \wedge x_{i+2} \wedge \neg x_{i+1}) \vee (x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge \neg y_i \wedge \neg b_i) \vee (x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge y_i \wedge b_i)$

With these formulas and the formulas for $a_{i+1}$ and $b_{i+1}$ we can rewrite $(\neg a_{i+2} \vee b_{i+2})$.
$(\neg a_{i+2} \vee b_{i+2})$
$\iff$
$(\neg(x_{i+2} \wedge x_{i+1}) \vee ((x_{i+2} \wedge \neg x_{i+1}) \vee (x_{i+2} \wedge x_{i+1} \wedge \neg y_i \wedge \neg b_i) \vee (x_{i+2} \wedge x_{i+1} \wedge y_i \wedge b_i)))$
$\iff$
$(\neg x_{i+2} \vee \neg x_{i+1} \vee \neg x_{i+1} \vee (\neg y_i \wedge \neg b_i) \vee (y_i \wedge b_i))$

Likewise for $(\neg a_{i+3} \vee b_{i+3})$.
$(\neg a_{i+3} \vee b_{i+3})$
$\iff$
$(\neg(x_{i+3} \wedge x_{i+2}) \vee ((x_{i+3} \wedge \neg x_{i+2}) \vee (x_{i+3} \wedge x_{i+2} \wedge \neg x_{i+1}) \vee (x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge \neg y_i \wedge \neg b_i) \vee (x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge y_i \wedge b_i)))$
$\iff$

$(\neg x_{i+3} \vee \neg x_{i+2} \vee \neg x_{i+1} \vee (\neg y_i \wedge \neg b_i) \vee (y_i \wedge b_i))$

With these formulas we can see that $(\neg a_{i+3} \vee b_{i+3}) \iff (\neg x_{i+3} \vee (\neg a_{i+2} \vee b_{i+2}))$. From this it follows that $(\neg a_{i+2} \vee b_{i+2}) \implies (\neg a_{i+3} \vee b_{i+3})$, which also means that $(\neg a_i \vee b_i) \wedge (\neg a_{i+1} \vee b_{i+1}) \wedge (\neg a_{i+2} \vee b_{i+2}) \implies (\neg a_{i+3} \vee b_{i+3})$. This concludes our proof.

**Example 5.4** We have also tried to verify Example 5.2 from section 5.3. Let us recall the example. We have a cyclic program with an integer variable $x$. Initially $x = 0$ and in every iteration we get $x = x + 2$. The property we want to prove is $x \neq 3$. We have run the verification of this program with K-Inductor, CBMC Incremental, and 2LS. To prevent the tools from using an overflow of the integer values, we have added an `if`-statement. The C program can be found in Listing 5.9.

```c
#include <stdbool.h>

int main(){
    int x = 0;

    while(true){
        if(x > 1000) {
            x = 1000;
        }
        x = x + 2;

        assert(x != 3);
    }
}
```

Listing 5.9: small example

As expected, none of these tools could prove this property with k-induction. However 2LS can prove this property with its $k$I$k$I algorithm. We have strengthened the property to $x \geq 0 \wedge x \neq 3$. 2LS is able to prove this property with k-induction, while K-Inductor and CBMC Incremental still cannot. When we replace the property by $x \geq 0$ CBMC Incremental can also prove the property, but K-Inductor still is not able to do this.

The examples above show us that the tools have used different implementations of k-induction and that not all implementations are equally strong.

# 6 Case Study

As discussed in Section 3.2 we will use the most promising tools for verifying a larger program. From the first group of model checking tools we have chosen to do the case study with: NuSMV with CTL properties, NuSMV with invariant properties, and NuXmv. From the second group we use CBMC Incremental, 2LS with k-induction, 2LS with the *k*I*k*I algorithm, CPA-checker, and SATABS. These experiments are done on the program CPC [1], which can also be found in the Appendix E. We will first look at the new aspects of this program and at the translation into SMV and C code. Then we shortly discuss the translation of the properties. This section ends with the experiments and results of the case study.

## 6.1 The CPC program

The CPC program uses a few types which we have not yet seen in the example programs. Namely `WORD`, `ARRAY`, `TIME`, `REAL`, and `UINT`.

In this program a variable with the type `WORD` is always used together with a variable with type `ARRAY` in the following construction:

```
Manreg01:  WORD;
Manreg01b AT Manreg01:  ARRAY [0..15] OF BOOL;
```

The variable `Manreg01` of type `WORD` reserves 16 bits in the memory. With `Manreg01b AT Manreg01` a variable `Manreg01b` is specified at the location of `Manreg01`. The type of `Manreg01b` is `ARRAY [0..15] OF BOOL`. This specifies that this variable is actually an array of 16 boolean variables.

Variables of type `TIME` can get values of the form `T#`*a*`h_`*b*`m_`*c*`s_`*d*`ms` where *a*, *b*, *c*, and *d* are numbers, *h* defines the number of hours, *m* the number of minutes, *s* the number of seconds, and *ms* the number of milliseconds. Any of these letters can be omitted when they have a value of 0, but there should be at least one letter. For instance `T#0ms` stands for zero milliseconds.

The program consists of 822 lines of code, which includes the main `FUNCTION BLOCK`, two other `FUNCTION BLOCK`s, three `FUNCTION`s, and one user defined structure. There are 32 function calls of which 21 call a `FUNCTION` and 11 call a `FUNCTION BLOCK`. This program does not contain loops. The program has 54 input variables, 59 output variables, 91 internal variables, and 2 global variables. Variables in an array are counted separately. Variables in a `STRUCT` or in a `FUNCTION BLOCK` are included in these numbers.

Besides the main `FUNCTION BLOCK` the CPC program has two additional `FUNCTION BLOCK`s. Since we have only seen the `FUNCTION BLOCK` as the main `FUNCTION BLOCK`, we will explain this construct. A `FUNCTION BLOCK` is similar to a `FUNCTION`. In addition variables in a `FUNCTION BLOCK` can be stored in the memory while the program has returned. These variables can also have initial values. Listing 6.1 shows such a `FUNCTION BLOCK`.

```
FUNCTION_BLOCK TP
VAR_INPUT
        PT : TIME;
END_VAR

VAR_IN_OUT
        IN : BOOL;
END_VAR
```

```
 9
10  VAR_OUTPUT
11          Q : BOOL := FALSE;
12          ET : TIME; // elapsed time
13  END_VAR
14
15  VAR
16          old_in : BOOL := FALSE;
17          due : TIME := T#0ms;
18  END_VAR
19
20  BEGIN
21          if (in and not old_in) and not Q then
22                  due := __GLOBAL_TIME + pt;
23          end_if;
24          if __GLOBAL_TIME <= due then
25                  Q := true;
26                  ET := PT - (due - __GLOBAL_TIME);
27          else Q := false;
28                  if in then
29                          ET := PT;
30                  else ET := 0;
31                  end_if;
32          end_if;
33          old_in := in;
34  END_FUNCTION_BLOCK
```

Listing 6.1: `FUNCTION BLOCK` in SCL

We can see that there are four types of variables in this `FUNCTION BLOCK`. We have already seen these groups of variables in Section 2.2. The input variables and the in-output variables always get their values from the parameters of the function call. The output variables and the static variables in a `FUNCTION BLOCK` keep their value in the memory after the function has returned. We can see that some of these variables have an initial value (lines 12, 16, 17). To use a `FUNCTION BLOCK` we have to declare a variable with the name of the `FUNCTION BLOCK` as type. If there are multiple variable with this `FUNCTION BLOCK` as type, each instance gets their own variables. An example of the declaration of a `FUNCTION BLOCK` is shown below.

**Example** Consider the `FUNCTION BLOCK TP` as shown in Listing 6.1. We will declare a variable with this block as type with: `Timer_PulseOn: TP;`. We can now use the following variables in our code: `Timer_PulseOn.Q`, `Timer_PulseOn.ET`, `Timer_PulseOn.old_in`, and `Timer_PulseOn.due`. Inside the function these variables are used without the prefix `Timer_PulseOn`, i.e. with `Q`, `ET`, `old_in`, and `due`.

Another new aspect are timers. The implementation of timers in the SCL code is with a global `TIME` variable `__GLOBAL_TIME`. After each cycle, this variable is increased with the value of the global variable `T_CYCLE`.

## 6.2 Translation into SMV

We have made a translation of the CPC program into SMV in the same way as described in Section 4.2. The full program can be found in the Appendix F. This translation contains 1912 lines of code and 224 values of the location variable. The types that we have not yet seen in that translation are translated as follows. The combination of a variable of type `WORD` and a variable of type `ARRAY` in SCL, as seen earlier in this section, can be translated to a variable of type `array` in SMV. To make an array of 16 boolean variables `array 0..15 of boolean` is used. Variables of the type `TIME` and type `REAL` in SCL are represented by variables of the type `signed word[32]` in SMV and the variable of the type `UINT` in SCL is represented by a variable of the type `unsigned word[16]` in SMV.

At some points in the code a comparison or operation is done with variables of different "lengths". To be able to compare variables of different lengths of the type `word` in SMV, we have used the standard SMV operator `extend(`*`variable`*`,`*`size`*`)` to scale the variables. Here *`variable`* is the variable we want to scale and *`size`* is the size with which we want to extend this variable. For instance if we want to know if the variable `FSIinc` of type `signed word[16]` is larger than a variable `PulseWidth` of type `singed word[32]` we will get the following piece of code: `extend(FSIinc,16) > PulseWidth`.

To model the timer, we give the variable `T_CYCLE` a non-deterministic value between 5 and 100 in each cycle. We have chosen these values following the same convention as used at CERN [24]. To get a value in this range we have used an additional variable `random_t_cycle` with type `unsigned word[8]` which we scale to be inside the range. For the next state of `T_CYCLE` we get the following case in the case distinction:

```
(loc = start) :  ((extend(random_t_cycle,8)) mod 0ud16_95 + 0ud16_5);
```

To model a `FUNCTION BLOCK` in SMV we create the variables of the `FUNCTION BLOCK` for each variable with the `FUNCTION BLOCK` as type. The function part of the `FUNCTION BLOCK` is modeled by substituting the function calls by the code of the function in the same way as we have seen for a `FUNCTION`.

## 6.3 Translation into C

The translation of the CPC program into C is done in the same way as described in Section 5.1. The full program can be found in Appendix G. This translation has 779 lines of code, three structs, and five functions. For the new variable types the following C types are used: For a `WORD` in combination with an `ARRAY` in SCL, as seen earlier in this section, an array is used in C. To declare an array of 16 boolean variables `bool `*`variable name`*`[16];` is used. Variables of the types `TIME`, `REAL`, and `UINT` in SCL are modeled with the types `int`, `double`, and `unsigned short`, respectively.

Similar to the SMV translation, we model the timer variable `T_CYCLE` by giving it a non-deterministic value between 5 and 100 in each cycle. To do this we use the following line of code: `T_CYCLE = 5 + (nondet_unsignedshort() % 95);`.

To model a `FUNCTION BLOCK` from SCL code we have used a struct in combination with a function. The struct is used to create the variables for each variable with the `FUNCTION BLOCK` as type. For the `FUNCTION BLOCK` in Listing 6.1 this gives us the following Struct:

```
struct TP{
   bool Q;
   int ET;
   bool old_in;
   int due;
};
```

In the declaration and initialization of the variables this gives us the following:

```
struct TP Timer_PulseOn;
```

```
Timer_PulseOn.Q = false;
Timer_PulseOn.old_in = false;
Timer_PulseOn.due = 0;
```

The body of the `FUNCTION BLOCK` is translated in the same way as a `FUNCTION`.

## 6.4 Properties

A list of the properties for this code can be found in the Appendix H. The properties of the CPC program differ slightly from the properties in the example programs. Some properties are not only based on the current value of variables but also on the value of variables at the end of the previous cycle and on the value of variables at the beginning of the current cycle.

To keep track of these values we have chosen to add some additional variables. Variables that are used to model a variable at the beginning of the current cycle or at the end of the previous cycle have a prefix. We have chosen not to use `old` because this has already been used in the other variables. Instead we have chosen to use an `'s'` as a prefix for the **s**tart of the current cycle and an `'p'` as a prefix for the end of the **p**revious cycle. For instance for the variable `AuAuMoR` the value at the start of the cycle is needed. To have this value we have added a variable `sAuAuMoR` which gets the value of `AuAuMoR` immediately after it has got its non-deterministic value. For the variable `TStopI` the value at the previous cycle is needed. To have this value we have added a variable `pTStopI` which gets the value of `TStopI` at the end of the cycle. In the SMV code we have added a location `pvar` between the locations `end` and `start` where these variables are assigned. In the C code these values are assigned at the beginning of the cycle, before the non-deterministic assignments. Properties that consider the value of variables in the previous cycle should not be verified in the first cycle. To do this we have used a variable `first` that states whether or not this is the first cycle. We have added this variable with a disjunction to the properties where needed.

## 6.5 Experiments and results

Table 6 shows the running times of the case study. In this case study we have given all processes a maximal running time of 15 minutes.

When we compare the results of our experiments we can see that CBMC Incremental had the best running times for all properties. We can also see that there are four properties which could not be proven by any of the used tools. Note that while CBMC Incremental and 2LS could not prove all TRUE-properties in the examples, both tools can prove most of the TRUE-properties in this program. CPA-checker only got results on some, but not all, TRUE-properties, while for the example program it could prove and disprove all properties. SATABS could not verify or refute any of the properties within the given time. For 2LS we can see that the $k$I$k$I algorithm has a significant larger running time than the k-induction algorithm for a number of FALSE-properties. While NuSMV and nuXmv had better running times on the `Example` programs, only the Invariant verification of NuSMV could refute a few properties within the given time.

| **Property** | **2LS** | | **CBMC** | **CPA-checker** | **SATABS** | **NuSMV** | | **nuXmv** |
|---|---|---|---|---|---|---|---|---|
| | **k-induction** | ***k*I*k*I** | **Incremental** | | | **CTL** | **Invariants** | |
| 1-1(false) | 2.06s | 15.49s | **1.01s** | - | - | - | - | - |
| 1-2(true) | - | - | - | - | - | - | - | - |
| 1-3(false) | 2.12s | 13.62s | **0.99s** | - | - | - | - | - |
| 1-4(false) | 2.07s | 14.04s | **1.02s** | - | - | - | - | - |
| 1-5(false) | 1.74s | 14.13s | **1.02s** | - | - | - | - | - |
| 1-6(true) | - | - | - | - | - | - | - | - |
| 1-7(true) | 1.17s | 1.15s | **0.70s** | 19.90s | - | - | - | - |
| 1-8(true) | 1.12s | 1.26s | **0.73s** | 18.83s | - | - | - | - |
| 1-9(true) | 1.17s | 1.18s | **0.71s** | 20.60s | - | - | - | - |
| 1-11a(false) | 1.55s | 6.56s | **0.83s** | - | - | - | - | - |
| 1-11b(false) | 1.58s | 5.76s | **0.86s** | - | - | - | - | - |
| 2-1(true) | 1.12s | 1.22s | **0.67s** | 19.56s | - | - | - | - |
| 2-2(false) | 1.10s | 1.17s | **0.71s** | - | - | - | 6m39.52s | - |
| 2-3(true) | 1.18s | 1.20s | **0.68s** | 20.87s | - | - | - | - |
| 2-4(true) | 0.97s | 0.96s | **0.69s** | 20.28s | - | - | - | - |
| 2-5(true) | - | - | - | - | - | - | - | - |
| 2-6(true) | - | - | - | - | - | - | - | - |
| 2-7(false) | 1.23s | 1.24s | **0.69s** | - | - | - | 6m53.17s | - |
| 2-8(true) | 1.20s | 1.20s | **0.67s** | 19.90s | - | - | - | - |
| 3-1(true) | 1.00s | 1.19s | **0.73s** | 30.05s | - | - | - | - |
| 3-2(false) | 1.00s | 1.07s | **0.64s** | - | - | - | 5m51.64s | - |
| 3-3(true) | 1.00s | 1.28s | **0.72s** | 21.27s | - | - | - | - |
| 3-4(false) | 1.01s | 1.13s | **0.63s** | - | - | - | 4m37.65s | - |
| 3-5(false) | 2.04s | 13.70s | **1.09s** | - | - | - | - | - |
| 3-6(true) | 1.12s | 1.21s | **0.74s** | 33.06s | - | - | - | - |
| 3-7(false) | 2.03s | 14.65s | **0.73s** | - | - | - | - | - |
| 4-1(true) | 1.21s | 1.19s | **0.68s** | 21.27s | - | - | - | - |

Tab. 6: Running times of the experiments on the CPC program

Table 7 shows the running times of the case study done at CERN. These results are not comparable with our results, because the verifications at CERN are done on reduced models and with different computers. These results are included to show that reductions are very important for the SMV tools and to show an indication of the complexity of the program. The bold values are running times that are faster than the running times of all of our tools. This shows that for some properties our tools, especially CBMC Incremental, already had a faster running time, despite the fact that we had not used any reductions. For properties 1-11a, 1-11b, 3-5, and 3-7 we can see that the running times for some of our tools are significantly better than the running time from CERN. For most of the properties the running times from CERN are faster, but the differences are very small.

| **Property** | **CERN** |
|---|---|
| 1-1(false) | **0.485s** |
| 1-2(true) | **1.600s** |
| 1-3(false) | 1.271s |
| 1-4(false) | **0.548s** |
| 1-5(false) | **0.586s** |
| 1-6(true) | **1.900s** |
| 1-7(true) | **0.410s** |
| 1-8(true) | **0.404s** |
| 1-9(true) | **0.386s** |
| 1-11a(false) | 5m1.000s |
| 1-11b(false) | 4m39.300s |
| 2-1(true) | **0.422s** |
| 2-2(false) | **0.375s** |
| 2-3(true) | 1.113s |
| 2-4(true) | 1.030s |
| 2-5(true) | **4.205s** |
| 2-6(true) | **4.895s** |
| 2-7(false) | 1.634s |
| 2-8(true) | 1.060s |
| 3-1(true) | 0.992s |
| 3-2(false) | 1.530s |
| 3-3(true) | 0.772s |
| 3-4(false) | **0.319s** |
| 3-5(false) | 21.406s |
| 3-6(true) | 1.065s |
| 3-7(false) | 16.009s |
| 4-1(true) | 0.797s |

Tab. 7: Running times of the CPC program from CERN

# 7 Conclusion

In this thesis we have looked at verification tools for PLC code used at CERN. We have discussed the tools in two groups: Verification tools on Models and Verification tools on Code. We have compared the tools on three aspects: Ease of translation, Results and Running times, and Counterexamples.

For most of the tools the translation was easy. Both the C language and the PROMELA language are very similar to the SCL language. For SMV the translation was a bit more difficult because there were a lot of choices we had to make. All these choices can result in multiple possible translations of the same program. These possible translations might give different results and different running times.

With the exception of K-Inductor, all tools are able to give clear counterexamples. The counterexamples from NuSVM and nuXmv are not readable without the translated code, while the counterexamples from the other tools are readable with only the SCL code.

To get the results and running times we have first done some experiments with example programs and later we have done a larger case study with the most promising tools. For the example programs we have seen that Spin, NuSMV, nuXmv, CPA-checker, and SATABS were able to prove or refute all properties correctly while CBMC Incremental and 2LS could not prove one of the TRUE-properties. For the running times we have seen that nuXmv, CBMC Incremental, and 2LS all had good running times for most or all of the properties. NuSMV, CPA-checker, and SATABS had reasonable running times for some of the properties but took longer for other properties. We could also see that the reductions that we did improved the running times for Spin, NuSMV, nuXmv, SATABS, and CPA-checker, but not for the other tools.

We did a larger case study with 2LS, CBMC Incremental, CPA-checker, SATABS, NuSMV, and nuXmv. We have compared the results with each other as well as with the results from CERN. While NuSMV, nuXmv, SATABS, and CPA-checker could prove or refute all properties in the example program, in the case study they could hardly prove or refute any properties. The $k$I$k$I algorithm of 2LS took significantly longer than the k-induction algorithm for some of the properties. This is interesting because it shows that although the $k$I$k$I algorithm might be able to prove more properties, as we have seen in Example 5.4, it is slower on large programs, so the k-induction algorithm might be preferable. We have also seen that, despite the fact that we have done no reductions, the running times of CBMC Incremental were smaller than the running times of CERN for some properties. For the other properties the difference between our running times and the running times of CERN were very small.

Overall it seems that the verification tools on code gave better results than the verification tools on models. The tool that had the best performances is CBMC Incremental, but since it could not prove all properties there is still some room for improvement. We would recommend CERN to use CBMC Incremental or 2LS for their verifications. For the properties that these tools are unable to prove further research should be done. Until then CERN could use their own methods if these tools fail to prove or refute a property.

## 7.1 Future work

In this thesis we have discussed a number of verification tools to find out which tools can be used for verification of PLC code at CERN. Further research is needed on a number of aspects to find out which tool could best be used for this verification.

While we have considered a number of tools, there was not enough time to explore every option of these tools. Additionally there are other verification tools that can be considered. Since there are multiple contests for verification tools, results from these contests could be used to choose different tools.

The translation that we have used for SMV creates a lot of states. Some parallel assignments are used in this translation, but more research on parallel assignments would probably improve the running times of both NuSMV and nuXmv significantly. Note that the C language does not have any support for parallel assignments, so this would not improve the running times of the programs that use C code.

In the case study we did not use any reduction techniques. Some research about these techniques could be done to improve the verification times. When reduction techniques are used, a better comparison to the running times from CERN could be made. For SMV these reductions could be similar to the reductions of CERN, but for the C programs there might be other reduction techniques. Research of reduction techniques on C programs might make the use of an intermediate model unnecessary.

To model the timers we have chosen to use the same technique as CERN [24]. Different ways of modeling the timers could possibly improve the verification times. There are multiple possible ways to model the timers. The used time for a cycle could be set inside a different range or to a fixed value. If the only aspect that matters is whether the time has passed a timeout value, a single non-deterministic boolean variable that states this could be used.

## References

[1] CPC program. `http://cern.ch/unicos/`. Accessed: 2016-05-13.

[2] C. Baier and J. Katoen. *Principles of model checking.* MIT Press, 2008.

[3] H. Barbosa and D. Déharbe. Formal verification of PLC programs using the B method. In *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, pages 353–356, 2012.

[4] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, pages 504–518. Springer, 2007.

[5] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.

[6] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 189–198. FMCAD Inc, 2010.

[7] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures Instead of BDDs. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference*, DAC '99, pages 317–320, New York, NY, USA, 1999. ACM.

[8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 193–207, 1999.

[9] W. Bolton. *Programmable logic controllers.* Newnes, 2015.

[10] M. Brain, S. Joshi, D. Kroening, and P. Schrammel. *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, chapter Safety Verification and Refutation by k-Invariants and k-Induction, pages 145–161. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[11] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[12] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. In A. Biere and R. Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.

[14] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

[15] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[16] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking.* MIT press, 1999.

[17] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.

[18] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.

[19] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, chapter An Overview of the mCRL2 Toolset and Its Recent Advances, pages 199–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[20] D. Darvas. Personal e-mail communication, 2015.

[21] L. De Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. *Lecture notes in computer science*, pages 14–26, 2003.

[22] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *Static Analysis*, pages 351–368. Springer, 2011.

[23] B. Fernández Adiego. *Bringing Automated Formal Verification to PLC Program Development.* PhD thesis, University of Oviedo, 2014.

[24] B. Fernandez Adiego, V. Gonzalez Suarez, J. Blech, E. Blanco Vinuela, D. Darvas, and J. Tournier. Modelling and formal verification of timing aspects in large PLC programs. Technical report, 2014.

[25] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.

[26] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[27] M. Meulen. Verification of PLC source code using propositional logic. Master's thesis, Technical University of Eindhoven, 2010.

[28] S. C. Park, C. M. Park, G. Wang, J. Kwak, and S. Yeo. Plcstudio: Simulation based PLC code verification. In *Proceedings of the 2008 Winter Simulation Conference, Global Gateway to Discovery, WSC 2008, InterContinental Hotel, Miami, Florida, USA, December 7-10, 2008*, pages 222–228, 2008.

[29] O. Pavlovic, R. Pinger, and M. Kollmann. Automated formal verification of PLC programs written in IL. In *Conference on Automated Deduction (CADE)*, pages 152–163, 2007.

[30] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.

[31] M. Rausch and B. H. Krogh. Formal verification of PLC programs. In *American Control Conference, 1998. Proceedings of the 1998*, volume 1, pages 234–238. IEEE, 1998.

[32] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller. Incremental Bounded Model Checking for Embedded Software (extended version). *CoRR*, abs/1409.5872, 2014.

[33] S. SIMATIC. Structured Control Language (SCL) for S7-300/S7-400 Programming Manual, 1998.

[34] F. Somenzi and A. R. Bradley. IC3: where monolithic and incremental meet. In *FMCAD*, pages 3–8, 2011.

[35] M. Stokely, S. Chaki, and J. Ouaknine. Parallel assignments in software model checking. *Electronic Notes in Theoretical Computer Science*, 157(1):77–94, 2006.

# Appendices

## A Example programs in SCL

In this appendix the example programs from CERN can be found.

```
// Type definition
TYPE ComplexSignal
STRUCT
        out1 : BOOL;
        out2 : BOOL;
        remaining : INT;
        elapsed : INT;
END_STRUCT
END_TYPE

// The function block to be verified
FUNCTION_BLOCK ComplexExample
CONST
        cntr_max := 5;
END_CONST
VAR_INPUT
        signal : BOOL;           // input signal
        error : BOOL;            // not used (but it happens that we have non-used variables)

        toMode1 : BOOL;          // request to switch to mode1
        toMode2 : BOOL;          // request to switch to mode2
        toMode3 : BOOL;          // request to switch to mode3
        mode3Forbidden : BOOL;   // if it is true, it is forbidden to be in mode3
END_VAR
VAR
        signal_old : BOOL := FALSE;      // signal value from the last cycle
        cntr : INT := 0;         // counter to delay the out2 signal

        mode1 : BOOL;            // true if the block is in mode1
        mode2 : BOOL;            // true if the block is in mode2
        mode3 : BOOL;            // true if the block is in mode3
END_VAR
VAR_TEMP
        edge_signal : BOOL;      // rising edge of out1
END_VAR
VAR_OUTPUT
        out1 : BOOL;             // out1 is true if the signal is true
        out2 : BOOL;             // out2 is true if the signal is true AND out1 was true for
     'cntr_max' cycles
        out3 : ComplexSignal;
END_VAR

BEGIN
        (* Signal handling *)
        edge_signal := R_EDGE(new := signal, old := signal_old);

        IF NOT signal THEN
                // outputs are false if the signal is false
                out1 := FALSE;
                out2 := FALSE;
                cntr := 0;
        ELSIF edge_signal THEN
                // if the signal has a rising edge, out1 should be true
                out1 := NOT out1;
                out3.out1 := out1;
        ELSE
                cntr := cntr + 1;
                IF cntr > cntr_max AND signal THEN
                        out2 := TRUE;
                ELSE
                        out2 := FALSE;
                END_IF;
                out3.out1 := out1;
                out3.out2 := out2;
                out3.remaining := cntr_max - cntr;
                out3.elapsed := cntr;
        END_IF;

        (* ---------------------------------------------------------------- *)

        (* Operation mode handling *)
        IF not mode1 and not mode2 and not mode3 THEN
                mode1 := TRUE;
        END_IF;

        IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
                mode1 := TRUE;
        END_IF;
        IF toMode2 THEN
                mode2 := TRUE;
        ELSIF toMode3 THEN
                mode3 := TRUE;
        END_IF;

        IF mode1 THEN
                ModeDB.mode := 1;
        ELSIF mode2 THEN
                ModeDB.mode := 2;
        ELSIF mode3 THEN
                ModeDB.mode := 3;
        ELSE
                ModeDB.mode := 0;
        END_IF;
END_FUNCTION_BLOCK

// Global data storage
DATA_BLOCK ModeDB
STRUCT
        mode : INT;
END_STRUCT
BEGIN
        mode := -1; // default value for the mode variable in the data block
END_DATA_BLOCK

// Helper function to determine the rising edge on a signal.
FUNCTION R_EDGE : BOOL
        VAR_INPUT
                new : BOOL;
        END_VAR
        VAR_IN_OUT
                old : BOOL;
        END_VAR

        BEGIN
                IF (new = true AND old = false) THEN
                        R_EDGE := true;
                        old := true;
                ELSE R_EDGE := false;
                        old := new;
                END_IF;
END_FUNCTION
```

Listing A.1: `Example.SCL`

```
// Type definition
TYPE ComplexSignal
STRUCT
        out1 : BOOL;
        out2 : BOOL;
        remaining : INT;
        elapsed : INT;
END_STRUCT
END_TYPE

// The function block to be verified
FUNCTION_BLOCK ComplexExample
CONST
        cntr_max := 5;
END_CONST
VAR_INPUT
        signal : BOOL;            // input signal
        error : BOOL;             // not used (but it happens that we have non-used variables)

        toMode1 : BOOL;           // request to switch to mode1
        toMode2 : BOOL;           // request to switch to mode2
        toMode3 : BOOL;           // request to switch to mode3
        mode3Forbidden : BOOL;    // if it is true, it is forbidden to be in mode3
END_VAR
VAR
        signal_old : BOOL := FALSE;      // signal value from the last cycle
        cntr : INT := 0;          // counter to delay the out2 signal
        cntr2 : INT := 0;

        mode1 : BOOL;             // true if the block is in mode1
        mode2 : BOOL;             // true if the block is in mode2
        mode3 : BOOL;             // true if the block is in mode3
END_VAR
VAR_TEMP
        edge_signal : BOOL;       // rising edge of out1
END_VAR
VAR_OUTPUT
        out1 : BOOL;              // out1 is true if the signal is true
        out2 : BOOL;              // out2 is true if the signal is true AND out1 was true for
              'cntr_max' cycles
        out3 : ComplexSignal;
END_VAR

BEGIN
```

```
        (* Signal handling *)
        edge_signal := R_EDGE(new := signal, old := signal_old);

        IF NOT signal THEN
                // outputs are false if the signal is false
                out1 := FALSE;
                out2 := FALSE;
                cntr := 0;
        ELSIF edge_signal THEN
                // if the signal has a rising edge, out1 should be true
                out1 := NOT out1;
                out3.out1 := out1;
        ELSE
                cntr := cntr + 1;
                // --- ADDITIONAL PART
                IF cntr > cntr_max THEN
                        cntr := 0;
                        cntr2 := cntr2 + 1;
                END_IF;
                // ---

                IF cntr2 > cntr_max AND signal THEN
                        out2 := TRUE;
                ELSE
                        out2 := FALSE;
                END_IF;
                out3.out1 := out1;
                out3.out2 := out2;
                out3.remaining := cntr_max - cntr;
                out3.elapsed := cntr;
        END_IF;

        (* ---------------------------------------------------------------- *)

        (* Operation mode handling *)
        IF not mode1 and not mode2 and not mode3 THEN
                mode1 := TRUE;
        END_IF;

        IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
                mode1 := TRUE;
        END_IF;
        IF toMode2 THEN
                mode2 := TRUE;
        ELSIF toMode3 THEN
                mode3 := TRUE;
        END_IF;

        IF mode1 THEN
                ModeDB.mode := 1;
        ELSIF mode2 THEN
                ModeDB.mode := 2;
        ELSIF mode3 THEN
                ModeDB.mode := 3;
        ELSE
                ModeDB.mode := 0;
        END_IF;
END_FUNCTION_BLOCK

// Global data storage
DATA_BLOCK ModeDB
STRUCT
        mode : INT;
END_STRUCT
BEGIN
        mode := -1; // default value for the mode variable in the data block
END_DATA_BLOCK

// Helper function to determine the rising edge on a signal.
FUNCTION R_EDGE : BOOL
        VAR_INPUT
                new : BOOL;
        END_VAR
        VAR_IN_OUT
                old : BOOL;
        END_VAR

        BEGIN
                IF (new = true AND old = false) THEN
                        R_EDGE := true;
                        old := true;
                ELSE R_EDGE := false;
                        old := new;
                END_IF;
END_FUNCTION
```

Listing A.2: Example_int.SCL

```
// Type definition
TYPE ComplexSignal
STRUCT
        out1 : BOOL;
        out2 : BOOL;
        remaining : INT;
        elapsed : INT;
END_STRUCT
END_TYPE

// The function block to be verified
FUNCTION_BLOCK ComplexExample
CONST
        cntr_max := 5;
END_CONST
VAR_INPUT
        signal : BOOL;          // input signal
        error : BOOL;           // not used (but it happens that we have non-used variables)

        toMode1 : BOOL;         // request to switch to mode1
        toMode2 : BOOL;         // request to switch to mode2
        toMode3 : BOOL;         // request to switch to mode3
        mode3Forbidden : BOOL;  // if it is true, it is forbidden to be in mode3
END_VAR
VAR
        signal_old : BOOL := FALSE;     // signal value from the last cycle
        cntr : INT := 0;        // counter to delay the out2 signal
        cntr2 : INT := 0;

        mode1 : BOOL;           // true if the block is in mode1
        mode2 : BOOL;           // true if the block is in mode2
        mode3 : BOOL;           // true if the block is in mode3
END_VAR
VAR_TEMP
        edge_signal : BOOL;     // rising edge of out1
END_VAR
VAR_OUTPUT
        out1 : BOOL;            // out1 is true if the signal is true
        out2 : BOOL;            // out2 is true if the signal is true AND out1 was true for
             'cntr_max' cycles
        out3 : ComplexSignal;
END_VAR

BEGIN
        (* Signal handling *)
        edge_signal := R_EDGE(new := signal, old := signal_old);

        IF NOT signal THEN
                // outputs are false if the signal is false
                out1 := FALSE;
                out2 := FALSE;
                cntr := 0;
        ELSIF edge_signal THEN
                // if the signal has a rising edge, out1 should be true
                out1 := NOT out1;
                out3.out1 := out1;
        ELSE
                cntr := cntr + 1;
                // --- ADDITIONAL PART
                WHILE cntr > cntr_max DO
                        cntr := cntr - 1;
                        cntr2 := cntr2 + 1;
                END_WHILE;
                // ---

                IF cntr2 > cntr_max AND signal THEN
                        out2 := TRUE;
                ELSE
                        out2 := FALSE;
                END_IF;
                out3.out1 := out1;
                out3.out2 := out2;
                out3.remaining := cntr_max - cntr;
                out3.elapsed := cntr;
        END_IF;

        (* ---------------------------------------------------------- *)

        (* Operation mode handling *)
        IF not mode1 and not mode2 and not mode3 THEN
                mode1 := TRUE;
        END_IF;

        IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
                mode1 := TRUE;
        END_IF;
        IF toMode2 THEN
                mode2 := TRUE;
        ELSIF toMode3 THEN
```

```
                mode3 := TRUE;
        END_IF;

        IF mode1 THEN
                ModeDB.mode := 1;
        ELSIF mode2 THEN
                ModeDB.mode := 2;
        ELSIF mode3 THEN
                ModeDB.mode := 3;
        ELSE
                ModeDB.mode := 0;
        END_IF;
END_FUNCTION_BLOCK

// Global data storage
DATA_BLOCK ModeDB
STRUCT
        mode : INT;
END_STRUCT
BEGIN
        mode := -1; // default value for the mode variable in the data block
END_DATA_BLOCK

// Helper function to determine the rising edge on a signal.
FUNCTION R_EDGE : BOOL
        VAR_INPUT
                new : BOOL;
        END_VAR
        VAR_IN_OUT
                old : BOOL;
        END_VAR

        BEGIN
                IF (new = true AND old = false) THEN
                        R_EDGE := true;
                        old := true;
                ELSE R_EDGE := false;
                        old := new;
                END_IF;
END_FUNCTION
```

Listing A.3: `Example_while.SCL`

# B Example programs in PROMELA

In this appendix the translations from the example programs to PROMELA can be found, both the full program and the reduced program are shown.

```
typedef ComplexSignal{
    bool out1;
    bool out2;
    short remaining;
    short elapsed;
};
bool error, toMode1, toMode2, toMode3, mode3Forbidden;
bool mode1 = 0;
bool mode2 = 0;
bool mode3 = 0;
bool signal;
bool signal_old = 0;
bool edge_signal;
bool out1, out2;
int mode = 0;
int cntr_max = 5;
short cntr = 0;
ComplexSignal out3;

active proctype go()
{
    do
    ::if
        ::error = 0
        ::error = 1
    fi;
        if
        ::toMode1 = 0
        ::toMode1 = 1
    fi;
    if
        ::toMode2 = 0
        ::toMode2 = 1
    fi;
    if
```

```
            ::toMode3 = 0
            ::toMode3 = 1
        fi;
        if
            ::mode3Forbidden = 0
            ::mode3Forbidden = 1
        fi;
        if
            ::signal = 0
            ::signal = 1
        fi;

        d_step{if
            ::signal && !signal_old ->       edge_signal = 1;
                                             signal_old = 1
            :: else ->                       edge_signal = 0;
                                             signal_old = signal;
        fi;
        if
            :: !signal ->   out1 = 0;
                            out2 = 0;
                            cntr = 0
            :: signal && edge_signal -> out1 = !out1;
                                         out3.out1 = out1
            :: else -> cntr = cntr + 1; if
                                             :: (cntr > cntr_max) && signal -> out2 = 1
                                             :: else -> out2 = 0
                                         fi;
                                         out3.out1 = out1;
                                         out3.out2 = out2;
                                         out3.remaining = cntr_max - cntr;
                                         out3.elapsed = cntr
        fi;
        if
            :: !mode1 && !mode2 && !mode3 -> mode1 = 1
            :: else
        fi;
        if
            :: toMode1 || (toMode3 && mode3Forbidden) -> mode1 = 1
            :: else
        fi;
        if
            :: toMode2 -> mode2 = 1
            :: toMode3 -> mode3 = 1
            :: else
        fi;
        if
            :: mode1 -> mode = 1
            :: !mode1 && mode2 -> mode = 2
            :: !mode1 && !mode2 && mode3 -> mode = 3
            :: else -> mode = 0
        fi;}

        assert (!out2 || out1)
        assert (signal || !out2)
        assert (out3.out1 == out1)
        assert (out1 || (out3.elapsed == 0))

        od
}
```

Listing B.1: `Example.pml`

```
typedef ComplexSignal{
        bool out1;
        short elapsed;
};
bool signal;
bool signal_old = 0;
bool edge_signal;
bool out1, out2;
int cntr_max = 5;
short cntr = 0;
ComplexSignal out3;

active proctype go()
{
        do
        ::if
            ::signal = 0
            ::signal = 1
        fi;

        d_step{if
            ::(signal==1) && (signal_old==0) ->       edge_signal = 1;
                                             signal_old = 1
            :: else ->                       edge_signal = 0;
                                             signal_old = signal;
```

```
    fi;
    if
        :: !signal ->    out1 = 0;
                         out2 = 0;
                         cntr = 0
        :: signal && edge_signal -> out1 = !out1;
                                     out3.out1 = out1
        :: else -> cntr = cntr + 1; if
                                         :: (cntr > cntr_max) && signal -> out2 = 1
                                         :: else -> out2 = 0
                                     fi;
                                     out3.out1 = out1;
                                     out3.elapsed = cntr
    fi;}

    assert (!out2 || out1)
    assert (signal || !out2)
    assert (out3.out1 == out1)
    assert (out1 || (out3.elapsed == 0))

    od
}
```

Listing B.2: Reduced version of `Example.pml`

```
typedef ComplexSignal{
    bool out1;
    bool out2;
    short remaining;
    short elapsed;
};
bool error, toMode1, toMode2, toMode3, mode3Forbidden;
bool mode1 = 0;
bool mode2 = 0;
bool mode3 = 0;
bool signal;
bool signal_old = 0;
bool edge_signal;
bool out1, out2;
int mode = 0;
int cntr_max = 5;
short cntr = 0;
short cntr2 = 0;
ComplexSignal out3;

active proctype go()
{
    do
    ::if
        ::error = 0
        ::error = 1
    fi;
    if
        ::toMode1 = 0
        ::toMode1 = 1
    fi;
    if
        ::toMode2 = 0
        ::toMode2 = 1
    fi;
    if
        ::toMode3 = 0
        ::toMode3 = 1
    fi;
    if
        ::mode3Forbidden = 0
        ::mode3Forbidden = 1
    fi;
    if
        ::signal = 0
        ::signal = 1
    fi;

    d_step{if
        ::(signal==1) && (signal_old==0) ->      edge_signal = 1;
                                           signal_old = 1
        ::else ->                          edge_signal = 0;
                                           signal_old = signal;
    fi;
    if
        :: !signal ->    out1 = 0;
                         out2 = 0;
                         cntr = 0
        :: signal && edge_signal -> out1 = !out1;
                                     out3.out1 = out1
        :: else -> cntr = cntr + 1; if
                                         :: cntr > cntr_max -> cntr = 0; cntr2 = cntr2 + 1
                                         :: else
```

```
                                        fi;
                                        if
                                            :: (cntr2 > cntr_max) && signal -> out2 = 1
                                            :: else -> out2 = 0
                                        fi;
                                        out3.out1 = out1;
                                        out3.out2 = out2;
                                        out3.remaining = cntr_max - cntr;
                                        out3.elapsed = cntr
    fi;
    if
        :: !mode1 && !mode2 && !mode3 -> mode1 = 1
        :: else
    fi;
    if
        :: toMode1 || (toMode3 && mode3Forbidden) -> mode1 = 1
        :: else
    fi;
    if
        :: toMode2 -> mode2 = 1
        :: toMode3 -> mode3 = 1
        :: else
    fi;
    if
        :: mode1 -> mode = 1
        :: !mode1 && mode2 -> mode = 2
        :: !mode1 && !mode2 && mode3 -> mode = 3
        :: else -> mode = 0
    fi;}

    assert (!out2 || out1)
    assert (signal || !out2)
    assert (out3.out1 == out1)
    assert (out1 || (out3.elapsed == 0))

    od
}
```

Listing B.3: Example_int.pml

```
typedef ComplexSignal{
    bool out1;
    short elapsed;
};
bool signal;
bool signal_old = 0;
bool edge_signal;
bool out1, out2;
int cntr_max = 5;
short cntr = 0;
short cntr2 = 0;
ComplexSignal out3;

active proctype go()
{
    do
    ::if
        ::signal = 0
        ::signal = 1
    fi;

    d_step{if
        ::(signal==1) && (signal_old==0) ->      edge_signal = 1;
                                             signal_old = 1
        ::else ->                            edge_signal = 0;
                                             signal_old = signal;
    fi;
    if
        :: !signal ->   out1 = 0;
                        out2 = 0;
                        cntr = 0
        :: signal && edge_signal -> out1 = !out1;
                                        out3.out1 = out1
        :: else -> cntr = cntr + 1; if
                                            :: cntr > cntr_max -> cntr = 0; cntr2 = cntr2 + 1
                                            :: else
                                        fi;
                                        if
                                            :: (cntr2 > cntr_max) && signal -> out2 = 1
                                            :: else -> out2 = 0
                                        fi;
                                        out3.out1 = out1;
                                        out3.elapsed = cntr
    fi;}

    assert (!out2 || out1)
    assert (signal || !out2)
    assert (out3.out1 == out1)
    assert (out1 || (out3.elapsed == 0))

    od
}
```

Listing B.4: Reduced version of `Example_int.pml`

```
typedef ComplexSignal{
    bool out1;
    bool out2;
    short remaining;
    short elapsed;
};
bool error, toMode1, toMode2, toMode3, mode3Forbidden;
bool mode1 = 0;
bool mode2 = 0;
bool mode3 = 0;
bool signal;
bool signal_old = 0;
bool edge_signal;
bool out1, out2;
int mode = 0;
int cntr_max = 5;
short cntr = 0;
short cntr2 = 0;
ComplexSignal out3;

active proctype go()
{
    do
    ::if
        ::error = 0
        ::error = 1
    fi;
        if
        ::toMode1 = 0
        ::toMode1 = 1
    fi;
    if
        ::toMode2 = 0
        ::toMode2 = 1
    fi;
    if
        ::toMode3 = 0
        ::toMode3 = 1
    fi;
    if
        ::mode3Forbidden = 0
        ::mode3Forbidden = 1
    fi;
    if
        ::signal = 0
        ::signal = 1
    fi;

    d_step{if
        ::(signal==1) && (signal_old==0) ->      edge_signal = 1;
                                                 signal_old = 1
        ::!signal || signal_old ->               edge_signal = 0;
                                                 signal_old = signal;
    fi;
    if
        :: !signal ->   out1 = 0;
                        out2 = 0;
                        cntr = 0
        :: signal && edge_signal -> out1 = !out1;
                                    out3.out1 = out1
        :: else -> cntr = cntr + 1;
                                    do
                                        :: cntr > cntr_max -> cntr = cntr - 1; cntr2 = cntr2
                                            + 1
                                        :: else -> break
                                    od;
                                    if
                                        :: (cntr2 > cntr_max) && signal -> out2 = 1
                                        :: else -> out2 = 0
                                    fi;
                                    out3.out1 = out1;
                                    out3.out2 = out2;
                                    out3.remaining = cntr_max - cntr;
                                    out3.elapsed = cntr
    fi;
    if
        :: !mode1 && !mode2 && !mode3 -> mode1 = 1
        :: else
    fi;
    if
        :: toMode1 || (toMode3 && mode3Forbidden) -> mode1 = 1
```

```
            :: else
    fi;
    if
            :: toMode2 -> mode2 = 1
            :: toMode3 -> mode3 = 1
            :: else
    fi;
    if
            :: mode1 -> mode = 1
            :: !mode1 && mode2 -> mode = 2
            :: !mode1 && !mode2 && mode3 -> mode = 3
            :: else -> mode = 0
    fi;}

    assert (!out2 || out1)
    assert (signal || !out2)
    assert (out3.out1 == out1)
    assert (out1 || (out3.elapsed == 0))

    od
}
```

Listing B.5: `Example_while.pml`

```
typedef ComplexSignal{
    bool out1;
    short elapsed;
};
bool signal;
bool signal_old = 0;
bool edge_signal;
bool out1, out2;
int cntr_max = 5;
short cntr = 0;
short cntr2 = 0;
ComplexSignal out3;

active proctype go()
{
    do
    ::
    if
        ::signal = 0
        ::signal = 1
    fi;

    d_step{if
        ::(signal==1) && (signal_old==0) ->       edge_signal = 1;
                                          signal_old = 1
        ::!signal || signal_old ->        edge_signal = 0;
                                          signal_old = signal;
    fi;
    if
        :: !signal ->    out1 = 0;
                         out2 = 0;
                         cntr = 0
        :: signal && edge_signal -> out1 = !out1;
                                    out3.out1 = out1
        :: else -> cntr = cntr + 1;
                                    do
                                        :: cntr > cntr_max -> cntr = cntr - 1; cntr2 = cntr2
                                            + 1
                                        :: else -> break
                                    od;
                                    if
                                        :: (cntr2 > cntr_max) && signal -> out2 = 1
                                        :: else -> out2 = 0
                                    fi;
                                    out3.out1 = out1;
                                    out3.elapsed = cntr
    fi;}

    assert (!out2 || out1)
    assert (signal || !out2)
    assert (out3.out1 == out1)
    assert (out1 || (out3.elapsed == 0))

    od
}
```

Listing B.6: Reduced version of `Example_while.pml`

# C Example programs in SMV

This appendix shows the translations from the example programs to SMV, both the full program and the reduced are given.

```
MODULE main
    VAR
        signal : boolean;
        error : boolean;
        toMode1 : boolean;
        toMode2 : boolean;
        toMode3 : boolean;
        mode3Forbidden : boolean;
        signal_old : boolean;
        edge_signal : boolean;
        cntr : signed word[16];
        out1 : boolean;
        out2 : boolean;
        out3.out1 : boolean;
        out3.out2 : boolean;
        out3.remaining : signed word[16];
        out3.elapsed : signed word[16];
        mode1 : boolean;
        mode2 : boolean;
        mode3 : boolean;
        mode : -1..3;
        loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
              step11, step12, step13, step14, step15, step16, step17, step18, step19, step20,
              step21, step22, end};

    ASSIGN
        init(signal_old) := FALSE;
        init(edge_signal) := FALSE;
        init(cntr) := 0sd16_0;
        init(out1) := FALSE;
        init(out2) := FALSE;
        init(out3.out1) := FALSE;
        init(out3.out2) := FALSE;
        init(out3.remaining) := 0sd16_0;
        init(out3.elapsed) := 0sd16_0;
        init(mode1) := FALSE;
        init(mode2) := FALSE;
        init(mode3) := FALSE;
        init(mode) := -1;
        init(loc) := start;

        next(loc) :=
            case
                (loc = start) : step1;
                (loc = step1) & (signal & !signal_old) : step2;
                (loc = step1) : step3;
                (loc = step2) : step4;
                (loc = step3) : step4;
                (loc = step4) & (!signal) : step5;
                (loc = step4) & (signal) & (edge_signal) : step6;
                (loc = step4) : step7;
                (loc = step5) : step11;
                (loc = step6) : step11;
                (loc = step7) & (cntr > 0sd16_5) & (signal) : step8;
                (loc = step7) : step9;
                (loc = step8) : step10;
                (loc = step9) : step10;
                (loc = step10) : step11;
                (loc = step11) & (!mode1 & !mode2 & !mode3) : step12;
                (loc = step11) : step13;
                (loc = step12) : step13;
                (loc = step13) & (toMode1) | (toMode3 & mode3Forbidden) : step14;
                (loc = step13) : step15;
                (loc = step14) : step15;
                (loc = step15) & (toMode2) : step16;
                (loc = step15) & (!toMode2) & (toMode3) : step17;
                (loc = step15) : step18;
                (loc = step16) : step18;
                (loc = step17) : step18;
                (loc = step18) & (mode1) : step19;
                (loc = step18) & (!mode1) & (mode2) : step20;
                (loc = step18) & (!mode1) & (!mode2) & (mode3) : step21;
                (loc = step18) : step22;
                (loc = step19) : end;
                (loc = step20) : end;
                (loc = step21) : end;
                (loc = step22) : end;
                (loc = end) : start;
            esac;

        next(signal) :=
            case
```

```
                (loc = start) : {TRUE, FALSE};
                TRUE : signal;
            esac;
        next(error) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : error;
            esac;
        next(toMode1) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode1;
            esac;
        next(toMode2) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode2;
            esac;
        next(toMode3) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode3;
            esac;
        next(mode3Forbidden) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : mode3Forbidden;
            esac;

        next(signal_old) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : signal;
                TRUE : signal_old;
            esac;
        next(edge_signal) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : FALSE;
                TRUE : edge_signal;
            esac;
        next(cntr) :=
            case
                (loc = step5) : 0sd16_0;
                (loc = step7) : cntr + 0sd16_1;
                TRUE : cntr;
            esac;
        next(out1) :=
            case
                (loc = step5) : FALSE;
                (loc = step6) : !out1;
                TRUE : out1;
            esac;
        next(out2) :=
            case
                (loc = step5) : FALSE;
                (loc = step8) : TRUE;
                (loc = step9) : FALSE;
                TRUE : out2;
            esac;
        next(out3.out1) :=
            case
                (loc = step6) : !out1;
                (loc = step10) : out1;
                TRUE : out3.out1;
            esac;
        next(out3.out2) :=
            case
                (loc = step10) : out2;
                TRUE : out3.out2;
            esac;
        next(out3.remaining) :=
            case
                (loc = step10) : 0sd16_5 - cntr;
                TRUE : out3.remaining;
            esac;
        next(out3.elapsed) :=
            case
                (loc = step10) : cntr;
                TRUE : out3.elapsed;
            esac;
        next(mode1) :=
            case
                (loc = step12) : TRUE;
                (loc = step14) : TRUE;
                TRUE : mode1;
            esac;
        next(mode2) :=
            case
                (loc = step16) : TRUE;
                TRUE : mode2;
            esac;
        next(mode3) :=
            case
                (loc = step17) : TRUE;
                TRUE : mode3;
            esac;
        next(mode) :=
            case
                (loc = step19) : 1;
                (loc = step20) : 2;
                (loc = step21) : 3;
                (loc = step22) : 0;
                TRUE : mode;
            esac;

--CTL properties
SPEC AG(loc = end -> (out2 -> out1))
SPEC AG(loc = end -> (!signal -> !out2))
SPEC AG(loc = end -> (out3.out1 = out1))
SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

--INVARIANT properties
INVARSPEC(loc = end -> (out2 -> out1))
INVARSPEC(loc = end -> (!signal -> !out2))
INVARSPEC(loc = end -> (out3.out1 = out1))
INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
```

Listing C.1: `Example.smv`

```
MODULE main
    VAR
        signal : boolean;
        signal_old : boolean;
        edge_signal : boolean;
        cntr : signed word[16];
        out1 : boolean;
        out2 : boolean;
        out3.out1 : boolean;
        out3.elapsed : signed word[16];
        loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
            end};

    ASSIGN
        init(signal_old) := FALSE;
        init(edge_signal) := FALSE;
        init(cntr) := 0sd16_0;
        init(out1) := FALSE;
        init(out2) := FALSE;
        init(out3.out1) := FALSE;
        init(out3.elapsed) := 0sd16_0;
        init(loc) := start;

        next(loc) :=
            case
                (loc = start) : step1;
                (loc = step1) & (signal & !signal_old) : step2;
                (loc = step1) : step3;
                (loc = step2) : step4;
                (loc = step3) : step4;
                (loc = step4) & (!signal) : step5;
                (loc = step4) & (signal) & (edge_signal) : step6;
                (loc = step4) : step7;
                (loc = step5) : end;
                (loc = step6) : end;
                (loc = step7) & (cntr > 0sd16_5) & (signal) : step8;
                (loc = step7) : step9;
                (loc = step8) : step10;
                (loc = step9) : step10;
                (loc = step10) : end;
                (loc = end) : start;
            esac;

        next(signal) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : signal;
            esac;

        next(signal_old) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : signal;
                TRUE : signal_old;
            esac;
        next(edge_signal) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : FALSE;
                TRUE : edge_signal;
            esac;
        next(cntr) :=
            case
                (loc = step5) : 0sd16_0;
                (loc = step7) : cntr + 0sd16_1;
                TRUE : cntr;
            esac;
        next(out1) :=
            case
                (loc = step5) : FALSE;
                (loc = step6) : !out1;
                TRUE : out1;
            esac;
        next(out2) :=
            case
                (loc = step5) : FALSE;
                (loc = step8) : TRUE;
                (loc = step9) : FALSE;
                TRUE : out2;
            esac;
        next(out3.out1) :=
            case
                (loc = step6) : out1;
                (loc = step10) : out1;
                TRUE : out3.out1;
            esac;
        next(out3.elapsed) :=
            case
                (loc = step10) : cntr;
                TRUE : out3.elapsed;
            esac;

--CTL properties
SPEC AG(loc = end -> (out2 -> out1))
SPEC AG(loc = end -> (!signal -> !out2))
SPEC AG(loc = end -> (out3.out1 = out1))
SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

--INVARIANT properties
INVARSPEC(loc = end -> (out2 -> out1))
INVARSPEC(loc = end -> (!signal -> !out2))
INVARSPEC(loc = end -> (out3.out1 = out1))
INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
```

Listing C.2: Reduced version of `Example.smv`

```
MODULE main
    VAR
        signal : boolean;
        error : boolean;
        toMode1 : boolean;
        toMode2 : boolean;
        toMode3 : boolean;
        mode3Forbidden : boolean;
        signal_old : boolean;
        edge_signal : boolean;
        cntr : signed word[16];
        cntr2 : signed word[16];
        out1 : boolean;
        out2 : boolean;
        out3.out1 : boolean;
        out3.out2 : boolean;
        out3.remaining : signed word[16];
        out3.elapsed : signed word[16];
        mode1 : boolean;
        mode2 : boolean;
        mode3 : boolean;
        mode : -1..3;
        loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
             step11, step12, step13, step14, step15, step16, step17, step18, step19, step20,
             step21, step22, step23, step24, end};

    ASSIGN
        init(signal_old) := FALSE;
        init(edge_signal) := FALSE;
        init(cntr) := 0sd16_0;
        init(cntr2) := 0sd16_0;
        init(out1) := FALSE;
        init(out2) := FALSE;
        init(out3.out1) := FALSE;
        init(out3.out2) := FALSE;
        init(out3.remaining) := 0sd16_0;
        init(out3.elapsed) := 0sd16_0;
        init(mode1) := FALSE;
        init(mode2) := FALSE;
        init(mode3) := FALSE;
        init(mode) := -1;
        init(loc) := start;

        next(loc) :=
            case
                (loc = start) : step1;
                (loc = step1) & (signal & !signal_old) : step2;
                (loc = step1) : step3;
                (loc = step2) : step4;
                (loc = step3) : step4;
                (loc = step4) & (!signal) : step5;
                (loc = step4) & (signal) & (edge_signal) : step6;
                (loc = step4) : step7;
                (loc = step5) : step13;
                (loc = step6) : step13;
                (loc = step7) & (cntr > 0sd16_5) : step8;
                (loc = step7) : step9;
                (loc = step8) : step9;
                (loc = step9) & (cntr2 > 0sd16_5) & (signal) : step10;
                (loc = step9) : step11;
                (loc = step10) : step12;
                (loc = step11) : step12;
                (loc = step12) : step13;
                (loc = step13) & (!mode1 & !mode2 & !mode3) : step14;
                (loc = step13) : step15;
                (loc = step14) : step15;
                (loc = step15) & (toMode1) | (toMode3 & mode3Forbidden) : step16;
                (loc = step15) : step17;
                (loc = step16) : step17;
                (loc = step17) & (toMode2) : step18;
                (loc = step17) & (!toMode2) & (toMode3) : step19;
                (loc = step17) : step20;
                (loc = step18) : step20;
                (loc = step19) : step20;
                (loc = step20) & (mode1) : step21;
                (loc = step20) & (!mode1) & (mode2) : step22;
                (loc = step20) & (!mode1) & (!mode2) & (mode3) : step23;
                (loc = step20) : step24;
                (loc = step21) : end;
                (loc = step22) : end;
                (loc = step23) : end;
                (loc = step24) : end;
                (loc = end) : start;
            esac;

        next(signal) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : signal;
            esac;
        next(error) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : error;
            esac;
        next(toMode1) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode1;
            esac;
        next(toMode2) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode2;
            esac;
        next(toMode3) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode3;
            esac;
        next(mode3Forbidden) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : mode3Forbidden;
            esac;

        next(signal_old) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : signal;
                TRUE : signal_old;
            esac;
        next(edge_signal) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : FALSE;
                TRUE : edge_signal;
            esac;
        next(cntr) :=
            case
```

```
                (loc = step5) : 0sd16_0;
                (loc = step7) : cntr + 0sd16_1;
                (loc = step8) : 0sd16_0;
                TRUE : cntr;
            esac;
        next(cntr2) :=
            case
                (loc = step8) : cntr2 + 0sd16_1;
                TRUE : cntr2;
            esac;
        next(out1) :=
            case
                (loc = step5) : FALSE;
                (loc = step6) : !out1;
                TRUE : out1;
            esac;
        next(out2) :=
            case
                (loc = step5) : FALSE;
                (loc = step10) : TRUE;
                (loc = step11) : FALSE;
                TRUE : out2;
            esac;
        next(out3.out1) :=
            case
                (loc = step6) : !out1;
                (loc = step12) : out1;
                TRUE : out3.out1;
            esac;
        next(out3.out2) :=
            case
                (loc = step12) : out2;
                TRUE : out3.out2;
            esac;
        next(out3.remaining) :=
            case
                (loc = step12) : 0sd16_5 - cntr;
                TRUE : out3.remaining;
            esac;
        next(out3.elapsed) :=
            case
                (loc = step12) : cntr;
                TRUE : out3.elapsed;
            esac;
        next(mode1) :=
            case
                (loc = step14) : TRUE;
                (loc = step16) : TRUE;
                TRUE : mode1;
            esac;
        next(mode2) :=
            case
                (loc = step18) : TRUE;
                TRUE : mode2;
            esac;
        next(mode3) :=
            case
                (loc = step19) : TRUE;
                TRUE : mode3;
            esac;
        next(mode) :=
            case
                (loc = step21) : 1;
                (loc = step22) : 2;
                (loc = step23) : 3;
                (loc = step24) : 0;
                TRUE : mode;
            esac;

--CTL properties
SPEC AG(loc = end -> (out2 -> out1))
SPEC AG(loc = end -> (!signal -> !out2))
SPEC AG(loc = end -> (out3.out1 = out1))
SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

--INVARIANT properties
INVARSPEC(loc = end -> (out2 -> out1))
INVARSPEC(loc = end -> (!signal -> !out2))
INVARSPEC(loc = end -> (out3.out1 = out1))
INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
```

Listing C.3: `Example_int.smv`

```
MODULE main
    VAR
        signal : boolean;
        signal_old : boolean;
        edge_signal : boolean;
        cntr : signed word[16];
        cntr2 : signed word[16];
        out1 : boolean;
        out2 : boolean;
        out3.out1 : boolean;
        out3.elapsed : signed word[16];
        loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
             step11, step12, end};

    ASSIGN
        init(signal_old) := FALSE;
        init(edge_signal) := FALSE;
        init(cntr) := 0sd16_0;
        init(cntr2) := 0sd16_0;
        init(out1) := FALSE;
        init(out2) := FALSE;
        init(out3.out1) := FALSE;
        init(out3.elapsed) := 0sd16_0;
        init(loc) := start;

        next(loc) :=
            case
                (loc = start) : step1;
                (loc = step1) & (signal & !signal_old) : step2;
                (loc = step1) : step3;
                (loc = step2) : step4;
                (loc = step3) : step4;
                (loc = step4) & (!signal) : step5;
                (loc = step4) & (signal) & (edge_signal) : step6;
                (loc = step4) : step7;
                (loc = step5) : end;
                (loc = step6) : end;
                (loc = step7) & (cntr > 0sd16_5) : step8;
                (loc = step7) : step9;
                (loc = step8) : step9;
                (loc = step9) & (cntr2 > 0sd16_5) & (signal) : step10;
                (loc = step9) : step11;
                (loc = step10) : step12;
                (loc = step11) : step12;
                (loc = step12) : end;
                (loc = end) : start;
            esac;

        next(signal) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : signal;
            esac;

        next(signal_old) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : signal;
                TRUE : signal_old;
            esac;
        next(edge_signal) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : FALSE;
                TRUE : edge_signal;
            esac;
        next(cntr) :=
            case
                (loc = step5) : 0sd16_0;
                (loc = step7) : cntr + 0sd16_1;
                (loc = step8) : 0sd16_0;
                TRUE : cntr;
            esac;
        next(cntr2) :=
            case
                (loc = step8) : cntr2 + 0sd16_1;
                TRUE : cntr2;
            esac;
        next(out1) :=
            case
                (loc = step5) : FALSE;
                (loc = step6) : !out1;
                TRUE : out1;
            esac;
        next(out2) :=
            case
                (loc = step5) : FALSE;
                (loc = step10) : TRUE;
                (loc = step11) : FALSE;
                TRUE : out2;
            esac;
        next(out3.out1) :=
            case
                (loc = step6) : !out1;
                (loc = step12) : out1;
                TRUE : out3.out1;
            esac;
        next(out3.elapsed) :=
            case
                (loc = step12) : cntr;
                TRUE : out3.elapsed;
            esac;

--CTL properties
SPEC AG(loc = end -> (out2 -> out1))
SPEC AG(loc = end -> (!signal -> !out2))
SPEC AG(loc = end -> (out3.out1 = out1))
SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

--INVARIANT properties
INVARSPEC(loc = end -> (out2 -> out1))
INVARSPEC(loc = end -> (!signal -> !out2))
INVARSPEC(loc = end -> (out3.out1 = out1))
INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
```

Listing C.4: Reduced version of `Example_int.smv`

```
MODULE main
    VAR
        signal : boolean;
        error : boolean;
        toMode1 : boolean;
        toMode2 : boolean;
        toMode3 : boolean;
        mode3Forbidden : boolean;
        signal_old : boolean;
        edge_signal : boolean;
        cntr : signed word[16];
        cntr2 : signed word[16];
        out1 : boolean;
        out2 : boolean;
        out3.out1 : boolean;
        out3.out2 : boolean;
        out3.remaining : signed word[16];
        out3.elapsed : signed word[16];
        mode1 : boolean;
        mode2 : boolean;
        mode3 : boolean;
        mode : -1..3;
        loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
             step11, step12, step13, step14, step15, step16, step17, step18, step19, step20,
             step21, step22, step23, step24, step25, end};

    ASSIGN
        init(signal_old) := FALSE;
        init(edge_signal) := FALSE;
        init(cntr) := 0sd16_0;
        init(cntr2) := 0sd16_0;
        init(out1) := FALSE;
        init(out2) := FALSE;
        init(out3.out1) := FALSE;
        init(out3.out2) := FALSE;
        init(out3.remaining) := 0sd16_0;
        init(out3.elapsed) := 0sd16_0;
        init(mode1) := FALSE;
        init(mode2) := FALSE;
        init(mode3) := FALSE;
        init(mode) := -1;
        init(loc) := start;

        next(loc) :=
            case
                (loc = start) : step1;
                (loc = step1) & (signal & !signal_old) : step2;
                (loc = step1) : step3;
                (loc = step2) : step4;
                (loc = step3) : step4;
                (loc = step4) & (!signal) : step5;
                (loc = step4) & (signal) & (edge_signal) : step6;
                (loc = step4) : step7;
                (loc = step5) : step14;
                (loc = step6) : step14;
                (loc = step7) : step8;
                (loc = step8) & (cntr > 0sd16_5) : step9;
                (loc = step8) : step10;
                (loc = step9) : step8;
                (loc = step10) & (cntr2 > 0sd16_5) & (signal) : step11;
                (loc = step10) : step12;
                (loc = step11) : step13;
                (loc = step12) : step13;
                (loc = step13) : step14;
                (loc = step14) & (!mode1 & !mode2 & !mode3) : step15;
                (loc = step14) : step16;
                (loc = step15) : step16;
                (loc = step16) & (toMode1) | (toMode3 & mode3Forbidden) : step17;
                (loc = step16) : step18;
                (loc = step17) : step18;
                (loc = step18) & (toMode2) : step19;
                (loc = step18) & (!toMode2) & (toMode3) : step20;
                (loc = step18) : step21;
                (loc = step19) : step21;
                (loc = step20) : step21;
                (loc = step21) & (mode1) : step22;
                (loc = step21) & (!mode1) & (mode2) : step23;
                (loc = step21) & (!mode1) & (!mode2) & (mode3) : step24;
                (loc = step21) : step25;
                (loc = step22) : end;
                (loc = step23) : end;
                (loc = step24) : end;
                (loc = step25) : end;
                (loc = end) : start;
            esac;

        next(signal) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : signal;
            esac;
        next(error) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : error;
            esac;
        next(toMode1) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode1;
            esac;
        next(toMode2) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode2;
            esac;
        next(toMode3) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : toMode3;
            esac;
        next(mode3Forbidden) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : mode3Forbidden;
            esac;

        next(signal_old) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : signal;
                TRUE : signal_old;
            esac;
        next(edge_signal) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : FALSE;
                TRUE : edge_signal;
            esac;
        next(cntr) :=
            case
                (loc = step5) : 0sd16_0;
                (loc = step7) : cntr + 0sd16_1;
                (loc = step9) : cntr - 0sd16_1;
                TRUE : cntr;
            esac;
        next(cntr2) :=
            case
                (loc = step9) : cntr2 + 0sd16_1;
                TRUE : cntr2;
            esac;
        next(out1) :=
            case
                (loc = step5) : FALSE;
                (loc = step6) : !out1;
                TRUE : out1;
            esac;
        next(out2) :=
            case
                (loc = step5) : FALSE;
                (loc = step11) : TRUE;
                (loc = step12) : FALSE;
                TRUE : out2;
            esac;
        next(out3.out1) :=
            case
```

```
                (loc = step6) : !out1;
                (loc = step13) : out1;
                TRUE : out3.out1;
            esac;
        next(out3.out2) :=
            case
                (loc = step13) : out2;
                TRUE : out3.out2;
            esac;
        next(out3.remaining) :=
            case
                (loc = step13) : 0sd16_5 - cntr;
                TRUE : out3.remaining;
            esac;
        next(out3.elapsed) :=
            case
                (loc = step13) : cntr;
                TRUE : out3.elapsed;
            esac;
        next(mode1) :=
            case
                (loc = step15) : TRUE;
                (loc = step17) : TRUE;
                TRUE : mode1;
            esac;
        next(mode2) :=
            case
                (loc = step19) : TRUE;
                TRUE : mode2;
            esac;
        next(mode3) :=
            case
                (loc = step20) : TRUE;
                TRUE : mode3;
            esac;
        next(mode) :=
            case
                (loc = step22) : 1;
                (loc = step23) : 2;
                (loc = step24) : 3;
                (loc = step25) : 0;
                TRUE : mode;
            esac;

--CTL properties
SPEC AG(loc = end -> (out2 -> out1))
SPEC AG(loc = end -> (!signal -> !out2))
SPEC AG(loc = end -> (out3.out1 = out1))
SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

--INVARIANT properties
INVARSPEC(loc = end -> (out2 -> out1))
INVARSPEC(loc = end -> (!signal -> !out2))
INVARSPEC(loc = end -> (out3.out1 = out1))
INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
```

Listing C.5: `Example_while.smv`

```
MODULE main
    VAR
        signal : boolean;
        signal_old : boolean;
        edge_signal : boolean;
        cntr : signed word[16];
        cntr2 : signed word[16];
        out1 : boolean;
        out2 : boolean;
        out3.out1 : boolean;
        out3.elapsed : signed word[16];
        loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
             step11, step12, step13, end};

    ASSIGN
        init(signal_old) := FALSE;
        init(edge_signal) := FALSE;
        init(cntr) := 0sd16_0;
        init(cntr2) := 0sd16_0;
        init(out1) := FALSE;
        init(out2) := FALSE;
        init(out3.out1) := FALSE;
        init(out3.elapsed) := 0sd16_0;
        init(loc) := start;

        next(loc) :=
            case
                (loc = start) : step1;
                (loc = step1) & (signal & !signal_old) : step2;
                (loc = step1) : step3;
                (loc = step2) : step4;
                (loc = step3) : step4;
                (loc = step4) & (!signal) : step5;
                (loc = step4) & (signal) & (edge_signal) : step6;
                (loc = step4) : step7;
                (loc = step5) : end;
                (loc = step6) : end;
                (loc = step7) : step8;
                (loc = step8) & (cntr > 0sd16_5) : step9;
                (loc = step8) : step10;
                (loc = step9) : step8;
                (loc = step10) & (cntr2 > 0sd16_5) & (signal) : step11;
                (loc = step10) : step12;
                (loc = step11) : step13;
                (loc = step12) : step13;
                (loc = step13) : end;
                (loc = end) : start;
            esac;

        next(signal) :=
            case
                (loc = start) : {TRUE, FALSE};
                TRUE : signal;
            esac;

        next(signal_old) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : signal;
                TRUE : signal_old;
            esac;
        next(edge_signal) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : FALSE;
                TRUE : edge_signal;
            esac;
        next(cntr) :=
            case
                (loc = step5) : 0sd16_0;
                (loc = step7) : cntr + 0sd16_1;
                (loc = step9) : cntr - 0sd16_1;
                TRUE : cntr;
            esac;
        next(cntr2) :=
            case
                (loc = step9) : cntr2 + 0sd16_1;
                TRUE : cntr2;
            esac;
        next(out1) :=
            case
                (loc = step5) : FALSE;
                (loc = step6) : !out1;
                TRUE : out1;
            esac;
        next(out2) :=
            case
                (loc = step5) : FALSE;
                (loc = step11) : TRUE;
                (loc = step12) : FALSE;
                TRUE : out2;
            esac;
        next(out3.out1) :=
            case
                (loc = step6) : !out1;
                (loc = step13) : out1;
                TRUE : out3.out1;
            esac;
        next(out3.elapsed) :=
            case
                (loc = step13) : cntr;
                TRUE : out3.elapsed;
            esac;

--CTL properties
SPEC AG(loc = end -> (out2 -> out1))
SPEC AG(loc = end -> (!signal -> !out2))
SPEC AG(loc = end -> (out3.out1 = out1))
SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

--INVARIANT properties
INVARSPEC(loc = end -> (out2 -> out1))
INVARSPEC(loc = end -> (!signal -> !out2))
INVARSPEC(loc = end -> (out3.out1 = out1))
INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
```

Listing C.6: Reduced version of `Example_while.smv`

# D Example programs in C

The translations from the example programs to SMV can be found in this appendix, both the full program and the reduced are shown.

```c
#include <stdbool.h>
bool nondet_bool();

bool R_edge(bool new, bool *old){
    if(new &&!*old){
        *old = true;
        return true;
    } else{
        *old = new;
        return false;
    }
}

int main(){
    struct ComplexSignal{
        bool out1;
        bool out2;
        short remaining;
        short elapsed;
    };
    bool error, signal, toMode1, toMode2, toMode3, mode3Forbidden;
    bool signal_old = false;
    bool edge_signal = false;
    bool mode1 = false;
    bool mode2 = false;
    bool mode3 = false;
    short mode = 0;
    short cntr = 0;
    short cntr_max = 5;
    bool out1 = false;
    bool out2 = false;
    struct ComplexSignal out3;
    out3.out1 = false;
    out3.out2 = false;
    out3.elapsed = 0;
    out3.remaining = 0;

    while(true){
        error = nondet_bool();
        toMode1 = nondet_bool();
        toMode2 = nondet_bool();
        toMode3 = nondet_bool();
        mode3Forbidden = nondet_bool();
        signal = nondet_bool();

        edge_signal = R_edge(signal,&signal_old);
        if(!signal){
            out1 = false;
            out2 = false;
            cntr = 0;
        } else if(edge_signal){
            out1 = !out1;
            out3.out1 = out1;
        } else{
            cntr = cntr + 1;
            if(cntr > cntr_max && signal){
                out2 = true;
            } else{
                out2 = false;
            }
            out3.out1 = out1;
            out3.out2 = out2;
            out3.remaining = cntr_max - cntr;
            out3.elapsed = cntr;
        }
        if (!mode1 && !mode2 && !mode3){
            mode1 = true;
        }
        if(toMode1 || (toMode3 && mode3Forbidden)){
            mode1 = true;
        }
        if(toMode2){
            mode2 = true;
        } else if(toMode3){
            mode3 = true;
        }
        if(mode1){
            mode = 1;
        } else if(mode2){
            mode = 2;
        } else if(mode3){
            mode = 3;
        } else{
            mode = 0;
        }

        assert(!out2 || out1);
        assert(signal || !out2);
        assert(out3.out1 == out1);
        assert(out1 || (out3.elapsed == 0));
    }
}
```

Listing D.1: Example.c

```
#include <stdbool.h>
bool nondet_bool();

bool R_edge(bool new, bool *old){
    if(new &&!*old){
        *old = true;
        return true;
    } else{
        *old = new;
        return false;
    }
}

int main(){
    struct ComplexSignal{
        bool out1;
        short elapsed;
    };
    bool signal;
    bool signal_old = false;
    bool edge_signal = false;
    short cntr = 0;
    short cntr_max = 5;
    bool out1 = false;
    bool out2 = false;
    struct ComplexSignal out3;
    out3.out1 = false;
    out3.elapsed = 0;

    while(true){
        signal = nondet_bool();

        edge_signal = R_edge(signal,&signal_old);

        if(!signal){
            out1 = false;
            out2 = false;
            cntr = 0;
        } else if(edge_signal){
            out1 = !out1;
            out3.out1 = out1;
        } else{
            cntr = cntr + 1;
            if(cntr > cntr_max && signal){
                out2 = true;
            } else{
                out2 = false;
            }
            out3.out1 = out1;
            out3.elapsed = cntr;
        }

        assert(!out2 || out1);
        assert(signal || !out2);
        assert(out3.out1 == out1);
        assert(out1 || (out3.elapsed == 0));
    }
}
```

Listing D.2: Reduced version of Example.c

```
#include <stdbool.h>
bool nondet_bool();

bool R_edge(bool new, bool *old){
    if(new &&!*old){
        *old = true;
        return true;
    } else{
        *old = new;
        return false;
    }
}
```

```

int main(){
    struct ComplexSignal{
        bool out1;
        bool out2;
        short remaining;
        short elapsed;
    };
    bool error, signal, toMode1, toMode2, toMode3, mode3Forbidden;
    bool signal_old = false;
    bool edge_signal = false;
    bool mode1 = false;
    bool mode2 = false;
    bool mode3 = false;
    short mode = 0;
    short cntr = 0;
    short cntr2 = 0;
    short cntr_max = 5;
    bool out1 = false;
    bool out2 = false;
    struct ComplexSignal out3;
    out3.out1 = false;
    out3.out2 = false;
    out3.elapsed = 0;
    out3.remaining = 0;

    while(true){
        error = nondet_bool();
        toMode1 = nondet_bool();
        toMode2 = nondet_bool();
        toMode3 = nondet_bool();
        mode3Forbidden = nondet_bool();
        signal = nondet_bool();

        edge_signal = R_edge(signal,&signal_old);
        if(!signal){
            out1 = false;
            out2 = false;
            cntr = 0;
        } else if(edge_signal){
            out1 = !out1;
            out3.out1 = out1;
        } else{
            cntr = cntr + 1;
            if(cntr > cntr_max){
                cntr = 0;
                cntr2 = cntr2 + 1;
            }
            if(cntr2 > cntr_max && signal){
                out2 = true;
            } else{
                out2 = false;
            }
            out3.out1 = out1;
            out3.out2 = out2;
            out3.remaining = cntr_max - cntr;
            out3.elapsed = cntr;
        }
        if (!mode1 && !mode2 && !mode3){
            mode1 = true;
        }
        if(toMode1 || (toMode3 && mode3Forbidden)){
            mode1 = true;
        }
        if(toMode2){
            mode2 = true;
        } else if(toMode3){
            mode3 = true;
        }
        if(mode1){
            mode = 1;
        } else if(mode2){
            mode = 2;
        } else if(mode3){
            mode = 3;
        } else{
            mode = 0;
        }

        assert(!out2 || out1);
        assert(signal || !out2);
        assert(out3.out1 == out1);
        assert(out1 || (out3.elapsed == 0));
    }
}
```

Listing D.3: `Example_int.c`

```c
#include <stdbool.h>
bool nondet_bool();

bool R_edge(bool new, bool *old){
    if(new &&!*old){
        *old = true;
        return true;
    } else{
        *old = new;
        return false;
    }
}

int main(){
    struct ComplexSignal{
        bool out1;
        short elapsed;
    };
    bool signal;
    bool signal_old = false;
    bool edge_signal = false;
    short cntr = 0;
    short cntr2 = 0;
    short cntr_max = 5;
    bool out1 = false;
    bool out2 = false;
    struct ComplexSignal out3;
    out3.out1 = false;
    out3.elapsed = 0;

    while(true){
        signal = nondet_bool();

        edge_signal = R_edge(signal,&signal_old);
        if(!signal){
            out1 = false;
            out2 = false;
            cntr = 0;
        } else if(edge_signal){
            out1 = !out1;
            out3.out1 = out1;
        } else{
            cntr = cntr + 1;
            if(cntr > cntr_max){
                cntr = 0;
                cntr2 = cntr2 + 1;
            }
            if(cntr2 > cntr_max && signal){
                out2 = true;
            } else{
                out2 = false;
            }
            out3.out1 = out1;
            out3.elapsed = cntr;
        }

        assert(!out2 || out1);
        assert(signal || !out2);
        assert(out3.out1 == out1);
        assert(out1 || (out3.elapsed == 0));
    }
}
```

Listing D.4: Reduced version of `Example_int.c`

```c
#include <stdbool.h>
bool nondet_bool();

bool R_edge(bool new, bool *old){
    if(new &&!*old){
        *old = true;
        return true;
    } else{
        *old = new;
        return false;
    }
}

int main(){
    struct ComplexSignal{
        bool out1;
        bool out2;
        short remaining;
        short elapsed;
    };
    bool error, signal, toMode1, toMode2, toMode3, mode3Forbidden;
    bool signal_old = false;
    bool edge_signal = false;;
    bool mode1 = false;
    bool mode2 = false;
    bool mode3 = false;
    short mode = 0;
    short cntr = 0;
    short cntr2 = 0;
    short cntr_max = 5;
    bool out1 = false;
    bool out2 = false;
    struct ComplexSignal out3;
    out3.out1 = false;
    out3.out2 = false;
    out3.elapsed = 0;
    out3.remaining = 0;

    while(true){
        error = nondet_bool();
        toMode1 = nondet_bool();
        toMode2 = nondet_bool();
        toMode3 = nondet_bool();
        mode3Forbidden = nondet_bool();
        signal = nondet_bool();

        edge_signal = R_edge(signal,&signal_old);
        if(!signal){
            out1 = false;
            out2 = false;
            cntr = 0;
        } else if(edge_signal){
            out1 = !out1;
            out3.out1 = out1;
        } else{
            cntr = cntr + 1;
            while(cntr > cntr_max){
                cntr = cntr - 1;;
                cntr2 = cntr2 + 1;
            }
            if(cntr2 > cntr_max && signal){
                out2 = true;
            } else{
                out2 = false;
            }
            out3.out1 = out1;
            out3.out2 = out2;
            out3.remaining = cntr_max - cntr;
            out3.elapsed = cntr;
        }
        if (!mode1 && !mode2 && !mode3){
            mode1 = true;
        }
        if(toMode1 || (toMode3 && mode3Forbidden)){
            mode1 = true;
        }
        if(toMode2){
            mode2 = true;
        } else if(toMode3){
            mode3 = true;
        }
        if(mode1){
            mode = 1;
        } else if(mode2){
            mode = 2;
        } else if(mode3){
            mode = 3;
        } else{
            mode = 0;
        }

        assert(!out2 || out1);
        assert(signal || !out2);
        assert(out3.out1 == out1);
        assert(out1 || (out3.elapsed == 0));
    }
}
```

Listing D.5: Example_while.c

```
#include <stdbool.h>
bool nondet_bool();

bool R_edge(bool new, bool *old){
    if(new &&!*old){
        *old = true;
        return true;
    } else{
        *old = new;
        return false;
    }
}

int main(){
    struct ComplexSignal{
        bool out1;
        short elapsed;
    };
    bool signal;
    bool signal_old = false;
    bool edge_signal = false;
    short cntr = 0;
    short cntr2 = 0;
    short cntr_max = 5;
    bool out1 = false;
    bool out2 = false;
    struct ComplexSignal out3;
    out3.out1 = false;
    out3.elapsed = 0;

    while(true){
        signal = nondet_bool();

        edge_signal = R_edge(signal,&signal_old);
        if(!signal){
            out1 = false;
            out2 = false;
            cntr = 0;
        } else if(edge_signal){
            out1 = !out1;
            out3.out1 = out1;
        } else{
            cntr = cntr + 1;
            while(cntr > cntr_max){
                cntr = cntr - 1;;
                cntr2 = cntr2 + 1;
            }
            if(cntr2 > cntr_max && signal){
                out2 = true;
            } else{
                out2 = false;
            }
            out3.out1 = out1;
            out3.elapsed = cntr;
        }

        assert(!out2 || out1);
        assert(signal || !out2);
        assert(out3.out1 == out1);
        assert(out1 || (out3.elapsed == 0));
    }
}
```

Listing D.6: Reduced version of `Example_while.smv`

# E CPC program in SCL

```
//UNICOS
//Copyright CERN 2013 all rights reserved

(* ON/OFF OBJECT FUNCTION BLOCK *******************************************)

FUNCTION_BLOCK CPC_FB_ONOFF
TITLE = 'CPC_FB_ONOFF'
//
//  ONOFF Object
//
VERSION: '6.5'
AUTHOR: 'EN/ICE'
NAME: 'OBJECT'
FAMILY: 'FO'

VAR_INPUT

    HFOn:                   BOOL;
    HFOff:                  BOOL;
    HLD:                    BOOL;
    IOError:                BOOL;
    IOSimu:                 BOOL;
    AlB:                    BOOL;
    Manreg01:               WORD;
    Manreg01b AT Manreg01:  ARRAY [0..15] OF BOOL;
    HOnR:                   BOOL;
    HOffR:                      BOOL;
    StartI:                     BOOL;
    TStopI:                     BOOL;
    FuStopI:                    BOOL;
    Al:                         BOOL;
    AuOnR:                      BOOL;
    AuOffR:                     BOOL;
    AuAuMoR:                    BOOL;
    AuIhMMo:                    BOOL;
    AuIhFoMo:                   BOOL;
    AuAlAck:                    BOOL;
    IhAuMRW:                    BOOL;
    AuRstart:                   BOOL;
    POnOff:                     CPC_ONOFF_PARAM;
    POnOffb AT POnOff:          STRUCT
    ParRegb:                    ARRAY [0..15] OF BOOL;
    PPulseLeb:                  TIME;
    PWDtb:                      TIME;
                                END_STRUCT;


END_VAR

VAR_OUTPUT

    Stsreg01:                   WORD;
    Stsreg01b AT Stsreg01:      ARRAY [0..15] OF BOOL;
    Stsreg02:                   WORD;
    Stsreg02b AT Stsreg02:      ARRAY [0..15] OF BOOL;
    OutOnOV:                    BOOL;
    OutOffOV:                   BOOL;
    OnSt:                       BOOL;
    OffSt:                      BOOL;
    AuMoSt:                     BOOL;
    MMoSt:                      BOOL;
    LDSt:                       BOOL;
    SoftLDSt:                   BOOL;
    FoMoSt:                     BOOL;
    AuOnRSt:                    BOOL;
    AuOffRSt:                   BOOL;
    MOnRSt:                     BOOL;
    MOffRSt:                    BOOL;
    HOnRSt:                     BOOL;
    HOffRSt:                    BOOL;
    IOErrorW:                   BOOL;
    IOSimuW:                    BOOL;
    AuMRW:                      BOOL;
    AlUnAck:                    BOOL;
    PosW:                       BOOL;
    StartISt:                   BOOL;
    TStopISt:                   BOOL;
    FuStopISt:                  BOOL;
    AlSt:                       BOOL;
    AlBW:                       BOOL;
    EnRstartSt:                 BOOL := TRUE;
    RdyStartSt:                 BOOL;


END_VAR

VAR  //Internal Variables

    //Variables for Edge detection
    E_MAuMoR:                   BOOL;
    E_MMMoR:                    BOOL;
    E_MFoMoR:                   BOOL;
    E_MOnR:                     BOOL;
    E_MOffR:                    BOOL;
    E_MAlAckR:                  BOOL;
    E_StartI:                   BOOL;
    E_TStopI:                   BOOL;
    E_FuStopI:                  BOOL;
    E_Al:                       BOOL;
    E_AuAuMoR:                  BOOL;
    E_AuAlAck:                  BOOL;
    E_MSoftLDR:                 BOOL;
    E_MEnRstartR:               BOOL;
    RE_AlUnAck:                 BOOL;
    FE_AlUnAck:                 BOOL;
    RE_PulseOn:                 BOOL;
    FE_PulseOn:                 BOOL;
    RE_PulseOff:                BOOL;
    RE_OutOVSt_aux:             BOOL;
    FE_OutOVSt_aux:             BOOL;
    FE_InterlockR:              BOOL;

    //Variables for old values
    MAuMoR_old:                 BOOL;
    MMMoR_old:                  BOOL;
    MFoMoR_old:                 BOOL;
```

```
     MOnR_old:                  BOOL;
     MOffR_old:                 BOOL;
     MAlAckR_old:               BOOL;
     AuAuMoR_old:               BOOL;
     AuAlAck_old:               BOOL;
     StartI_old:                BOOL;
     TStopI_old:                BOOL;
     FuStopI_old:               BOOL;
     Al_old:                    BOOL;
     AlUnAck_old:               BOOL;
     MSoftLDR_old:              BOOL;
     MEnRstartR_old:            BOOL;
     RE_PulseOn_old:            BOOL;
     FE_PulseOn_old:            BOOL;
     RE_PulseOff_old:           BOOL;
     RE_OutOVSt_aux_old:        BOOL;
     FE_OutOVSt_aux_old:        BOOL;
     FE_InterlockR_old:         BOOL;

     //General internal variables
     PFsPosOn:                  BOOL;
     PFsPosOn2:                 BOOL;
     PHFOn:                     BOOL;
     PHFOff:                    BOOL;
     PPulse:                    BOOL;
     PPulseCste:                BOOL;
     PHLD:                      BOOL;
     PHLDCmd:                   BOOL;
     PAnim:                     BOOL;
     POutOff:                   BOOL;
     PEnRstart:                 BOOL;
     PRstartFS:                 BOOL;
     OutOnOVSt:                 BOOL;
     OutOffOVSt:                BOOL;
     AuMoSt_aux:                BOOL;
     MMoSt_aux:                 BOOL;
     FoMoSt_aux:                BOOL;
     SoftLDSt_aux:              BOOL;
     PulseOn:                   BOOL;
     PulseOff:                  BOOL;
     PosW_aux:                  BOOL;
     OutOVSt_aux:               BOOL;
     fullNotAcknowledged:       BOOL;
     PulseOnR:                  BOOL;
     PulseOffR:                 BOOL;
     InterlockR:                BOOL;

     //Variables for IEC Timers
     Time_Warning:              TIME;
     Timer_PulseOn:             TP;
     Timer_PulseOff:            TP;
     Timer_Warning:             TON;

     //Variables for interlock Ststus delay handling
     PulseWidth:                REAL;
     FSIinc:                    INT;
     TSIinc:                    INT;
     SIinc:                     INT;
     Alinc:                     INT;
     WTStopISt:                 BOOL;
     WStartISt:                 BOOL;
     WAlSt:                     BOOL;
     WFuStopISt:                BOOL;

END_VAR

BEGIN

(* INPUT MANAGER *)

      E_MAuMoR      := R_EDGE(new:=ManReg01b[8],old:=MAuMoR_old);
    (* Manual Auto Mode Request    *)
      E_MMMoR       := R_EDGE(new:=ManReg01b[9],old:=MMMoR_old);
    (* Manual Manual Mode Request *)
      E_MFoMoR      := R_EDGE(new:=ManReg01b[10],old:=MFoMoR_old);
    (* Manual Forced Mode Request *)
      E_MSoftLDR    := R_EDGE(new:=ManReg01b[11],old:=MSoftLDR_old);
    (* Manual Software Local Drive Request *)
      E_MOnR        := R_EDGE(new:=ManReg01b[12],old:=MOnR_old);
    (* Manual On/Open Request      *)
      E_MOffR       := R_EDGE(new:=ManReg01b[13],old:=MOffR_old);
    (* Manual Off/close Request    *)
      E_MEnRstartR  := R_EDGE(new:=ManReg01b[1],old:=MEnRstartR_old);
    (* Manual Restart after full stop Request *)
      E_MAlAckR     := R_EDGE(new:=ManReg01b[7],old:=MAlAckR_old);
    (* Manual Alarm Ack. Request   *)

      PFsPosOn   := POnOffb.ParRegb[8];
    (* 1st Parameter bit to define Fail safe position behaviour *)
      PHFOn      := POnOffb.ParRegb[9];
    (* Hardware feedback On present*)
      PHFOff     := POnOffb.ParRegb[10];
    (* Hardware feedback Off present*)
      PPulse     := POnOffb.ParRegb[11];
    (* Object is pulsed pulse duration : POnOff.PulseLe*)
      PHLD       := POnOffb.ParRegb[12];
    (* Local Drive mode Allowed *)
      PHLDCmd    := POnOffb.ParRegb[13];
    (* Local Drive Command allowed *)
      PAnim      := POnOffb.ParRegb[14];
    (* Inverted Output*)
      POutOff    := POnOffb.ParRegb[15];
      PEnRstart  := POnOffb.ParRegb[0];
    (* Enable Restart after Full Stop *)
      PRstartFS  := POnOffb.ParRegb[1];
    (* Enable Restart when Full Stop still active *)
      PFsPosOn2  := POnOffb.ParRegb[2];
    (* 2nd Parameter bit to define Fail safe position behaviour *)
      PPulseCste := POnOffb.ParRegb[3];
    (* Pulse Constant duration irrespective of the feedback status *)

      E_AuAuMoR := R_EDGE(new:=AuAuMoR, old:=AuAuMoR_old);
    (* Auto Auto Mode Request  *)
      E_AuAlAck := R_EDGE(new:=AuAlAck, old:=AuAlAck_old);
    (* Auto Alarm Ack. Request *)

      E_StartI    := R_EDGE(new:=StartI, old:=StartI_old);
      E_TStopI    := R_EDGE(new:=TStopI, old:=TStopI_old);
      E_FuStopI   := R_EDGE(new:=FuStopI, old:=FuStopI_old);
      E_Al        := R_EDGE(new:=Al, old:=Al_old);

      StartISt   := StartI;
    (* Start Interlock present *)
      TStopISt   := TStopI;
    (* Temporary Stop Interlock present  *)
      FuStopISt  := FuStopI;
    (* Full Stop Interlock present  *)

(* INTERLOCK & ACKNOWLEDGE *)

     IF (E_MAlAckR OR E_AuAlAck) THEN
         fullNotAcknowledged := FALSE;
         AlUnAck := FALSE;
     ELSIF (E_TStopI OR  E_StartI OR E_FuStopI OR E_Al) THEN
         AlUnAck := TRUE;
     END_IF;

     IF ((PEnRstart AND (E_MEnRstartR OR AuRstart) AND NOT FuStopISt) OR (PEnRstart AND
          PRstartFS AND (E_MEnRstartR OR AuRstart))) AND NOT fullNotAcknowledged THEN
         EnRstartSt := TRUE;
     END_IF;

     IF E_FuStopI THEN
         fullNotAcknowledged := TRUE;
         IF PEnRstart THEN
             EnRstartSt := FALSE;
         END_IF;
     END_IF;

     InterlockR :=   TStopISt OR FuStopISt OR FullNotAcknowledged OR NOT EnRstartSt OR
                     (StartISt AND NOT POutOff AND NOT OutOnOV) OR
                     (StartISt AND POutOff AND ((PFsPosOn AND OutOVSt_aux) OR (NOT PFsPosOn
                          AND NOT OutOVSt_aux)));

     FE_InterlockR   := F_EDGE (new:=InterlockR, old:=FE_InterlockR_old);

(* MODE MANAGER *)

      IF NOT (HLD AND PHLD) THEN

(* Forced Mode *)
               IF (AuMoSt_aux OR MMoSt_aux OR SoftLDSt_aux) AND
                    E_MFoMoR AND NOT(AuIhFoMo) THEN
                        AuMoSt_aux    := FALSE;
                        MMoSt_aux     := FALSE;
                        FoMoSt_aux    := TRUE;
                        SoftLDSt_aux  := FALSE;
             END_IF;

(* Manual Mode *)
               IF (AuMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux) AND
                    E_MMMoR AND NOT(AuIhMMo) THEN
                        AuMoSt_aux    := FALSE;
                        MMoSt_aux     := TRUE;
                        FoMoSt_aux    := FALSE;
                        SoftLDSt_aux  := FALSE;
               END_IF;

(* Auto Mode *)
               IF (MMoSt_aux AND (E_MAuMoR OR E_AuAuMoR )) OR
```

```
                    (FoMoSt_aux AND E_MAuMoR) OR
                    (SoftLDSt_aux AND E_MAuMoR) OR
                    (MMoSt_aux AND AuIhMMo) OR
                    (FoMoSt_aux AND AuIhFoMo)OR
                    (SoftLDSt_aux AND AuIhFoMo) OR
                     NOT(AuMoSt_aux OR MMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux) THEN
                         AuMoSt_aux    := TRUE;
                         MMoSt_aux     := FALSE;
                         FoMoSt_aux    := FALSE;
                         SoftLDSt_aux  := FALSE;
                END_IF;

(* Software Local Mode *)
                IF (AuMoSt_aux OR MMoSt_aux) AND E_MSoftLDR AND NOT AuIhFoMo THEN
                         AuMoSt_aux   := FALSE;
                         MMoSt_aux    := FALSE;
                         FoMoSt_aux   := FALSE;
                         SoftLDSt_aux:= TRUE;
                END_IF;

(* Status setting *)
               LDSt      := FALSE;
               AuMoSt    := AuMoSt_aux;
               MMoSt     := MMoSt_aux;
               FoMoSt    := FoMoSt_aux;
               SoftLDSt  := SoftLDSt_aux;
      ELSE
(* Local Drive Mode *)
               AuMoSt   := FALSE;
               MMoSt    := FALSE;
               FoMoSt   := FALSE;
               LDSt     := TRUE;
               SoftLDSt:= FALSE;
      END_IF;

(* LIMIT MANAGER *)

(* On/Open Evaluation *)
          OnSt:= (HFOn AND PHFOn) OR
            (*Feedback ON present*)
                   (NOT PHFOn AND PHFOff AND PAnim AND NOT HFOff) OR
            (*Feedback ON not present and PAnim = TRUE*)
                   (NOT PHFOn AND NOT PHFOff AND OutOVSt_aux);



(* Off/Closed Evaluation *)
          OffSt:=(HFOff AND PHFOff) OR
          (*Feedback OFF present*)
                   (NOT PHFOff AND PHFOn AND PAnim AND NOT HFOn) OR
          (*Feedback OFF not present and PAnim = TRUE*)
                   (NOT PHFOn AND NOT PHFOff AND NOT OutOVSt_aux);

(* REQUEST MANAGER *)

(* Auto On/Off Request*)

         IF AuOffR THEN
               AuOnRSt := FALSE;
         ELSIF AuOnR THEN
               AuOnRSt := TRUE;
         ELSIF fullNotAcknowledged OR FuStopISt OR NOT EnRstartSt THEN
               AuOnRSt :=  PFsPosOn;
         END_IF;
         AuOffRSt:= NOT AuOnRSt;

(* Manual On/Off Request*)

         IF (((E_MOffR AND (MMoSt OR FoMoSt OR SoftLDSt))
             OR (AuOffRSt AND AuMoSt)
             OR (LDSt AND PHLDCmd AND HOffRSt)
             OR (FE_PulseOn AND PPulse AND NOT POutOff) AND EnRstartSt)
             OR (E_FuStopI AND NOT PFsPosOn)) THEN

                    MOnRSt := FALSE;

         ELSIF (((E_MOnR  AND (MMoSt OR FoMoSt OR SoftLDSt))
                OR (AuOnRSt  AND AuMoSt)
                OR (LDSt AND PHLDCmd AND HOnRSt) AND EnRstartSt)
                OR (E_FuStopI AND PFsPosOn)) THEN

                    MOnRSt := TRUE;
         END_IF;

         MOffRSt:= NOT MOnRSt;

(* Local Drive Request *)

         IF HOffR THEN
               HOnRSt := FALSE;
```

```
        ELSE IF HOnR THEN
            HOnRSt := TRUE;
             END_IF;
        END_IF;
        HOffRSt :=  NOT(HOnRSt);


(* PULSE REQUEST MANAGER*)
  IF PPulse THEN
        IF InterlockR THEN
           PulseOnR:= (PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2);
           PulseOffR:= (NOT PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2);
        ELSIF FE_InterlockR THEN
        (*Clear PulseOnR/PulseOffR to be sure you get a new pulse after InterlockR*)
           PulseOnR:= FALSE;
           PulseOffR:= FALSE;
           Timer_PulseOn (IN:=FALSE,PT:=T#0s);
           Timer_PulseOff (IN:=FALSE,PT:=T#0s);
        ELSIF (MOffRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuOffRSt AND AuMoSt) OR (HOffR
             AND LDSt AND  PHLDCmd) THEN  //Off Request
           PulseOnR:= FALSE;
           PulseOffR:= TRUE;
        ELSIF (MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuOnRSt AND AuMoSt) OR (HOnR
            AND LDSt AND PHLDCmd) THEN //On Request
            PulseOnR:= TRUE;
            PulseOffR:= FALSE;
        ELSE
           PulseOnR:= FALSE;
           PulseOffR:= FALSE;
        END_IF;


        //Pulse functions
        Timer_PulseOn (IN:= PulseOnR,PT:=POnOffb.PPulseLeb);
        Timer_PulseOff (IN:=PulseOffR,PT:=POnOffb.PPulseLeb);

        RE_PulseOn    := R_EDGE(new:=PulseOn,old:=RE_PulseOn_old);
        FE_PulseOn    := F_EDGE(new:=PulseOn,old:=FE_PulseOn_old);
        RE_PulseOff   := R_EDGE(new:=PulseOff,old:=RE_PulseOff_old);

        //The pulse functions have to be reset when changing from On to Off
        IF RE_PulseOn THEN
            Timer_PulseOff (IN:=FALSE,PT:=T#0s);
        END_IF;

        IF RE_PulseOff THEN
           Timer_PulseOn (IN:=FALSE,PT:=T#0s);
        END_IF;

        IF PPulseCste THEN
        (* Pulse constant duration irrespective of feedback status *)
            PulseOn  := Timer_PulseOn.Q AND NOT PulseOffR;
            PulseOff := Timer_PulseOff.Q AND NOT PulseOnR;
        ELSE
            PulseOn  := Timer_PulseOn.Q AND NOT PulseOffR AND (NOT PHFOn OR (PHFOn AND NOT
                HFOn));
            PulseOff := Timer_PulseOff.Q AND NOT PulseOnR AND (NOT PHFOff OR (PHFOff AND NOT
                 HFOff));
        END_IF;
 END_IF;

(* Output On Request *)
       OutOnOVSt := (PPulse AND PulseOn) OR
                    (NOT PPulse AND ((MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR
                    (AuOnRSt AND AuMoSt) OR
                    (HOnRST AND LDSt AND PHLDCmd)));

(* Output Off Request *)
       IF POutOff THEN
            OutOffOVSt := (PulseOff AND PPulse) OR
                          (NOT(PPulse) AND ((MOffRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (
                              AuOffRSt AND AuMoSt) OR (HOffRST AND LDSt AND  PHLDCmd)));
       END_IF;

(* Interlocks / FailSafe *)

       IF POutOff THEN
            IF InterlockR THEN
                IF PPulse AND NOT PFsPosOn2 THEN
                    IF PFsPosOn THEN
                        OutOnOVSt := PulseOn;
                        OutOffOVSt :=  FALSE;
                    ELSE
                        OutOnOVSt := FALSE;
                        OutOffOVSt :=  PulseOff;
                    END_IF;
                ELSE
                    OutOnOVSt := (PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2);
                    OutOffOVSt:= (NOT PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2
                        );
                END_IF;
            END_IF;
        ELSE
            IF InterlockR THEN
                    OutOnOVSt:= PFsPosOn;
            END_IF;
        END_IF;

(* Ready to Start Status *)

     RdyStartSt := NOT InterlockR;

(*Alarms*)

     AlSt := Al;

(* SURVEILLANCE *)

(* I/O Warning *)
     IOErrorW := IOError;
     IOSimuW  := IOSimu;

(* Auto<> Manual Warning *)
     AuMRW     := (MMoSt OR FoMoSt OR SoftLDSt) AND
                  ((AuOnRSt XOR MOnRSt) OR (AuOffRSt XOR MOffRSt)) AND NOT IhAuMRW;



(* OUTPUT_MANAGER AND OUTPUT REGISTER *)
    IF NOT POutOff THEN
        IF PFsPosOn THEN
             OutOnOV := NOT OutOnOVSt;
        ELSE
             OutOnOV := OutOnOVSt;
        END_IF;
    ELSE
        OutOnOV   := OutOnOVSt;
        OutOffOV  := OutOffOVSt;
    END_IF;

(* Position warning *)

(* Set reset of the OutOnOVSt *)
         IF OutOnOVSt OR (PPulse AND PulseOnR) THEN
               OutOVSt_aux := TRUE;
        END_IF;
        IF  (OutOffOVSt AND POutOff) OR (NOT OutOnOVSt AND NOT POutOff) OR (PPulse AND
             PulseOffR) THEN
               OutOVSt_aux := FALSE;
        END_IF;

        RE_OutOVSt_aux    := R_EDGE(new:=OutOVSt_aux,old:=RE_OutOVSt_aux_old);
        FE_OutOVSt_aux    := F_EDGE(new:=OutOVSt_aux,old:=FE_OutOVSt_aux_old);

    IF ((OutOVSt_aux AND ((PHFOn AND NOT OnSt) OR (PHFOff AND OffSt)))
        OR (NOT OutOVSt_aux AND ((PHFOff AND NOT OffSt) OR (PHFON AND OnSt)))
        OR (OffSt AND OnSt))
        AND (NOT PPulse OR (POutOff AND PPulse AND NOT OutOnOV AND NOT OutOffOV))
    THEN
        PosW_aux:= TRUE;
    END_IF;

    IF  NOT ((OutOVSt_aux AND ((PHFOn AND NOT OnSt) OR (PHFOff AND OffSt)))
        OR (NOT OutOVSt_aux AND ((PHFOff AND NOT OffSt) OR (PHFON AND OnSt)))
        OR (OffSt AND OnSt))
        OR RE_OutOVSt_aux
        OR FE_OutOVSt_aux
        OR (PPulse AND POutOff AND OutOnOV)
        OR (PPulse AND POutOff AND OutOffOV)
    THEN
        PosW_aux := FALSE;
    END_IF;

    Timer_Warning(IN := PosW_aux,
                  PT := POnOffb.PWDtb);

    PosW := Timer_Warning.Q;
    Time_Warning := Timer_Warning.ET;

(* Alarm Blocked Warning*)

    AlBW := AlB;

(* Maintain Interlock status 1.5s in Stsreg for PVSS *)

PulseWidth := 1500 (* msec*) / DINT_TO_REAL(TIME_TO_DINT(T_CYCLE));


IF FuStopISt OR FSIinc > 0 THEN
    FSIinc := FSIinc + 1;
```

```
     WFuStopISt := TRUE;
END_IF;

IF FSIinc > PulseWidth OR (NOT FuStopISt AND FSIinc = 0) THEN
     FSIinc := 0;
     WFuStopISt := FuStopISt;
END_IF;

IF TStopISt OR TSIinc > 0 THEN
     TSIinc := TSIinc + 1;
     WTStopISt := TRUE;
END_IF;

IF TSIinc > PulseWidth OR (NOT TStopISt AND TSIinc = 0) THEN
     TSIinc := 0;
     WTStopISt := TStopISt;
END_IF;

IF StartISt OR SIinc > 0 THEN
     SIinc := SIinc + 1;
     WStartISt:= TRUE;
END_IF;

IF SIinc > PulseWidth OR (NOT StartISt AND SIinc = 0) THEN
     SIinc := 0;
     WStartISt := StartISt;
END_IF;

IF AlSt OR Alinc > 0 THEN
     Alinc := Alinc + 1;
     WAlSt := TRUE;
END_IF;

IF Alinc > PulseWidth OR (NOT AlSt AND Alinc = 0) THEN
     Alinc := 0;
     WAlSt := AlSt;
END_IF;


(* STATUS REGISTER *)

     Stsreg01b[8]  := OnSt;            //StsReg01 Bit 00
     Stsreg01b[9]  := OffSt;           //StsReg01 Bit 01
     Stsreg01b[10] := AuMoSt;          //StsReg01 Bit 02
     Stsreg01b[11] := MMoSt;           //StsReg01 Bit 03
     Stsreg01b[12] := FoMoSt;          //StsReg01 Bit 04
     Stsreg01b[13] := LDSt;            //StsReg01 Bit 05
     Stsreg01b[14] := IOErrorW;        //StsReg01 Bit 06
     Stsreg01b[15] := IOSimuW;         //StsReg01 Bit 07
     stsreg01b[0]  := AuMRW;           //StsReg01 Bit 08
     Stsreg01b[1]  := PosW;            //StsReg01 Bit 09
     Stsreg01b[2]  := WStartISt;       //StsReg01 Bit 10
     Stsreg01b[3]  := WTStopISt;       //StsReg01 Bit 11
     Stsreg01b[4]  := AlUnAck;         //StsReg01 Bit 12
     Stsreg01b[5]  := AuIhFoMo;        //StsReg01 Bit 13
     Stsreg01b[6]  := WAlSt;           //StsReg01 Bit 14
     Stsreg01b[7]  := AuIhMMo;         //StsReg01 Bit 15

     Stsreg02b[8]  := OutOnOVSt;       //StsReg02 Bit 00
     Stsreg02b[9]  := AuOnRSt;         //StsReg02 Bit 01
     Stsreg02b[10] := MOnRSt;          //StsReg02 Bit 02
     Stsreg02b[11] := AuOffRSt;        //StsReg02 Bit 03
     Stsreg02b[12] := MOffRSt;         //StsReg02 Bit 04
     Stsreg02b[13] := HOnRSt;          //StsReg02 Bit 05
     Stsreg02b[14] := HOffRSt;         //StsReg02 Bit 06
     Stsreg02b[15] := 0;               //StsReg02 Bit 07
     stsreg02b[0]  := 0;               //StsReg02 Bit 08
     Stsreg02b[1]  := 0;               //StsReg02 Bit 09
     Stsreg02b[2]  := WFuStopISt ;     //StsReg02 Bit 10
     Stsreg02b[3]  := EnRstartSt;      //StsReg02 Bit 11
     Stsreg02b[4]  := SoftLDSt;        //StsReg02 Bit 12
     Stsreg02b[5]  := AlBW;            //StsReg02 Bit 13
     Stsreg02b[6]  := OutOffOVSt;      //StsReg02 Bit 14
     Stsreg02b[7]  := 0;               //StsReg02 Bit 15

(* Edges *)

  DETECT_EDGE(new:=AlUnAck, old:=AlUnAck_old, re:=RE_AlUnAck, fe:=FE_AlUnAck);


END_FUNCTION_BLOCK


/////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////
//                                                                                 //
//  Common functions for UNICOS applications + implementation of platform FBs.     //
//                                                                                 //
/////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////////////////////////////////
```

```



//UNICOS
//  Copyright CERN 2013 all rights reserved

(* DATA STRUCTURES ********************************************************************)

TYPE CPC_ONOFF_PARAM
TITLE = 'CPC_ONOFF_PARAM'
//
// Parameters of ONOFF
//
AUTHOR : 'EN/ICE'
NAME : 'DataType'
FAMILY : 'Base'
    STRUCT
        ParReg : WORD;
        PPulseLe : TIME;
        PWDt : TIME;
    END_STRUCT
END_TYPE

(* OTHER FUNCTIONS ********************************************************************)

(*Rising Edge*)
FUNCTION R_EDGE : BOOL
TITLE = 'R_EDGE'
//
// Detect a Rising Edge on a signal
//
AUTHOR : 'EN/ICE'
NAME : 'Function'
FAMILY : 'Base'
VAR_INPUT
    new : BOOL;
END_VAR
VAR_IN_OUT
    old : BOOL;
END_VAR
BEGIN

    IF (new = 1 AND old = 0) THEN //Raising edge detected
        R_EDGE := 1;
        old := 1;
    ELSE R_EDGE := 0;
        old := new;
    END_IF;
END_FUNCTION

(*Falling Edge*)
FUNCTION F_EDGE : BOOL
TITLE = 'F_EDGE'
//
// Detect a Falling Edge on a signal
//
AUTHOR : 'EN/ICE'
NAME : 'Function'
FAMILY : 'Base'
VAR_INPUT
    new : BOOL;
END_VAR
VAR_IN_OUT
    old : BOOL;
END_VAR
BEGIN

    IF (new = 0 AND old = 1) THEN //Falling edge detected
        F_EDGE := 1;
        old := 0;
    ELSE F_EDGE := 0;
        old := new;
    END_IF;
END_FUNCTION

(*Rising and Falling Edge*)
FUNCTION DETECT_EDGE : VOID
TITLE = 'DETECT_EDGE'
//
// Detect a Rising and Falling Edge of a signal
//
AUTHOR : 'EN/ICE'
NAME : 'Function'
FAMILY : 'Base'
VAR_INPUT
    new : BOOL;
END_VAR
VAR_IN_OUT
    old : BOOL;
END_VAR
```

```
VAR_OUTPUT
     re : BOOL;
     fe : BOOL;
END_VAR

BEGIN
     IF new <> old THEN
          IF new = true THEN // Raising edge
               re := true;
               fe := false;
          ELSE // Falling edge
               re := false;
               fe := true;
          END_IF;
          old := new; // shift new to old

     ELSE re := false; // reset edge detection
          fe := false;
     END_IF;
END_FUNCTION

// TIMERS
//# GLOBALVAR __GLOBAL_TIME : TIME;
//# GLOBALVAR T_CYCLE : UINT;

// Pulse timer
FUNCTION_BLOCK TP
// updated on 10 Oct 2013
VAR_INPUT
     PT : TIME;
END_VAR

VAR_IN_OUT
     IN : BOOL;
END_VAR

VAR_OUTPUT
     Q : BOOL := FALSE;
     ET : TIME; // elapsed time

END_VAR

VAR
     old_in : BOOL := FALSE;
     due : TIME := T#0ms;
END_VAR

BEGIN
     if (in and not old_in) and not Q then
          due := __GLOBAL_TIME + pt;
     end_if;
     if __GLOBAL_TIME <= due then
          Q := true;
          ET := PT - (due - __GLOBAL_TIME);
     else Q := false;
          if in then
               ET := PT;
          else ET := 0;
          end_if;
     end_if;
     old_in := in;
END_FUNCTION_BLOCK

// On-delay timer
FUNCTION_BLOCK TON
// !!! It is assumed that PT>0, if IN=true. !!!
// updated on 20/05/2014
VAR_INPUT
     PT : TIME; // pulse time
     IN : BOOL;
END_VAR

VAR_OUTPUT
     Q : BOOL := FALSE;
     ET : TIME := T#0s; // elapsed time

END_VAR

VAR
     running : BOOL := FALSE;
     start : TIME := T#0ms; //STIME in Siemens implementation (?)
END_VAR

BEGIN
     if IN = false then
          Q := false;
          ET := T#0s;
          running := false;
     else
     // in this case IN == TRUE
```

```
        if running = false then
        // just started
            start := __GLOBAL_TIME;
            running := true;
            ET := 0;
            // Q = false , ET = 0

        else
            if not (__GLOBAL_TIME - (start + pt) >= T#0s) then
            // running , but no timeout
            // do not modify Q
                if not Q then
                    ET := __GLOBAL_TIME - start;
                end_if;
                // ET should be good even if GT>0 & due < 0

            else
            // timeout
                Q := true;
                ET := PT;
            end_if;
        end_if;
    end_if;
END_FUNCTION_BLOCK
```

Listing E.1: CPC.scl

## F CPC program in SMV

This appendix shows the translation from the CPC program to SMV. Note that the properties can be found in Appendix H.

```
MODULE main
Baseline_version
--Variable declaration
VAR
    --VAR_INPUT
    HFOn : boolean;
    HFOff : boolean;
    HLD : boolean;
    IOError : boolean;
    IOSimu : boolean;
    AlB : boolean;
    Manreg01b : array 0..15 of boolean;
    HOnR : boolean;
    HOffR : boolean;
    StartI : boolean;
    TStopI : boolean;
    FuStopI : boolean;
    Al : boolean;
    AuOnR : boolean;
    AuOffR : boolean;
    AuAuMoR : boolean;
    AuIhMMo : boolean;
    AuIhFoMo : boolean;
    AuAlAck : boolean;
    IhAuMRW : boolean;
    AuRstart : boolean;
    POnOffb.ParRegb : array 0..15 of boolean;
    POnOffb.PPulseLeb : signed word[32];
    POnOffb.PWDtb : signed word[32];

    --VAR_OUTPUT
    Stsreg01b : array 0..15 of boolean;
    Stsreg02b : array 0..15 of boolean;
    OutOnOV : boolean;
    OutOffOV : boolean;
    OnSt : boolean;
    OffSt : boolean;
    AuMoSt : boolean;
    MMoSt : boolean;
    LDSt : boolean;
    SoftLDSt : boolean;
    FoMoSt : boolean;
    AuOnRSt : boolean;
    AuOffRSt : boolean;
    MOnRSt : boolean;
    MOffRSt : boolean;
    HOnRSt : boolean;
    HOffRSt : boolean;
    IOErrorW : boolean;
    IOSimuW : boolean;
    AuMRW : boolean;
    AlUnAck : boolean;
    PosW : boolean;
    StartISt : boolean;
    TStopISt : boolean;
    FuStopISt : boolean;
    AlSt : boolean;
    AlBW : boolean;
    EnRstartSt : boolean;
    RdyStartSt : boolean;

    --Internal Variables
    --Variables for Edge detection
    E_MAuMoR : boolean;
    E_MMMoR : boolean;
    E_MFoMoR : boolean;
    E_MOnR : boolean;
    E_MOffR : boolean;
    E_MAlAckR : boolean;
    E_StartI : boolean;
    E_TStopI : boolean;
    E_FuStopI : boolean;
    E_Al : boolean;
    E_AuAuMoR : boolean;
    E_AuAlAck : boolean;
    E_MSoftLDR : boolean;
    E_MEnRstartR : boolean;
    RE_AlUnAck : boolean;
    FE_AlUnAck : boolean;
    RE_PulseOn : boolean;
    FE_PulseOn : boolean;
    RE_PulseOff : boolean;
    RE_OutOVSt_aux : boolean;
    FE_OutOVSt_aux : boolean;
    FE_InterlockR : boolean;

    --Variables for old values
    MAuMoR_old : boolean;
    MMMoR_old : boolean;
    MFoMoR_old : boolean;
    MOnR_old : boolean;
    MOffR_old : boolean;
    MAlAckR_old : boolean;
    AuAuMoR_old : boolean;
    AuAlAck_old : boolean;
    StartI_old : boolean;
    TStopI_old : boolean;
    FuStopI_old : boolean;
    Al_old : boolean;
    AlUnAck_old : boolean;
    MSoftLDR_old : boolean;
    MEnRstartR_old : boolean;
    RE_PulseOn_old : boolean;
    FE_PulseOn_old : boolean;
    RE_PulseOff_old : boolean;
    RE_OutOVSt_aux_old : boolean;
    FE_OutOVSt_aux_old : boolean;
    FE_InterlockR_old : boolean;

    --General internal variables
    PFsPosOn : boolean;
    PFsPosOn2 : boolean;
    PHFOn : boolean;
    PHFOff : boolean;
    PPulse : boolean;
    PPulseCste : boolean;
    PHLD : boolean;
    PHLDCmd : boolean;
    PAnim : boolean;
    POutOff : boolean;
    PEnRstart : boolean;
    PRstartFS : boolean;
    OutOnOVSt : boolean;
    OutOffOVSt : boolean;
    AuMoSt_aux : boolean;
    MMoSt_aux : boolean;
    FoMoSt_aux : boolean;
    SoftLDSt_aux : boolean;
    PulseOn : boolean;
    PulseOff : boolean;
    PosW_aux : boolean;
    OutOVSt_aux : boolean;
    fullNotAcknowledged : boolean;
    PulseOnR : boolean;
    PulseOffR : boolean;
    InterlockR : boolean;

    --Variables for IEC Timers
    Time_Warning : signed word[32];
    Timer_PulseOn.Q : boolean;
    Timer_PulseOn.ET : signed word[32];
    Timer_PulseOn.old_in : boolean;
    Timer_PulseOn.due : signed word[32];
    Timer_PulseOff.Q : boolean;
    Timer_PulseOff.ET : signed word[32];
    Timer_PulseOff.old_in : boolean;
    Timer_PulseOff.due : signed word[32];
    Timer_Warning.Q : boolean;
    Timer_Warning.ET : signed word[32];
    Timer_Warning.running : boolean;
    Timer_Warning.Tstart : signed word[32];

    --Variables for interlock Ststus delay handling
    PulseWidth : signed word[32];
    FSIinc : signed word[16];
    TSIinc : signed word[16];
    SIinc : signed word[16];
    Alinc : signed word[16];
    WTStopISt : boolean;
    WStartISt : boolean;
    WAlSt : boolean;
    WFuStopISt : boolean;
    __GLOBAL_TIME : signed word[32];
    T_CYCLE : unsigned word[16];
    random_t_cycle : unsigned word[8];

    --extra assertion variables
    sFoMoSt_aux : boolean;
    sAuAuMoR : boolean;
    sManreg01b8 : boolean;
    sAuIhFoMo : boolean;
    sAuIhMMo : boolean;
    sMMoSt_aux : boolean;
    pOutOnOV : boolean;
    pTStopI : boolean;
    pFuStopI : boolean;
    pStartI : boolean;
    pOutOnOVSt : boolean;
    pMMoSt : boolean;
    pManreg01b12 : boolean;
    pManreg01b13 : boolean;
    first : boolean;

    loc:{start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10, step
        11, step12, step13, step14, step15,
    step16, step17, step18, step19, step20, step21, step22, step23, step24, step25, step26,
        step27, step28, step29, step30,
    step31, step32, step33, step34, step35, step36, step37, step38, step39, step40, step41,
        step42, step43, step44, step45,
    step46, step47, step48, step49, step50, step51, step52, step53, step54, step55, step56,
        step57, step58, step59, step60,
    step61, step62, step63, step64, step65, step66, step67, step68, step69, step70, step71,
        step72, step73, step74, step75,
    step76, step77, step78, step79, step80, step81, step82, step83, step84, step85, step86,
        step87, step88, step89, step90,
    step91, step92, step93, step94, step95, step96, step97, step98, step99, step100, step
        101, step102, step103, step104, step105,
    step106, step107, step108, step109, step110, step111, step112, step113, step114, step
        115, step116, step117, step118, step119,
    step120, step121, step122, step123, step124, step125, step126, step127, step128, step
        129, step130, step131, step132, step133,
    step134, step135, step136, step137, step138, step139, step140, step141, step142, step
        143, step144, step145, step146, step147,
    step148, step149, step150, step151, step152, step153, step154, step155, step156, step
        157, step158, step159, step160, step161,
    step162, step163, step164, step165, step166, step167, step168, step169, step170, step
        171, step172, step173, step174, step175,
    step176, step177, step178, step179, step180, step181, step182, step183, step184, step
        185, step186, step187, step188, step189,
    step190, step191, step192, step193, step194, step195, step196, step197, step198, step
        199, step200, step201, step202, step203,
    step204, step205, step206, step207, step208, step209, step210, step211, step212, step
        213, step214, step215, step216, step217,
    step218, step219, step220, step221, end, nvar};

ASSIGN
    --Location specification
    init(loc) := start;
    next(loc) :=
        case
            (loc = start) : step1;
            (loc = step1) & (Manreg01b[8] & !MAuMoR_old) : step2;
            (loc = step1) : step3;
            (loc = step2) : step4;
            (loc = step3) : step4;
            (loc = step4) & (Manreg01b[9] & !MMMoR_old) : step5;
            (loc = step4) : step6;
            (loc = step5) : step7;
            (loc = step6) : step7;
            (loc = step7) & (Manreg01b[10] & !MFoMoR_old) : step8;
            (loc = step7) : step9;
            (loc = step8) : step10;
            (loc = step9) : step10;
            (loc = step10) & (Manreg01b[11] & !MSoftLDR_old) : step11;
            (loc = step10) : step12;
            (loc = step11) : step13;
            (loc = step12) : step13;
            (loc = step13) & (Manreg01b[12] & !MOnR_old) : step14;
            (loc = step13) : step15;
            (loc = step14) : step16;
            (loc = step15) : step16;
            (loc = step16) & (Manreg01b[13] & !MOffR_old) : step17;
            (loc = step16) : step18;
            (loc = step17) : step19;
            (loc = step18) : step19;
            (loc = step19) & (Manreg01b[1] & !MEnRstartR_old) : step20;
            (loc = step19) : step21;
            (loc = step20) : step22;
            (loc = step21) : step22;
            (loc = step22) & (Manreg01b[7] & !MAlAckR_old) : step23;
            (loc = step22) : step24;
            (loc = step23) : step25;
            (loc = step24) : step25;
            (loc = step25) : step26;
            (loc = step26) & (AuAuMoR & !AuAuMoR_old) : step27;
            (loc = step26) : step28;
            (loc = step27) : step29;
            (loc = step28) : step29;
            (loc = step29) & (AuAlAck & !AuAlAck_old) : step30;
            (loc = step29) : step31;
            (loc = step30) : step32;
            (loc = step31) : step32;
            (loc = step32) & (StartI & !StartI_old) : step33;
            (loc = step32) : step34;
            (loc = step33) : step35;
            (loc = step34) : step35;
            (loc = step35) & (TStopI & !TStopI_old) : step36;
            (loc = step35) : step37;
            (loc = step36) : step38;
            (loc = step37) : step38;
            (loc = step38) & (FuStopI & !FuStopI_old) : step39;
            (loc = step38) : step40;
            (loc = step39) : step41;
            (loc = step40) : step41;
            (loc = step41) & (Al & !Al_old) : step42;
            (loc = step41) : step43;
            (loc = step42) : step44;
            (loc = step43) : step44;
            (loc = step44) : step45;
            (loc = step45) & (E_MAlAckR | E_AuAlAck) : step46;
            (loc = step45) & (E_TStopI |  E_StartI | E_FuStopI | E_Al) : step47;
            (loc = step45) : step48;
            (loc = step46) : step48;
            (loc = step47) : step48;
            (loc = step48) & (((PEnRstart & (E_MEnRstartR | AuRstart) & !FuStopISt) | (
                PEnRstart & PRstartFS & (E_MEnRstartR | AuRstart))) & !fullNotAcknowledged)
                : step49;
            (loc = step48) : step50;
            (loc = step49) : step50;
            (loc = step50) & (E_FuStopI) : step51;
            (loc = step50) : step54;
            (loc = step51) : step52;
            (loc = step52) & (PEnRstart) : step53;
            (loc = step52) : step54;
            (loc = step53) : step54;
            (loc = step54) : step55;
            (loc = step55) & (!InterlockR & FE_InterlockR_old) : step56;
            (loc = step55) : step57;
            (loc = step56) : step58;
            (loc = step57) : step58;
            (loc = step58) & (!(HLD & PHLD)) : step59;
            (loc = step58) : step68;
            (loc = step59) & ((AuMoSt_aux | MMoSt_aux | SoftLDSt_aux) & E_MFoMoR & !(
                AuIhFoMo)) : step60;
            (loc = step59) : step61;
            (loc = step60) : step61;
            (loc = step61) & ((AuMoSt_aux | FoMoSt_aux | SoftLDSt_aux) & E_MMMoR & !(AuIhMMo
                )) : step61;
            (loc = step61) : step63;
            (loc = step62) : step63;
            (loc = step63) & ((MMoSt_aux & (E_MAuMoR | E_AuAuMoR )) | (FoMoSt_aux & E_MAuMoR
                ) | (SoftLDSt_aux & E_MAuMoR) | (MMoSt_aux & AuIhMMo) | (FoMoSt_aux &
                AuIhFoMo) | (SoftLDSt_aux & AuIhFoMo) | !(AuMoSt_aux | MMoSt_aux | FoMoSt_
                aux | SoftLDSt_aux)) : step64;
            (loc = step63) : step65;
            (loc = step64) : step65;
            (loc = step65) & ((AuMoSt_aux | MMoSt_aux) & E_MSoftLDR & !AuIhFoMo) : step66;
            (loc = step65) : step67;
            (loc = step66) : step67;
            (loc = step67) : step69;
            (loc = step68) : step69;
```

```
        (loc = step69) : step70;
        (loc = step70) & (AuOffR) : step71;
        (loc = step70) & (AuOnR) : step72;
        (loc = step70) & (fullNotAcknowledged | FuStopISt | !EnRstartSt) : step73;
        (loc = step70) : step74;
        (loc = step71) : step74;
        (loc = step72) : step74;
        (loc = step73) : step74;
        (loc = step74) : step75;
        (loc = step75) & (((E_MOffR & (MMoSt | FoMoSt | SoftLDSt)) | (AuOffRSt & AuMoSt)
            | (LDSt & PHLDCmd & HOffRSt) | (FE_PulseOn & PPulse & !POutOff) &
            EnRstartSt) | (E_FuStopI & !PFsPosOn)) : step76;
        (loc = step75) & (((E_MOnR  & (MMoSt | FoMoSt | SoftLDSt)) | (AuOnRSt  & AuMoSt)
            | (LDSt & PHLDCmd & HOnRSt) & EnRstartSt) | (E_FuStopI & PFsPosOn)) : step
            77;
        (loc = step75) : step78;
        (loc = step76) : step78;
        (loc = step77) : step78;
        (loc = step78) : step79;
        (loc = step79) & (HOffR) : step80;
        (loc = step79) & (HOnR) : step81;
        (loc = step79) : step82;
        (loc = step80) : step82;
        (loc = step81) : step82;
        (loc = step82) : step83;
        (loc = step83) & (PPulse) : step84;
        (loc = step83) : step158;
        (loc = step84) & (InterlockR) : step85;
        (loc = step84) & (FE_InterlockR) : step86;
        (loc = step84) & ((MOffRSt & (MMoSt | FoMoSt | SoftLDSt)) | (AuOffRSt & AuMoSt)
            | (HOffR & LDSt &  PHLDCmd)) : step105;
        (loc = step84) & ((MOnRSt & (MMoSt | FoMoSt | SoftLDSt)) | (AuOnRSt & AuMoSt) |
            (HOnR & LDSt & PHLDCmd)) : step106;
        (loc = step84) : step107;
        (loc = step85) : step108;
        (loc = step86) : step87;
        (loc = step87) & ((FALSE & !Timer_PulseOn.old_in) & !Timer_PulseOn.Q) : step88;
        (loc = step87) : step89;
        (loc = step88) : step89;
        (loc = step89) & (__GLOBAL_TIME <= Timer_PulseOn.due) : step90;
        (loc = step89) : step91;
        (loc = step90) : step95;
        (loc = step91) : step92;
        (loc = step92) & (FALSE) : step93;
        (loc = step92) : step94;
        (loc = step93) : step95;
        (loc = step94) : step95;
        (loc = step95) : step96;
        (loc = step96) & ((FALSE & !Timer_PulseOff.old_in) & !Timer_PulseOff.Q) : step
            97;
        (loc = step96) : step98;
        (loc = step97) : step98;
        (loc = step98) & (__GLOBAL_TIME <= Timer_PulseOff.due) : step99;
        (loc = step98) : step100;
        (loc = step99) : step104;
        (loc = step100) : step101;
        (loc = step101) & (FALSE) : step102;
        (loc = step101) : step103;
        (loc = step102) : step104;
        (loc = step103) : step104;
        (loc = step104) : step105;
        (loc = step105) : step108;
        (loc = step106) : step108;
        (loc = step107) : step108;
        (loc = step108) & ((PulseOnR & !Timer_PulseOn.old_in) & !Timer_PulseOn.Q): step
            109;
        (loc = step108) : step110;
        (loc = step109) : step110;
        (loc = step110) & (__GLOBAL_TIME <= Timer_PulseOn.due) : step111;
        (loc = step110) : step112;
        (loc = step111) : step116;
        (loc = step112) : step113;
        (loc = step113) & (PulseOnR) : step114;
        (loc = step113) : step115;
        (loc = step114) : step116;
        (loc = step115) : step116;
        (loc = step116) : step117;
        (loc = step117) & ((PulseOffR & !Timer_PulseOff.old_in) & !Timer_PulseOff.Q):
            step118;
        (loc = step117) : step119;
        (loc = step118) : step119;
        (loc = step119) & (__GLOBAL_TIME <= Timer_PulseOff.due) : step120;
        (loc = step119) : step121;
        (loc = step120) : step125;
        (loc = step121) : step122;
        (loc = step122) & (PulseOffR) : step123;
        (loc = step122) : step124;
        (loc = step123) : step125;
        (loc = step124) : step125;
        (loc = step125) : step126;
```

```
            (loc = step126) & (PulseOn & !RE_PulseOn_old) : step127;
            (loc = step126) : step128;
            (loc = step127) : step129;
            (loc = step128) : step129;
            (loc = step129) & (!PulseOn & FE_PulseOn_old) : step130;
            (loc = step129) : step131;
            (loc = step130) : step132;
            (loc = step131) : step132;
            (loc = step132) & (PulseOff & !RE_PulseOff_old) : step133;
            (loc = step132) : step134;
            (loc = step133) : step135;
            (loc = step134) : step135;
            (loc = step135) & (RE_PulseOn) : step136;
            (loc = step135) : step145;
            (loc = step136) & ((FALSE & !Timer_PulseOff.old_in) & !Timer_PulseOff.Q): step
                137;
            (loc = step136) : step138;
            (loc = step137) : step138;
            (loc = step138) & (__GLOBAL_TIME <= Timer_PulseOff.due) : step139;
            (loc = step138) : step140;
            (loc = step139) : step144;
            (loc = step140) : step141;
            (loc = step141) & (FALSE) : step142;
            (loc = step141) : step143;
            (loc = step142) : step144;
            (loc = step143) : step144;
            (loc = step144) : step145;
            (loc = step145) & (RE_PulseOff) : step146;
            (loc = step145) : step155;
            (loc = step146) & ((FALSE & !Timer_PulseOn.old_in) & !Timer_PulseOn.Q): step147;
            (loc = step146) : step148;
            (loc = step147) : step149;
            (loc = step148) & (__GLOBAL_TIME <= Timer_PulseOn.due) : step149;
            (loc = step148) : step150;
            (loc = step149) : step154;
            (loc = step150) : step151;
            (loc = step151) & (FALSE) : step152;
            (loc = step151) : step153;
            (loc = step152) : step154;
            (loc = step153) : step154;
            (loc = step154) : step155;
            (loc = step155) & (PPulseCste) : step156;
            (loc = step155) : step157;
            (loc = step156) : step158;
            (loc = step157) : step158;
            (loc = step158) : step159;
            (loc = step159) & (POutOff) : step160;
            (loc = step159) : step161;
            (loc = step160) : step161;
            (loc = step161) & (POutOff) : step162;
            (loc = step161) : step168;
            (loc = step162) & (InterlockR) : step163;
            (loc = step162) : step170;
            (loc = step163) & (PPulse & !PFsPosOn2) : step164;
            (loc = step163) : step167;
            (loc = step164) & (PFsPosOn) : step165;
            (loc = step164) : step166;
            (loc = step165) : step170;
            (loc = step166) : step170;
            (loc = step167) : step170;
            (loc = step168) & (InterlockR) : step169;
            (loc = step168) : step170;
            (loc = step169) : step170;
            (loc = step170) : step171;
            (loc = step171) & (!POutOff) : step172;
            (loc = step171) : step175;
            (loc = step172) & (PFsPosOn) : step173;
            (loc = step172) : step174;
            (loc = step173) : step176;
            (loc = step174) : step176;
            (loc = step175) : step176;
            (loc = step176) & (OutOnOVSt | (PPulse & PulseOnR)) : step177;
            (loc = step176) : step178;
            (loc = step177) : step178;
            (loc = step178) & ((OutOffOVSt & POutOff) | (!OutOnOVSt & !POutOff) | (PPulse &
                PulseOffR)) : step179;
            (loc = step178) : step180;
            (loc = step179) : step180;
            (loc = step180) & (OutOVSt_aux & !RE_OutOVSt_aux_old): step181;
            (loc = step180) : step182;
            (loc = step181) : step183;
            (loc = step182) : step183;
            (loc = step183) & (!OutOVSt_aux & FE_OutOVSt_aux_old) : step184;
            (loc = step183) : step185;
            (loc = step184) : step186;
            (loc = step185) : step186;
            (loc = step186) & (((OutOVSt_aux & ((PHFOn & !OnSt) | (PHFOff & OffSt))) | (!
                OutOVSt_aux & ((PHFOff & !OffSt) | (PHFOn & OnSt))) | (OffSt & OnSt)) & (!
                PPulse | (POutOff & PPulse & !OutOnOV & !OutOffOV))) : step187;
            (loc = step186) : step188;
            (loc = step187) : step188;
            (loc = step188) & (!((OutOVSt_aux & ((PHFOn & !OnSt) | (PHFOff & OffSt))) | (!OutOVSt_aux & ((PHFOff & !OffSt) | (PHFOn & OnSt))) | (OffSt & OnSt)) | RE_OutOVSt_aux | FE_OutOVSt_aux | (PPulse & POutOff & OutOnOV) | (PPulse & POutOff & OutOffOV)) : step189;
            (loc = step188) : step190;
            (loc = step189) : step190;
            (loc = step190) & (!PosW_aux) : step191;
            (loc = step190) : step192;
            (loc = step191) : step198;
            (loc = step192) & (!Timer_Warning.running) : step193;
            (loc = step192) : step194;
            (loc = step193) : step198;
            (loc = step194) & (!(__GLOBAL_TIME - (Timer_Warning.Tstart + POnOffb.PWDtb) >= 0sd32_0)) : step195;
            (loc = step194) : step197;
            (loc = step195) & (!Timer_Warning.Q) : step196;
            (loc = step195) : step198;
            (loc = step196) : step198;
            (loc = step197) : step198;
            (loc = step198) : step199;
            (loc = step199) & (FuStopISt | FSIinc > 0sd16_0) : step200;
            (loc = step199) : step201;
            (loc = step200) : step201;
            (loc = step201) & (extend(FSIinc,16) > PulseWidth | (!FuStopISt & FSIinc = 0sd16_0)) : step202;
            (loc = step201) : step203;
            (loc = step202) : step203;
            (loc = step203) & (TStopISt | TSIinc > 0sd16_0) : step204;
            (loc = step203) : step205;
            (loc = step204) : step205;
            (loc = step205) & (extend(TSIinc,16) > PulseWidth | (!TStopISt & TSIinc = 0sd16_0)) : step206;
            (loc = step205) : step207;
            (loc = step206) : step207;
            (loc = step207) & (StartISt | SIinc > 0sd16_0) : step208;
            (loc = step207) : step209;
            (loc = step208) : step209;
            (loc = step209) & (extend(SIinc,16) > PulseWidth | (!StartISt & SIinc = 0sd16_0)) : step210;
            (loc = step209) : step211;
            (loc = step210) : step211;
            (loc = step211) & (AlSt | Alinc > 0sd16_0) : step212;
            (loc = step211) : step213;
            (loc = step212) : step213;
            (loc = step213) & (extend(Alinc,16) > PulseWidth | (!AlSt & Alinc = 0sd16_0)) : step214;
            (loc = step213) : step215;
            (loc = step214) : step215;
            (loc = step215) : step216;
            (loc = step216) & (AlUnAck != AlUnAck_old) : step217;
            (loc = step216) : step221;
            (loc = step217) & (AlUnAck) : step218;
            (loc = step217) : step219;
            (loc = step218) : step220;
            (loc = step219) : step220;
            (loc = step220) : end;
            (loc = step221) : end;
            (loc = end) : nvar;
            (loc = nvar) : start;
        esac;

    --Variable initialization
    init(Stsreg01b[0]) := FALSE;
    init(Stsreg01b[1]) := FALSE;
    init(Stsreg01b[2]) := FALSE;
    init(Stsreg01b[3]) := FALSE;
    init(Stsreg01b[4]) := FALSE;
    init(Stsreg01b[5]) := FALSE;
    init(Stsreg01b[6]) := FALSE;
    init(Stsreg01b[7]) := FALSE;
    init(Stsreg01b[8]) := FALSE;
    init(Stsreg01b[9]) := FALSE;
    init(Stsreg01b[10]) := FALSE;
    init(Stsreg01b[11]) := FALSE;
    init(Stsreg01b[12]) := FALSE;
    init(Stsreg01b[13]) := FALSE;
    init(Stsreg01b[14]) := FALSE;
    init(Stsreg01b[15]) := FALSE;
    init(Stsreg02b[0]) := FALSE;
    init(Stsreg02b[1]) := FALSE;
    init(Stsreg02b[2]) := FALSE;
    init(Stsreg02b[3]) := FALSE;
    init(Stsreg02b[4]) := FALSE;
    init(Stsreg02b[5]) := FALSE;
    init(Stsreg02b[6]) := FALSE;
    init(Stsreg02b[7]) := FALSE;
    init(Stsreg02b[8]) := FALSE;
    init(Stsreg02b[9]) := FALSE;
    init(Stsreg02b[10]) := FALSE;
    init(Stsreg02b[11]) := FALSE;
    init(Stsreg02b[12]) := FALSE;
    init(Stsreg02b[13]) := FALSE;
    init(Stsreg02b[14]) := FALSE;
    init(Stsreg02b[15]) := FALSE;

    init(OutOnOV) := FALSE;
    init(OutOffOV) := FALSE;
    init(OnSt) := FALSE;
    init(OffSt) := FALSE;
    init(AuMoSt) := FALSE;
    init(MMoSt) := FALSE;
    init(LDSt) := FALSE;
    init(SoftLDSt) := FALSE;
    init(FoMoSt) := FALSE;
    init(AuOnRSt) := FALSE;
    init(AuOffRSt) := FALSE;
    init(MOnRSt) := FALSE;
    init(MOffRSt) := FALSE;
    init(HOnRSt) := FALSE;
    init(HOffRSt) := FALSE;
    init(IOErrorW) := FALSE;
    init(IOSimuW) := FALSE;
    init(AuMRW) := FALSE;
    init(AlUnAck) := FALSE;
    init(PosW) := FALSE;
    init(StartISt) := FALSE;
    init(TStopISt) := FALSE;
    init(FuStopISt) := FALSE;
    init(AlSt) := FALSE;
    init(AlBW) := FALSE;
    init(EnRstartSt) := TRUE;
    init(RdyStartSt) := FALSE;

    --Internal Variables
    init(E_MAuMoR) := FALSE;
    init(E_MMMoR) := FALSE;
    init(E_MFoMoR) := FALSE;
    init(E_MOnR) := FALSE;
    init(E_MOffR) := FALSE;
    init(E_MAlAckR) := FALSE;
    init(E_StartI) := FALSE;
    init(E_TStopI) := FALSE;
    init(E_FuStopI) := FALSE;
    init(E_Al) := FALSE;
    init(E_AuAuMoR) := FALSE;
    init(E_AuAlAck) := FALSE;
    init(E_MSoftLDR) := FALSE;
    init(E_MEnRstartR) := FALSE;
    init(RE_AlUnAck) := FALSE;
    init(FE_AlUnAck) := FALSE;
    init(RE_PulseOn) := FALSE;
    init(FE_PulseOn) := FALSE;
    init(RE_PulseOff) := FALSE;
    init(RE_OutOVSt_aux) := FALSE;
    init(FE_OutOVSt_aux) := FALSE;
    init(FE_InterlockR) := FALSE;

    init(MAuMoR_old) := FALSE;
    init(MMMoR_old) := FALSE;
    init(MFoMoR_old) := FALSE;
    init(MOnR_old) := FALSE;
    init(MOffR_old) := FALSE;
    init(MAlAckR_old) := FALSE;
    init(AuAuMoR_old) := FALSE;
    init(AuAlAck_old) := FALSE;
    init(StartI_old) := FALSE;
    init(TStopI_old) := FALSE;
    init(FuStopI_old) := FALSE;
    init(Al_old) := FALSE;
    init(AlUnAck_old) := FALSE;
    init(MSoftLDR_old) := FALSE;
    init(MEnRstartR_old) := FALSE;
    init(RE_PulseOn_old) := FALSE;
    init(FE_PulseOn_old) := FALSE;
    init(RE_PulseOff_old) := FALSE;
    init(RE_OutOVSt_aux_old) := FALSE;
    init(FE_OutOVSt_aux_old) := FALSE;
    init(FE_InterlockR_old) := FALSE;

    init(PFsPosOn) := FALSE;
    init(PFsPosOn2) := FALSE;
    init(PHFOn) := FALSE;
    init(PHFOff) := FALSE;
    init(PPulse) := FALSE;
    init(PPulseCste) := FALSE;
    init(PHLD) := FALSE;
    init(PHLDCmd) := FALSE;
    init(PAnim) := FALSE;
    init(POutOff) := FALSE;
    init(PEnRstart) := FALSE;
    init(PRstartFS) := FALSE;
    init(OutOnOVSt) := FALSE;
    init(OutOffOVSt) := FALSE;
    init(AuMoSt_aux) := FALSE;
    init(MMoSt_aux) := FALSE;
    init(FoMoSt_aux) := FALSE;
    init(SoftLDSt_aux) := FALSE;
    init(PulseOn) := FALSE;
    init(PulseOff) := FALSE;
    init(PosW_aux) := FALSE;
    init(OutOVSt_aux) := FALSE;
    init(fullNotAcknowledged) := FALSE;
    init(PulseOnR) := FALSE;
    init(PulseOffR) := FALSE;
    init(InterlockR) := FALSE;

    --Variables for IEC Timers
    init(Time_Warning) := 0sd32_0;
    init(Timer_PulseOn.Q) := FALSE;
    init(Timer_PulseOn.ET) := 0sd32_0;
    init(Timer_PulseOn.old_in) := FALSE;
    init(Timer_PulseOn.due) := 0sd32_0;
    init(Timer_PulseOff.Q) := FALSE;
    init(Timer_PulseOff.ET) := 0sd32_0;
    init(Timer_PulseOff.old_in) := FALSE;
    init(Timer_PulseOff.due) := 0sd32_0;
    init(Timer_Warning.Q) := FALSE;
    init(Timer_Warning.ET) := 0sd32_0;
    init(Timer_Warning.running) := FALSE;
    init(Timer_Warning.Tstart) := 0sd32_0;

    --Variables for interlock Ststus delay handling
    init(PulseWidth) := 0sd32_0;
    init(FSIinc) := 0sd16_0;
    init(TSIinc) := 0sd16_0;
    init(SIinc) := 0sd16_0;
    init(Alinc) := 0sd16_0;
    init(WTStopISt) := FALSE;
    init(WStartISt) := FALSE;
    init(WAlSt) := FALSE;
    init(WFuStopISt) := FALSE;
    init(__GLOBAL_TIME) := 0sd32_0;
    init(T_CYCLE) := 0ud16_0;
    init(random_t_cycle) := 0ud8_0;

    --Non-deterministic input
    next(HFOn) :=
        case
            (loc = start)    : {TRUE,FALSE};
            TRUE             : HFOn ;
        esac;
    next(HFOff) :=
        case
            (loc = start)   : {TRUE,FALSE};
            TRUE    : HFOff ;
        esac;
    next(HLD) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : HLD;
        esac;
    next(IOError) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : IOError;
        esac;
    next(IOSimu) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : IOSimu;
        esac;
    next(AlB) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : AlB;
        esac;
    next(Manreg01b[0]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[0];
        esac;
    next(Manreg01b[1]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[1];
        esac;
    next(Manreg01b[2]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[2];
        esac;
    next(Manreg01b[3]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[3];
        esac;
    next(Manreg01b[4]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[4];
        esac;
    next(Manreg01b[5]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[5];
        esac;
    next(Manreg01b[6]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[6];
        esac;
    next(Manreg01b[7]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[7];
        esac;
    next(Manreg01b[8]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[8];
        esac;
    next(Manreg01b[9]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[0];
        esac;
    next(Manreg01b[10]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[10];
        esac;
    next(Manreg01b[11]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[11];
        esac;
    next(Manreg01b[12]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[12];
        esac;
    next(Manreg01b[13]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[13];
        esac;
    next(Manreg01b[14]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[14];
        esac;
    next(Manreg01b[15]) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : Manreg01b[15];
        esac;
    next(HOnR) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : HOnR;
        esac;
    next(HOffR) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : HOffR;
        esac;
    next(StartI) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : StartI;
        esac;
    next(TStopI) :=
        case
            (loc = start) : {TRUE,FALSE};
            TRUE : TStopI;
        esac;
    next(FuStopI) :=
        case
            (loc = start) : {TRUE,FALSE};
```

```
                TRUE : FuStopI;
            esac;
        next(Al) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : Al;
            esac;
        next(AuOnR) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : AuOnR;
            esac;
        next(AuOffR) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : AuOffR;
            esac;
        next(AuAuMoR) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : AuAuMoR;
            esac;
        next(AuIhMMo) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : AuIhMMo;
            esac;
        next(AuIhFoMo) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : AuIhFoMo;
            esac;
        next(AuAlAck) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : AuAlAck;
            esac;
        next(IhAuMRW) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : IhAuMRW;
            esac;
        next(AuRstart) :=
            case
                (loc = start) : {TRUE,FALSE};
                TRUE : AuRstart;
            esac;

        --Values that are non-deterministic, but do not got a new value every iteration
        next(POnOffb.ParRegb[0]) :=
            case
                TRUE : POnOffb.ParRegb[0];
            esac;
        next(POnOffb.ParRegb[1]) :=
            case
                TRUE : POnOffb.ParRegb[1];
            esac;
        next(POnOffb.ParRegb[2]) :=
            case
                TRUE : POnOffb.ParRegb[2];
            esac;
        next(POnOffb.ParRegb[3]) :=
            case
                TRUE : POnOffb.ParRegb[3];
            esac;
        next(POnOffb.ParRegb[4]) :=
            case
                TRUE : POnOffb.ParRegb[4];
            esac;
        next(POnOffb.ParRegb[5]) :=
            case
                TRUE : POnOffb.ParRegb[5];
            esac;
        next(POnOffb.ParRegb[6]) :=
            case
                TRUE : POnOffb.ParRegb[6];
            esac;
        next(POnOffb.ParRegb[7]) :=
            case
                TRUE : POnOffb.ParRegb[7];
            esac;
        next(POnOffb.ParRegb[8]) :=
            case
                TRUE : POnOffb.ParRegb[8];
            esac;
        next(POnOffb.ParRegb[9]) :=
            case
                TRUE : POnOffb.ParRegb[9];
            esac;
        next(POnOffb.ParRegb[10]) :=
            case
                TRUE : POnOffb.ParRegb[10];
            esac;
        next(POnOffb.ParRegb[11]) :=
            case
                TRUE : POnOffb.ParRegb[11];
            esac;
        next(POnOffb.ParRegb[12]) :=
            case
                TRUE : POnOffb.ParRegb[12];
            esac;
        next(POnOffb.ParRegb[13]) :=
            case
                TRUE : POnOffb.ParRegb[13];
            esac;
        next(POnOffb.ParRegb[14]) :=
            case
                TRUE : POnOffb.ParRegb[14];
            esac;
        next(POnOffb.ParRegb[15]) :=
            case
                TRUE : POnOffb.ParRegb[15];
            esac;
        next(POnOffb.PPulseLeb) :=
            case
                TRUE : POnOffb.PPulseLeb;
            esac;
        next(POnOffb.PWDtb) :=
            case
                TRUE : POnOffb.PWDtb;
            esac;

        --Program body
        next(E_MAuMoR) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : FALSE;
                TRUE : E_MAuMoR;
            esac;
        next(MAuMoR_old) :=
            case
                (loc = step2) : TRUE;
                (loc = step3) : Manreg01b[8];
                TRUE : MAuMoR_old;
            esac;
        next(E_MMMoR) :=
            case
                (loc = step5) : TRUE;
                (loc = step6) : FALSE;
                TRUE : E_MMMoR;
            esac;
        next(MMMoR_old) :=
            case
                (loc = step5) : TRUE;
                (loc = step6) : Manreg01b[9];
                TRUE : MMMoR_old;
            esac;
        next(E_MFoMoR) :=
            case
                (loc = step8) : TRUE;
                (loc = step9) : FALSE;
                TRUE : E_MFoMoR;
            esac;
        next(MFoMoR_old) :=
            case
                (loc = step8) : TRUE;
                (loc = step9) : Manreg01b[10];
                TRUE : MFoMoR_old;
            esac;
        next(E_MSoftLDR) :=
            case
                (loc = step11) : TRUE;
                (loc = step12) : FALSE;
                TRUE : E_MSoftLDR;
            esac;
        next(MSoftLDR_old) :=
            case
                (loc = step11) : TRUE;
                (loc = step12) : Manreg01b[11];
                TRUE : MSoftLDR_old;
            esac;
        next(E_MOnR) :=
            case
                (loc = step14) : TRUE;
                (loc = step15) : FALSE;
                TRUE : E_MOnR;
            esac;
        next(MOnR_old) :=
            case
                (loc = step14) : TRUE;
                (loc = step15) : Manreg01b[12];
                TRUE : MOnR_old;
            esac;
        next(E_MOffR) :=
            case
                (loc = step17) : TRUE;
                (loc = step18) : FALSE;
                TRUE : E_MOffR;
            esac;
        next(MOffR_old) :=
            case
                (loc = step17) : TRUE;
                (loc = step18) : Manreg01b[13];
                TRUE : MOffR_old;
            esac;
        next(E_MEnRstartR) :=
            case
                (loc = step20) : TRUE;
                (loc = step21) : FALSE;
                TRUE : E_MEnRstartR;
            esac;
        next(MEnRstartR_old) :=
            case
                (loc = step20) : TRUE;
                (loc = step21) : Manreg01b[1];
                TRUE : MEnRstartR_old;
            esac;
        next(E_MAlAckR) :=
            case
                (loc = step23) : TRUE;
                (loc = step24) : FALSE;
                TRUE : E_MAlAckR;
            esac;
        next(MAlAckR_old) :=
            case
                (loc = step23) : TRUE;
                (loc = step24) : Manreg01b[7];
                TRUE : MAlAckR_old;
            esac;
        next(PFsPosOn) :=
            case
                (loc = step25) : POnOffb.ParRegb[8];
                TRUE : PFsPosOn;
            esac;
        next(PHFOn) :=
            case
                (loc = step25) : POnOffb.ParRegb[9];
                TRUE : PHFOn;
            esac;
        next(PHFOff) :=
            case
                (loc = step25) : POnOffb.ParRegb[10];
                TRUE : PHFOff;
            esac;
        next(PPulse) :=
            case
                (loc = step25) : FALSE;
                TRUE : PPulse;
            esac;
        next(PHLD) :=
            case
                (loc = step25) : POnOffb.ParRegb[12];
                TRUE : PHLD;
            esac;
        next(PHLDCmd) :=
            case
                (loc = step25) : POnOffb.ParRegb[13];
                TRUE : PHLDCmd;
            esac;
        next(PAnim) :=
            case
                (loc = step25) : POnOffb.ParRegb[14];
                TRUE : PAnim;
            esac;
        next(POutOff) :=
            case
                (loc = step25) : POnOffb.ParRegb[15];
                TRUE : POutOff;
            esac;
        next(PEnRstart) :=
            case
                (loc = step25) : POnOffb.ParRegb[0];
                TRUE : PEnRstart;
            esac;
        next(PRstartFS) :=
            case
                (loc = step25) : POnOffb.ParRegb[1];
                TRUE : PRstartFS;
            esac;
        next(PFsPosOn2) :=
            case
                (loc = step25) : POnOffb.ParRegb[2];
                TRUE : PFsPosOn2;
            esac;
        next(PPulseCste) :=
            case
                (loc = step25) : POnOffb.ParRegb[3];
                TRUE : PPulseCste;
            esac;
        next(E_AuAuMoR) :=
            case
                (loc = step27) : TRUE;
                (loc = step28) : FALSE;
                TRUE : E_AuAuMoR;
            esac;
        next(AuAuMoR_old) :=
            case
                (loc = step27) : TRUE;
                (loc = step28) : AuAuMoR;
                TRUE : AuAuMoR_old;
            esac;
        next(E_AuAlAck) :=
            case
                (loc = step30) : TRUE;
                (loc = step31) : FALSE;
                TRUE : E_AuAlAck;
            esac;
        next(AuAlAck_old) :=
            case
                (loc = step30) : TRUE;
                (loc = step31) : AuAlAck;
                TRUE : AuAlAck_old;
            esac;
        next(E_StartI) :=
            case
                (loc = step33) : TRUE;
                (loc = step34) : FALSE;
                TRUE : E_StartI;
            esac;
        next(StartI_old) :=
            case
                (loc = step33) : TRUE;
                (loc = step34) : StartI;
                TRUE : StartI_old;
            esac;
        next(E_TStopI) :=
            case
                (loc = step36) : TRUE;
                (loc = step37) : FALSE;
                TRUE : E_TStopI;
            esac;
        next(TStopI_old) :=
            case
                (loc = step36) : TRUE;
                (loc = step37) : TStopI;
                TRUE : TStopI_old;
            esac;
        next(E_FuStopI) :=
            case
                (loc = step39) : TRUE;
                (loc = step40) : FALSE;
                TRUE : E_FuStopI;
            esac;
        next(FuStopI_old) :=
            case
                (loc = step39) : TRUE;
                (loc = step40) : FuStopI;
                TRUE : FuStopI_old;
            esac;
        next(E_Al) :=
            case
                (loc = step42) : TRUE;
                (loc = step43) : FALSE;
                TRUE : E_Al;
            esac;
        next(Al_old) :=
            case
                (loc = step42) : TRUE;
                (loc = step43) : Al;
                TRUE : Al_old;
            esac;
        next(StartISt) :=
            case
                (loc = step44) : StartI;
                TRUE : StartISt;
            esac;
        next(TStopISt) :=
            case
                (loc = step44) : TStopI;
                TRUE : TStopISt;
            esac;
        next(FuStopISt) :=
            case
                (loc = step44) : FuStopI;
                TRUE : FuStopISt;
            esac;
        next(fullNotAcknowledged) :=
            case
                (loc = step46) : FALSE;
                (loc = step51) : TRUE;
                TRUE : fullNotAcknowledged;
            esac;
        next(AlUnAck) :=
            case
                (loc = step46) : FALSE;
                (loc = step47) : TRUE;
                TRUE : AlUnAck;
            esac;
        next(EnRstartSt) :=
            case
                (loc = step49) : TRUE;
                (loc = step53) : FALSE;
                TRUE : EnRstartSt;
            esac;
        next(InterlockR) :=
            case
                (loc = step54) : (TStopISt | FuStopISt | fullNotAcknowledged | !EnRstartSt | (
                    StartISt & !POutOff & !OutOnOV) | (StartISt & POutOff & ((PFsPosOn & OutOVSt
                    _aux) | (!PFsPosOn & !OutOVSt_aux))));
                TRUE : InterlockR;
            esac;
        next(FE_InterlockR) :=
            case
                (loc = step56) : TRUE;
                (loc = step57) : FALSE;
                TRUE : FE_InterlockR;
            esac;
        next(FE_InterlockR_old) :=
            case
                (loc = step56) : FALSE;
                (loc = step57) : InterlockR;
                TRUE : FE_InterlockR_old;
            esac;
        next(AuMoSt_aux) :=
            case
                (loc = step60) : FALSE;
                (loc = step62) : FALSE;
                (loc = step64) : TRUE;
                (loc = step66) : FALSE;
                TRUE : AuMoSt_aux;
            esac;
        next(MMoSt_aux) :=
            case
                (loc = step60) : FALSE;
                (loc = step62) : TRUE;
                (loc = step64) : FALSE;
                (loc = step66) : FALSE;
                TRUE : MMoSt_aux;
            esac;
        next(FoMoSt_aux) :=
            case
                (loc = step60) : TRUE;
                (loc = step62) : FALSE;
                (loc = step64) : FALSE;
                (loc = step66) : FALSE;
                TRUE : FoMoSt_aux;
            esac;
        next(SoftLDSt_aux) :=
            case
                (loc = step60) : FALSE;
                (loc = step62) : FALSE;
                (loc = step64) : FALSE;
                (loc = step66) : TRUE;
                TRUE : SoftLDSt_aux;
            esac;
        next(LDSt) :=
            case
                (loc = step67) : FALSE;
                (loc = step68) : TRUE;
                TRUE : LDSt;
            esac;
        next(AuMoSt) :=
            case
                (loc = step67) : AuMoSt_aux;
                (loc = step68) : FALSE;
                TRUE : AuMoSt;
            esac;
        next(MMoSt) :=
            case
                (loc = step67) : MMoSt_aux;
```

```
                (loc = step68) : FALSE;
                TRUE : MMoSt;
            esac;
        next(FoMoSt) :=
            case
                (loc = step67) : FoMoSt_aux;
                (loc = step68) : FALSE;
                TRUE : FoMoSt;
            esac;
        next(SoftLDSt) :=
            case
                (loc = step67) : SoftLDSt_aux;
                (loc = step68) : FALSE;
                TRUE : SoftLDSt;
            esac;
        next(OnSt) :=
            case
                (loc = step69) : (HFOn & PHFOn) | (!PHFOn & PHFOff & PAnim & !HFOff) | (!PHFOn &
                     !PHFOff & OutOVSt_aux);
                TRUE : OnSt;
            esac;
        next(OffSt) :=
            case
                (loc = step69) : (HFOff & PHFOff) | (!PHFOff & PHFOn & PAnim & !HFOn) | (!PHFOn
                    & !PHFOff & !OutOVSt_aux);
                TRUE : OffSt;
            esac;
        next(AuOnRSt) :=
            case
                (loc = step71) : FALSE;
                (loc = step72) : TRUE;
                (loc = step73) : PFsPosOn;
                TRUE : AuOnRSt;
            esac;
        next(AuOffRSt) :=
            case
                (loc = step74) : !AuOnRSt;
                TRUE : AuOffRSt;
            esac;
        next(MOnRSt) :=
            case
                (loc = step76) : FALSE;
                (loc = step77) : TRUE;
                TRUE : MOnRSt;
            esac;
        next(HOnRSt) :=
            case
                (loc = step80) : FALSE;
                (loc = step81) : TRUE;
                TRUE : HOnRSt;
            esac;
        next(HOffRSt) :=
            case
                (loc = step82) : !HOnRSt;
                TRUE : HOffRSt;
            esac;
        next(PulseOnR) :=
            case
                (loc = step85) : (PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
                (loc = step86) : FALSE;
                (loc = step105) : FALSE;
                (loc = step106) : TRUE;
                (loc = step107) : FALSE;
                TRUE : PulseOnR;
            esac;
        next(PulseOffR) :=
            case
                (loc = step85) : (!PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
                (loc = step86) : FALSE;
                (loc = step105) : TRUE;
                (loc = step106) : FALSE;
                (loc = step107) : FALSE;
                TRUE : PulseOffR;
            esac;
        next(Timer_PulseOn.due) :=
            case
                (loc = step88) : __GLOBAL_TIME + 0sd32_0;
                (loc = step109) : __GLOBAL_TIME + POnOffb.PPulseLeb;
                (loc = step147) : __GLOBAL_TIME + 0sd32_0;
                TRUE : Timer_PulseOn.due;
            esac;
        next(Timer_PulseOn.Q) :=
            case
                (loc = step90) : TRUE;
                (loc = step91) : FALSE;
                (loc = step110) : TRUE;
                (loc = step111) : FALSE;
                (loc = step149) : TRUE;
                (loc = step150) : FALSE;
                TRUE : Timer_PulseOn.Q;
            esac;
        next(Timer_PulseOn.ET) :=
            case
                (loc = step90) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
                (loc = step93) : 0sd32_0;
                (loc = step94) : 0sd32_0;
                (loc = step111) : (POnOffb.PPulseLeb - (Timer_PulseOn.due - __GLOBAL_TIME));
                (loc = step114) : POnOffb.PPulseLeb;
                (loc = step115) : 0sd32_0;
                (loc = step149) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
                (loc = step152) : 0sd32_0;
                (loc = step153) : 0sd32_0;
                TRUE : Timer_PulseOn.ET;
            esac;
        next(Timer_PulseOn.old_in) :=
            case
                (loc = step95) : FALSE;
                (loc = step116) : PulseOnR;
                (loc = step154) : FALSE;
                TRUE : Timer_PulseOn.old_in;
            esac;
        next(Timer_PulseOff.due) :=
            case
                (loc = step97) : __GLOBAL_TIME + 0sd32_0;
                (loc = step118) : __GLOBAL_TIME + POnOffb.PPulseLeb;
                (loc = step137) : __GLOBAL_TIME + 0sd32_0;
                TRUE : Timer_PulseOff.due;
            esac;
        next(Timer_PulseOff.Q) :=
            case
                (loc = step99) : TRUE;
                (loc = step100) : FALSE;
                (loc = step120) : TRUE;
                (loc = step121) : FALSE;
                (loc = step139) : TRUE;
                (loc = step140) : FALSE;
                TRUE : Timer_PulseOff.Q;
            esac;
        next(Timer_PulseOff.ET) :=
            case
                (loc = step99) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
                (loc = step102) : 0sd32_0;
                (loc = step103) : 0sd32_0;
                (loc = step120) : (POnOffb.PPulseLeb - (Timer_PulseOn.due - __GLOBAL_TIME));
                (loc = step123) : POnOffb.PPulseLeb;
                (loc = step124) : 0sd32_0;
                (loc = step139) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
                (loc = step142) : 0sd32_0;
                (loc = step143) : 0sd32_0;
                TRUE : Timer_PulseOff.ET;
            esac;
        next(Timer_PulseOff.old_in) :=
            case
                (loc = step104) : FALSE;
                (loc = step125) : PulseOffR;
                (loc = step144) : FALSE;
                TRUE : Timer_PulseOff.old_in;
            esac;
        next(RE_PulseOn) :=
            case
                (loc = step127) : TRUE;
                (loc = step128) : FALSE;
                TRUE : RE_PulseOn;
            esac;
        next(RE_PulseOn_old) :=
            case
                (loc = step127) : TRUE;
                (loc = step128) : PulseOn;
                TRUE : RE_PulseOn_old;
            esac;
        next(FE_PulseOn) :=
            case
                (loc = step130) : TRUE;
                (loc = step131) : FALSE;
                TRUE : FE_PulseOn;
            esac;
        next(FE_PulseOn_old) :=
            case
                (loc = step130) : FALSE;
                (loc = step131) : PulseOn;
                TRUE : FE_PulseOn_old;
            esac;
        next(RE_PulseOff) :=
            case
                (loc = step133) : TRUE;
                (loc = step134) : FALSE;
                TRUE : RE_PulseOff;
            esac;
        next(RE_PulseOff_old) :=
            case
```

```
            (loc = step133) : TRUE;
            (loc = step134) : PulseOff;
            TRUE : RE_PulseOff_old;
        esac;
    next(PulseOn) :=
        case
            (loc = step156) : Timer_PulseOn.Q & !PulseOffR;
            (loc = step157) : Timer_PulseOn.Q & !PulseOffR & (!PHFOn | (PHFOn & !HFOn));
            TRUE : PulseOn;
        esac;
    next(PulseOff) :=
        case
            (loc = step156) : Timer_PulseOff.Q & !PulseOnR;
            (loc = step157) : Timer_PulseOff.Q & !PulseOnR & (!PHFOff | (PHFOff & !HFOff));
            TRUE : PulseOff;
        esac;
    next(OutOnOVSt) :=
        case
            (loc = step158) : (PPulse & PulseOn) | (!PPulse & ((MOnRSt & (MMoSt | FoMoSt |
                SoftLDSt)) | (AuOnRSt & AuMoSt) | (HOnRSt & LDSt & PHLDCmd)));
            (loc = step165) : PulseOn;
            (loc = step166) : FALSE;
            (loc = step167) : (PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
            (loc = step169) : PFsPosOn;
            TRUE : OutOnOVSt;
        esac;
    next(OutOffOVSt) :=
        case
            (loc = step160) : (PulseOff & PPulse) | (!(PPulse) & ((MOffRSt & (MMoSt | FoMoSt
                 | SoftLDSt)) | (AuOffRSt & AuMoSt) | (HOffRSt & LDSt & PHLDCmd)));
            (loc = step165) : FALSE;
            (loc = step166) : PulseOff;
            (loc = step167) : (!PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
            TRUE : OutOffOVSt;
        esac;
    next(RdyStartSt) :=
        case
            (loc = step170) : !InterlockR;
            TRUE : RdyStartSt;
        esac;
    next(AlSt) :=
        case
            (loc = step170) : Al;
            TRUE : AlSt;
        esac;
    next(IOErrorW) :=
        case
            (loc = step170) : IOError;
            TRUE : IOErrorW;
        esac;
    next(IOSimuW) :=
        case
            (loc = step170) : IOSimu;
            TRUE : IOSimuW;
        esac;
    next(AuMRW) :=
        case
            (loc = step170) : (MMoSt | FoMoSt | SoftLDSt) & ((AuOnRSt xor MOnRSt) | (
                AuOffRSt xor MOffRSt)) & !IhAuMRW;
            TRUE : AuMRW;
        esac;
    next(OutOnOV) :=
        case
            (loc = step173) : !OutOnOVSt;
            (loc = step174) : OutOnOVSt;
            (loc = step175) : OutOnOVSt;
            TRUE : OutOnOV;
        esac;
    next(OutOffOV) :=
        case
            (loc = step175) : OutOffOVSt;
            TRUE : OutOffOV;
        esac;
    next(OutOVSt_aux) :=
        case
            (loc = step177) : TRUE;
            (loc = step179) : FALSE;
            TRUE : OutOVSt_aux;
        esac;
    next(RE_OutOVSt_aux) :=
        case
            (loc = step181) : TRUE;
            (loc = step182) : FALSE;
            TRUE : RE_OutOVSt_aux;
        esac;
    next(RE_OutOVSt_aux_old) :=
        case
            (loc = step181) : TRUE;
            (loc = step182) : OutOVSt_aux;
            TRUE : RE_OutOVSt_aux_old;
        esac;
    next(FE_OutOVSt_aux) :=
        case
            (loc = step184) : TRUE;
            (loc = step185) : FALSE;
            TRUE : FE_OutOVSt_aux;
        esac;
    next(FE_OutOVSt_aux_old) :=
        case
            (loc = step184) : FALSE;
            (loc = step185) : OutOVSt_aux;
            TRUE : FE_OutOVSt_aux_old;
        esac;
    next(PosW_aux) :=
        case
            (loc = step187) : TRUE;
            (loc = step189) : FALSE;
            TRUE : PosW_aux;
        esac;
    next(Timer_Warning.Q) :=
        case
            (loc = step191) : FALSE;
            (loc = step197) : TRUE;
            TRUE : Timer_Warning.Q;
        esac;
    next(Timer_Warning.ET) :=
        case
            (loc = step191) : 0sd32_0;
            (loc = step193) : 0sd32_0;
            (loc = step196) : __GLOBAL_TIME - Timer_Warning.Tstart;
            (loc = step197) : POnOffb.PWDtb;
            TRUE : Timer_Warning.ET;
        esac;
    next(Timer_Warning.running) :=
        case
            (loc = step191) : FALSE;
            (loc = step193) : TRUE;
            TRUE : Timer_Warning.running;
        esac;
    next(Timer_Warning.Tstart) :=
        case
            (loc = step193) : __GLOBAL_TIME;
            TRUE : Timer_Warning.Tstart;
        esac;
    next(PosW) :=
        case
            (loc = step198) : Timer_Warning.Q;
            TRUE : PosW;
        esac;
    next(Time_Warning) :=
        case
            (loc = step198) : Timer_Warning.ET;
            TRUE : Time_Warning;
        esac;
    next(AlBW) :=
        case
            (loc = step198) : AlB;
            TRUE : AlBW;
        esac;
    next(PulseWidth) :=
        case
            (loc = step198) : (0sd32_150000 * 0sd32_100) / (signed(extend(T_CYCLE, 16)) * 0
                sd32_100 + 0sd32_1);
            TRUE : PulseWidth;
        esac;
    next(FSIinc) :=
        case
            (loc = step200) : FSIinc + 0sd16_1;
            (loc = step202) : 0sd16_0;
            TRUE : FSIinc;
        esac;
    next(WFuStopISt) :=
        case
            (loc = step200) : TRUE;
            (loc = step202) : FuStopISt;
            TRUE : WFuStopISt;
        esac;
    next(TSIinc) :=
        case
            (loc = step204) : TSIinc + 0sd16_1;
            (loc = step206) : 0sd16_0;
            TRUE : TSIinc;
        esac;
    next(WTStopISt) :=
        case
            (loc = step204) : TRUE;
            (loc = step206) : TStopISt;
            TRUE : WTStopISt;
        esac;
    next(SIinc) :=
        case
            (loc = step208) : SIinc + 0sd16_1;
            (loc = step210) : 0sd16_0;
            TRUE : SIinc;
        esac;
    next(WStartISt) :=
        case
            (loc = step208) : TRUE;
            (loc = step210) : StartISt;
            TRUE : WStartISt;
        esac;
    next(AIinc) :=
        case
            (loc = step212) : AIinc + 0sd16_1;
            (loc = step214) : 0sd16_0;
            TRUE : AIinc;
        esac;
    next(WAlSt) :=
        case
            (loc = step212) : TRUE;
            (loc = step214) : AlSt;
            TRUE : WAlSt;
        esac;
    next(Stsreg01b[8]) :=
        case
            (loc = step215) : OnSt;
            TRUE : Stsreg01b[8];
        esac;
    next(Stsreg01b[9]) :=
        case
            (loc = step215) : OffSt;
            TRUE : Stsreg01b[9];
        esac;
    next(Stsreg01b[10]) :=
        case
            (loc = step215) : AuMoSt;
            TRUE : Stsreg01b[10];
        esac;
    next(Stsreg01b[11]) :=
        case
            (loc = step215) : MMoSt;
            TRUE : Stsreg01b[11];
        esac;
    next(Stsreg01b[12]) :=
        case
            (loc = step215) : FoMoSt;
            TRUE : Stsreg01b[12];
        esac;
    next(Stsreg01b[13]) :=
        case
            (loc = step215) : LDSt;
            TRUE : Stsreg01b[13];
        esac;
    next(Stsreg01b[14]) :=
        case
            (loc = step215) : IOErrorW;
            TRUE : Stsreg01b[14];
        esac;
    next(Stsreg01b[15]) :=
        case
            (loc = step215) : IOSimuW;
            TRUE : Stsreg01b[15];
        esac;
    next(Stsreg01b[0]) :=
        case
            (loc = step215) : AuMRW;
            TRUE : Stsreg01b[0];
        esac;
    next(Stsreg01b[1]) :=
        case
            (loc = step215) : PosW;
            TRUE : Stsreg01b[1];
        esac;
    next(Stsreg01b[2]) :=
        case
            (loc = step215) : WStartISt;
            TRUE : Stsreg01b[2];
        esac;
    next(Stsreg01b[3]) :=
        case
            (loc = step215) : WTStopISt;
            TRUE : Stsreg01b[3];
        esac;
    next(Stsreg01b[4]) :=
        case
            (loc = step215) : AlUnAck;
            TRUE : Stsreg01b[4];
        esac;
    next(Stsreg01b[5]) :=
        case
```

```
                (loc = step215) : AuIhFoMo;
                TRUE : Stsreg01b[5];
            esac;
        next(Stsreg01b[6]) :=
            case
                (loc = step215) : WAlSt;
                TRUE : Stsreg01b[6];
            esac;
        next(Stsreg01b[7]) :=
            case
                (loc = step215) : AuIhMMo;
                TRUE : Stsreg01b[7];
            esac;
        next(Stsreg02b[8]) :=
            case
                (loc = step215) : OutOnOVSt;
                TRUE : Stsreg02b[8];
            esac;
        next(Stsreg02b[9]) :=
            case
                (loc = step215) : AuOnRSt;
                TRUE : Stsreg02b[9];
            esac;
        next(Stsreg02b[10]) :=
            case
                (loc = step215) : MOnRSt;
                TRUE : Stsreg02b[10];
            esac;
        next(Stsreg02b[11]) :=
            case
                (loc = step215) : AuOffRSt;
                TRUE : Stsreg02b[11];
            esac;
        next(Stsreg02b[12]) :=
            case
                (loc = step215) : MOffRSt;
                TRUE : Stsreg02b[12];
            esac;
        next(Stsreg02b[13]) :=
            case
                (loc = step215) : HOnRSt;
                TRUE : Stsreg02b[13];
            esac;
        next(Stsreg02b[14]) :=
            case
                (loc = step215) : HOffRSt;
                TRUE : Stsreg02b[14];
            esac;
        next(Stsreg02b[15]) :=
            case
                (loc = step215) : FALSE;
                TRUE : Stsreg02b[15];
            esac;
        next(Stsreg02b[0]) :=
            case
                (loc = step215) : FALSE;
                TRUE : Stsreg02b[0];
            esac;
        next(Stsreg02b[1]) :=
            case
                (loc = step215) : FALSE;
                TRUE : Stsreg02b[1];
            esac;
        next(Stsreg02b[2]) :=
            case
                (loc = step215) : WFuStopISt;
                TRUE : Stsreg02b[2];
            esac;
        next(Stsreg02b[3]) :=
            case
                (loc = step215) : EnRstartSt;
                TRUE : Stsreg02b[3];
            esac;
        next(Stsreg02b[4]) :=
            case
                (loc = step215) : SoftLDSt;
                TRUE : Stsreg02b[4];
            esac;
        next(Stsreg02b[5]) :=
            case
                (loc = step215) : AlBW;
                TRUE : Stsreg02b[5];
            esac;
        next(Stsreg02b[6]) :=
            case
                (loc = step215) : OutOffOVSt;
                TRUE : Stsreg02b[6];
            esac;
        next(Stsreg02b[7]) :=
            case
                (loc = step215) : FALSE;
                TRUE : Stsreg02b[7];
            esac;
        next(RE_AlUnAck) :=
            case
                (loc = step218) : TRUE;
                (loc = step219) : FALSE;
                (loc = step221) : FALSE;
                TRUE : RE_AlUnAck;
            esac;
        next(FE_AlUnAck) :=
            case
                (loc = step218) : FALSE;
                (loc = step219) : TRUE;
                (loc = step221) : FALSE;
                TRUE : FE_AlUnAck;
            esac;
        next(AlUnAck_old) :=
            case
                (loc = step220) : AlUnAck;
                TRUE : AlUnAck_old;
            esac;

        next(__GLOBAL_TIME) :=
            case
                (loc = end) : __GLOBAL_TIME + signed(extend(T_CYCLE, 16));
                TRUE : __GLOBAL_TIME;
            esac;

        next(T_CYCLE) :=
            case
                (loc = start) : ((extend(random_t_cycle,8)) mod 0ud16_95 + 0ud16_5);
                TRUE : T_CYCLE;
            esac;

        --extra assertion variables
        next(sFoMoSt_aux) :=
            case
                (loc = step1) : FoMoSt_aux;
                TRUE : sFoMoSt_aux;
            esac;
        next(sAuAuMoR) :=
            case
                (loc = step1) : AuAuMoR;
                TRUE : sAuAuMoR;
            esac;
        next(sManreg01b8) :=
            case
                (loc = step1) : Manreg01b[8];
                TRUE : sManreg01b8;
            esac;
        next(sAuIhFoMo) :=
            case
                (loc = step1) : AuIhFoMo;
                TRUE : sAuIhFoMo;
            esac;
        next(sAuIhMMo) :=
            case
                (loc = step1) : AuIhMMo;
                TRUE : sAuIhMMo;
            esac;
        next(sMMoSt_aux) :=
            case
                (loc = step1) : MMoSt_aux;
                TRUE : sMMoSt_aux;
            esac;
        next(pOutOnOV) :=
            case
                (loc = nvar) : OutOnOV;
                TRUE : pOutOnOV;
            esac;
        next(pTStopI) :=
            case
                (loc = nvar) : TStopI;
                TRUE : pTStopI;
            esac;
        next(pFuStopI) :=
            case
                (loc = nvar) : FuStopI;
                TRUE : pFuStopI;
            esac;
        next(pStartI) :=
            case
                (loc = nvar) : StartI;
                TRUE : pStartI;
            esac;
        next(pOutOnOVSt) :=
            case
                (loc = nvar) : OutOnOVSt;
                TRUE : pOutOnOVSt;
            esac;
        next(pMMoSt) :=
            case
                (loc = nvar) : MMoSt;
                TRUE : pMMoSt;
            esac;
        next(pManreg01b12) :=
            case
                (loc = nvar) : Manreg01b[12];
                TRUE : pManreg01b12;
            esac;
        next(pManreg01b13) :=
            case
                (loc = nvar) : Manreg01b[13];
                TRUE : pManreg01b13;
            esac;
        init(first) := TRUE;
        next(first) :=
            case
                (loc = end) : FALSE;
                TRUE : first;
            esac;

--Properties
```

Listing F.1: CPC.smv

## G CPC program in C

This appendix shows the translation from the CPC program to C. Note that the properties can be found in Appendix H.

```
#include <stdbool.h>
_Bool nondet_bool();
int nondet_int();
unsigned short nondet_unsignedshort();

struct TP{
    bool Q;
    int ET;
    bool old_in;
    int due;
};

struct TON{
    bool Q;
    int ET;
    bool running;
    int start;
};

//VAR_INPUT
bool HFOn, HFOff, HLD, IOError, IOSimu, AlB;
bool Manreg01b[16];
bool HOnR, HOffR, StartI, TStopI, FuStopI, Al, AuOnR, AuOffR, AuAuMoR, AuIhMMo, AuIhFoMo,
    AuAlAck, IhAuMRW, AuRstart;
struct CPC_ONOFF_PARAM{
    bool ParRegb[16];
    int PPulseLeb, PWDtb;
};
struct CPC_ONOFF_PARAM POnOffb;

//VAR_OUTPUT
bool Stsreg01b[16];
bool Stsreg02b[16];
bool OutOnOV = false;
bool OutOffOV = false;
bool OnSt = false;
bool OffSt = false;
bool AuMoSt = false;
bool MMoSt = false;
bool LDSt = false;
bool SoftLDSt = false;
bool FoMoSt = false;
bool AuOnRSt = false;
bool AuOffRSt = false;
bool MOnRSt = false;
bool MOffRSt = false;
bool HOnRSt = false;
bool HOffRSt = false;
bool IOErrorW = false;
bool IOSimuW = false;
bool AuMRW = false;
bool AlUnAck = false;
bool PosW = false;
bool StartISt = false;
bool TStopISt = false;
bool FuStopISt = false;
bool AlSt = false;
bool AlBW = false;
bool EnRstartSt = true;
bool RdyStartSt = false;

//Internal Variables
bool E_MAuMoR = false;
bool E_MMMoR = false;
bool E_MFoMoR = false;
bool E_MOnR = false;
bool E_MOffR = false;
bool E_MAlAckR = false;
bool E_StartI = false;
bool E_TStopI = false;
bool E_FuStopI = false;
bool E_Al = false;
bool E_AuAuMoR = false;
bool E_AuAlAck = false;
bool E_MSoftLDR = false;
bool E_MEnRstartR = false;
bool RE_AlUnAck = false;
bool FE_AlUnAck = false;
bool RE_PulseOn = false;
bool FE_PulseOn = false;
bool RE_PulseOff = false;
bool RE_OutOVSt_aux = false;
bool FE_OutOVSt_aux = false;
bool FE_InterlockR = false;

bool MAuMoR_old = false;
bool MMMoR_old = false;
bool MFoMoR_old = false;
bool MOnR_old = false;
bool MOffR_old = false;
bool MAlAckR_old = false;
bool AuAuMoR_old = false;
bool AuAlAck_old = false;
bool StartI_old = false;
bool TStopI_old = false;
bool FuStopI_old = false;
bool Al_old = false;
bool AlUnAck_old = false;
bool MSoftLDR_old = false;
bool MEnRstartR_old = false;
bool RE_PulseOn_old = false;
bool FE_PulseOn_old = false;
bool RE_PulseOff_old = false;
bool RE_OutOVSt_aux_old = false;
bool FE_OutOVSt_aux_old = false;
bool FE_InterlockR_old = false;

bool PFsPosOn = false;
bool PFsPosOn2 = false;
bool PHFOn = false;
bool PHFOff = false;
bool PPulse = false;
bool PPulseCste = false;
bool PHLD = false;
bool PHLDCmd = false;
bool PAnim = false;
bool POutOff = false;
bool PEnRstart = false;
bool PRstartFS = false;
bool OutOnOVSt = false;
bool OutOffOVSt = false;
bool AuMoSt_aux = false;
bool MMoSt_aux = false;
bool FoMoSt_aux = false;
bool SoftLDSt_aux = false;
bool PulseOn = false;
bool PulseOff = false;
bool PosW_aux = false;
bool OutOVSt_aux = false;
bool fullNotAcknowledged = false;
bool PulseOnR = false;
bool PulseOffR = false;
bool InterlockR = false;

//time Time_Warning;
int Time_Warning;
struct TP Timer_PulseOn;
struct TP Timer_PulseOff;
struct TON Timer_Warning;

double PulseWidth;
```

```
short FSIinc, TSIinc, SIinc, AIinc;
bool WTStopISt, WStartISt, WAlSt, WFuStopISt;
int __GLOBAL_TIME = 0;
unsigned short T_CYCLE;

bool R_EDGE(bool new, bool *old){
    if(new &&!*old){
        *old = true;
        return true;
    } else{
        *old = new;
        return false;
    }
}

bool F_EDGE(bool new, bool *old){
    if(!new && *old){
        *old = false;
        return true;
    } else{
        *old = new;
        return false;
    }
}

void DETECT_EDGE(bool new, bool old, bool *re, bool *fe){
    if(new!=old){
        if(new){
            *re = true;
            *fe = false;
        } else{
            *re = false;
            *fe = true;
        }
    } else{
        *re = false;
        *fe = false;
    }
}

void updateTP(bool *tpQ, int *tpET, bool *tpold_in, int *tpdue, bool in, int PT){
    if(in && ! *tpold_in && ! *tpQ){
        *tpdue = __GLOBAL_TIME + PT;
    }
    if(__GLOBAL_TIME <= *tpdue){
        *tpQ = true;
        *tpET = PT - (*tpdue - __GLOBAL_TIME);
    } else{
        *tpQ = false;
        if(in){
            *tpET = PT;
        } else{
            *tpET = 0;
        }
    }
    *tpold_in = in;
}

void updateTON(bool *tonQ, int *tonET, bool *tonrunning, int *tonstart, int PT, bool in){
    if(!in){
        *tonQ = false;
        *tonET = 0;
        *tonrunning = false;
    } else if(!*tonrunning){
        *tonstart = __GLOBAL_TIME;
        *tonrunning = true;
        *tonET = 0;
    } else if(!((__GLOBAL_TIME - (*tonstart + PT)) >= 0)){
        if(!*tonQ){
            *tonET = __GLOBAL_TIME - *tonstart;
        }
    } else{
        *tonQ = true;
        *tonET = PT;
    }
}

int main(){

Stsreg01b[0] = false;
Stsreg02b[0] = false;
Stsreg01b[1] = false;
Stsreg02b[1] = false;
Stsreg01b[2] = false;
Stsreg02b[2] = false;
Stsreg01b[3] = false;
Stsreg02b[3] = false;
Stsreg01b[4] = false;
Stsreg02b[4] = false;
Stsreg01b[5] = false;
```

```
Stsreg02b[5] = false;
Stsreg01b[6] = false;
Stsreg02b[6] = false;
Stsreg01b[7] = false;
Stsreg02b[7] = false;
Stsreg01b[8] = false;
Stsreg02b[8] = false;
Stsreg01b[9] = false;
Stsreg02b[9] = false;
Stsreg01b[10] = false;
Stsreg02b[10] = false;
Stsreg01b[11] = false;
Stsreg02b[11] = false;
Stsreg01b[12] = false;
Stsreg02b[12] = false;
Stsreg01b[13] = false;
Stsreg02b[13] = false;
Stsreg01b[14] = false;
Stsreg02b[14] = false;
Stsreg01b[15] = false;
Stsreg02b[15] = false;

Timer_PulseOn.Q = false;
Timer_PulseOn.old_in = false;
Timer_PulseOn.due = 0;
Timer_PulseOff.Q = false;
Timer_PulseOff.old_in = false;
Timer_PulseOff.due = 0;
Timer_Warning.Q = false;
Timer_Warning.ET = 0;
Timer_Warning.running = false;
Timer_Warning.start = 0;
int first = true;

POnOffb.ParRegb[0] = nondet_bool();
POnOffb.ParRegb[1] = nondet_bool();
POnOffb.ParRegb[2] = nondet_bool();
POnOffb.ParRegb[3] = nondet_bool();
POnOffb.ParRegb[4] = nondet_bool();
POnOffb.ParRegb[5] = nondet_bool();
POnOffb.ParRegb[6] = nondet_bool();
POnOffb.ParRegb[7] = nondet_bool();
POnOffb.ParRegb[8] = nondet_bool();
POnOffb.ParRegb[9] = nondet_bool();
POnOffb.ParRegb[10] = nondet_bool();
POnOffb.ParRegb[11] = nondet_bool();
POnOffb.ParRegb[12] = nondet_bool();
POnOffb.ParRegb[13] = nondet_bool();
POnOffb.ParRegb[14] = nondet_bool();
POnOffb.ParRegb[15] = nondet_bool();

POnOffb.PPulseLeb = nondet_int();
POnOffb.PWDtb = nondet_int();

    while(true){
        //extra assertion variables
        bool pOutOnOV = OutOnOV;
        bool pTStopI = TStopI;
        bool pFuStopI = FuStopI;
        bool pStartI = StartI;
        bool pOutOnOVSt = OutOnOVSt;
        bool pMMoSt = MMoSt;
        bool pManreg01b12 = Manreg01b[12];
        bool pManreg01b13 = Manreg01b[13];

        //nondet-input
        HFOn = nondet_bool();
        HFOff = nondet_bool();
        HLD = nondet_bool();
        IOError = nondet_bool();
        IOSimu = nondet_bool();
        AlB = nondet_bool();

        Manreg01b[0] = nondet_bool();
        Manreg01b[1] = nondet_bool();
        Manreg01b[2] = nondet_bool();
        Manreg01b[3] = nondet_bool();
        Manreg01b[4] = nondet_bool();
        Manreg01b[5] = nondet_bool();
        Manreg01b[6] = nondet_bool();
        Manreg01b[7] = nondet_bool();
        Manreg01b[8] = nondet_bool();
        Manreg01b[9] = nondet_bool();
        Manreg01b[10] = nondet_bool();
        Manreg01b[11] = nondet_bool();
        Manreg01b[12] = nondet_bool();
        Manreg01b[13] = nondet_bool();
        Manreg01b[14] = nondet_bool();
        Manreg01b[15] = nondet_bool();

        HOnR = nondet_bool();
        HOffR = nondet_bool();
        StartI = nondet_bool();
        TStopI = nondet_bool();
        FuStopI = nondet_bool();
        Al = nondet_bool();
        AuOnR = nondet_bool();
        AuOffR = nondet_bool();
        AuAuMoR = nondet_bool();
        AuIhMMo = nondet_bool();
        AuIhFoMo = nondet_bool();
        AuAlAck = nondet_bool();
        IhAuMRW = nondet_bool();
        AuRstart = nondet_bool();
        T_CYCLE = 5 + (nondet_unsignedshort() % 95);

        //extra assertion variables
        bool sFoMoSt_aux = FoMoSt_aux;
        bool sAuAuMoR = AuAuMoR;
        bool sManreg01b8 = Manreg01b[8];
        bool sAuIhFoMo = AuIhFoMo;
        bool sAuIhMMo = AuIhMMo;
        bool sMMoSt_aux = MMoSt_aux;

        //input manager
        E_MAuMoR = R_EDGE(Manreg01b[8],&MAuMoR_old);
        E_MMMoR = R_EDGE(Manreg01b[9],&MMMoR_old);
        E_MFoMoR = R_EDGE(Manreg01b[10],&MFoMoR_old);
        E_MSoftLDR = R_EDGE(Manreg01b[11],&MSoftLDR_old);
        E_MOnR = R_EDGE(Manreg01b[12],&MOnR_old);
        E_MOffR = R_EDGE(Manreg01b[13],&MOffR_old);
        E_MEnRstartR = R_EDGE(Manreg01b[1],&MEnRstartR_old);
        E_MAlAckR = R_EDGE(Manreg01b[7],&MAlAckR_old);

        PFsPosOn = POnOffb.ParRegb[8];
        PHFOn = POnOffb.ParRegb[9];
        PHFOff = POnOffb.ParRegb[10];
        //PPulse = POnOffb.ParRegb[11];
        PPulse = false;
        PHLD = POnOffb.ParRegb[12];
        PHLDCmd = POnOffb.ParRegb[13];
        PAnim = POnOffb.ParRegb[14];
        POutOff = POnOffb.ParRegb[15];
        PEnRstart = POnOffb.ParRegb[0];
        PRstartFS = POnOffb.ParRegb[1];
        PFsPosOn2 = POnOffb.ParRegb[2];
        PPulseCste = POnOffb.ParRegb[3];

        E_AuAuMoR = R_EDGE(AuAuMoR,&AuAuMoR_old);
        E_AuAlAck = R_EDGE(AuAlAck,&AuAlAck_old);
        E_StartI = R_EDGE(StartI,&StartI_old);
        E_TStopI = R_EDGE(TStopI,&TStopI_old);
        E_FuStopI = R_EDGE(FuStopI,&FuStopI_old);
        E_Al = R_EDGE(Al,&Al_old);

        StartISt = StartI;
        TStopISt = TStopI;
        FuStopISt = FuStopI;

        //interock & acknowledge
        if(E_MAlAckR || E_AuAlAck){
            fullNotAcknowledged = false;
            AlUnAck = false;
        }else if(E_TStopI || E_StartI || E_FuStopI || E_Al){
            AlUnAck = true;
        }
        if(((PEnRstart && (E_MEnRstartR || AuRstart) && !FuStopISt) || (PEnRstart &&
            PRstartFS && (E_MEnRstartR || AuRstart))) && !fullNotAcknowledged){
            EnRstartSt = true;
        }
        if(E_FuStopI){
            fullNotAcknowledged = true;
            if(PEnRstart){
                EnRstartSt = false;
            }
        }
        InterlockR =  TStopISt || FuStopISt || fullNotAcknowledged || !EnRstartSt ||
                      (StartISt && !POutOff && !OutOnOV) ||
                      (StartISt && POutOff && ((PFsPosOn && OutOVSt_aux) || (!PFsPosOn && !
                          OutOVSt_aux)));
        FE_InterlockR  = F_EDGE (InterlockR,&FE_InterlockR_old);

        //mode manager
        if(!(HLD && PHLD)){
            //forced mode
            if((AuMoSt_aux || MMoSt_aux || SoftLDSt_aux) && E_MFoMoR && !(AuIhFoMo)){
                    AuMoSt_aux = false;
                    MMoSt_aux = false;
                    FoMoSt_aux = true;
                    SoftLDSt_aux = false;
            }
            //manual mode
            if((AuMoSt_aux || FoMoSt_aux || SoftLDSt_aux) && E_MMMoR && !(AuIhMMo)){
                    AuMoSt_aux = false;
                    MMoSt_aux = true;
                    FoMoSt_aux = false;
                    SoftLDSt_aux = false;
            }
            //auto mode
            if((MMoSt_aux && (E_MAuMoR || E_AuAuMoR )) || (FoMoSt_aux && E_MAuMoR) ||
            (SoftLDSt_aux && E_MAuMoR) || (MMoSt_aux && AuIhMMo) || (FoMoSt_aux && AuIhFoMo)
                 ||
                        (SoftLDSt_aux && AuIhFoMo) || !(AuMoSt_aux || MMoSt_aux ||
                             FoMoSt_aux || SoftLDSt_aux)){
                    AuMoSt_aux   = true;
                    MMoSt_aux    = false;
                    FoMoSt_aux   = false;
                    SoftLDSt_aux = false;
            }
            // Software Local Mode
             if((AuMoSt_aux || MMoSt_aux) && E_MSoftLDR && !(AuIhFoMo)){
                    AuMoSt_aux = false;
                    MMoSt_aux = false;
                    FoMoSt_aux = false;
                    SoftLDSt_aux = true;
            }
            // Status setting
            LDSt = false;
            AuMoSt = AuMoSt_aux;
            MMoSt = MMoSt_aux;
            FoMoSt = FoMoSt_aux;
            SoftLDSt = SoftLDSt_aux;
        }else{
         // Local Drive Mode
            AuMoSt = false;
            MMoSt = false;
            FoMoSt = false;
            LDSt = true;
            SoftLDSt= false;
        }

            // LIMIT MANAGER
         // On/Open Evaluation
             OnSt= (HFOn && PHFOn) ||
                    (!PHFOn && PHFOff && PAnim && !HFOff) ||
                    (!PHFOn && !PHFOff && OutOVSt_aux);

         // Off/Closed Evaluation
             OffSt=(HFOff && PHFOff) ||
                    (!PHFOff && PHFOn && PAnim && !HFOn) ||
                    (!PHFOn && !PHFOff && !OutOVSt_aux);

    // REQUEST MANAGER
         // Auto On/Off Request
            if( AuOffR ){
                AuOnRSt = false;
            }else if( AuOnR ){
                AuOnRSt = true;
            }else if( fullNotAcknowledged || FuStopISt || !EnRstartSt ){
                AuOnRSt =  PFsPosOn;
            }
            AuOffRSt= !AuOnRSt;

         // Manual On/Off Request
            if( (((E_MOffR && (MMoSt || FoMoSt || SoftLDSt))
               || (AuOffRSt && AuMoSt)
               || (LDSt && PHLDCmd && HOffRSt)
               || (FE_PulseOn && PPulse && !POutOff) && EnRstartSt)
               || (E_FuStopI && !PFsPosOn)) ){

                    MOnRSt = false;

            }else if( (((E_MOnR  && (MMoSt || FoMoSt || SoftLDSt))
                  || (AuOnRSt  && AuMoSt)
                  || (LDSt && PHLDCmd && HOnRSt) && EnRstartSt)
                  || (E_FuStopI && PFsPosOn)) ){

                    MOnRSt = true;
            }

            MOffRSt= !MOnRSt;

         // Local Drive Request
            if( HOffR ){
                HOnRSt = false;
            }else if( HOnR ){
                HOnRSt = true;
            }
            HOffRSt =  !(HOnRSt);

    // PULSE REQUEST MANAGER
    if( PPulse ){
          if( InterlockR ){
             PulseOnR= (PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
             PulseOffR= (!PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
          }else if( FE_InterlockR ){
             PulseOnR= false;
             PulseOffR= false;
             updateTP(&Timer_PulseOn.Q, &Timer_PulseOn.ET, &Timer_PulseOn.old_in, &Timer_PulseOn.due, false,0);
             updateTP(&Timer_PulseOff.Q, &Timer_PulseOff.ET, &Timer_PulseOff.old_in, &Timer_PulseOff.due, false,0);
          }else if( (MOffRSt && (MMoSt || FoMoSt || SoftLDSt)) || (AuOffRSt && AuMoSt) || (HOffR && LDSt && PHLDCmd) ){  //Off Request
             PulseOnR= false;
             PulseOffR= true;
          }else if( (MOnRSt && (MMoSt || FoMoSt || SoftLDSt)) || (AuOnRSt && AuMoSt) || (HOnR && LDSt && PHLDCmd) ){ //On Request
              PulseOnR= true;
              PulseOffR= false;
          }else {
             PulseOnR= false;
             PulseOffR= false;
          }

          //Pulse functions
          updateTP(&Timer_PulseOn.Q, &Timer_PulseOn.ET, &Timer_PulseOn.old_in, &Timer_PulseOn.due, PulseOnR, POnOffb.PPulseLeb);
          updateTP(&Timer_PulseOff.Q, &Timer_PulseOff.ET, &Timer_PulseOff.old_in, &Timer_PulseOff.due, PulseOffR, POnOffb.PPulseLeb);
          RE_PulseOn    = R_EDGE(PulseOn, RE_PulseOn_old);
          FE_PulseOn    = F_EDGE(PulseOn, FE_PulseOn_old);
          RE_PulseOff   = R_EDGE(PulseOff, RE_PulseOff_old);

          //The pulse functions have to be reset when changing from On to Off
          if( RE_PulseOn ){
              updateTP(&Timer_PulseOff.Q, &Timer_PulseOff.ET, &Timer_PulseOff.old_in, &Timer_PulseOff.due, false,0);
          }
          if( RE_PulseOff ){
             updateTP(&Timer_PulseOn.Q, &Timer_PulseOn.ET, &Timer_PulseOn.old_in, &Timer_PulseOn.due, false,0);
          }
          if( PPulseCste ){
              PulseOn  = Timer_PulseOn.Q && !PulseOffR;
              PulseOff = Timer_PulseOff.Q && !PulseOnR;
          }else{
              PulseOn  = Timer_PulseOn.Q && !PulseOffR && (!PHFOn || (PHFOn && !HFOn));
              PulseOff = Timer_PulseOff.Q && !PulseOnR && (!PHFOff || (PHFOff && !HFOff));
          }
      }

       // Output On Request
         OutOnOVSt = (PPulse && PulseOn) ||
                       (!PPulse && ((MOnRSt && (MMoSt || FoMoSt || SoftLDSt)) ||
                       (AuOnRSt && AuMoSt) ||
                       (HOnRSt && LDSt && PHLDCmd)));

       // Output Off Request
         if( POutOff ){
              OutOffOVSt = (PulseOff && PPulse) ||
                  (!PPulse && ((MOffRSt && (MMoSt || FoMoSt || SoftLDSt)) || (AuOffRSt && AuMoSt) || (HOffRSt && LDSt &&  PHLDCmd)));
         }

       // Interlocks / FailSafe
         if( POutOff ){
              if( InterlockR ){
                  if( PPulse && !PFsPosOn2 ){
                      if( PFsPosOn ){
                          OutOnOVSt = PulseOn;
                          OutOffOVSt =  false;
                      }else{
                          OutOnOVSt = false;
                          OutOffOVSt =  PulseOff;
                      }
                  }else{
                      OutOnOVSt = (PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
                      OutOffOVSt= (!PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
                  }
              }
          }else{
              if( InterlockR ){
                      OutOnOVSt= PFsPosOn;
              }
          }

       // Ready to Start Status
       RdyStartSt = !InterlockR;

```

```
            //Alarms
            AlSt = Al;

        // SURVEILLANCE
            // I/O Warning
            IOErrorW = IOError;
            IOSimuW  = IOSimu;

            // Auto<> Manual Warning
            AuMRW    = (MMoSt || FoMoSt || SoftLDSt) &&
                        (((AuOnRSt || MOnRSt) && !(AuOnRSt && MOnRSt)) || ((AuOffRSt || MOffRSt
                             ) && !(AuOffRSt && MOffRSt))) && !IhAuMRW;

        // OUTPUT_MANAGER && OUTPUT REGISTER
           if( !POutOff ){
               if( PFsPosOn ){
                    OutOnOV = !OutOnOVSt;
               }else{
                    OutOnOV = OutOnOVSt;
               }
           }else{
               OutOnOV  = OutOnOVSt;
               OutOffOV = OutOffOVSt;
           }

       // Position warning
               // Set reset of the OutOnOVSt
               if( OutOnOVSt || (PPulse && PulseOnR) ){
                     OutOVSt_aux = true;
               }
               if(  (OutOffOVSt && POutOff) || (!OutOnOVSt && !POutOff) || (PPulse && PulseOffR
                    ) ){
                     OutOVSt_aux = false;
               }
               RE_OutOVSt_aux    = R_EDGE(OutOVSt_aux,RE_OutOVSt_aux_old);
               FE_OutOVSt_aux    = F_EDGE(OutOVSt_aux,FE_OutOVSt_aux_old);
           if( ((OutOVSt_aux && ((PHFOn && !OnSt) || (PHFOff && OffSt)))
               || (!OutOVSt_aux && ((PHFOff && !OffSt) || (PHFOn && OnSt)))
               || (OffSt && OnSt))
               && (!PPulse || (POutOff && PPulse && !OutOnOV && !OutOffOV))
           ){
               PosW_aux= true;
           }
           if(  !((OutOVSt_aux && ((PHFOn && !OnSt) || (PHFOff && OffSt)))
               || (!OutOVSt_aux && ((PHFOff && !OffSt) || (PHFOn && OnSt)))
               || (OffSt && OnSt))
               || RE_OutOVSt_aux
               || FE_OutOVSt_aux
               || (PPulse && POutOff && OutOnOV)
               || (PPulse && POutOff && OutOffOV)
           ){
               PosW_aux = false;
           }
           updateTON(&Timer_Warning.Q, &Timer_Warning.ET, &Timer_Warning.running, &
                Timer_Warning.start, PosW_aux, POnOffb.PWDtb);

           PosW = Timer_Warning.Q;
           Time_Warning = Timer_Warning.ET;

        // Alarm Blocked Warning
           AlBW = AlB;

        // Maintain Interlock status 1.5s in Stsreg for PVSS
       PulseWidth = 1500/T_CYCLE;

       if( FuStopISt || FSIinc > 0 ){
           FSIinc = FSIinc + 1;
           WFuStopISt = true;
       }
       if( FSIinc > PulseWidth || (!FuStopISt && FSIinc == 0) ){
           FSIinc = 0;
           WFuStopISt = FuStopISt;
       }
       if( TStopISt || TSIinc > 0 ){
           TSIinc = TSIinc + 1;
           WTStopISt = true;
       }
       if( TSIinc > PulseWidth || (!TStopISt && TSIinc == 0) ){
           TSIinc = 0;
           WTStopISt = TStopISt;
       }
       if( StartISt || SIinc > 0 ){
           SIinc = SIinc + 1;
           WStartISt= true;
       }
       if( SIinc > PulseWidth || (!StartISt && SIinc == 0) ){
           SIinc = 0;
           WStartISt = StartISt;
       }
       if( AlSt || Alinc > 0 ){
            Alinc = Alinc + 1;
            WAlSt = true;
        }
        if( Alinc > PulseWidth || (!AlSt && Alinc == 0) ){
            Alinc = 0;
            WAlSt = AlSt;
        }

         // STATUS REGISTER
        Stsreg01b[8]  = OnSt;             //StsReg01 Bit 00
        Stsreg01b[9]  = OffSt;            //StsReg01 Bit 01
        Stsreg01b[10] = AuMoSt;           //StsReg01 Bit 02
        Stsreg01b[11] = MMoSt;            //StsReg01 Bit 03
        Stsreg01b[12] = FoMoSt;           //StsReg01 Bit 04
        Stsreg01b[13] = LDSt;             //StsReg01 Bit 05
        Stsreg01b[14] = IOErrorW;         //StsReg01 Bit 06
        Stsreg01b[15] = IOSimuW;          //StsReg01 Bit 07
        Stsreg01b[0]  = AuMRW;            //StsReg01 Bit 08
        Stsreg01b[1]  = PosW;             //StsReg01 Bit 09
        Stsreg01b[2]  = WStartISt;        //StsReg01 Bit 10
        Stsreg01b[3]  = WTStopISt;        //StsReg01 Bit 11
        Stsreg01b[4]  = AlUnAck;          //StsReg01 Bit 12
        Stsreg01b[5]  = AuIhFoMo;         //StsReg01 Bit 13
        Stsreg01b[6]  = WAlSt;            //StsReg01 Bit 14
        Stsreg01b[7]  = AuIhMMo;          //StsReg01 Bit 15

        Stsreg02b[8]  = OutOnOVSt;        //StsReg02 Bit 00
        Stsreg02b[9]  = AuOnRSt;          //StsReg02 Bit 01
        Stsreg02b[10] = MOnRSt;           //StsReg02 Bit 02
        Stsreg02b[11] = AuOffRSt;         //StsReg02 Bit 03
        Stsreg02b[12] = MOffRSt;          //StsReg02 Bit 04
        Stsreg02b[13] = HOnRSt;           //StsReg02 Bit 05
        Stsreg02b[14] = HOffRSt;          //StsReg02 Bit 06
        Stsreg02b[15] = 0;                //StsReg02 Bit 07
        Stsreg02b[0]  = 0;                //StsReg02 Bit 08
        Stsreg02b[1]  = 0;                //StsReg02 Bit 09
        Stsreg02b[2]  = WFuStopISt ;      //StsReg02 Bit 10
        Stsreg02b[3]  = EnRstartSt;       //StsReg02 Bit 11
        Stsreg02b[4]  = SoftLDSt;         //StsReg02 Bit 12
        Stsreg02b[5]  = AlBW;             //StsReg02 Bit 13
        Stsreg02b[6]  = OutOffOVSt;       //StsReg02 Bit 14
        Stsreg02b[7]  = 0;                //StsReg02 Bit 15

        // Edges
        DETECT_EDGE(AlUnAck, AlUnAck_old,&RE_AlUnAck,&FE_AlUnAck);
        __GLOBAL_TIME = __GLOBAL_TIME + T_CYCLE;

        //Properties
        first = false;
        }
}
```

Listing G.1: CPC.c

## H Properties of the CPC program

In this appendix we show the properties of the CPC program. The property is given as an assertion for the C programs and in CTL for the SMV programs. Note that to get the invariant property for SMV the letters 'AG' should be removed from the CTL property.

- R1-1
  Assertion: `(!(sFoMoSt_aux && !sAuAuMoR && !sManreg01b8) || !AuMoSt)`
  CTL: `AG(loc = end -> (!(sFoMoSt_aux & !sAuAuMoR & !sManreg01b8) | !AuMoSt))`
  Result: `false`
- R1-2
  Assertion `(!(AuMoSt) || (sAuAuMoR || sManreg01b8 || sAuIhFoMo || !sFoMoSt_aux))`
  CTL: `AG(loc = end -> (!(AuMoSt) | (sAuAuMoR | sManreg01b8 | sAuIhFoMo | !sFoMoSt_aux)))`
  Result: `true`
- R1-3
  Assertion `(!(sFoMoSt_aux && sAuAuMoR && !sManreg01b8) || !AuMoSt)`
  CTL: `AG(loc = end -> (!(sFoMoSt_aux & sAuAuMoR & !sManreg01b8) | !AuMoSt))`
  Result: `false`
- R1-4
  Assertion `(!(sFoMoSt_aux && sAuAuMoR && !sManreg01b8 && !sAuIhFoMo && !sAuIhMMo) || !AuMoSt)`

CTL: `AG(loc = end -> (!(sFoMoSt_aux & sAuAuMoR & !sManreg01b8 & !sAuIhFoMo & !sAuIhMMo) | !AuMoSt))`
Result: `false`

- R1-5
  Assertion `(!(sMMoSt_aux && !sAuAuMoR && !sManreg01b8) || !AuMoSt)`
  CTL: `AG(loc = end -> (!(sMMoSt_aux & !sAuAuMoR & !sManreg01b8) | !AuMoSt))`
  Result: `false`
- R1-6
  Assertion `(!(AuMoSt) || (sAuAuMoR || sManreg01b8 || sAuIhMMo || !sMMoSt_aux))`
  CTL: `AG(loc = end -> (!(AuMoSt) | (sAuAuMoR | sManreg01b8 | sAuIhMMo | !sMMoSt_aux)))`
  Result: `true`
- R1-7
  Assertion `(!AuIhFoMo || !SoftLDSt)`
  CTL: `AG(loc = end -> (!AuIhFoMo | !SoftLDSt))`
  Result: `true`
- R1-8
  Assertion `(!AuIhFoMo || !FoMoSt)`
  CTL: `AG(loc = end -> (!AuIhFoMo | !FoMoSt))`
  Result: `true`
- R1-9
  Assertion `(!AuIhMMo || !MMoSt)`
  CTL: `AG(loc = end -> (!AuIhMMo | !MMoSt))`
  Result: `true`
- R1-11a
  We have chosen to split property R1-11 in two properties.
  Assertion `(!(!pOutOnOV && !pTStopI && !pFuStopI && !pStartI && HLD && !HOnR && !HOffR && !TStopI && !FuStopI && !StartI) || !OutOnOV)`
  CTL: `AG(loc = end -> (!(!pOutOnOV & !pTStopI & !pFuStopI & !pStartI & HLD & !HOnR & !HOffR & !TStopI & !FuStopI & !StartI) | !OutOnOV | first))`
  Result: `false`
- R1-11b
  Assertion `(!(pOutOnOV && !pTStopI && !pFuStopI && !pStartI && HLD && !HOnR && !HOffR && !TStopI && !FuStopI && !StartI) || OutOnOV)`
  CTL: `AG(loc = end -> (!(pOutOnOV & !pTStopI & !pFuStopI & !pStartI & HLD & !HOnR & !HOffR & !TStopI & !FuStopI & !StartI) | OutOnOV | first))`
  Result: `false`
- R2-1
  Assertion `(!AuOffR || AuOffRSt)`
  CTL: `AG(loc = end -> (!AuOffR | AuOffRSt))`
  Result: `true`
- R2-2
  Assertion `(!AuOnR || AuOnRSt)`
  CTL: `AG(loc = end -> (!AuOnR | AuOnRSt))`
  Result: `false`
- R2-3
  Assertion `(!(AuOnR && !AuOffR && AuMoSt && !InterlockR && !PFsPosOn) || OutOnOV)`
  CTL: `AG(loc = end -> (!(AuOnR & !AuOffR & AuMoSt & !InterlockR & !PFsPosOn) | OutOnOV))`
  Result: `true`
- R2-4
  Assertion `(!(AuOffR && !AuOnR && AuMoSt && !InterlockR && !PFsPosOn) || !OutOnOV)`
  CTL: `AG(loc = end -> (!(AuOffR & !AuOnR & AuMoSt & !InterlockR & !PFsPosOn) | !OutOnOV))`
  Result: `true`
- R2-5
  Assertion `(!(!pManreg01b12 && Manreg01b[12] && !Manreg01b[13] && MMoSt && !InterlockR && !PFsPosOn) || OutOnOV)`
  CTL: `AG(loc = end -> (!(!pManreg01b12 & Manreg01b[12] & !Manreg01b[13] & MMoSt & !InterlockR & !PFsPosOn) | OutOnOV | first))`

Result: `true`

- R2-6
  Assertion `(!(!pManreg01b13 && Manreg01b[13] && !Manreg01b[12] && MMoSt && !InterlockR && !PFsPosOn) || !OutOnOV)`
  CTL: `AG(loc = end -> (!(!pManreg01b13 & Manreg01b[13] & !Manreg01b[12] & MMoSt & !InterlockR & !PFsPosOn) | !OutOnOV | first))`
  Result: `true`
- R2-7
  Assertion `(!(HOnR && !HOffR && LDSt && !InterlockR && !PFsPosOn) || OutOnOV)`
  CTL: `AG(loc = end -> (!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) | OutOnOV))`
  Result: `false`
- R2-8
  Assertion `(!(HOffR && !HOnR && LDSt && !InterlockR && !PFsPosOn) || !OutOnOV)`
  CTL: `AG(loc = end -> (!(HOffR & !HOnR & LDSt & !InterlockR & !PFsPosOn) | !OutOnOV))`
  Result: `true`
- R3-1
  Assertion `(!(InterlockR && PFsPosOn) || OutOnOVSt)`
  CTL: `AG(loc = end -> (!(InterlockR & PFsPosOn) | OutOnOVSt))`
  Result: `true`
- R3-2
  Assertion `(!(InterlockR && PFsPosOn)|| !OutOffOVSt)`
  CTL: `AG(loc = end -> (!(InterlockR & PFsPosOn)| !OutOffOVSt))`
  Result: `false`
- R3-3
  Assertion `(!((TStopI || FuStopI || !EnRstartSt) && !PFsPosOn && PEnRstart && !PRstartFS) || !OutOnOVSt)`
  CTL: `AG(loc = end -> (!((TStopI | FuStopI | !EnRstartSt) & !PFsPosOn & PEnRstart & !PRstartFS) | !OutOnOVSt))`
  Result: `true`
- R3-4
  Assertion `(!((FuStopI || fullNotAcknowledged) && PEnRstart && !PRstartFS) || EnRstartSt)`
  CTL: `AG(loc = end -> (!((FuStopI | fullNotAcknowledged) & PEnRstart & !PRstartFS) | EnRstartSt))`
  Result: `false`
- R3-5
  Assertion `(!(!pOutOnOVSt && (StartI || FuStopI || TStopI || !EnRstartSt) && !PFsPosOn && PEnRstart && !PRstartFS) || !OutOnOVSt)`
  CTL: `AG(loc = end -> (!(!pOutOnOVSt & (StartI | FuStopI | TStopI | !EnRstartSt) & !PFsPosOn & PEnRstart & !PRstartFS) | !OutOnOVSt | first))`
  Result: `false`
- R3-6
  Assertion `(!(FuStopISt && !PFsPosOn && !POutOff && !PPulse) || !OutOnOV)`
  CTL: `AG(loc = end -> (!(FuStopISt & !PFsPosOn & !POutOff & !PPulse) | !OutOnOV))`
  Result: `true`
- R3-7
  Assertion `(!(pFuStopI && pMMoSt && !FuStopI && !Manreg01b[12] && MMoSt) || ((MOnRSt && PFsPosOn) || (!MOnRSt && !PFsPosOn)))`
  CTL: `AG(loc = end -> (!(pFuStopI & pMMoSt & !FuStopI & !Manreg01b[12] & MMoSt) | ((MOnRSt & PFsPosOn) | (!MOnRSt & !PFsPosOn)) | first))`
  Result: `false`
- R4-1
  Assertion `(!(!HFOn || !HFOff) || (!OnSt || !OffSt))`
  CTL: `AG(loc = end -> (!(!HFOn | !HFOff) | (!OnSt | !OffSt)))`
  Result: `true`