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 5

2 Preliminaries 7
   2.1 Programmable Logic Controller (PLC) 7
   2.2 Structured Control Language (SCL) 7

3 Problem Description 12
   3.1 Verification at CERN 12
   3.2 Approach 15

4 Verification tools on Models 18
   4.1 Spin 18
   4.2 NuSMV and nuXmv 23
   4.3 Experiments and results 31

5 Verification tools on Code 33
   5.1 C 33
   5.2 CBMC 36
   5.3 K-Inductor 37
   5.4 CBMC Incremental 39
   5.5 2LS 41
   5.6 CPA-checker 41
   5.7 SATABS 42
   5.8 Experiments and results 43

6 Case Study 50
   6.1 The CPC program 50
   6.2 Translation into SMV 51
   6.3 Translation into C 52
   6.4 Properties 53
   6.5 Experiments and results 53

7 Conclusion 56
   7.1 Future work 56

Appendices 61

A Example programs in SCL 61
B Example programs in PROMELA 65
C Example programs in SMV 71
D Example programs in C 82
E CPC program in SCL 87
F  CPC program in SMV  97
G  CPC program in C  119
H  Properties of the CPC program  127
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 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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Bits</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td>1</td>
<td>true, false</td>
</tr>
<tr>
<td>INT</td>
<td>16</td>
<td>Signed Integer</td>
</tr>
<tr>
<td>UINT</td>
<td>16</td>
<td>Unsigned Integer</td>
</tr>
<tr>
<td>WORD</td>
<td>16</td>
<td>Bit combinations</td>
</tr>
<tr>
<td>TIME</td>
<td>32</td>
<td>-24d 20h 31m 23s 647ms to 24d 20h 31m 23s 647ms</td>
</tr>
<tr>
<td>REAL</td>
<td>32</td>
<td>Floating Point</td>
</tr>
</tbody>
</table>

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].

```scl
1  // The main program
2 FUNCTION_BLOCK SimpleExample
3 VAR_INPUT
4   error : BOOL; // not used
5   toMode1 : BOOL; // request to switch to mode1
6   toMode2 : BOOL; // request to switch to mode2
7   toMode3 : BOOL; // request to switch to mode3
8   mode3Forbidden : BOOL; // if it is true, it is forbidden to be in mode3
9 END_VAR
10
11 VAR
12   mode1 : BOOL; // true if the block is in mode1
13   mode2 : BOOL; // true if the block is in mode2
14   mode3 : BOOL; // true if the block is in mode3
15 END_VAR
16 VAR_OUTPUT
17   mode : INT;
18 END_VAR
19
20 BEGIN
21  // Operation mode handling
22  IF NOT mode1 AND NOT mode2 AND NOT mode3 THEN
23     mode1 := TRUE;
24  END_IF;
25
26  IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
27     mode1 := TRUE;
28  END_IF;
29  IF toMode2 THEN
30     mode2 := TRUE;
31  ELSIF toMode3 THEN
32     mode3 := TRUE;
33  END_IF;
34
35  IF mode1 THEN
36     mode := 1;
37  ELSIF mode2 THEN
38     mode := 2;
39  ELSIF mode3 THEN
40     mode := 3;
41  ELSE
42     mode := 0;
43  END_IF;
44 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.

```plaintext
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 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`. 

```plaintext
10
```
Listing 2.3: User-defined data type in SCL

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

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 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 \( a = (L, T, l_0, V_a, Val_0) \) where \( L = \{l_0, l_1, \ldots \} \) 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 \( Val_0 \) 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 \( l_0 \) to \( l_1 \);

- 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 \( l_2 \) to \( l_3 \) and from \( l_3 \) to \( l_4 \). The transition from \( l_2 \) to \( l_3 \) first assigns the parameter of the function. In this case it assigns the value \texttt{TRUE} to variable \( a \) in the function. It also has a synchronization \( i_1! \). This ensures that this transition is synchronized with the transition with \( i_1? \) in the Function block. The other transition (from \( l_3 \) to \( l_4 \)) with synchronization \( i_2? \) has
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 10 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
```

We can see that there are three transitions added from 10. These transitions are the three branches of the control statement. One transition has the condition from the
IF-statement as guard (from 10 to 11), one has the negation of this condition and the condition from the ELSEIF-statement as guard (from 10 to 12), and one transition has the negation of both conditions as guard (from 10 to 15). We can see that all three branches will end up in location 15. 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 10 up to 112 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.
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

<table>
<thead>
<tr>
<th>Variable</th>
<th>Data type</th>
<th>Variable type</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal</td>
<td>Boolean</td>
<td>Input variable</td>
</tr>
<tr>
<td>out1</td>
<td>Boolean</td>
<td>Output variable</td>
</tr>
<tr>
<td>out2</td>
<td>Boolean</td>
<td>Output variable</td>
</tr>
<tr>
<td>out3.out1</td>
<td>Boolean</td>
<td>Output variable</td>
</tr>
<tr>
<td>out3.elapsed</td>
<td>Integer</td>
<td>Output variable</td>
</tr>
</tbody>
</table>

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 SCI 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.

```plaintext
do::program
do
```

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:

```plaintext
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.

```plaintext
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.

```
1 bool error , toMode1 , toMode2 , toMode3 , mode3Forbidden ;
2 bool model1 = 0 ;
3 bool mode2 = 0 ;
4 bool mode3 = 0 ;
5 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 ;
```
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
  ::!b && b2 -> s2
  ::else
fi;
```

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: \texttt{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 \texttt{spin -a Example.pml} Spin makes an exhaustive state space searching program for the model, which results in five files named \texttt{pan.[bcnmt]}. We can compile this program with \texttt{gcc pan pan.c}, which gives us an executable \texttt{pan}. Executing this executable completes the verification. With this last step we can use the option \texttt{-m N} to set maximal search depth to \texttt{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 \texttt{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 \texttt{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: \texttt{s1; s2; s3}. This gives us 4 states: one at the beginning and one after each statement. Now if we use a \texttt{d_step} we get \texttt{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 \texttt{d_step} sequence can only contain deterministic code. We add the \texttt{d_step} after the non-deterministic assignments of the input variables. The structure of the program with the \texttt{d_step} is shown in Listing 4.4.

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

Listing 4.4: PROMELA structure with \texttt{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.

\[ \text{next}(a) := \begin{cases} 
\text{case} & \\
\text{b1} : s1; & \\
\text{b2} : s2; & \\
\text{TRUE}: a; & \\
\text{esac}; & 
\end{cases} \]

In this example, if in the boolean condition $\text{b1}$ holds in the current state, the value of variable $a$ in the next state will be $s1$. If $\text{b1}$ does not hold and $\text{b2}$ does, the value of $a$ in the next state will be $s2$. If neither $\text{b1}$ nor $\text{b2}$ 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 \texttt{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 \texttt{loc} variable. In the initial state of the FSM the variable \texttt{loc} has the value \texttt{start}. All possible paths from this location will reach the location where \texttt{loc} = \texttt{end}. To make the unconditional loop we add a transition from \texttt{loc} = \texttt{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 \texttt{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:

\[
\text{next}(a) := \{\text{TRUE, FALSE}\};
\]

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 1. At any other location its value will remain the same, as we can see in the following construction:

\[
\text{next}(a) := \\
\text{case} \\
\text{ (loc = 1) : } \{\text{TRUE, FALSE}\}; \\
\text{ TRUE : a; } \\
\text{ esac;}
\]

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 \( \text{loc} \) variable at the corresponding SMV code is shown.

\[
\begin{align*}
\text{(l1)} & \text{ IF } a \text{ THEN} \\
\text{(l2)} & \text{ b := TRUE} \\
\text{(l1)} & \text{ ELSE} \\
\text{(l3)} & \text{ b := FALSE} \\
\text{(l4)} & \text{ END IF}
\end{align*}
\]

\[
\begin{align*}
\text{init(loc)} := & \text{ l1;} \\
\text{next(loc)} := & \text{ case} \\
\text{ (loc = 11) & (a) : } & \text{ l2;} \\
\text{ (loc = 11) : } & \text{ l3;} \\
\text{ (loc = 12) : } & \text{ l4;} \\
\text{ (loc = 13) : } & \text{ l4;} \\
\text{ esac;} \\
\text{next(b)} := & \text{ case} \\
\text{ (loc = 12) : } & \text{ TRUE;} \\
\text{ (loc = 13) : } & \text{ FALSE;} \\
\text{ TRUE : b;} \\
\text{ esac;}
\end{align*}
\]

The \( \text{loc} \) variable starts at location 11. For the conditional choice we use the condition as a guard in the case distinction of the \( \text{loc} \) variable. If the condition holds we go to location 12 and if it does not we go to location 13. 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 12 and 13. 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 14. 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 \( \text{loc} = l1 \). The condition \( i < 10 \) is checked at the location variable. If \( i < 10 \), the program will go to the body of the loop (location 12). Otherwise the program will continue after the loop (location 13). At the end of the body of the loop the location returns to the beginning of the loop (location 11). When we are at location 12, \( i \) 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.

```
1 MODULE main
2 Variable declaration
```
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.

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 \textit{out3} of this type.

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

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

**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.

```plaintext
next(error):=
  case
    (loc = start) : {TRUE, FALSE};
    TRUE : error;
```

**Listing 4.7: SMV location specification**

**Listing 4.8: SMV Variable initialization**

**Listing 4.9: SMV Variable initialization**
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 \texttt{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 \texttt{loc}. The last case has condition \texttt{TRUE} and does not change the value. Note that this \texttt{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 \texttt{loc}, which divides the program into steps. At the start of the program body the variable \texttt{loc} has value \texttt{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: \(!, |, &, \to, \leftrightarrow\) 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 \( \forall G \) 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 \( \text{out2} \) is true then \( \text{out1} \) should be true too.
  Logical expression: \((\text{loc} = \text{end} \rightarrow (\text{out2} \rightarrow \text{out1}))\)

- If \( \text{signal} \) is false then \( \text{out2} \) should be false too.
  Logical expression: \((\text{loc} = \text{end} \rightarrow (!\text{signal} \rightarrow !\text{out2}))\)

- \( \text{out3.out1} \) equals \( \text{out1} \).
  Logical expression: \((\text{loc} = \text{end} \rightarrow (\text{out3}.\text{out1} = \text{out1}))\)

- \( \text{out3.elapsed} \) is 0 when \( \text{out1} \) is false.
  Logical expression: \((\text{loc} = \text{end} \rightarrow (!\text{out1} \rightarrow \text{out3.elapsed} = 0sd16_0))\)
For all expressions an implication with \texttt{loc = end} is used because we want to check the property at the end of the program. CTL specifications are added with \texttt{SPEC property} and invariant specifications are added with \texttt{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 \texttt{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:

\begin{verbatim}
NuSMV -int Example.smv
go
cHECK cTlSPEc
\end{verbatim}

For the invariant properties we change the last command to \texttt{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:

\begin{verbatim}
nuXmv -int Example.smv
go
build_boolean_model
check_invar_Ic3
\end{verbatim}

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.

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

- \texttt{-dynamic} Enables dynamic reordering of variables, this can reduce the size of the BDDs

- \texttt{-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.

<table>
<thead>
<tr>
<th>Program</th>
<th>Property</th>
<th>Spin</th>
<th>NuSMV</th>
<th>nuXmv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>CTL</td>
<td>Invariants</td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>3m0.03s</td>
<td>44.85s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2(true)</td>
<td>2m52.44s</td>
<td><strong>0.34s</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>35.28s</td>
<td>0.31s</td>
<td><strong>0.26s</strong></td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>11.43s</td>
<td><strong>0.30s</strong></td>
<td>0.41s</td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>12.15s</td>
<td>20.47s</td>
<td>-</td>
</tr>
<tr>
<td>reduced</td>
<td>2(true)</td>
<td>12.41s</td>
<td><strong>0.07s</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>11.61s</td>
<td><strong>0.07s</strong></td>
<td>0.09s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>13.17s</td>
<td>0.18s</td>
<td><strong>0.16s</strong></td>
</tr>
<tr>
<td>Example_int</td>
<td>1(true)</td>
<td>22m1.45s</td>
<td>58m31.38s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2(true)</td>
<td>21m57.84s</td>
<td><strong>0.28s</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>13.39s</td>
<td><strong>0.35s</strong></td>
<td>0.47s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>12.13s</td>
<td>0.64s</td>
<td><strong>0.52s</strong></td>
</tr>
<tr>
<td>Example_int</td>
<td>1(true)</td>
<td>17.19s</td>
<td>69m36.24s</td>
<td>-</td>
</tr>
<tr>
<td>reduced</td>
<td>2(true)</td>
<td>17.93s</td>
<td><strong>0.14s</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>11.40s</td>
<td>0.18s</td>
<td><strong>0.15s</strong></td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>12.20s</td>
<td><strong>0.19s</strong></td>
<td><strong>0.19s</strong></td>
</tr>
<tr>
<td>Example_while</td>
<td>1(true)</td>
<td>14m24.09s</td>
<td>22m56.11s</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2(true)</td>
<td>14m1.97s</td>
<td><strong>0.31s</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>12.23s</td>
<td>30.07s</td>
<td><strong>0.33s</strong></td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>13.22s</td>
<td>10.73s</td>
<td><strong>0.54s</strong></td>
</tr>
<tr>
<td>Example_while</td>
<td>1(true)</td>
<td><strong>17.26s</strong></td>
<td>7m25.84s</td>
<td>-</td>
</tr>
<tr>
<td>reduced</td>
<td>2(true)</td>
<td>16.71s</td>
<td><strong>0.11s</strong></td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>11.50s</td>
<td>3.62s</td>
<td><strong>0.12s</strong></td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>11.97s</td>
<td>3.40s</td>
<td><strong>0.16s</strong></td>
</tr>
</tbody>
</table>

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.

\[
\text{while}(b)\
\{\text{s}\}
\]

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.

```c
#include <stdbool.h>

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.

```c
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.

```
1 error = nondet_bool();
2 toMode1 = nondet_bool();
3 toMode2 = nondet_bool();
4 toMode3 = nondet_bool();
5 mode3Forbidden = nondet_bool();
6 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: \texttt{assert(property);}.

\textbf{Verification} To perform verification with CBMC we use the following command in the command line:

\texttt{cbmc Example.c --no-unwinding-assertions -unwind k} where \texttt{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 \texttt{--unwindset l:k} where \texttt{l} is the name of a loop and \texttt{k} is the bound on this loop. The names of the loops can be checked with the option \texttt{--show-loops}. The option \texttt{--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 \texttt{k = 3}. For all properties that are violated in the programs, a violation is found within this bound.

\section{K-Inductor}

K-Inductor \cite{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. \cite{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:

- \textbf{base case} \( p_0 \)
- \textbf{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 \( k \).

- \textbf{base case} \( p_0 \cdots p_{k-1} \)
- \textbf{step case} \( p_n \cdots p_{n+k-1} \implies p_{n+k} \)
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**

```c
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 \( (a_i \neq b_i) \land (a_{i+1} \neq b_{i+1}) \land (a_{i+2} \neq b_{i+2}) \implies a_{i+3} \neq b_{i+3} \). 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 \((x_i \neq 3 \land \cdots \land x_{i+k-1} \neq 3) \implies x_{i+k} \neq 3\). We can rewrite this to \((x_i \neq 3 \land \cdots \land x_i + 2(k-1) \neq 3) \implies x_i + 2k \neq 3\). 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.

```c
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();
        #endif
    }
}
```
• **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: `C PROVER 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 $kI$-I. For the experiments we have used 2LS version 0.3.4.

$kI$-I The $kI$-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 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 $kI$-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 ab-
straction 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:

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

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

```c
#include <stdbool.h>

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: \texttt{assert(property);}. To run a verification with SATABS, we use the following commands in the command line.

\begin{verbatim}
SATABS Example.c --modelchecker nusmv
\end{verbatim}

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 \texttt{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 \texttt{Example_int} and \texttt{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 \texttt{klkl} 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.
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.
<table>
<thead>
<tr>
<th>Program</th>
<th>Property</th>
<th>CBMC</th>
<th>K-Inductor</th>
<th>CBMC Incremental</th>
<th>2LS</th>
<th>CPA-checker</th>
<th>SATABS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2(true)</td>
<td>-</td>
<td>0.52s</td>
<td>0.42s</td>
<td>0.48s</td>
<td>0.43s</td>
<td>52.23s</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>0.42s</td>
<td>-</td>
<td>0.43s</td>
<td>0.51s</td>
<td>0.64s</td>
<td>9.65s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>0.45s</td>
<td>-</td>
<td>0.47s</td>
<td>0.51s</td>
<td>0.83s</td>
<td>12.25s</td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>reduced</td>
<td>2(true)</td>
<td>-</td>
<td>0.50s</td>
<td>0.41s</td>
<td>0.56s</td>
<td>0.43s</td>
<td>9.31s</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>0.43s</td>
<td>-</td>
<td>0.41s</td>
<td>0.45s</td>
<td>0.53s</td>
<td>10.60s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>0.41s</td>
<td>-</td>
<td>0.40s</td>
<td>0.51s</td>
<td>0.65s</td>
<td>11.99s</td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Int</td>
<td>2(true)</td>
<td>-</td>
<td>0.51s</td>
<td>0.46s</td>
<td>0.50s</td>
<td>0.45s</td>
<td>12m52.55s</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>0.43s</td>
<td>-</td>
<td>0.43s</td>
<td>0.49s</td>
<td>0.66s</td>
<td>9.04s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>0.44s</td>
<td>-</td>
<td>0.40s</td>
<td>0.53s</td>
<td>0.88s</td>
<td>12.51s</td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>reduced</td>
<td>2(true)</td>
<td>-</td>
<td>0.47s</td>
<td>0.41s</td>
<td>0.45s</td>
<td>0.35s</td>
<td>13.622s</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>0.43s</td>
<td>-</td>
<td>0.42s</td>
<td>0.48s</td>
<td>0.51s</td>
<td>9.20s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>0.42s</td>
<td>-</td>
<td>0.41s</td>
<td>0.49s</td>
<td>0.67s</td>
<td>12.55s</td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>While</td>
<td>2(true)</td>
<td>-</td>
<td>0.54s</td>
<td>-</td>
<td>0.45s</td>
<td>0.37s</td>
<td>2m34.81s</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>0.44s</td>
<td>-</td>
<td>0.45s</td>
<td>0.51s</td>
<td>0.75s</td>
<td>9.15s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>0.45s</td>
<td>-</td>
<td>0.41s</td>
<td>0.58s</td>
<td>1.14s</td>
<td>12.73s</td>
</tr>
<tr>
<td>Example</td>
<td>1(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>While</td>
<td>2(true)</td>
<td>-</td>
<td>0.49s</td>
<td>-</td>
<td>0.44s</td>
<td>0.43s</td>
<td>1m19.46s</td>
</tr>
<tr>
<td></td>
<td>3(false)</td>
<td>0.43s</td>
<td>-</td>
<td>0.41s</td>
<td>0.48s</td>
<td>0.65s</td>
<td>8.48s</td>
</tr>
<tr>
<td></td>
<td>4(false)</td>
<td>0.44s</td>
<td>-</td>
<td>0.43s</td>
<td>0.51s</td>
<td>0.95s</td>
<td>11.14s</td>
</tr>
</tbody>
</table>

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.

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

int main()
{
    bool a = false;
    bool b = false;
    bool x = false;
    bool y = false;
    bool z = false;

    while(true)
    {
        x = nondet_bool();
        z = nondet_bool();

        if(!x)
        {
            a = false;
            b = false;
        }
        else if(!y)
        {
            b = !b;
        }
        else
        {
            if(z)
            {
                a = true;
            }
            else
            {
                a = false;
            }
        }
        y = x;
        assert(!a || b);
    }
}
```

Listing 5.7: small example

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

int main()
{
    bool a = false;
    bool b = false;
    bool x = false;
    bool y = false;

    while(true)
    {
        x = nondet_bool();

        if(!x)
        {
            a = false;
            b = false;
        }
        else if(!y)
        {
            b = !b;
        }
        else
        {
            a = true;
        }
        y = x;
        assert(!a || b);
    }
}
```

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** $(\neg a_0 \lor b_0) \land (\neg a_1 \lor b_1) \land (\neg a_2 \lor b_2)$
• step case \((\neg a_i \lor b_i) \land (\neg a_{i+1} \lor b_{i+1}) \land (\neg a_{i+2} \lor b_{i+2}) \implies (\neg a_{i+3} \lor b_{i+3})\)

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.

\[
a_{i+1} = \begin{cases} 
false & \text{if } \neg x_{i+1} \\
    a_i & \text{if } x_{i+1} \land \neg y_i \\
    true & \text{if } x_{i+1} \land y_i 
\end{cases}
\]  

\(1\)

\[
b_{i+1} = \begin{cases} 
false & \text{if } \neg x_{i+1} \\
    \neg b_i & \text{if } x_{i+1} \land y_i \\
    b_i & \text{if } x_{i+1} \land \neg y_i 
\end{cases}
\]  

\(2\)

\[
y_{i+1} = x_{i+1}
\]  

\(3\)

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.

\[
a_{i+1} \iff (x_{i+1} \land \neg y_i \land x_i) \lor (x_{i+1} \land y_i)
\]

\[
b_{i+1} \iff (x_{i+1} \land \neg y_i \land \neg b_i) \lor (x_{i+1} \land y_i \land b_i)
\]

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

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

\[
a_1 = \begin{cases} 
false & \text{if } \neg x_1 \\
    false & \text{if } x_1 
\end{cases}
\]  

\(4\)

\[
b_1 = \begin{cases} 
false & \text{if } \neg x_1 \\
    true & \text{if } x_1 
\end{cases}
\]  

\(5\)

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 \((\neg false \lor b_1)\) which is true regardless of the value of \(b_1\).

The third case is \((\neg a_2 \lor b_2)\). 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 \(b_2\).

\[
a_2 = \begin{cases} 
false & \text{if } \neg x_2 \\
    false & \text{if } x_2 \land \neg x_1 \\
    true & \text{if } x_2 \land x_i 
\end{cases}
\]  

\(6\)

\[
b_2 = \begin{cases} 
false & \text{if } \neg x_2 \\
    true & \text{if } x_2 \land \neg x_1 \\
    true & \text{if } x_2 \land x_i 
\end{cases}
\]  

\(7\)

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 \((\neg a_2 \lor b_2)\) 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 $y_{i+3}$.

\[
\begin{align*}
    a_{i+2} &= \begin{cases}
        \text{false} & \text{if } \neg x_{i+2} \\
        \text{false} & \text{if } x_{i+2} \land \neg x_{i+1} \\
        \text{true} & \text{if } x_{i+2} \land x_{i+1}
    \end{cases} \\
    b_{i+2} &= \begin{cases}
        \text{false} & \text{if } \neg x_{i+2} \\
        \text{true} & \text{if } x_{i+2} \land \neg x_{i+1} \\
        \neg b_i & \text{if } x_{i+2} \land x_{i+1} \land \neg y_i \\
        b_i & \text{if } x_{i+2} \land x_{i+1} \land y_i
    \end{cases}
\end{align*}
\]

\[\text{if } x_{i+2} \land x_{i+1} \land y_i \implies y_{i+2} = x_{i+2}\] (10)

\[
\begin{align*}
    a_{i+3} &= \begin{cases}
        \text{false} & \text{if } \neg x_{i+3} \\
        \text{false} & \text{if } x_{i+3} \land \neg x_{i+2} \\
        \text{true} & \text{if } x_{i+3} \land x_{i+2}
    \end{cases} \\
    b_{i+3} &= \begin{cases}
        \text{false} & \text{if } \neg x_{i+3} \\
        \text{true} & \text{if } x_{i+3} \land \neg x_{i+2} \\
        \neg b_i & \text{if } x_{i+3} \land x_{i+2} \land \neg x_{i+1} \\
        b_i & \text{if } x_{i+3} \land x_{i+2} \land x_{i+1} \land y_i
    \end{cases}
\end{align*}
\]

\[\text{if } x_{i+3} \land x_{i+2} \land x_{i+1} \land y_i \implies y_{i+3} = x_{i+3}\] (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.

\[
a_{i+2} \iff (x_{i+2} \land x_{i+1})
\]

\[
b_{i+2} \iff (x_{i+2} \land \neg x_{i+1}) \lor (x_{i+2} \land x_{i+1} \land \neg y_i \land \neg b_i) \lor (x_{i+2} \land x_{i+1} \land y_i \land b_i)
\]

\[
a_{i+3} \iff (x_{i+3} \land x_{i+2})
\]

\[
b_{i+3} \iff (x_{i+3} \land \neg x_{i+2}) \lor (x_{i+3} \land x_{i+2} \land \neg x_{i+1}) \lor (x_{i+3} \land x_{i+2} \land x_{i+1} \land \neg y_i \land \neg b_i) \lor (x_{i+3} \land x_{i+2} \land x_{i+1} \land y_i \land b_i)
\]

With these formulas and the formulas for $a_{i+1}$ and $b_{i+1}$ we can rewrite $(-a_{i+2} \lor b_{i+2})$.

\[
(-a_{i+2} \lor b_{i+2}) \iff (\neg (x_{i+2} \land x_{i+1}) \lor ((x_{i+2} \land \neg x_{i+1}) \lor (x_{i+2} \land x_{i+1} \land \neg y_i \land \neg b_i) \lor (x_{i+2} \land x_{i+1} \land y_i \land b_i))) \iff (\neg x_{i+2} \lor \neg x_{i+1} \lor \neg x_{i+1} \lor (\neg y_i \land \neg b_i) \lor (y_i \land b_i))
\]

Likewise for $(-a_{i+3} \lor b_{i+3})$.

\[
(-a_{i+3} \lor b_{i+3}) \iff (\neg (x_{i+3} \land x_{i+2}) \lor ((x_{i+3} \land \neg x_{i+2}) \lor (x_{i+3} \land x_{i+2} \land \neg x_{i+1}) \lor (x_{i+3} \land x_{i+2} \land x_{i+1} \land \neg y_i \land \neg b_i) \lor (x_{i+3} \land x_{i+2} \land x_{i+1} \land y_i \land b_i))) \iff (\neg x_{i+3} \land (x_{i+2} \lor (\neg x_{i+1} \lor (\neg x_{i+1} \lor (\neg y_i \land \neg b_i) \lor (y_i \land b_i))))$
\((\neg x_{i+3} \lor \neg x_{i+2} \lor \neg x_{i+1} \lor (\neg y_i \land \neg b_i) \lor (y_i \land b_i))\)

With these formulas we can see that \((\neg a_{i+3} \lor b_{i+3}) \iff (\neg a_{i+3} \lor (\neg a_{i+2} \lor b_{i+2})).\) From this it follows that \((\neg a_{i+2} \lor b_{i+2}) \implies (\neg a_{i+3} \lor b_{i+3}),\) which also means that \((\neg a_i \lor b_i) \land (\neg a_{i+1} \lor b_{i+1}) \land (\neg a_{i+2} \lor b_{i+2}) \implies (\neg a_{i+3} \lor b_{i+3}).\) 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 \texttt{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 \(kI\) algorithm. We have strengthened the property to \(x \geq 0 \land x \neq 3.\) 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 kI&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:

\texttt{Manreg01: \text{WORD};
Manreg01b AT Manreg01: \text{ARRAY [0..15] OF BOOL};
}

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

Variables of type \text{TIME} can get values of the form \text{T#ah bm cs 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 \text{T#0ms} stands for zero milliseconds.

The program consists of 822 lines of code, which includes the main \text{FUNCTION BLOCK}, two other \text{FUNCTION BLOCKs}, three \text{FUNCTIONs}, and one user defined structure. There are 32 function calls of which 21 call a \text{FUNCTION} and 11 call a \text{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 \text{STRUCT} or in a \text{FUNCTION BLOCK} are included in these numbers.

Besides the main \text{FUNCTION BLOCK} the CPC program has two additional \text{FUNCTION BLOCKs}. Since we have only seen the \text{FUNCTION BLOCK} as the main \text{FUNCTION BLOCK}, we will explain this construct. A \text{FUNCTION BLOCK} is similar to a \text{FUNCTION}. In addition variables in a \text{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 \text{FUNCTION BLOCK}.

```c
1 FUNCTION BLOCK TP
2 VAR INPUT
3 PT : TIME;
4 END_VAR
5 VAR_IN_OUT
6 IN : BOOL;
7 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

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 \texttt{WORD} and a variable of type \texttt{ARRAY} in SCL, as seen earlier in this section, can be translated to a variable of type \texttt{array} in SMV. To make an array of 16 boolean variables \texttt{array 0..15 of boolean} is used. Variables of the type \texttt{TIME} and type \texttt{REAL} in SCL are represented by variables of the type \texttt{signed word[32]} in SMV and the variable of the type \texttt{UINT} in SCL is represented by a variable of the type \texttt{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 \texttt{word} in SMV, we have used the standard SMV operator \texttt{extend(variable, size)} to scale the variables. Here \texttt{variable} is the variable we want to scale and \texttt{size} is the size with which we want to extend this variable. For instance if we want to know if the variable \texttt{FSIinc} of type \texttt{signed word[16]} is larger than a variable \texttt{PulseWidth} of type \texttt{singed word[32]} we will get the following piece of code: \texttt{extend(FSIinc,16) > PulseWidth}.

To model the timer, we give the variable \texttt{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 \texttt{random\_t\_cycle} with type \texttt{unsigned word[8]} which we scale to be inside the range. For the next state of \texttt{T\_CYCLE} we get the following case in the case distinction:

\[
\text{(loc = start)} : ((\text{extend}((\text{random\_t\_cycle}),8)) \mod 0u16.95 + 0u16.5);
\]

To model a \texttt{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 \texttt{FUNCTION BLOCK} as type. For the \texttt{FUNCTION BLOCK} in Listing 6.1 this gives us the following Struct:

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

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

```c
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 start of the current cycle and an ’p’ as a prefix for the end of the previous 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 kIHiI 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.
<table>
<thead>
<tr>
<th>Property</th>
<th>2LS</th>
<th>CBMC</th>
<th>CPA-checker</th>
<th>SATABS</th>
<th>NuSMV</th>
<th>nuXmv</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>k-induction</td>
<td>kIลำ</td>
<td>Incremental</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1-1(false)</td>
<td>2.06s</td>
<td>15.49s</td>
<td>1.01s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-2(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-3(false)</td>
<td>2.12s</td>
<td>13.62s</td>
<td>0.99s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-4(false)</td>
<td>2.07s</td>
<td>14.04s</td>
<td>1.02s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-5(false)</td>
<td>1.74s</td>
<td>14.13s</td>
<td>1.02s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-6(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-7(true)</td>
<td>1.17s</td>
<td>1.15s</td>
<td>0.70s</td>
<td>19.90s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-8(true)</td>
<td>1.12s</td>
<td>1.26s</td>
<td>0.73s</td>
<td>18.83s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-9(true)</td>
<td>1.17s</td>
<td>1.18s</td>
<td>0.71s</td>
<td>20.60s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-11a(false)</td>
<td>1.55s</td>
<td>6.56s</td>
<td>0.83s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-11b(false)</td>
<td>1.58s</td>
<td>5.76s</td>
<td>0.86s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-1(true)</td>
<td>1.12s</td>
<td>1.22s</td>
<td>0.67s</td>
<td>19.56s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-2(false)</td>
<td>1.10s</td>
<td>1.17s</td>
<td>0.71s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-3(true)</td>
<td>1.18s</td>
<td>1.20s</td>
<td>0.68s</td>
<td>20.87s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-4(true)</td>
<td>0.97s</td>
<td>0.96s</td>
<td>0.69s</td>
<td>20.28s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-5(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-6(true)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-7(false)</td>
<td>1.23s</td>
<td>1.24s</td>
<td>0.69s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-8(true)</td>
<td>1.20s</td>
<td>1.20s</td>
<td>0.67s</td>
<td>19.90s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-1(true)</td>
<td>1.00s</td>
<td>1.19s</td>
<td>0.73s</td>
<td>30.05s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-2(false)</td>
<td>1.00s</td>
<td>1.07s</td>
<td>0.64s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-3(true)</td>
<td>1.00s</td>
<td>1.28s</td>
<td>0.72s</td>
<td>21.27s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-4(false)</td>
<td>1.01s</td>
<td>1.13s</td>
<td>0.63s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-5(false)</td>
<td>2.04s</td>
<td>13.70s</td>
<td>1.09s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-6(true)</td>
<td>1.12s</td>
<td>1.21s</td>
<td>0.74s</td>
<td>33.06s</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-7(false)</td>
<td>2.03s</td>
<td>14.65s</td>
<td>0.73s</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4-1(true)</td>
<td>1.21s</td>
<td>1.19s</td>
<td>0.68s</td>
<td>21.27s</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

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.

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

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 \( kI/kI \) 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 \( kI/kI \) 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


Appendices

A  Example programs in SCL

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

```plaintext
// 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 model
  toMode2 : BOOL; // request to switch to model2
  toMode3 : BOOL; // request to switch to model3
  mode3Forbidden : BOOL; // if it is true, it is forbidden to be in mode3
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 model1
  mode2 : BOOL; // true if the block is in model2
  mode3 : BOOL; // true if the block is in model3
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;
    out1.out1 := out1;
  ELSE
    cntr := cntr + 1;
    IF cntr > cntr_max AND signal THEN
      out2 := TRUE;
    ELSE
      out2 := FALSE;
    END_IF;
    out1.out1 := out1;
    out3.out1 := out1;
    out3.out2 := out2;
    out3.remaining := cntr_max - cntr;
    out3.elapsed := cntr;
  END_IF;

  (* -------------------------------- *)
  IF not mode1 and not mode2 and not mode3 THEN
    model := TRUE;
  END_IF;
  IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
    model := TRUE;
  END_IF;
  IF toMode2 THEN
    mode2 := TRUE;
```

61
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;

FUNCTION BLOCK

BEGIN
  mode := -1; // default value for the mode variable in the data block
END_FUNCTION_BLOCK

// Global data storage
DATA_BLOCK ModeDB
STRUCT
  mode : INT;
END_STRUCT

// Helper function to determine the rising edge on a signal.
FUNCTION R_EDGE : BOOL
VAR
  new : BOOL;
END_VAR
VAR
  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)
  toModel1 : BOOL; // request to switch to model1
toModel2 : BOOL; // request to switch to model2
toModel3 : BOOL; // request to switch to model3
mode3Forbidden : BOOL; // if it is true, it is forbidden to be in mode3
END_VAR
VAR
  signalOld : BOOL := FALSE; // signal value from the last cycle
cntr : INT := 0; // counter to delay the out2 signal
cntr2 : INT := 0;
model1 : BOOL; // true if the block is in model1
model2 : BOOL; // true if the block is in model2
model3 : BOOL; // true if the block is in model3
edgeSignal : BOOL; // rising edge of out1
END_VAR
VAR_TEMP
  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
VAR_OUTPUT
END_VAR
BEGIN

Listing A.2: Example int.SCL

44 (* Signal handling *)
45 edge_signal := R_EDGE(new := signal, old := signal_old);
46
47 IF NOT signal THEN
48 // outputs are false if the signal is false
49 out1 := FALSE;
50 out2 := FALSE;
51 cntr := 0;
52 ELSE
53 // if the signal has a rising edge, out1 should be true
54 out1 := NOT out1;
55 out3.out1 := out1;
56 ELSE
57 cntr := cntr + 1;
58 // ------ ADDITIONAL PART
59 IF cntr > cntr_max THEN
60 cntr := 0;
61 cntr2 := cntr2 + 1;
62 END_IF;
63 // ------
64 IF cntr2 > cntr_max AND signal THEN
65 out2 := TRUE;
66 ELSE
67 out2 := FALSE;
68 END_IF;
69 out3.out1 := out1;
70 out3.out2 := out2;
71 out3.remaining := cntr_max - cntr;
72 out3.elapsed := cntr;
73 END_IF;
74
75 (* ----------------------------------------------- *)
76
77 (* Operation mode handling *)
78 IF not mode1 and not mode2 and not mode3 THEN
79 mode1 := TRUE;
80 END_IF;
81
82 IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
83 mode1 := TRUE;
84 END_IF;
85 IF toMode2 THEN
86 mode2 := TRUE;
87 ELSIF toMode3 THEN
88 mode3 := TRUE;
89 ELSE
90 ModeDB.mode := 0;
91 END_IF;
92 END_FUNCTION BLOCK
93
94 // Global data storage
95 DATA_BLOCK ModeDB
96 STRUCT
97 mode : INT;
98 END_STRUCT
99 BEGIN
100 mode := -1; // default value for the mode variable in the data block
101 END_DATA_BLOCK
102
103 // Helper function to determine the rising edge on a signal.
104 FUNCTION R_EDGE : BOOL
105 VAR_INPUT
106 new : BOOL;
107 VAR_IN_OUT
108 old : BOOL;
109 END_VAR
110 BEGIN
111 IF (new = true AND old = false) THEN
112 R_EDGE := true;
113 ELSE R_EDGE := false;
114 END_IF;
115 END_FUNCTION
// 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 model
  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_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;
  END_IF;

  IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
    mode1 := TRUE;
  END_IF;

  IF toMode2 THEN
    mode2 := TRUE;
  ELSEIF toMode3 THEN
    mode3 := TRUE;
  END_IF;

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

  IF not cntr2 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
### 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.

```c
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 cntr = 0;
int cntr_max = 5;
short cntr = 0;
ComplexSignal out3;

active proctype go()
{
  do
    :if
      :error = 0
    :else
      :error = 1
    :fi;
  :if
    :toMode1 = 0
  :else
    :toMode1 = 1
  :fi;
  :if
    :toMode2 = 0
  :else
    :toMode2 = 1
  :fi;
  :if
    :toModel3 = 0
  :else
    :toModel3 = 1
  :fi;
  END_IF;
END_FUNCTION
```
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
  else
    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
  if
    !signal &= !signal_old -> 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
  if
    !mode1 & !mode2 & !mode3 -> mode = 1
  else
    : : else
  fi;
if
  if
    toMode1 || (toMode3 & mode3Forbidden) -> mode = 1
  : : else
  fi;
if
  if
    toMode2 -> mode = 2
    : : else
  fi;
if
  if
    !mode1 & !mode2 & mode3 -> mode = 3
  : : else -> mode = 0
  fi;
assert (!out2 || out1)
assert (signal || !out2)
assert (out3.out1 == out1)
assert (out1 || (out3.elapsed == 0))
}
```c
typedef ComplexSignal {
  bool out1;
  bool out2;
  short remaining;
  short elapsed;
};

typedef ComplexSignal {
  bool error, toMode1, toMode2, toMode3, mode3Forbidden;
  bool model = 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;
  int cntr = 0;
  int 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;
    :: else ->
      edge_signal = 0;
    :: signal = signal_old;
  fi;
  if
    :: !signal ->
      out1 = 0;
      out2 = 0;
      cntr = 0
    :: signal && edge_signal ->
      out1 = !out1;
      out3.out1 = out1
      out3.elapsed = cntr
    fi;
  else
    :: cntr > cntr_max ->
      cntr = 0;
      cntr2 = cntr2 + 1
    :: else
```
```plaintext
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
    ::
        signal = 0
    ::
        signal = 1
    fi;

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

    if
    :: !signal -> out1 = 0;
    :: out2 = 0;
    :: cntr = 0;
    :: signal && edge_signal -> out1 = !out1;
    :: out3.out1 = out1
    :: else ->
        cntr = cntr + 1;
    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
    :: model -> mode = 1
    :: !model && !mode2 && !mode3 -> 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))
}
```

Listing B.3: Example_int.pml
assert (out1 || (out3.elapsed == 0))
}

Listing B.4: Reduced version of Example_int.pml

typedef ComplexSignal{
  bool out1,
  bool out2,
  short remaining;
}

bool error, toModel1, toModel2, toModel3, 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
      :: toModel1 = 0
    :: toModel1 = 1
    fi;
    :: if
      :: toModel2 = 0
    :: toModel2 = 1
    fi;
    :: if
      :: toModel3 = 0
    :: toModel3 = 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 || signal_old -> edge_signal = 0;
    :: signal_old = signal;
    fi;
  
  
  :: if
    :: !signal -> out1 = 0;
    :: out2 = 0;
    :: cntr = 0
  
  :: signal & edge_signal -> out1 = !out1;
  :: else -> cntr = cntr + 1;
  :: d_step(if
    :: cntr > cntr_max -> cntr = cntr - 1; cntr2 = cntr2 + 1
    :: else -> break
    od;
  
  :: if
    :: (cntr2 > cntr_max) && signal -> out2 = 1
    :: else -> out2 = 0
    fi;
  
  :: if
    :: !mode1 && !mode2 && !mode3 -> model = 1
    :: else
    fi;
  
  :: if
    :: toModel1 || (toModel3 && mode3Forbidden) -> model = 1
  }
Listing B.5: Example while.pml

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.

```plaintext
1 MODULE main
2 VAR
3 signal : boolean;
4 error : boolean;
5 toModel : boolean;
6 toMode2 : boolean;
7 toMode3 : boolean;
8 modeForbidden : boolean;
9 signal_old : boolean;
10 edge_signal : boolean;
11 cntr : signed word[16];
12 out1 : boolean;
13 out2 : boolean;
14 out3.out1 : boolean;
15 out3.out2 : boolean;
16 out3.remaining : signed word[16];
17 out3.elapsed : signed word[16];
18 model : boolean;
19 mode2 : boolean;
20 mode3 : boolean;
21 mode : 1..3;

24 ASSIGN
25 init(signal_old) := FALSE;
26 init(edge_signal) := FALSE;
27 init(cntr) := 0x16,0;
28 init(out1) := FALSE;
29 init(out2) := FALSE;
30 init(out3.out1) := FALSE;
31 init(out3.out2) := FALSE;
32 init(out3.remaining) := 0x16,0;
33 init(out3.elapsed) := 0x16,0;
34 init(model) := FALSE;
35 init(mode2) := FALSE;
36 init(mode3) := FALSE;
37 init(mode) := 1;
38 init(loc) := start;

40 next(loc) :=
41 case
42 (loc = start) : step1;
43 (loc = step1) & (signal & !signal_old) : step2;
44 (loc = step1) : step3;
45 (loc = step2) : step4;
46 (loc = step3) : step4;
47 (loc = step4) & (edge_signal) : step5;
48 (loc = step4 & (signal)) & (edge_signal) : step6;
49 (loc = step4) : step7;
50 (loc = step5) : step8;
51 (loc = step6) : step11;
52 (loc = step7) & (cntr > 0x16,5) & (signal) : step8;
53 (loc = step7) : step9;
54 (loc = step8) : step10;
55 (loc = step9) : step10;
56 (loc = step10) : step11;
57 (loc = step11) & (!model & !mode2 & !mode3) : step12;
58 (loc = step11) : step13;
59 (loc = step12) : step13;
60 (loc = step13) & (toMode1) | (toMode3 & modeForbidden) : step14;
61 (loc = step13) : step15;
62 (loc = step14) : step15;
63 (loc = step15) & (toMode2) : step16;
64 (loc = step15) & (!toMode2 & (toMode3)) : step17;
65 (loc = step15) : step18;
66 (loc = step16) : step18;
67 (loc = step17) : step18;
68 (loc = step18) & (model) : step19;
69 (loc = step18) & (!model) & (mode2) : step20;
70 (loc = step18) & (!model) & (!mode2) & (mode3) : step21;
71 (loc = step18) : step22;
72 (loc = step19) : end;
73 (loc = step20) : end;
74 (loc = step21) : end;
75 (loc = step22) : end;
76 (loc = end) : start;
77 esac;
78 next(signal) :=
79 case
(loc = start) : {TRUE, FALSE};
TRUE : signal;
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(model) :=
case
  (loc = step12) : TRUE;
  (loc = step14) : TRUE;
  TRUE : model1;
  esac;
next(mode2) :=
case
  (loc = step16) : TRUE;
TRUE : mode 2;

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
(loc = step3) : FALSE;
TRUE : edgesignal;
esac;
next(cnt) :=
\texttt{case}
(\texttt{loc = step5}) : 0sd16\_0;
(\texttt{loc = step7}) : \texttt{cnt} + 0sd16\_1;
TRUE : cnt;
esac;
next(out) :=
\texttt{case}
(\texttt{loc = step5}) : FALSE;
(\texttt{loc = step6}) : \texttt{!out1};
TRUE : out;
esac;
next(out1) :=
\texttt{case}
(\texttt{loc = step5}) : FALSE;
(\texttt{loc = step6}) : \texttt{out1};
TRUE : \texttt{out};
esac;
next(out2) :=
\texttt{case}
(\texttt{loc = step5}) : FALSE;
(\texttt{loc = step10}) : \texttt{out};
TRUE : out2;
esac;
next(out3.out1) :=
\texttt{case}
(\texttt{loc = step5}) : FALSE;
(\texttt{loc = step8}) : \texttt{TRUE};
(\texttt{loc = step9}) : FALSE;
TRUE : out3.out1;
esac;
next(out3.elapsed) :=
\texttt{case}
(\texttt{loc = step5}) : FALSE;
(\texttt{loc = step10}) : \texttt{out};
TRUE : out3.elapsed;
esac;
\texttt{--CTL properties}
SPEC AG(\texttt{loc = end} \rightarrow (\texttt{out2} \rightarrow \texttt{out1}))
SPEC AG(\texttt{loc = end} \rightarrow (\texttt{signal} \rightarrow \texttt{!out2}))
SPEC AG(\texttt{loc = end} \rightarrow (\texttt{out3.out1 = out1}))
SPEC AG(\texttt{loc = end} \rightarrow (\texttt{out1} \rightarrow \texttt{out3.elapsed = 0sd16\_0}))
\texttt{--INVARIANT properties}
INVARSPEC(\texttt{loc = end} \rightarrow (\texttt{out2} \rightarrow \texttt{out1}))
INVARSPEC(\texttt{loc = end} \rightarrow (\texttt{signal} \rightarrow \texttt{!out2}))
INVARSPEC(\texttt{loc = end} \rightarrow (\texttt{out3.out1 = out1}))
INVARSPEC(\texttt{loc = end} \rightarrow (\texttt{out1} \rightarrow \texttt{out3.elapsed = 0sd16\_0}))
init (mode) := -1;
init (loc) := start;

next (loc) :=
case
  (loc = start) : step 1;
  (loc = step 1) & (signal & !signal_old) : step 2;
  (loc = step 1) : step 3;
  (loc = step 2) : step 4;
  (loc = step 3) : step 4;
  (loc = step 4) & (!signal) : step 5;
  (loc = step 4) & (signal) & (edge_signal) : step 6;
  (loc = step 4) : step 7;
  (loc = step 5) : step 8;
  (loc = step 6) : step 9;
  (loc = step 7) & (cnt > 0sd16_5) : step 8;
  (loc = step 7) : step 10;
  (loc = step 8) : step 11;
  (loc = step 9) & (!mode1 & !mode2 & !mode3) : step 11;
  (loc = step 9) : step 12;
  (loc = step 10) : step 12;
  (loc = step 11) : step 12;
  (loc = step 12) : step 13;
  (loc = step 13) & (edge_signal) : step 15;
  (loc = step 13) : step 15;
  (loc = step 14) : step 15;
  (loc = step 15) : step 15;
  (loc = step 16) : step 15;
  (loc = step 17) : step 15;
  (loc = step 18) : step 15;
  (loc = step 19) : step 15;
  (loc = step 20) : step 15;
  (loc = step 21) : step 15;
  (loc = step 22) : step 15;
  (loc = step 23) : step 15;
  (loc = step 24) : step 15;
  (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 = step 2) : TRUE;
  (loc = step 3) : signal;
  TRUE : signal_old;

esac;

next (edge_signal) :=
case
  (loc = step 2) : TRUE;
  (loc = step 3) : FALSE;
  TRUE : edge_signal;

esac;

next (cnt) :=
case
Listing C.3: Example_int.smv

```plaintext
MODULE main
VAR
  signal : boolean;
  signal_old : boolean;
  edge,signal : boolean;
```
Master's Thesis

ASSIGN

\[\begin{align*}
\text{cntr} & : \text{signed word}[16]; \\
\text{cntr2} & : \text{signed word}[16]; \\
\text{out} & : \text{boolean}; \\
\text{out2} & : \text{boolean}; \\
\text{out3.out} & : \text{boolean}; \\
\text{out3.elapsed} & : \text{signed word}[16];
\end{align*}\]

\[\begin{align*}
\text{loc} & : \{\text{start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10, step11, step12, end}\};
\end{align*}\]

\[\begin{align*}
\text{init}(\text{signal.old}) & := \text{FALSE}; \\
\text{init}(\text{edge.signal}) & := \text{FALSE}; \\
\text{init}(\text{cntr}) & := \text{0sd16}_0; \\
\text{init}(\text{cntr2}) & := \text{0sd16}_0; \\
\text{init}(\text{out}) & := \text{FALSE}; \\
\text{init}(\text{out2}) & := \text{FALSE}; \\
\text{init}(\text{out3.elapsed}) & := \text{0sd16}_0; \\
\text{init}(\text{loc}) & := \text{start};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{loc}) & := \\
\text{case} \\
(\text{loc} = \text{start}) & : \text{step1}; \\
(\text{loc} = \text{step1}) & : \{\text{signal} \& \text{!signal.old}\} : \text{step2}; \\
(\text{loc} = \text{step1}) & : \text{step3}; \\
(\text{loc} = \text{step2}) & : \text{step4}; \\
(\text{loc} = \text{step3}) & : \text{step4}; \\
(\text{loc} = \text{step4}) & : \text{!signal} : \text{step5}; \\
(\text{loc} = \text{step4}) & : \{\text{signal} \& \text{edge.signal}\} : \text{step6}; \\
(\text{loc} = \text{step4}) & : \text{step7}; \\
(\text{loc} = \text{step5}) & : \text{end}; \\
(\text{loc} = \text{step6}) & : \text{end}; \\
(\text{loc} = \text{step7}) & : \{\text{cntr} > \text{0sd16}_5\} : \text{step8}; \\
(\text{loc} = \text{step7}) & : \text{step9}; \\
(\text{loc} = \text{step8}) & : \text{step9}; \\
(\text{loc} = \text{step9}) & : \{\text{cntr2} > \text{0sd16}_5\} \& \{\text{signal}\} : \text{step10}; \\
(\text{loc} = \text{step9}) & : \text{step11}; \\
(\text{loc} = \text{step10}) & : \text{step12}; \\
(\text{loc} = \text{step11}) & : \text{step12}; \\
(\text{loc} = \text{step12}) & : \text{end}; \\
(\text{loc} = \text{end}) & : \text{start};
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{signal}) & := \\
\text{case} \\
(\text{loc} = \text{start}) & : \{\text{TRUE, FALSE}\}; \\
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{signal.old}) & := \\
\text{case} \\
(\text{loc} = \text{step2}) & : \text{TRUE}; \\
(\text{loc} = \text{step3}) & : \text{signal}; \\
\text{TRUE} & : \text{signal.old};
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{edge.signal}) & := \\
\text{case} \\
(\text{loc} = \text{step2}) & : \text{TRUE}; \\
(\text{loc} = \text{step3}) & : \text{FALSE}; \\
\text{TRUE} & : \text{edge.signal};
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{cntr}) & := \\
\text{case} \\
(\text{loc} = \text{step5}) & : \text{0sd16}_0; \\
(\text{loc} = \text{step7}) & : \text{cntr} + \text{0sd16}_1; \\
(\text{loc} = \text{step8}) & : \text{0sd16}_0; \\
\text{TRUE} & : \text{cntr};
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{cntr2}) & := \\
\text{case} \\
(\text{loc} = \text{step8}) & : \text{cntr2} + \text{0sd16}_1; \\
\text{TRUE} & : \text{cntr2};
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{out}) & := \\
\text{case} \\
(\text{loc} = \text{step5}) & : \text{FALSE}; \\
(\text{loc} = \text{step6}) & : \text{!out1}; \\
\text{TRUE} & : \text{out1};
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{out2}) & := \\
\text{case} \\
(\text{loc} = \text{step5}) & : \text{FALSE}; \\
(\text{loc} = \text{step10}) & : \text{TRUE}; \\
(\text{loc} = \text{step11}) & : \text{FALSE}; \\
\text{TRUE} & : \text{out2};
\text{esac};
\end{align*}\]

\[\begin{align*}
\text{next}(\text{out3.out}) & := \\
\text{case} \\
(\text{loc} = \text{step6}) & : \text{!out1}; \\
(\text{loc} = \text{step12}) & : \text{out1};
\end{align*}\]
TRUE : out3.out1;

\[\text{next(out3.elapsed)} := \]

\[\begin{cases}
\text{(loc = step12) : cntr} ; \\
\text{TRUE : out3.elapsed} ;
\end{cases}\]

---CTL properties

\[\text{SPEC AG(loc = end \rightarrow (out2 \rightarrow out1))} \]

\[\text{SPEC AG(loc = end \rightarrow (!signal \rightarrow !out2))} \]

\[\text{SPEC AG(loc = end \rightarrow (out3.out1 = out1))} \]

\[\text{SPEC AG(loc = end \rightarrow (!out1 \rightarrow out3.elapsed = 0 sd 16_0))} \]

---INVARIANT properties

\[\text{INVARSPEC(loc = end \rightarrow (out2 \rightarrow out1))} \]

\[\text{INVARSPEC(loc = end \rightarrow (!signal \rightarrow !out2))} \]

\[\text{INVARSPEC(loc = end \rightarrow (out3.out1 = out1))} \]

\[\text{INVARSPEC(loc = end \rightarrow (!out1 \rightarrow out3.elapsed = 0 sd 16_0))} \]

---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\};

ASSIGN

\[\text{init(signal_old)} := \text{FALSE} ; \]
\[\text{init(edge_signal)} := \text{FALSE} ; \]
\[\text{init(cntr)} := \text{0 sd 16}_0 ; \]
\[\text{init(cntr2)} := \text{0 sd 16}_0 ; \]
\[\text{init(out1)} := \text{FALSE} ; \]
\[\text{init(out2)} := \text{FALSE} ; \]
\[\text{init(out3.out1)} := \text{FALSE} ; \]
\[\text{init(out3.remaining)} := \text{0 sd 16}_0 ; \]
\[\text{init(out3.elapsed)} := \text{0 sd 16}_0 ; \]
\[\text{init(mode1)} := \text{FALSE} ; \]
\[\text{init(mode2)} := \text{FALSE} ; \]
\[\text{init(mode3)} := \text{FALSE} ; \]
\[\text{init(mode)} := \text{-1} ; \]
\[\text{init(loc)} := \text{start} ; \]

\[\text{next(loc)} := \]

\[\begin{cases}
\text{(loc = start) : step1} ; \\
\text{(loc = step1) & (signal & !signal_old) : step2} ; \\
\text{(loc = step1) : step3} ; \\
\text{(loc = step2) : step4} ; \\
\text{(loc = step3) : step4} ; \\
\text{(loc = step4) & (!signal) : step5} ; \\
\text{(loc = step4) & (signal) & (edge_signal) : step6} ; \\
\text{(loc = step4) : step7} ; \\
\text{(loc = step5) : step14} ; \\
\text{(loc = step6) : step14} ; \\
\text{(loc = step7) : step8} ; \\
\text{(loc = step8) & (cntr > 0 sd 16_5) : step9} ; \\
\text{(loc = step8) : step10} ; \\
\text{(loc = step9) : step8} ; \\
\text{(loc = step10) & (cntr2 > 0 sd 16_5) & (signal) : step11} ; \\
\text{(loc = step10) : step12} ; \\
\text{(loc = step11) : step13} ; \\
\text{(loc = step12) : step13} ; \\
\text{(loc = step13) : step14} ; \\
\text{(loc = step14) & (!model1 & !mode2 & !mode3) : step15} ; \\
\text{(loc = step14) : step16} ;
\end{cases}\]
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 = step 6) : !out 1;
(loc = step 13) : out 1;
TRUE : out 3 . out 1;

next (out 3 . out 2) :=
case
(loc = step 13) : out 2;
TRUE : out 3 . out 2;

next (out 3 . remaining) :=
case
(loc = step 13) : 0sd16_5 = cntr;
TRUE : out 3 . remaining;

next (out 3 . elapsed) :=
case
(loc = step 13) : 0sd16_0;
TRUE : out 3 . elapsed;

next (mode) :=
case
(loc = step 22) : 1;
(loc = step 23) : 2;
(loc = step 24) : 3;
(loc = step 25) : 0;
TRUE : mode;

--- CTL properties
SPEC AG( loc = end -> (out 2 -> out 1))
SPEC AG( loc = end -> (!signal -> !out 2))
SPEC AG( loc = end -> (out 3 . out 1 = out 1))
SPEC AG( loc = end -> (out 1 -> out 3 . elapsed = 0sd16_0))

--- INVARIANT properties
INVARSPEC( loc = end -> (out 2 -> out 1))
INVARSPEC( loc = end -> (!signal -> !out 2))
INVARSPEC( loc = end -> (out 3 . out 1 = out 1))
INVARSPEC( loc = end -> (out 1 -> out 3 . elapsed = 0sd16_0))

Listing C.5: Example_while.smv
30 (loc = step2) : step4;
31 (loc = step3) : step4;
32 (loc = step4) & (!signal) : step5;
33 (loc = step4) & (signal) & (edge_signal) : step6;
34 (loc = step4) : step7;
35 (loc = step5) : end;
36 (loc = step6) : step8;
37 (loc = step7) : step8;
38 (loc = step8) & (cnt > 0sd16_5) : step9;
39 (loc = step8) : step10;
40 (loc = step9) : step8;
41 (loc = step10) & (cnt2 > 0sd16_5) & (signal) : step11;
42 (loc = step10) : step12;
43 (loc = step11) : step13;
44 (loc = step12) : step13;
45 (loc = step13) : end;
46 (loc = end) : start;

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(cnt) :=
  case
    (loc = step5) : 0sd16_0;
    (loc = step7) : cnt + 0sd16_1;
    (loc = step9) : cnt - 0sd16_1;
    TRUE : cnt;
  esac;

next(cnt2) :=
  case
    (loc = step9) : cnt2 + 0sd16_1;
    TRUE : cnt2;
  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) : cnt;
    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();
        toModel = 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;
        } else if (toModel | (toMode3 & mode3Forbidden)){
            mode1 = true;
        }
        if(toMode2){
            mode2 = true;
        } else if(toMode3){
            mode3 = true;
        }
        if(model){
            mode = 1;
        } else if(mode2){
            mode = 2;
        } else if(mode3){
            mode = 3;
        }
    }
}
```
83     } else{
84         mode = 0;
85     }
86
87     assert(!out2 || out1);
88     assert(signal || !out2);
89     assert(out3.out1 == out1);
90     assert(out1 || (out3.elapsed == 0));
91     }
92 }

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){
                out3 = 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;
    }
}
```c
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 model = false;
    bool mode2 = false;
    bool mode3 = false;
    short mode = 0;
    short cnt = 0;
    short cnt2 = 0;
    short cnt_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 = edge(signal, &signal_old);
        if (!signal) {
            out1 = false;
            out2 = false;
            cnt = 0;
        } else if (edge_signal) {
            out1 = !out1;
            out3.out1 = out1;
        } else {
            cnt = cnt + 1;
            if (cnt > cnt_max) {
                cnt = 0;
                cnt2 = cnt2 + 1;
            }
            if (cnt2 > cnt_max && signal) {
                out2 = true;
            } else {
                out2 = false;
            }
            out3.out1 = out1;
            out3.out2 = out2;
            out3.remaining = cnt_max - cnt;
            out3.elapsed = cnt;
        }
        if (!model && !mode2 && !mode3) {
            model = true;
        } else if (toMode1) {
            mode1 = true;
        } else if (toMode2) {
            mode2 = true;
        } else if (toMode3) {
            mode3 = true;
        } else {
            mode = 1;
        }
        assert (!out2 || out1);
        assert (signal || !out2);
        assert (out3.out1 == out1);
        assert (out1 || (out3.elapsed == 0));
    }
}
```

Listing D.3: Example_int.c
#include <stdbool.h>

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){
                cntr = 0;
            } 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
bool edge_signal = false;
bool mode1 = false;
bool mode2 = false;
bool mode3 = false;
short mode = 0;
short cnt2 = 0;
short cnt_max = 5;
bool out1 = false;
bool out2 = false;
bool out3 = false;

struct ComplexSignal out3;
out3.out1 = false;
out3.out2 = false;
out3.elapsed = 0;
out3.remaining = 0;

while(true) {
    error = nondet_bool();
toModel = nondet_bool();
toMode2 = nondet_bool();
toMode3 = nondet_bool();
mode3Forbidden = nondet_bool();
signal = nondet_bool();
    if(!signal){
        out1 = false;
        out2 = false;
cnt = 0;
    } else if(edge_signal){
        out1 = !out1;
out3.out1 = out1;
    } else{
        cnt = cnt + 1;
        while((cnt > cnt_max){
            cnt = cnt - 1;
cnt2 = cnt2 + 1;
        } if((cnt2 > cnt_max && signal){
            out2 = true;
        } else{
            out2 = false;
        }
out3.out1 = out1;
out3.out2 = out2;
out3.remaining = out3.max - cnt;
out3.elapsed = cnt;
    } if ((!mode1 && !mode2 && !mode3){
        model = true;
    } if(toModel { (toMode3 && !mode3Forbidden)){
        model = true;
    } if(toMode2){
        mode2 = true;
    } else if(toModel){
        mode3 = true;
    } if(model){
        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;
Listing D.6: Reduced version of Example\_while.smv

E  

CPC program in SCL

```c
def main()
{
    struct ComplexSignal{
        bool out1;
        short elapsed;
    };
    bool signal;
    bool signal\_old = false;
    bool edge\_signal = false;
    short cntr = 0;
    short cntr\_2 = 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();
        if (!signal)
        {
            out1 = false;
            out2 = false;
            cntr = 0;
            out3\_.out1 = out1;
        }
        else if (edge\_signal)
        {
            out1 = !out1;
            out3\_.out1 = out1;
        }
        else
        {
            cntr = cntr + 1;
            while (cntr > cntr\_max)
            {
                cntr = cntr - 1;
                cntr\_2 = cntr\_2 + 1;
            }
            if (cntr\_2 > 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));
    }
}
```
27 | HOffR: BOOL;
28 | StartI: BOOL;
29 | TStopI: BOOL;
30 | FuStopI: BOOL;
31 | Al: BOOL;
32 | AuOnR: BOOL;
33 | AuOffR: BOOL;
34 | AuAuMoR: BOOL;
35 | AuAlAck: BOOL;
36 | IhAuMRW: BOOL;
37 | AuRstart: BOOL;
38 | POnOff: CPCQNOFFPARAM;
39 | POnOffb: STRUCT AT POnOff: STRUCT;
40 | ParRegb: ARRAY [0..15] OF BOOL;
41 | PPulseLeb: TIME;
42 | PWDtb: TIME;
43 | END STRUCT;
44 | END VAR
45 |
46 | VAR OUTPUT
47 |
48 | Stsreg01: WORD;
49 | Stsreg01b: ARRAY [0..15] OF BOOL;
50 | Stsreg02: WORD;
51 | Stsreg02b: ARRAY [0..15] OF BOOL;
52 | OutOnOV: BOOL;
53 | OutOffOV: BOOL;
54 | OnSt: BOOL;
55 | OffSt: BOOL;
56 | AuMoSt: BOOL;
57 | MMoSt: BOOL;
58 | LDSt: BOOL;
59 | SoftLDS: BOOL;
60 | FwMoSt: BOOL;
61 | AuOnRS: BOOL;
62 | AuOffRS: BOOL;
63 | M3oRt: BOOL;
64 | MoIRSt: BOOL;
65 | HOnRS: BOOL;
66 | HOnRS: BOOL;
67 | IOErrorW: BOOL;
68 | IOSimow: BOOL;
69 | AuM3RW: BOOL;
70 | AIUnAck: BOOL;
71 | AlUnAck: BOOL;
72 | POSW: BOOL;
73 | Stsreg01: WORD;
74 | Stsreg01b: ARRAY [0..15] OF BOOL;
75 | Stsreg02: WORD;
76 | Stsreg02b: ARRAY [0..15] OF BOOL;
77 | OutOnOV: BOOL;
78 | OutOffOV: BOOL;
79 | OnSt: BOOL;
80 | OffSt: BOOL;
81 | AuRS: BOOL;
82 | IBMW: BOOL;
83 | AlUnAck: BOOL;
84 | MEnRstartR: BOOL := TRUE;
85 | END VAR
86 |
87 | VAR // Internal Variables
88 |
89 | // Variables for Edge detection
90 | E_MnMoR: BOOL;
91 | E_MMMoR: BOOL;
92 | E_MnMoR: BOOL;
93 | E_MnMoR: BOOL;
94 | E_MnMoR: BOOL;
95 | E_MnMoR: BOOL;
96 | E_MnMoR: BOOL;
97 | E_MnMoR: BOOL;
98 | E_MnMoR: BOOL;
99 | E_A: BOOL;
100 | E_MnMoR: BOOL;
101 | E_MnMoR: BOOL;
102 | E_MnMoR: BOOL;
103 | E_MnMoR: BOOL;
104 | E_MnMoR: BOOL;
105 | E_MnMoR: BOOL;
106 | E_MnMoR: BOOL;
107 | E_MnMoR: BOOL;
108 | E_MnMoR: BOOL;
109 | E_MnMoR: BOOL;
110 | E_MnMoR: BOOL;
111 | E_MnMoR: BOOL;
112 |
113 | // Variables for old values
114 | M3nMoR: BOOL;
115 | M3nMoR: BOOL;
116 | M3nMoR: BOOL;
Master's Thesis

BEGIN

(* INPUT MANAGER *)

E_ManMoR := R_EDGE(new:=ManReg01b[8], old:=MAuMoR_old); 
(* Manual Auto Mode Request *)

E_ManManR := R_EDGE(new:=ManReg01b[9], old:=MMoMoR_old); 
(* Manual Manual Mode Request *)

E_FoMoR := R_EDGE(new:=ManReg01b[10], old:=MFoMoR_old); 
(* Manual Forced Mode Request *)

E_SoftLDR := R_EDGE(new:=ManReg01b[11], old:=MSofLDR_old); 
(* Manual Software Local Drive Request *)

E_OnR := R_EDGE(new:=ManReg01b[12], old:=MOnR_old); 
(* Manual On/Open Request *)

E_OffR := R_EDGE(new:=ManReg01b[13], old:=MOffR_old); 
(* Manual Off/Close Request *)

E_EnRestartR := R_EDGE(new:=ManReg01b[14], old:=MEnRestartR_old); 
(* Manual Restart after Full Stop Request *)

E_MAIAlAckR := R_EDGE(new:=ManReg01b[15], old:=MAuAlAckR_old); 
(* Manual Alarm Ack Request *)

PFsPosOn := POnOffb.ParRegb[8]; 
(* 1st Parameter bit to define Fail safe position behaviour *)

PHFOff := POnOffb.ParRegb[9];
(• Hardware feedback On present •)

PHFOff := POnOffb . ParRegb [1 0];

(• Hardware feedback Off present •)

PPulse := POnOffb . ParRegb [1 1];

(• Local Drive mode Allowed •)

PHLD := POnOffb . ParRegb [1 2];

(• Local Drive Command allowed •)

PAnim := POnOffb . ParRegb [1 3];

(• Inverted Outputs •)

POutOff := POnOffb . ParRegb [1 5];

(• Enable Restart after Full Stop •)

PEnRstart := POnOffb . ParRegb [0];

(• Enable Restart when Full Stop still active •)

PFsPosOn2 := POnOffb . ParRegb [2];

(• 2nd Parameter bit to define Fail safe position behaviour •)

PulseCste := POnOffb . ParRegb [3];

(• Pulse Constant duration irrespective of the feedback status •)

E AuAuMoR := R EDGE (new:=AuAuMoR, old :=AuAuMoR);

(• Auto Auto Mode Request •)

E AuAlAck := R EDGE (new:=AuAlAck, old :=AuAlAck);

(• Auto Alarm Ack. Request •)

E StartI := R EDGE (new:=StartI, old :=StartI);

E TStopI := R EDGE (new:=TStopI, old :=TStopI);

E FuStopI := R EDGE (new:=FuStopI, old :=FuStopI);

E Al := R EDGE (new:=Al, old :=Al);

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 MEnRstartR OR AuRstart) AND NOT FuStopISt THEN
  fullNotAcknowledged := TRUE;
END_IF;

IF (E FuStopI THEN
  fullNotAcknowledged := TRUE;
IF PEnRstart THEN
  EnRstartSt := FALSE;
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 OutOVStaux) OR (NOT PFsPosOn 
AND NOT OutOVStaux))

FE InterlockR := F EDGE (new:=InterlockR, old :=FE InterlockR);

(• MODE MANAGER •)

IF NOT (HLD AND PHLD) THEN

(• Forced Mode •)

IF (AuMoStaux OR MMoStaux OR SoftLDSaux) AND 
E FMoFFoR AND NOT(AuIhFoMo) THEN
  AuMoStaux := FALSE;
  MMoStaux := FALSE;
  FoMoStaux := TRUE;
  SoftLDSaux := FALSE;
END_IF;

(• Manual Mode •)

IF (AuMoStaux OR FoMoStaux OR SoftLDSaux) AND 
E FMoRFoR AND NOT(AuIhMe) THEN
  AuMoStaux := FALSE;
  MMoStaux := TRUE;
  FoMoStaux := FALSE;
  SoftLDSaux := FALSE;
END_IF;

(• Auto Mode •)

IF (MMoStaux AND (E FMoRFoR OR E AuAnMoR)) OR OR
(FoMoSt\text{aux} \text{ AND } EM\text{AuMoR}) \text{ OR }
(SoftLDSt\text{aux} \text{ AND } EM\text{AuMoR}) \text{ OR }
(MMoSt\text{aux} \text{ AND } AuthMoM) \text{ OR }
(FoMoSt\text{aux} \text{ AND } AuthFoMo) \text{ OR }
(SoftLDSt\text{aux} \text{ AND } AuthFoMo) \text{ OR }
NOT(Em\text{MoSt\text{aux}} \text{ OR } MM\text{MoSt\text{aux}} \text{ OR } FoMoSt\text{aux} \text{ OR } SoftLDSt\text{aux}) \text{ THEN }

AuMoSt\text{aux} := \text{TRUE};
MlMoSt\text{aux} := \text{FALSE};
FoMoSt\text{aux} := \text{FALSE};
SoftLDSt\text{aux} := \text{FALSE};

END IF;

(* Software Local Mode *)
IF (AuMoSt\text{aux} \text{ OR } MM\text{MoSt\text{aux}}) \text{ AND } EM\text{SoftLDR} \text{ AND } NOT \text{ AuIhFoMo} \text{ THEN }

AuMoSt\text{aux} := \text{FALSE};
MlMoSt\text{aux} := \text{FALSE};
FoMoSt\text{aux} := \text{FALSE};
SoftLDSt\text{aux} := \text{TRUE};

END IF;

(* Status setting *)
LDSt := \text{FALSE};
AuMoSt := AuMoSt\text{aux};
MlMoSt := MM\text{MoSt\text{aux}};
FoMoSt := FoMoSt\text{aux};
SoftLDSt := SoftLDSt\text{aux};

ELSE

(* Local Drive Mode *)
AuMoSt := \text{FALSE};
MlMoSt := \text{FALSE};
FoMoSt := \text{FALSE};
LDSt := \text{TRUE};
SoftLDSt := \text{FALSE};

END IF;

(* LIMIT MANAGER *)

(* On/Open Evaluation *)
OnSt := (HFOn \text{ AND } PHFOn) \text{ OR }

(* Feedback ON present *)
(\text{NOT PHFOn } \text{ AND } PHFOff \text{ AND } PAnim \text{ AND } NOT \text{ HFOff}) \text{ OR }
(\text{NOT PHFOn } \text{ AND } NOT \text{ PHFOff } \text{ AND } OutOVSt\text{aux})

(* Feedback OFF not present and PAnim = \text{TRUE} *)
(\text{NOT PHFOn } \text{ AND } NOT \text{ PHFOff } \text{ AND } OutOVSt\text{aux})

(* REQUEST MANAGER *)

(* Auto On/Off Request *)
IF AuOffR \text{ THEN }

AuOnRSt := \text{FALSE};
ELSIF AuOnR \text{ THEN }

AuOnRSt := \text{TRUE};
ELSIF fullNotAcknowledged \text{ OR } FuStopIS \text{ OR } \text{ EnRstartSt} \text{ THEN }

AuOnRSt := PFsPosOn;

END IF;

AuOffRSt := \text{NOT AuOnRSt};

(* Manual On/Off Request *)

IF (EM\text{On/R AND MM}0\text{St OR FoMoSt OR SoftLDSt})

OR (\\text{ AuOHRSt AND AuMoSt})

OR (LDSt \text{ AND PHLDCmd AND HOnRSt})

OR (\text{ FuPulseOn AND PFpulse AND NOT POstRef}) \text{ AND } \text{ EnRstartSt})

OR (\text{ FuStopPl AND NOT PFsPosOn}) \text{ THEN }

MoRSt := \text{FALSE};

ELSIF (\text{ EM\text{OffR AND MM}0\text{St OR FoMoSt OR SoftLDSt})}

OR (\text{ AuOHRSt AND AuMoSt})

OR (LDSt \text{ AND PHLDCmd AND HOnRSt}) \text{ AND } \text{ EnRstartSt})

OR (\text{ FuStopPl AND PFsPosOn}) \text{ THEN }

MoRSt := \text{TRUE};

END IF;

MOHRSt := \text{NOT MoRSt};

(* Local Drive Request *)
IF HOffR \text{ THEN }

HOnRSt := \text{FALSE};
ELSE IF HOnR THEN
  HOnRSt := TRUE;
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 (MOHrSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuoHrSt AND AuoMoSt) OR (HOnR AND LDSt AND PHLDCmd) THEN // Off Request
      PulseOnR := FALSE;
      PulseOffR := FALSE;
    ELSIF (MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuoOnRSt AND AuoMoSt) 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:=PulseOnR, old:=RE_PulseOnR);
  FE_PulseOn := F_EDGE(new:=PulseOnR, old:=FE_PulseOnR);
  RE_PulseOff := R_EDGE(new:=PulseOffR, old:=RE_PulseOffR);
  FE_PulseOff := F_EDGE(new:=PulseOffR, old:=FE_PulseOffR);

  // 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 HFOn));
  END IF;
END IF;

(* Output On Request *)
OutOnOVSt := (PPulse AND PulseOn) OR
             (NOT PPulse AND ((MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR
             (AuoOnrSt AND AuoMoSt) OR
             (HOnRSt AND LDSt AND PHLDCmd)));

(* Output Off Request *)
IF POOff THEN
  OutOffOVSt := (PulseOff AND PPulse) OR
                (NOT PPulse AND ((MOHrSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR
                (AuoHrSt AND AuoMoSt) OR
                (HOnRSt AND LDSt AND PHLDCmd)));
END IF;

(* Interlocks / FailSafe *)

IF POOff THEN
  IF InterlockR THEN
    IF PPulse AND NOT PFsPosOn2 THEN
      OutOnOVSt := PulseOn;
    END IF;
    ELSE
      OutOnOVSt := FALSE;
    END IF;

    ELSE
      OutOnOVSt := (PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2);
    END IF;
END IF;

92
IF InterlockR THEN
  OutOnOVSt := PFsPosOn;
END_IF;
END_IF;
ELSE
  IF InterlockR THEN
    OutOnOVSt := PFsPosOn;
  END_IF;
END_IF;

(* Ready to Start Status *)
RdyStartSt := NOT InterlockR;

(* Alarms *)
A1St := A1;

(* SURVEILLANCE *)

(* I/O Warning *)
IOErrW := IOErr;
IOSimuW := IOSimu;

(* Auto<> Manual Warning *)
AaMRW := (MMoSt OR FoMoSt OR SoftLDSt) AND
((AuOnSt XOR MOSt) OR (AuOffSt XOR MOSt)) AND NOT RaAaMRW;

(* OUTPUT MANAGER AND OUTPUT REGISTER *)
IF NOT POOutOff THEN
  IF PFsPosOn THEN
    OutOnOV := NOT OutOnOVSt;
  ELSE
    OutOnOV := OutOnOVSt;
  END_IF;
ELSE
  OutOnOV := OutOnOVSt;
END_IF;

(* Position warning *)

(* Set reset of the OutOnOVSt *)
IF OutOnOVSt OR (PPulse AND PulseOnR) THEN
  OutOVSt_aux := TRUE;
END_IF;
IF (OutOnOVSt AND POOutOff) OR (NOT OutOnOVSt AND NOT POOutOff) OR (PPulse AND PulseOffR) THEN
  OutOVSt_aux := FALSE;
END_IF;
RE_OutOVSt_aux := R_EDGE(new:=OutOVSt_aux, old:=RE_OutOVSt_aux); FE_OutOVSt_aux := F_EDGE(new:=OutOVSt_aux, old:=FE_OutOVSt_aux);

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 (OIFS1 AND OnSt)
OR (NOT OutOVSt_aux AND (NOT OIFS1 AND OffSt))
AND (NOT PPulse OR (POOutOff 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 (OIFS1 AND OnSt)
OR (RE_OutOVSt_aux)
OR (FE_OutOVSt_aux)
OR (PPulse AND POOutOff AND OutOnOV)
OR (PPulse AND POOutOff AND OutOffOV)
THEN
  PosW_aux := FALSE;
END_IF;

Timer_Warning (IN := PosW_aux,
PT := POOutOff.PWn8);
PosW := Timer_Warning.Q;
Time_Warning := Timer_Warning.ET;

(* Alarm Blocked Warning *)
AIBW := AIB;

(* 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;
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 SInc > 0 THEN
SInc := SInc + 1;
WStartISt := TRUE;
END_IF;

IF SInc > PulseWidth OR (NOT StartISt AND SInc = 0) THEN
SInc := 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 *)

(* Edges *)

END_FUNCTION_BLOCK
DATA STRUCTURES

TYPE CPC\_ONOFF\_PARAM
TITLE = 'CPC\_ONOFF\_PARAM'

PARAM
AUTHOR : 'EN/ICE'
NAME : 'DataType'
FAMILY : 'Base'
STRUCT
ParReg : WORD;
P\_PulseLe : TIME;
PWDt : TIME;
END STRUCT
END TYPE

OTHER FUNCTIONS

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

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

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
BEGIN
IF (new = 1 AND old = 0) THEN // Raising edge detected
F\_EDGE := 1;
old := 0;
ELSE F\_EDGE := 0;
old := new;
END IF;
END FUNCTION
VAR
    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 : 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;
    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
    running : BOOL := FALSE;
    start : TIME := T#0ns; //STIME in Siemens implementation (?)
END_VAR
BEGIN
    IF IN = false then
        Q := false;
        ET := T#0ns;
        running := false;
    ELSE
        // in this case IN == TRUE
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.

```plaintext
1 MODULE main
2 Baseline_version
3 --- Variable declaration
4 VAR
5    ---VAR_INPUT
6 HFOn : boolean;
7 HFOff : boolean;
8 HLD : boolean;
9 IOError : boolean;
10 IOSimu : boolean;
11 AIb : boolean;
12 Manreg01b : array 0..15 of boolean;
13 HOnR : boolean;
14 HOffR : boolean;
15 StartI : boolean;
16 TStopI : boolean;
17 FuStop1 : boolean;
18 AI : boolean;
19 AuOnR : boolean;
20 AuOFR : boolean;
21 AuMeR : boolean;
22 AuMMe : boolean;
23 AuFOrMo : boolean;
24 AuAACK : boolean;
25 BaAuMRW : boolean;
26 AuRstart : boolean;
27 POnOffb.ParRegb : array 0..15 of boolean;
28 POnOffb.PFulseLeb : signed word[32];
29 POnOffb.PWDb : signed word[32];
30 ---VAR_OUTPUT
31 Streg01b : array 0..15 of boolean;
32 Streg02b : array 0..15 of boolean;
33 OutOnOVO : boolean;
34 OutOHOVO : boolean;
35 OnSt : boolean;
36 OHS1 : boolean;
37 AuOSt : boolean;
38 MMOSt : boolean;
39 LST : boolean;
40 SoftLDS1 : boolean;
41 FoMoS : boolean;
42 AuOnRSt : boolean;
43 AuOHS1S : boolean;
44 MOHS1S : boolean;
45 MHOHS1S : boolean;
46 IOEerrorW : boolean;
47 IOSimuW : boolean;
48 AuMRW : boolean;
```

97
--- Internal Variables

--- Variables for edge detection

--- Variables for old values

--- General internal variables

--- Variables for IEC Timers

--- Variables for IEC Timers
--- Variables for interlock Status delay handling
PulseWidth : signed word [32];
FSInc : signed word [16];
TSInc : signed word [16];
SInc  : signed word [16];
AInc  : signed word [16];
WTStopISt : boolean;
WStartISt : boolean;
WAlSt : boolean;
WFuStopISt : boolean;

GLOBAL
TIME : signed word [32];
CYCLE : unsigned word [16];
tcycle : unsigned word [8];

--- extra assertion variables
sFoMoSt_aux : boolean;
AuMoR : boolean;
Manreg01b8 : boolean;
AldMo : boolean;
MMoSt_aux : boolean;
OstOnOV : boolean;
TStopI : boolean;
PStartI : boolean;
PStartOVSt : boolean;
MManreg01b12 : boolean;
PManreg01b13 : boolean;
First : boolean;


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] & !MMoS_old) => step5;
(loc = step4) => step6;
(loc = step5) => step7;
(loc = step6) => step7;
(loc = step7) & (Manreg01b[10] & !MFoMoR_old) => step8;
(loc = step7) => step9;
(loc = step 8) : step 10;
(loc = step 9) : step 10;
(loc = step 10) : step 12;
(loc = step 11) : step 13;
(loc = step 12) : step 13;
(loc = step 13) & (Manreg01b[12] & !MOnR_old) : step 14;
(loc = step 13) : step 15;
(loc = step 14) : step 16;
(loc = step 15) : step 16;
(loc = step 16) & (Manreg01b[13] & !MOHR_old) : step 17;
(loc = step 16) : step 18;
(loc = step 17) : step 19;
(loc = step 18) : step 19;
(loc = step 19) & (Manreg01b[1] & !MEnRstartR_old) : step 20;
(loc = step 19) : step 21;
(loc = step 20) : step 22;
(loc = step 21) : step 22;
(loc = step 22) & (Manreg01b[7] & !MAlAckR_old) : step 23;
(loc = step 22) : step 24;
(loc = step 23) : step 25;
(loc = step 24) : step 25;
(loc = step 25) : step 26;
(loc = step 26) & (AuAuMoR & !AuAuMoR_old) : step 27;
(loc = step 26) : step 28;
(loc = step 27) : step 28;
(loc = step 28) & (Manreg01b[1] & !MEnRstartR_old) : step 29;
(loc = step 28) : step 30;
(loc = step 29) : step 31;
(loc = step 30) : step 32;
(loc = step 31) : step 32;
(loc = step 32) & (Start1 & !Start1_old) : step 33;
(loc = step 32) : step 34;
(loc = step 33) : step 35;
(loc = step 34) : step 35;
(loc = step 35) & (TStopI & !TStopI_old) : step 36;
(loc = step 35) : step 37;
(loc = step 36) : step 38;
(loc = step 37) : step 38;
(loc = step 38) & (FuStopI & !FuStopI_old) : step 39;
(loc = step 38) : step 40;
(loc = step 39) : step 41;
(loc = step 40) : step 41;
(loc = step 41) & (AI & !AI_old) : step 42;
(loc = step 41) : step 43;
(loc = step 42) : step 44;
(loc = step 43) : step 44;
(loc = step 44) : step 45;
(loc = step 45) & (E_MAIAck & E_AuAIack) : step 46;
(loc = step 45) & (E_TStopI | E_Start1 | E_FuStopI | E_AI) : step 47;
(loc = step 45) : step 48;
(loc = step 46) : step 48;
(loc = step 47) : step 48;
(loc = step 48) & ((PEnRstart & (E_MEnRstartR | AuRstart)) & !FuStopI) & (PEnRstart & PEnRstartFS & (E_MEnRstartR | AuRstart)) & (!fullNotAcknowledged) : step 49;
(loc = step 48) : step 50;
(loc = step 49) : step 50;
(loc = step 50) & (E_FuStopI) : step 51;
(loc = step 51) : step 52;
(loc = step 52) & (PEnRstart) : step 53;
(loc = step 53) : step 54;
(loc = step 54) : step 55;
(loc = step 55) & (!InterlockR & !FullNotAcknowledged) : step 56;
(loc = step 56) : step 57;
(loc = step 57) : step 58;
(loc = step 58) & (!IhLD & !HLD) : step 59;
(loc = step 59) : step 60;
(loc = step 60) : step 61;
(loc = step 61) & ((AuMoSt_aux | MMoSt_aux | SoftLDS1_aux) & E_MsoftLD & !{(AuMoStF0Mo)}) : step 62;
(loc = step 62) : step 63;
(loc = step 63) & (MMoSt_aux & (E_MsoftLD | E_AuAuMoR)) & (FoMoSt_aux & E_FoMoR) & (SoftLDS1_aux & E_MsoftLD & !{(AuMoStF0Mo)}) : step 64;
(loc = step 64) : step 65;
(loc = step 65) & (MMoSt_aux & MMoStF0Mo) | (MMoSt_aux & !AuMoSt) | (SoftLDS1_aux & E_MsoftLD) | (FoMoSt_aux & !FoMoR) | (SoftLDS1_aux & !AuMoStF0Mo) | (MMoStF0Mo) | (!AuMoSt) | (MMoStF0Mo) | (SoftLDS1_aux) : step 66;
(loc = step 66) : step 67;
(loc = step 67) : step 68;
(loc = step 68) : step 69;
(loc = step 69) : step 70;
(loc = step 70) & (MMoStF0Mo) : step 71;
(loc = step 71) : step 72;
(loc = step 72) : step 73;
(loc = step 73) : step 74;
(loc = step 74) : step 75;
(loc = step 75) : step 76;
(loc = step 76) : step 77;
(loc = step 77) : step 78;
(loc = step 78) : step 79;
(loc = step 79) : step 80;
(loc = step 80) : step 81;
(loc = step 81) : step 82;
(loc = step 82) : step 83;
(loc = step 83) : step 84;
(loc = step 84) : step 85;
(loc = step 85) : step 86;
(loc = step 86) : step 87;
(loc = step 87) : step 88;
(loc = step 88) : step 89;
(loc = step 89) : step 90;
(loc = step 90) : step 91;
(loc = step 91) : step 92;
(loc = step 92) : step 93;
(loc = step 93) : step 94;
(loc = step 94) : step 95;
(loc = step 95) & (MMoStF0Mo) : step 96;
300 (loc = step 69) : step 70;
301 (loc = step 70) & (AuOffR) : step 71;
302 (loc = step 70) & (AuOnR) : step 72;
303 (loc = step 70) & (fullNotAcknowledged | FuStopSt) : step 73;
304 (loc = step 70) : step 74;
305 (loc = step 71) : step 74;
306 (loc = step 72) : step 74;
307 (loc = step 73) : step 74;
308 (loc = step 74) : step 75;
309 (loc = step 75) & ((E | MOHR & (MMoSt & FoMoSt & SoftLDSt)) & (AuOffRSt & AuMoSt) & (LDSt & PHLDCmd & HOnRSt) & EnRstartSt) : step 76;
310 (loc = step 75) & (E | MoR & (MMoSt & FoMoSt & SoftLDSt)) & (AuOnRSt & AuMoSt) & (LDSt & PHLDCmd & HOnRSt & EnRstartSt) : step 77;
311 (loc = step 75) : step 78;
312 (loc = step 76) : step 78;
313 (loc = step 77) : step 78;
314 (loc = step 78) : step 79;
315 (loc = step 79) & (HOnR) : step 80;
316 (loc = step 79) & (HOffR) : step 81;
317 (loc = step 79) : step 82;
318 (loc = step 80) : step 82;
319 (loc = step 81) : step 82;
320 (loc = step 82) : step 83;
321 (loc = step 83) & (PFpulse) : step 84;
322 (loc = step 83) : step 158;
323 (loc = step 84) & (EnRstartSt) : step 85;
324 (loc = step 84) & (E | FuStopR & !PFpulse) : step 86;
325 (loc = step 84) & (E | MOHR & (MMoSt & FoMoSt & SoftLDSt)) & (AuOffRSt & AuMoSt) & (LDSt & PHLDCmd & HOnRSt & EnRstartSt) : step 105;
326 (loc = step 84) & (E | MoR & (MMoSt & FoMoSt & SoftLDSt)) & (AuOnRSt & AuMoSt) & (LDSt & PHLDCmd & HOnRSt & EnRstartSt) : step 106;
327 (loc = step 84) & (InterlockR) : step 87;
328 (loc = step 84) & (E | InterlockR) : step 87;
329 (loc = step 84) & (MOffRSt & (MMoSt & FoMoSt & SoftLDSt)) & (AuOffRSt & AuMoSt) & (HOffR & LDSt & PHLDCmd) : step 97;
330 (loc = step 84) & (MOnRSt & (MMoSt & FoMoSt & SoftLDSt)) & (AuOnRSt & AuMoSt) & (HOnR & LDSt & PHLDCmd) : step 98;
331 (loc = step 85) : step 98;
332 (loc = step 86) : step 98;
333 (loc = step 87) : step 98;
334 (loc = step 88) : step 98;
335 (loc = step 89) : step 98;
336 (loc = step 90) : step 98;
337 (loc = step 91) : step 98;
338 (loc = step 92) : step 98;
339 (loc = step 93) : step 98;
340 (loc = step 94) : step 98;
341 (loc = step 95) : step 98;
342 (loc = step 96) & (FALSE & !TimerPulseOn.old) & !TimerPulseOn.Q : step 97;
343 (loc = step 96) : step 98;
344 (loc = step 97) : step 98;
345 (loc = step 98) : step 98;
346 (loc = step 99) : step 98;
347 (loc = step 100) : step 101;
348 (loc = step 101) & (FALSE) : step 102;
349 (loc = step 101) : step 103;
350 (loc = step 102) : step 104;
351 (loc = step 103) : step 104;
352 (loc = step 104) : step 105;
353 (loc = step 105) : step 106;
354 (loc = step 106) : step 108;
355 (loc = step 107) : step 108;
357 (loc = step 108) : step 110;
358 (loc = step 109) : step 110;
359 (loc = step 110) & (GLOBAL_TIME <= TimerPulseOn.due) : step 111;
360 (loc = step 110) & (GLOBAL_TIME <= TimerPulseOff.due) : step 112;
361 (loc = step 110) : step 112;
362 (loc = step 111) : step 113;
363 (loc = step 112) : step 113;
364 (loc = step 113) & (PulseOnR) : step 114;
365 (loc = step 113) : step 115;
366 (loc = step 114) : step 116;
367 (loc = step 115) : step 117;
368 (loc = step 116) : step 118;
369 (loc = step 117) & (PulseOffR & !TimerPulseOn.old) & !TimerPulseOff.Q) : step 118;
370 (loc = step 117) : step 119;
371 (loc = step 118) : step 119;
372 (loc = step 119) & (GLOBAL_TIME <= TimerPulseOff.due) : step 120;
373 (loc = step 119) : step 121;
374 (loc = step 120) : step 122;
375 (loc = step 121) : step 122;
376 (loc = step 122) & (PulseOffR) : step 123;
377 (loc = step 122) : step 124;
378 (loc = step 123) : step 125;
379 (loc = step 124) : step 126;
380 (loc = step 125) : step 126;
467 | ( loc = step 187) : step 188;
468 | ( loc = step 188) & ( !(OutOVSt_aux & (PHFOn & !OnSt) | (PHFOff & !OffSt)) | !(OutOVSt_aux & (PHFOn & !OnSt) | (PHFOff & OffSt)) | RE_OutOVSt | FE_OutOVSt AUX | (PFpulse & POnOff & OutOnOV) | (PPulse & POutOff & OutOnOV)) : step 189;
469 | ( loc = step 189) : step 190;
470 | ( loc = step 190) & ( !PosWaux) : step 191;
471 | ( loc = step 191) : step 192;
472 | ( loc = step 192) & ( !TimerWarning.running) : step 193;
473 | ( loc = step 193) : step 198;
474 | ( loc = step 194) & ( ! (GLOBAL_TIME - (TimerWarning.Tstart + POnOffb.POnbb) >= 0 sd 32)) : step 195;
475 | ( loc = step 195) : step 196;
476 | ( loc = step 196) & ( !FuStopISt) : step 197;
477 | ( loc = step 197) & ( (FuStopISt | FSIinc > 0 sd 16) | !FuStopISt & FSIinc = 0 sd 16) : step 202;
478 | ( loc = step 202) : step 203;
479 | ( loc = step 203) & ( ! (StartISt | SInc > 0 sd 16)) : step 204;
480 | ( loc = step 204) : step 205;
481 | ( loc = step 205) & ( (StartISt & SInc = 0 sd 16) | !StartISt & SInc = 0 sd 16) : step 210;
482 | ( loc = step 210) : step 211;
483 | ( loc = step 211) & ( !AiSt) : step 212;
484 | ( loc = step 212) : step 213;
486 | ( loc = step 216) : step 217;
487 | ( loc = step 217) : step 218;
488 | ( loc = step 218) : step 219;
489 | ( loc = step 219) : step 220;
490 | ( loc = step 220) : end;
491 | ( loc = end) : nvar;
492 | ( loc = nvar) : start;
493 | esac;
494
516 | ( loc = step 221) : end;
517 | ( loc = end) : nvar;
518 | ( loc = nvar) : start;
519
521 --- Variable initialization
522 init (Sfreg01b[0]) := FALSE;
523 init (Sfreg01b[1]) := FALSE;
524 init (Sfreg01b[2]) := FALSE;
525 init (Sfreg01b[3]) := FALSE;
526 init (Sfreg01b[4]) := FALSE;
527 init (Sfreg01b[5]) := FALSE;
528 init (Sfreg01b[6]) := FALSE;
529 init (Sfreg01b[7]) := FALSE;
530 init (Sfreg01b[8]) := FALSE;
531 init (Sfreg01b[9]) := FALSE;
532 init (Sfreg01b[10]) := FALSE;
533 init (Sfreg01b[11]) := FALSE;
534 init (Sfreg01b[12]) := FALSE;
535 init (Sfreg01b[13]) := FALSE;
536 init (Sfreg01b[14]) := FALSE;
537 init (Sfreg01b[15]) := FALSE;
538 init (Sfreg02b[0]) := FALSE;
539 init (Sfreg02b[1]) := FALSE;
540 init (Sfreg02b[2]) := FALSE;
541 init (Sfreg02b[3]) := FALSE;
542 init (Sfreg02b[4]) := FALSE;
543 init (Sfreg02b[5]) := FALSE;
544 init (Sfreg02b[6]) := FALSE;
545 init (Sfreg02b[7]) := FALSE;
546 init (Sfreg02b[8]) := FALSE;
547 init (Sfreg02b[9]) := FALSE;
548 init (Sfreg02b[10]) := FALSE;
init (Streg02b[11]) := FALSE;
init (Streg02b[12]) := FALSE;
init (Streg02b[13]) := FALSE;
init (Streg02b[14]) := FALSE;
init (Streg02b[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 (AuOHRS) := FALSE;
init (MOHRs) := FALSE;
init (HOnRSt) := FALSE;
init (HOffRSt) := FALSE;
init (IOErrorW) := FALSE;
init (IOSimuW) := FALSE;
init (AuMRW) := FALSE;
init (AlUnAck) := FALSE;
init (PosW) := FALSE;
init (StaISt) := 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_MAIAckR) := FALSE;
init (E_StartI) := FALSE;
init (E_TStopI) := FALSE;
init (E_FuStopI) := FALSE;
init (E_AI) := FALSE;
init (E_AuAmMoR) := FALSE;
init (E_AuAmMoR_old) := FALSE;
init (E_MSisLDR) := FALSE;
init (E_MEnRstartR) := FALSE;
init (E_ReAIUnAck) := FALSE;
init (E_FeAIUnAck) := FALSE;
init (E_RePulseOn) := FALSE;
init (E_FePulseOn) := FALSE;
init (E_RePulseOff) := FALSE;
init (E_ReOutOVShAux) := FALSE;
init (E_ReOutOVShAux_old) := FALSE;
init (E_FeOutOVShAux) := FALSE;
init (E_FeOutOVShAux_old) := FALSE;
init (E_ReInterlockR) := FALSE;
init (E_RePulseOn_old) := FALSE;
init (E_RePulseOff_old) := FALSE;
init (E_RePulseOffAux) := FALSE;
init (E_RePulseOffAux_old) := FALSE;
init (E_ReOutOVShAux_old) := FALSE;
init (E_ReOutOVShAux_old) := FALSE;
init (E_ReInterlockR_old) := FALSE;
init (E_RePulseOn_old) := FALSE;
init (E_RePulseOff_old) := FALSE;
init (E_RePulseOffAux) := FALSE;
init (E_RePulseOffAux_old) := FALSE;
init (E_ReOutOVShAux) := FALSE;
init (E_ReOutOVShAux) := FALSE;
init (E_ReInterlockR) := FALSE;
init (E_RePulseOn) := FALSE;
init (E_RePulseOff) := FALSE;
init (E_RePulseOffAux) := FALSE;
init (E_RePulseOffAux) := FALSE;
init (E_ReOutOVShAux) := FALSE;
init (E_ReOutOVShAux) := FALSE;
init (E_ReInterlockR) := FALSE;
639  init (PEnRstart) := FALSE;
640  init (PRstartFS) := FALSE;
641  init (OutOVOFS) := FALSE;
642  init (OutOVSt) := FALSE;
643  init (AuMoSt_aux) := FALSE;
644  init (MMoSt_aux) := FALSE;
645  init (FoMoSt_aux) := FALSE;
646  init (SoftLDSt_aux) := FALSE;
647  init (PulseOn) := FALSE;
648  init (PulseOff) := FALSE;
649  init (PosW_aux) := FALSE;
650  init (OutOVSt_aux) := FALSE;
651  init (fullNotAcknowledged) := FALSE;
652  init (PulseOnR) := FALSE;
653  init (PulseOffR) := FALSE;
654  init (InterlockR) := FALSE;
655
656  -- Variables for IEC Timers
657  init (TimeWarning) := 0 sd 32_0;
658  init (TimerPulseOn.Q) := FALSE;
659  init (TimerPulseOn.ET) := 0 sd 32_0;
660  init (TimerPulseOn.old_in) := FALSE;
661  init (TimerPulseOn.due) := 0 sd 32_0;
662  init (TimerPulseOff.Q) := FALSE;
663  init (TimerPulseOff.ET) := 0 sd 32_0;
664  init (TimerPulseOff.old_in) := FALSE;
665  init (TimerPulseOff.due) := 0 sd 32_0;
666  init (TimerWarning.Q) := FALSE;
667  init (TimerWarning.ET) := 0 sd 32_0;
668  init (TimerWarning.running) := FALSE;
669  init (TimerWarning.Tstart) := 0 sd 32_0;
670
671  -- Variables for interlock Status delay handling
672  init (PulseWidth) := 0 sd 32_0;
673  init (FSInc) := 0 sd 16_0;
674  init (TSInc) := 0 sd 16_0;
675  init (StInc) := 0 sd 16_0;
676  init (AtInc) := 0 sd 16_0;
677  init (WTStopISt) := FALSE;
678  init (WTstartISt) := FALSE;
679  init (WAISt) := FALSE;
680  init (WPuStopISt) := FALSE;
681  init (_GLOBAL_TIME) := 0 sd 32_0;
682  init (T_CYCLE) := 0 sd 16_0;
683  init (random_i_cycle) := 0 sd 8_0;
684
685  -- Non-deterministic input
686  next (HFOn) :=
687      case
688          (loc = start) : (TRUE, FALSE);
689          TRUE : HFOn ;
690          esac ;
691  next (HOff) :=
692      case
693          (loc = start) : (TRUE, FALSE);
694          TRUE : HOFF ;
695          esac ;
696  next (HLD) :=
697      case
698          (loc = start) : (TRUE, FALSE);
699          TRUE : HLD ;
700          esac ;
701  next (IOError) :=
702      case
703          (loc = start) : (TRUE, FALSE);
704          TRUE : IOERROR ;
705          esac ;
706  next (IOSimu) :=
707      case
708          (loc = start) : (TRUE, FALSE);
709          TRUE : IOSIMU ;
710          esac ;
711  next (AIB) :=
712      case
713          (loc = start) : (TRUE, FALSE);
714          TRUE : AIB ;
715          esac ;
716  next (Manreg01b[0]) :=
717      case
718          (loc = start) : (TRUE, FALSE);
719          TRUE : Manreg01b[0] ;
720          esac ;
721  next (Manreg01b[1]) :=
722      case
723          (loc = start) : (TRUE, FALSE);
724          TRUE : Manreg01b[1] ;
725          esac ;
726  next (Manreg01b[2]) :=
727      case
728          (loc = start) : (TRUE, FALSE);
TRUE : Manreg01b[2];
next(Manreg01b[3]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[3];
ext;
next(Manreg01b[4]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[4];
ext;
next(Manreg01b[5]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[5];
ext;
next(Manreg01b[6]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[6];
ext;
next(Manreg01b[7]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[7];
ext;
next(Manreg01b[8]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[8];
ext;
next(Manreg01b[9]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[9];
ext;
next(Manreg01b[10]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[10];
ext;
next(Manreg01b[11]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[11];
ext;
next(Manreg01b[12]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[12];
ext;
next(Manreg01b[13]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[13];
ext;
next(Manreg01b[14]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[14];
ext;
next(Manreg01b[15]) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : Manreg01b[15];
ext;
next(HOnR) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : HOnR;
ext;
next(HOffR) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : HOffR;
ext;
next(StartI) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : StartI;
ext;
next(TStopI) :=
case
  (loc = start) : (TRUE, FALSE);
  TRUE : TStopI;
ext;
next(FuStopI) :=
case
  (loc = start) : (TRUE, FALSE);
TRUE : FuStopI;

next (Al) :=
  case
    (loc = start) : (TRUE, FALSE);
    TRUE : Al;
  esac;

next (AuOhR) :=
  case
    (loc = start) : (TRUE, FALSE);
    TRUE : AuOhR;
  esac;

next (AuOHR) :=
  case
    (loc = start) : (TRUE, FALSE);
    TRUE : AuOHR;
  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;

¬¬¬Values that are non-deterministic, but do not get 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

case TRUE : POnOffb . ParRegb[10];
}
}
next (POnOffb . ParRegb[11]) :=
}
next (POnOffb . ParRegb[12]) :=
}
next (POnOffb . ParRegb[13]) :=
}
next (POnOffb . ParRegb[14]) :=
}
next (POnOffb . ParRegb[15]) :=
}
next (POnOffb . PPulseLeb) :=
}
next (POnOffb . PWDtb) :=
}
next (E0MAuMoR) :=
}
next (MAuMoRold) :=
}
next (EMoM0R) :=
}
next (MoM0Rold) :=
}
next (EM0F0MoR) :=
}
next (MF0MoRold) :=
}
next (EM0LDR) :=
}
next (M0LDRold) :=
}
next (EM0R) :=
}

Program body

next (E0MAuMoR) :=
}

next (MAuMoRold) :=
}

next (EMoM0R) :=
}

next (MoM0Rold) :=
}

next (EM0F0MoR) :=
}

next (MF0MoRold) :=
}

next (EM0LDR) :=
}

next (M0LDRold) :=
}

next (EM0R) :=
}
(loc = step 15) : Manreg01b[12];
TRUE : MOffR_old;

next (E_MOHR) :=
case
(loc = step 17) : TRUE;
(loc = step 18) : FALSE;
TRUE : E_MOHR;

next (MOHR_old) :=
case
(loc = step 17) : TRUE;
(loc = step 18) : Manreg01b[13];
TRUE : E_MOHR;

next (E_MOEnRstartR) :=
case
(loc = step 17) : TRUE;
(loc = step 18) : FALSE;
TRUE : E_MOEnRstartR;

next (E_MOUSERstartR) :=
case
(loc = step 20) : TRUE;
(loc = step 21) : FALSE;
TRUE : E_MOUSERstartR;

next (E_MAlAckR) :=
case
(loc = step 23) : TRUE;
(loc = step 24) : FALSE;
TRUE : E_MAlAckR;

next (PFsPosOn) :=
case
(loc = step 25) : POnOffb.ParRegb[0];
TRUE : PFsPosOn;

next (PHFOn) :=
case
(loc = step 25) : POnOffb.ParRegb[1];
TRUE : PHFOn;

next (PHFOff) :=
case
(loc = step 25) : POnOffb.ParRegb[2];
TRUE : PHFOff;

next (PAnim) :=
case
(loc = step 25) : POnOffb.ParRegb[3];
TRUE : PAnim;

next (POutOff) :=
case
(loc = step 25) : POnOffb.ParRegb[4];
TRUE : POutOff;

next (PEnRstart) :=
case
(loc = step 25) : POnOffb.ParRegb[5];
TRUE : PEnRstart;

next (PRstartFS) :=
case
(loc = step 25) : POnOffb.ParRegb[6];
TRUE : PRstartFS;
Master's Thesis

case
(\text{loc} = \text{step} 25): \text{POnOffb} . ParRegb[2] ;
TRUE: PFsPosOn2 ;
\text{esac} ;
next (PPulseCste) :=
case
(\text{loc} = \text{step} 25): \text{POnOffb} . ParRegb[3] ;
TRUE: PPulseCste ;
\text{esac} ;
next (E_{\text{AuAuMoR}}) :=
case
(\text{loc} = \text{step} 27): \text{TRUE} ;
(\text{loc} = \text{step} 28): \text{FALSE} ;
TRUE: E_{\text{AuAuMoR}} ;
\text{esac} ;
next (AuAuMoR_{old}) :=
case
(\text{loc} = \text{step} 27): \text{TRUE} ;
(\text{loc} = \text{step} 28): \text{AuAuMoR} ;
TRUE: AuAuMoR_{old} ;
\text{esac} ;
next (E_{\text{AuAlAck}}) :=
case
(\text{loc} = \text{step} 30): \text{TRUE} ;
(\text{loc} = \text{step} 31): \text{FALSE} ;
TRUE: E_{\text{AuAlAck}} ;
\text{esac} ;
next (AuAlAck_{old}) :=
case
(\text{loc} = \text{step} 30): \text{TRUE} ;
(\text{loc} = \text{step} 31): \text{AuAlAck} ;
TRUE: AuAlAck_{old} ;
\text{esac} ;
next (E_{\text{StartI}}) :=
case
(\text{loc} = \text{step} 33): \text{TRUE} ;
(\text{loc} = \text{step} 34): \text{FALSE} ;
TRUE: E_{\text{StartI}} ;
\text{esac} ;
next (\text{StartI}_{old}) :=
case
(\text{loc} = \text{step} 33): \text{TRUE} ;
(\text{loc} = \text{step} 34): \text{StartI} ;
TRUE: \text{StartI}_{old} ;
\text{esac} ;
next (E_{\text{TStopI}}) :=
case
(\text{loc} = \text{step} 36): \text{TRUE} ;
(\text{loc} = \text{step} 37): \text{FALSE} ;
TRUE: E_{\text{TStopI}} ;
\text{esac} ;
next (TStopI_{old}) :=
case
(\text{loc} = \text{step} 36): \text{TRUE} ;
(\text{loc} = \text{step} 37): \text{TStopI} ;
TRUE: TStopI_{old} ;
\text{esac} ;
next (E_{\text{FuStopI}}) :=
case
(\text{loc} = \text{step} 39): \text{TRUE} ;
(\text{loc} = \text{step} 40): \text{FALSE} ;
TRUE: E_{\text{FuStopI}} ;
\text{esac} ;
next (FuStopI_{old}) :=
case
(\text{loc} = \text{step} 39): \text{TRUE} ;
(\text{loc} = \text{step} 40): \text{FuStopI} ;
TRUE: FuStopI_{old} ;
\text{esac} ;
ext (E_{\text{Al}}) :=
case
(\text{loc} = \text{step} 42): \text{TRUE} ;
(\text{loc} = \text{step} 43): \text{FALSE} ;
TRUE: E_{\text{Al}} ;
\text{esac} ;
ext (Al_{old}) :=
case
(\text{loc} = \text{step} 42): \text{TRUE} ;
(\text{loc} = \text{step} 43): \text{Al} ;
TRUE: Al_{old} ;
\text{esac} ;
ext (\text{StartISt}) :=
case
(\text{loc} = \text{step} 44): \text{StartI} ;
TRUE: \text{StartISt} ;
\text{esac} ;
ext (TStopISt) :=
case
(\text{loc} = \text{step} 44): \text{TStopI} ;
TRUE: TStopISt ;
next (FuStopISt) :=
case
  (loc = step 44) : FuStopI;
  TRUE : FuStopISt;
  esac;

next (fullNotAcknowledged) :=
case
  (loc = step 46) : FALSE;
  (loc = step 51) : TRUE;
  TRUE : fullNotAcknowledged;
  esac;

next (AlUnAck) :=
case
  (loc = step 46) : FALSE;
  (loc = step 47) : TRUE;
  TRUE : AlUnAck;
  esac;

next (EnRstartSt) :=
case
  (loc = step 49) : TRUE;
  (loc = step 53) : FALSE;
  TRUE : EnRstartSt;
  esac;

next (InterlockR) :=
case
  (loc = step 54) : (TStopISt | FuStopISt | fullNotAcknowledged | !EnRstartSt | (StartISt & !POutOff & !OutOV) | (StartISt & POutOff & (PFsPosOn & OutOVStaux) | (!PEsPosOn & !OutOVStaux)));
  TRUE : InterlockR;
  esac;

ejnext (FEInterlockR) :=
case
  (loc = step 56) : TRUE;
  (loc = step 57) : FALSE;
  FALSE : FEInterlockR;
  esac;

next (FEInterlockR_old) :=
case
  (loc = step 56) : FALSE;
  (loc = step 57) : InterlockR;
  TRUE : FEInterlockR_old;
  esac;

next (AuMoSt_aux) :=
case
  (loc = step 60) : FALSE;
  (loc = step 62) : FALSE;
  (loc = step 64) : TRUE;
  TRUE : AuMoSt_aux;
  esac;

next (MMoSt_aux) :=
case
  (loc = step 60) : FALSE;
  (loc = step 62) : TRUE;
  (loc = step 64) : FALSE;
  TRUE : MMoSt_aux;
  esac;

next (FoMoSt_aux) :=
case
  (loc = step 60) : TRUE;
  (loc = step 62) : FALSE;
  (loc = step 64) : FALSE;
  TRUE : FoMoSt_aux;
  esac;

next (SoftLDSt_aux) :=
case
  (loc = step 60) : FALSE;
  (loc = step 62) : FALSE;
  (loc = step 64) : FALSE;
  TRUE : SoftLDSt_aux;
  esac;

next (LDSt) :=
case
  (loc = step 67) : FALSE;
  (loc = step 68) : TRUE;
  TRUE : LDSt;
  esac;

next (AuMoSt) :=
case
  (loc = step 67) : AuMoSt_aux;
  (loc = step 68) : TRUE;
  TRUE : AuMoSt;
  esac;

next (MMoSt) :=
case
  (loc = step 67) : MMoSt_aux;
1267 (loc = step 68) : FALSE;
1268 TRUE : MMoSt;
1269 esac ;
1270 next (FoMoSt) :=
1271 case
1272 (loc = step 67) : FoMoSt_aux;
1273 (loc = step 68) : FALSE;
1274 TRUE := FoMoSt;
1275 esac ;
1276 next (SoftLDSt) :=
1277 case
1278 (loc = step 67) : FALSE;
1279 (loc = step 68) : TRUE := SoftLDSt;
1280 esac ;
1281 next (OnSt) :=
1282 case
1283 (loc = step 69) : (HFOn & PHFOn)
1284 | ( !PHFOn & PHFOff & PAnim & !HFOff )
1285 | ( !PHFOn & !PHFOff & OutOVSt_aux )
1286 TRUE := OnSt;
1287 esac ;
1288 next (OffSt) :=
1289 case
1290 (loc = step 69) : (HFOff & PHFOff)
1291 | ( !PHFOff & PHFOn & PAnim & !HFOn )
1292 | ( !PHFOn & !PHFOff & !OutOVSt_aux )
1293 TRUE := OffSt;
1294 esac ;
1295 next (AuOnRSt) :=
1296 case
1297 (loc = step 71) : FALSE;
1298 (loc = step 72) : TRUE := AuOnRSt;
1299 (loc = step 73) : PFsPosOn;
1300 TRUE := AuOnRSt;
1301 esac ;
1302 next (AuOffRSt) :=
1303 case
1304 (loc = step 74) : ! AuOnRSt;
1305 (loc = step 75) : TRUE := AuOffRSt;
1306 (loc = step 77) : FALSE;
1307 (loc = step 78) : TRUE := AuOffRSt;
1308 (loc = step 79) : PFsPosOn;
1309 TRUE := AuOffRSt;
1310 esac ;
1311 next (MOnRSt) :=
1312 case
1313 (loc = step 76) : FALSE;
1314 (loc = step 77) : TRUE := MOnRSt;
1315 TRUE := MOnRSt;
1316 (loc = step 78) : PFsPosOn;
1317 (loc = step 79) : TRUE := MOnRSt;
1318 (loc = step 80) : FALSE;
1319 TRUE := MOnRSt;
1320 esac ;
1321 next (HOnRSt) :=
1322 case
1323 (loc = step 85) : (PFsPosOn & !PFsPosOn2)
1324 | (PFsPosOn & PFsPosOn2);
1325 (loc = step 86) : FALSE;
1326 (loc = step 87) : FALSE;
1327 (loc = step 88) : FALSE;
1328 TRUE := HOnRSt;
1329 esac ;
1330 next (HOffRSt) :=
1331 case
1332 (loc = step 89) : TRUE;
1333 (loc = step 90) : FALSE;
1334 (loc = step 91) : FALSE;
1335 (loc = step 92) : TRUE;
1336 (loc = step 93) : TRUE := HOffRSt;
1337 (loc = step 94) : FALSE;
1338 (loc = step 95) : FALSE;
1339 (loc = step 96) : FALSE;
1340 TRUE := HOffRSt;
1341 esac ;
1342 next (Timer_PulseOn . due ) :=
1343 case
1344 (loc = step 88) : GLOBAL_TIME + 0 sd 32_0;
1345 (loc = step 105) : GLOBAL_TIME + 0 sd 32_0;
1346 (loc = step 147) : GLOBAL_TIME + 0 sd 32_0;
1347 TRUE := Timer_PulseOn . due ;
1348 esac ;
1349 next (Timer_PulseOn . Q ) :=
1350 case
1351 (loc = step 90) : TRUE;
1352 (loc = step 91) : FALSE;
1353 (loc = step 92) : TRUE;
1354 (loc = step 93) : FALSE;
1355 (loc = step 94) : TRUE;
1356 (loc = step 95) : FALSE;
1357 (loc = step 96) : FALSE;
1358 TRUE := Timer_PulseOn . Q ;
next (Timer\_PulseOn\_ET) :=
\[\begin{align*}
(\text{loc} = \text{step 90}) & : (0 \Leftrightarrow 32_0 - (\text{Timer}\_\text{PulseOn}\_\text{due} - \text{GLOBAL}\_\text{TIME}) ; \\
(\text{loc} = \text{step 93}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 94}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 111}) & : (\text{POnOff}\_\text{FFPulseLeb} - (\text{Timer}\_\text{PulseOn}\_\text{due} - \text{GLOBAL}\_\text{TIME}) ) ; \\
(\text{loc} = \text{step 114}) & : \text{POnOff}\_\text{FFPulseLeb} ; \\
(\text{loc} = \text{step 115}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 149}) & : (0 \Leftrightarrow 32_0 - (\text{Timer}\_\text{PulseOn}\_\text{due} - \text{GLOBAL}\_\text{TIME}) ) ; \\
(\text{loc} = \text{step 152}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 153}) & : 0 \Leftrightarrow 32_0 ; \\
\text{TRUE} & : \text{Timer}\_\text{PulseOn}\_\text{ET} ; \\
\end{align*}\]

next (Timer\_\text{PulseOn\_old\_in}) :=
\[\begin{align*}
(\text{loc} = \text{step 95}) & : \text{FALSE} ; \\
(\text{loc} = \text{step 100}) & : \text{PulseOnR} ; \\
(\text{loc} = \text{step 154}) & : \text{FALSE} ; \\
\text{TRUE} & : \text{Timer}\_\text{PulseOn}\_\text{old\_in} ; \\
\end{align*}\]

next (Timer\_\text{PulseOff\_due}) :=
\[\begin{align*}
(\text{loc} = \text{step 97}) & : \text{GLOBAL}\_\text{TIME} + 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 118}) & : \text{GLOBAL}\_\text{TIME} + \text{POnOff}\_\text{FFPulseLeb} ; \\
(\text{loc} = \text{step 137}) & : \text{GLOBAL}\_\text{TIME} + 0 \Leftrightarrow 32_0 ; \\
\text{TRUE} & : \text{Timer}\_\text{PulseOff}\_\text{due} ; \\
\end{align*}\]

next (Timer\_\text{PulseOff\_Q}) :=
\[\begin{align*}
(\text{loc} = \text{step 99}) & : \text{TRUE} ; \\
(\text{loc} = \text{step 100}) & : \text{FALSE} ; \\
(\text{loc} = \text{step 121}) & : \text{FALSE} ; \\
(\text{loc} = \text{step 139}) & : \text{TRUE} ; \\
(\text{loc} = \text{step 140}) & : \text{FALSE} ; \\
\text{TRUE} & : \text{Timer}\_\text{PulseOff}\_\text{Q} ; \\
\end{align*}\]

next (Timer\_\text{PulseOff\_ET}) :=
\[\begin{align*}
(\text{loc} = \text{step 99}) & : (0 \Leftrightarrow 32_0 - (\text{Timer}\_\text{PulseOn}\_\text{due} - \text{GLOBAL}\_\text{TIME}) ) ; \\
(\text{loc} = \text{step 102}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 103}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 120}) & : (\text{POnOff}\_\text{FFPulseLeb} - (\text{Timer}\_\text{PulseOn}\_\text{due} - \text{GLOBAL}\_\text{TIME}) ) ; \\
(\text{loc} = \text{step 123}) & : \text{POnOff}\_\text{FFPulseLeb} ; \\
(\text{loc} = \text{step 124}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 139}) & : (0 \Leftrightarrow 32_0 - (\text{Timer}\_\text{PulseOn}\_\text{due} - \text{GLOBAL}\_\text{TIME}) ) ; \\
(\text{loc} = \text{step 142}) & : 0 \Leftrightarrow 32_0 ; \\
(\text{loc} = \text{step 143}) & : 0 \Leftrightarrow 32_0 ; \\
\text{TRUE} & : \text{Timer}\_\text{PulseOff}\_\text{ET} ; \\
\end{align*}\]

next (Timer\_\text{PulseOff\_old\_in}) :=
\[\begin{align*}
(\text{loc} = \text{step 104}) & : \text{FALSE} ; \\
(\text{loc} = \text{step 125}) & : \text{PulseOffR} ; \\
(\text{loc} = \text{step 144}) & : \text{FALSE} ; \\
\text{TRUE} & : \text{Timer}\_\text{PulseOff\_old\_in} ; \\
\end{align*}\]

next (RE\_PulseOn) :=
\[\begin{align*}
(\text{loc} = \text{step 127}) & : \text{TRUE} ; \\
(\text{loc} = \text{step 128}) & : \text{FALSE} ; \\
\text{TRUE} & : \text{RE}\_\text{PulseOn} ; \\
\end{align*}\]

next (RE\_PulseOn\_old) :=
\[\begin{align*}
(\text{loc} = \text{step 127}) & : \text{TRUE} ; \\
(\text{loc} = \text{step 128}) & : \text{PulseOn} ; \\
\text{TRUE} & : \text{RE}\_\text{PulseOn\_old} ; \\
\end{align*}\]

next (FE\_PulseOn) :=
\[\begin{align*}
(\text{loc} = \text{step 130}) & : \text{TRUE} ; \\
(\text{loc} = \text{step 131}) & : \text{FALSE} ; \\
\text{TRUE} & : \text{FE}\_\text{PulseOn} ; \\
\end{align*}\]

next (FE\_PulseOn\_old) :=
\[\begin{align*}
(\text{loc} = \text{step 130}) & : \text{FALSE} ; \\
(\text{loc} = \text{step 131}) & : \text{PulseOn} ; \\
\text{TRUE} & : \text{FE}\_\text{PulseOn\_old} ; \\
\end{align*}\]

next (RE\_PulseOff) :=
\[\begin{align*}
(\text{loc} = \text{step 133}) & : \text{TRUE} ; \\
(\text{loc} = \text{step 134}) & : \text{FALSE} ; \\
\text{TRUE} & : \text{RE}\_\text{PulseOff} ; \\
\end{align*}\]

next (RE\_PulseOff\_old) :=
\[\begin{align*}
(\text{loc} = \text{step 133}) & : \text{TRUE} ; \\
(\text{loc} = \text{step 134}) & : \text{FALSE} ; \\
\text{TRUE} & : \text{RE}\_\text{PulseOff\_old} ; \\
\end{align*}\]
(loc = step 133) : TRUE;
(loc = step 134) : PulseOff;
TRUE : RE_PulseOff_old;
next (PulseOn) :=
case
(loc = step 156) : Timer_PulseOn Q & !PulseOff;
(loc = step 157) : Timer_PulseOn Q & !PulseOff & (PHFOOn | (PHFOff & !HFOn));
TRUE : PulseOn;
esc;
next (PulseOff) :=
case
(loc = step 156) : Timer_PulseOff.Q & !PulseOn;
(loc = step 157) : Timer_PulseOff.Q & !PulseOn & (PHFOff | (PHFOff & !HFOn));
TRUE : PulseOff;
esc;
next (OutOnOVSt) :=
case
(loc = step 158) : (PPulse & PulseOn) | ((PPulse & ((MOrrSt & (MMoSt | FoMoSt | SoftLDSt)) | (AUonRS & AuMoSt)) | (HOnRS & LDSt & PHILCmd)));
(loc = step 165) : PulseOn;
(loc = step 166) : FALSE;
(loc = step 167) : (PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
(loc = step 169) : PFsPosOn;
TRUE : OutOnOVSt;
esc;
next (OutOffOVSt) :=
case
(loc = step 160) : (PulseOff & PPulse) | ((PPulse & ((MOrrSt & (MMoSt | FoMoSt | SoftLDSt)) | (AUonRS & AuMoSt)) | (HOnRS & LDSt & PHILCmd)));
(loc = step 165) : FALSE;
(loc = step 166) : PulseOff;
(loc = step 167) : (PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
TRUE : OutOffOVSt;
esc;
next (RdyStartSt) :=
case
(loc = step 170) : !InterlockR;
TRUE : RdyStartSt;
esc;
next (AlSt) :=
case
(loc = step 170) : Al;
TRUE : AlSt;
esc;
next (IOErrorW) :=
case
(loc = step 170) : IOError;
TRUE : IOErrorW;
esc;
next (IOSimuW) :=
case
(loc = step 170) : IOSimu;
TRUE : IOSimuW;
esc;
next (AaMBW) :=
case
(loc = step 170) : (MOrrSt | FoMoSt | SoftLDSt) & ((AUonRS xor MOrrSt) & (AUOHrs1 xor MOrrSt)) & !baAaMBW;
TRUE : AaMBW;
esc;
next (OutOnOV) :=
case
(loc = step 173) : !OutOnOVSt;
(loc = step 174) : OutOnOVSt;
(loc = step 175) : OutOnOVSt;
TRUE : OutOnOV;
esc;
next (OutOHIOV) :=
case
(loc = step 175) : OutOHIOV;
TRUE : OutOHIOV;
esc;
next (OutOVSt_aux) :=
case
(loc = step 177) : TRUE;
(loc = step 179) : FALSE;
TRUE : OutOVSt_aux;
esc;
next (RE_OutOVSt_aux) :=
case
(loc = step 181) : TRUE;
(loc = step 182) : FALSE;
TRUE : RE_OutOVSt_aux;
esc;
next (RE_OutOVSt_aux_old) :=
case
(loc = step 181) : TRUE;
(loc = step 182) : FALSE;
TRUE : RE_OutOVSt_aux_old;
master's thesis

1532 esac;
1533 next (FE_OutOVSt_aux) :=
1534 case
1535 (loc = step 184) : TRUE;
1536 (loc = step 185) : FALSE;
1537 true : FE_OutOVSt_aux;
1538 esac;
1539 next (FE_OutOVSt_aux_old) :=
1540 case
1541 (loc = step 184) : FALSE;
1542 (loc = step 185) : OutOVSt_aux;
1543 true : FE_OutOVSt_aux_old;
1544 esac;
1545 next (PswW_aux) :=
1546 case
1547 (loc = step 187) : TRUE;
1548 (loc = step 189) : FALSE;
1549 true : PswW_aux;
1550 esac;
1551 next (Timer_Warning.Q) :=
1552 case
1553 (loc = step 191) : FALSE;
1554 (loc = step 197) : TRUE;
1555 true : Timer_Warning.Q;
1556 esac;
1557 next (Timer_Warning.ET) :=
1558 case
1559 (loc = step 191) : 0 sd 32_0;
1560 (loc = step 193) : 0 sd 32_0;
1561 (loc = step 196) : Timer_WARNING_1:
1562 (loc = step 197) : PswWb_PWdb;
1563 true : Timer_Warning.ET;
1564 esac;
1565 next (Timer_Warning.running) :=
1566 case
1567 (loc = step 191) : FALSE;
1568 (loc = step 193) : TRUE;
1569 true : Timer_Warning.running;
1570 esac;
1571 next (Timer_Warning.Tstart) :=
1572 case
1573 (loc = step 193) : Timer_WARNING_1;
1574 true : Timer_Warning.Tstart;
1575 esac;
1576 next (PswW) :=
1577 case
1578 (loc = step 198) : Timer_Warning.Q;
1579 true : PswW;
1580 esac;
1581 next (Timer_Warning) :=
1582 case
1583 (loc = step 198) : Timer_Warning.ET;
1584 true : Timer_Warning;
1585 esac;
1586 next (ABW) :=
1587 case
1588 (loc = step 198) : ABW;
1589 true : ABW;
1590 esac;
1591 next (PulseWidth) :=
1592 case
1593 (loc = step 198) : (0 sd 32_150000 * 0 sd 32_100) / (signed (extend (T_CYCLE, 16)) * 0 sd 32_100 + 0 sd 32_1);
1594 true : PulseWidth;
1595 esac;
1596 next (FSlinC) :=
1597 case
1598 (loc = step 200) : FSlinC + 0 sd 16_1;
1599 (loc = step 202) : 0 sd 16_0;
1600 true : FSlinC;
1601 esac;
1602 next (WFuStopISt) :=
1603 case
1604 (loc = step 200) : TRUE;
1605 (loc = step 202) : FfuStopISt;
1606 true : WFuStopISt;
1607 esac;
1608 next (TSlinC) :=
1609 case
1610 (loc = step 204) : TSlinC + 0 sd 16_1;
1611 (loc = step 206) : 0 sd 16_0;
1612 true : TSlinC;
1613 esac;
1614 next (WStopISt) :=
1615 case
1616 (loc = step 204) : TRUE;
1617 (loc = step 206) : TStopISt;
1618 true : WStopISt;
1619 esac;
1620 next (SlinC) :=

115
Master's Thesis

case
  (loc = step 208) : SInc + 0sd16_1;
  (loc = step 210) : 0sd16_0;
  TRUE : SInc;
next(WStartISt) :=
case
  (loc = step 208) : TRUE;
  (loc = step 210) : StartISt;
  TRUE : WStartISt;
  esac;
next(AInc) :=
case
  (loc = step 212) : AInc + 0sd16_1;
  (loc = step 214) : 0sd16_0;
  TRUE : AInc;
  esac;
next(WAISI) :=
case
  (loc = step 212) : TRUE;
  (loc = step 214) : AISI;
  TRUE : WAISI;
  esac;
next(Sreg01b[8]) :=
case
  (loc = step 215) : OnSt;
  TRUE : Sreg01b[8];
  esac;
next(Sreg01b[9]) :=
case
  (loc = step 215) : OffSt;
  TRUE : Sreg01b[9];
  esac;
next(Sreg01b[10]) :=
case
  (loc = step 215) : AUSI;
  TRUE : Sreg01b[10];
  esac;
next(Sreg01b[11]) :=
case
  (loc = step 215) : MoSI;
  TRUE : Sreg01b[11];
  esac;
next(Sreg01b[12]) :=
case
  (loc = step 215) : PoMoSt;
  TRUE : Sreg01b[12];
  esac;
next(Sreg01b[13]) :=
case
  (loc = step 215) : LDSI;
  TRUE : Sreg01b[13];
  esac;
next(Sreg01b[14]) :=
case
  (loc = step 215) : IOErrorW;
  TRUE : Sreg01b[14];
  esac;
next(Sreg01b[15]) :=
case
  (loc = step 215) : IOSimuW;
  TRUE : Sreg01b[15];
  esac;
next(Sreg01b[16]) :=
case
  (loc = step 215) : AAMSw;
  TRUE : Sreg01b[16];
  esac;
next(Sreg01b[17]) :=
case
  (loc = step 215) : PwW;
  TRUE : Sreg01b[17];
  esac;
next(Sreg01b[18]) :=
case
  (loc = step 215) : WStartISt;
  TRUE : Sreg01b[18];
  esac;
next(Sreg01b[19]) :=
case
  (loc = step 215) : WTStopISt;
  TRUE : Sreg01b[19];
  esac;
next(Sreg01b[20]) :=
case
  (loc = step 215) : AIUnAck;
  TRUE : Sreg01b[20];
  esac;
next(Sreg01b[21]) :=
case
  (loc = step 215) : WStartISt;
  TRUE : Sreg01b[21];
  esac;
next(Sreg01b[22]) :=
case
  (loc = step 215) : WTStopISt;
  TRUE : Sreg01b[22];
  esac;
next(Sreg01b[23]) :=
case
  (loc = step 215) : AIUnAck;
  TRUE : Sreg01b[23];
  esac;
next(Sreg01b[24]) :=
case
  (loc = step 215) : WStartISt;
  TRUE : Sreg01b[24];
  esac;
next(Sreg01b[25]) :=
case
  (loc = step 215) : WTStopISt;
  TRUE : Sreg01b[25];
  esac;
next(Sreg01b[26]) :=
case
  (loc = step 215) : AIUnAck;
  TRUE : Sreg01b[26];
  esac;
next(Sreg01b[27]) :=
case
  (loc = step 215) : WStartISt;
  TRUE : Sreg01b[27];
  esac;
next(Sreg01b[28]) :=
case
  (loc = step 215) : WTStopISt;
  TRUE : Sreg01b[28];
  esac;
next(Sreg01b[29]) :=
case
  (loc = step 215) : AIUnAck;
  TRUE : Sreg01b[29];
  esac;
next(Sreg01b[30]) :=
case
  (loc = step 215) : WStartISt;
  TRUE : Sreg01b[30];
  esac;
next(Sreg01b[31]) :=
case
  (loc = step 215) : WTStopISt;
  TRUE : Sreg01b[31];
  esac;
next(Sreg01b[32]) :=
case
  (loc = step 215) : AIUnAck;
  TRUE : Sreg01b[32];
  esac;
next(Sreg01b[33]) :=
case
  (loc = step 215) : WStartISt;
  TRUE : Sreg01b[33];
  esac;
next(Sreg01b[34]) :=
case
  (loc = step 215) : WTStopISt;
  TRUE : Sreg01b[34];
  esac;
next(Sreg01b[35]) :=
case
  (loc = step 215) : AIUnAck;
  TRUE : Sreg01b[35];
  esac;
\begin{verbatim}
1711 (loc = step 215) : AuIhFoMo;
1712 TRUE : Stsreg01b[5];
1713 esac;
1714 next (Stsreg01b[6]) :=
1715 case
1716 (loc = step 215) : WAISt;
1717 TRUE : Stsreg01b[6];
1718 esac;
1719 next (Stsreg01b[7]) :=
1720 case
1721 (loc = step 215) : AuIhMMo;
1722 TRUE : Stsreg01b[7];
1723 esac;
1724 next (Stsreg02b[8]) :=
1725 case
1726 (loc = step 215) : OutOnOVAR;
1727 TRUE : Stsreg02b[8];
1728 esac;
1729 next (Stsreg02b[9]) :=
1730 case
1731 (loc = step 215) : AuOnRSt;
1732 TRUE : Stsreg02b[9];
1733 esac;
1734 next (Stsreg02b[10]) :=
1735 case
1736 (loc = step 215) : MoOnRSt;
1737 TRUE : Stsreg02b[10];
1738 esac;
1739 next (Stsreg02b[11]) :=
1740 case
1741 (loc = step 215) : AuOffRSt;
1742 TRUE : Stsreg02b[11];
1743 esac;
1744 next (Stsreg02b[12]) :=
1745 case
1746 (loc = step 215) : MoOffRSt;
1747 TRUE : Stsreg02b[12];
1748 esac;
1749 next (Stsreg02b[13]) :=
1750 case
1751 (loc = step 215) : HoOnRSt;
1752 TRUE : Stsreg02b[13];
1753 esac;
1754 next (Stsreg02b[14]) :=
1755 case
1756 (loc = step 215) : HoOffRSt;
1757 TRUE : Stsreg02b[14];
1758 esac;
1759 next (Stsreg02b[15]) :=
1760 case
1761 (loc = step 215) : FALSE;
1762 TRUE : Stsreg02b[15];
1763 esac;
1764 next (Stsreg02b[0]) :=
1765 case
1766 (loc = step 215) : FALSE;
1767 TRUE : Stsreg02b[0];
1768 esac;
1769 next (Stsreg02b[1]) :=
1770 case
1771 (loc = step 215) : FALSE;
1772 TRUE : Stsreg02b[1];
1773 esac;
1774 next (Stsreg02b[2]) :=
1775 case
1776 (loc = step 215) : WPuStopSt;
1777 TRUE : Stsreg02b[2];
1778 esac;
1779 next (Stsreg02b[3]) :=
1780 case
1781 (loc = step 215) : EnRstartSt;
1782 TRUE : Stsreg02b[3];
1783 esac;
1784 next (Stsreg02b[4]) :=
1785 case
1786 (loc = step 215) : SoftLDSt;
1787 TRUE : Stsreg02b[4];
1788 esac;
1789 next (Stsreg02b[5]) :=
1790 case
1791 (loc = step 215) : AiBw;
1792 TRUE : Stsreg02b[5];
1793 esac;
1794 next (Stsreg02b[6]) :=
1795 case
1796 (loc = step 215) : OutOHOVS;
1797 TRUE : Stsreg02b[6];
1798 esac;
1799 next (Stsreg02b[7]) :=
1800 case
\end{verbatim}
\texttt{(loc = step 215)} : FALSE;
\texttt{TRUE : Starreg02b[7];
next(RE\textsubscript{\textit{AIUnAck}}) :=
\texttt{case (loc = step 218)} : TRUE;
\texttt{(loc = step 219)} : FALSE;
\texttt{(loc = step 221)} : FALSE;
\texttt{TRUE : RE\textsubscript{\textit{AIUnAck}};}
\texttt{esac;)
\texttt{next(FE\textsubscript{\textit{AIUnAck}}) :=
\texttt{case (loc = step 218)} : FALSE;
\texttt{(loc = step 219)} : TRUE;
\texttt{(loc = step 221)} : FALSE;
\texttt{TRUE : FE\textsubscript{\textit{AIUnAck}};}
\texttt{esac;)
\texttt{next(AlUnAck\textsubscript{old}) :=
\texttt{case (loc = step 220)} : AlUnAck;
\texttt{TRUE : AlUnAck\textsubscript{old};}
\texttt{esac;)
\texttt{next(\_GLOBAL\_TIME) :=
\texttt{case (loc = end)} : \_GLOBAL\_TIME + \text{signed(extend(T\_CYCLE, 16));}
\texttt{TRUE : \_GLOBAL\_TIME;}
\texttt{esac;)
\texttt{next(T\_CYCLE) :=
\texttt{case (loc = start)} : \text{extend(random\_t\_cycle, 8)} \mod 0ud_{16}.05 + 0ud_{16}.5;}
\texttt{TRUE : T\_CYCLE;}
\texttt{esac;)
\texttt{---extra assertion variables
next(sFoMoSt\_aux) :=
\texttt{case (loc = step 1)} : FoMoSt\_aux;
\texttt{TRUE : sFoMoSt\_aux;}
\texttt{esac;)
\texttt{next(sAnAuMoR) :=
\texttt{case (loc = step 1)} : AnAuMoR;
\texttt{TRUE : sAnAuMoR;}
\texttt{esac;)
\texttt{next(sManreg01b8) :=
\texttt{case (loc = step 1)} : Manreg01b[8];}
\texttt{TRUE : sManreg01b8;}
\texttt{esac;)
\texttt{next(sAnhFoMo) :=
\texttt{case (loc = step 1)} : AnhFoMo;
\texttt{TRUE : sAnhFoMo;}
\texttt{esac;)
\texttt{next(sAnhMMo) :=
\texttt{case (loc = step 1)} : AnhMMo;
\texttt{TRUE : sAnhMMo;}
\texttt{esac;)
\texttt{next(sMMoSt\_aux) :=
\texttt{case (loc = step 1)} : MMoSt\_aux;
\texttt{TRUE : sMMoSt\_aux;}
\texttt{esac;)
\texttt{next(pOutOnOV) :=
\texttt{case (loc = nvar)} : OutOnOV;
\texttt{TRUE : pOutOnOV;}
\texttt{esac;)
\texttt{next(pTStopI) :=
\texttt{case (loc = nvar)} : TStopI;
\texttt{TRUE : pTStopI;}
\texttt{esac;)
\texttt{next(pFuStopI) :=
\texttt{case (loc = nvar)} : FuStopI;
\texttt{TRUE : pFuStopI;}
\texttt{esac;)
\texttt{next(pStartI) :=
\texttt{case (loc = nvar)} : StartI;
\texttt{TRUE : pStartI;}
\texttt{esac;)
\texttt{next(pOutOnOVSt) :=
\texttt{case (loc = nvar)} : OutOnOVSt;
\texttt{TRUE : pOutOnOVSt;}}
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, A1, AsOnR, AsOfR, AuAsMoR, AuMIMo, AuAlAck, BaAUMoR, AuIhMMo, AuIhFoMo,
AuAlAck, IhAuMRW, AuRstart;

struct CPC_ONOFFPARAM{
  bool ParRegb[16];
  int PPulseLeb, PWDtb;
};

//VAR_OUTPUT
bool Stsreg01b[16];
bool Stsreg02b[16];
bool OutOnOV = false;
bool OutOffOV = false;
bool MMoSt = false;
bool LDSt = false;
bool SoftLDSt = false;
bool FoMoSt = false;
bool AsOnRS = false;
bool AsOfRS = false;
bool MOIRSt = false;
bool MOIRSt = false;
bool HOnRS = false;
bool HOffRS = false;
bool IOErrorW = false;
bool IOSimuW = false;
bool AuMRW = false;

Listing F.1: CPC.smv
```
```c
bool AlUnAck = false;
bool PosW = false;
bool TStopISt = false;
bool FuStopISt = false;
bool AlSt = false;
bool AlBW = false;
bool EnRstartSt = true;
bool RdyStartSt = false;

// Internal Variables
bool EMAuMoR = false;
bool EMMMoR = false;
bool EMFoMoR = false;
bool EMOnR = false;
bool EMOffR = false;
bool EMAlAckR = false;
bool EStartI = false;
bool ETStopI = false;
bool EFuStopI = false;
bool EA = false;
bool EAuAuMoR = false;
bool EAlUnAck = false;
bool EOutOVStaux = false;
bool EMEnRstartR = false;
bool EOutOVStaux = false;
bool EInterlockR = false;

bool MAuMoRold = false;
bool MMMoRold = false;
bool MFoMoRold = false;
bool MOnRold = false;
bool MOGrdold = false;
bool MAIACKRold = false;
bool AnAnMoRold = false;
bool AuAIackold = false;
bool StartIold = false;
bool TStopIold = false;
bool FuStopIold = false;
bool AAlold = false;
bool AUnAckold = false;
bool MSoftLDR = false;
bool MEnRstartRold = false;
bool EOutOVStauxold = false;
bool EInterlockRold = false;

bool PfPosOn = false;
bool PfPosOn2 = false;
bool PFbOn = false;
bool PFHOF = false;
bool PFpulse = false;
bool PFpulseCste = false;
bool PFoLD = false;
bool PFoLDcmd = false;
bool FAnum = false;
bool POutOf = false;
bool FEnRestart = false;
bool PResetFS = false;
bool OutOnOVS = false;
bool OutOstOVS = false;
bool AAnMoStaux = false;
bool MMMoStaux = false;
bool FoMoStaux = false;
bool SoftLDStaux = false;
bool PulseOn = false;
bool PulseOff = false;
bool PosWaux = false;
bool OutOVStaux = false;
bool fullNotAcknowledged = false;
bool PulseOnR = false;
bool PulseOffR = false;
bool InterlockR = false;

// Time Warning
int TimeWarning;
struct TP TimerPulseOn;
struct TP timerPulseOff;
struct T0 TimerWarning;

double PulseWidth;
```
short FSIinc, TSIinc, Slinc, Alinc;
bool WTStopISt, WStartISt, WASt, WFStopISt;
int _GLOBAL_TIME = 0;

unsigned short T_CYCLE;

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

bool FEDGE(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;
    } else if(_GLOBAL_TIME <= *tpdue) {
        *tpQ = true;
        *tpET = PT - (*tpdue - _GLOBAL_TIME);
    } else {
        *tpQ = false;
        if(in){
            *tpET = PT;
        }
    }
    *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;
    Stsreg01b[1] = false;
    Stsreg02b[0] = 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;
231 Stsreg02b [5] = false;
232 Stsreg01b [6] = false;
233 Stsreg02b [6] = false;
234 Stsreg01b [7] = false;
235 Stsreg02b [7] = false;
236 Stsreg01b [8] = false;
237 Stsreg02b [8] = false;
238 Stsreg01b [9] = false;
239 Stsreg02b [9] = false;
240 Stsreg01b [10] = false;
241 Stsreg02b [10] = false;
244 Stsreg01b [12] = false;
245 Stsreg02b [12] = false;
246 Stsreg01b [13] = false;
247 Stsreg02b [13] = false;
248 Stsreg01b [14] = false;
249 Stsreg02b [14] = false;
250 Stsreg01b [15] = false;
251 Stsreg02b [15] = false;
252 Timer_PulseOn.Q = false;
253 Timer_PulseOn.old_in = false;
254 Timer_PulseOn.due = 0;
255 Timer_PulseOff.Q = false;
256 Timer_PulseOff.due = 0;
257 Timer_Warning.Q = false;
258 Timer_Warning.ET = 0;
259 Timer_Warning.running = false;
260 Timer_Warning.start = 0;
261 int first = true;
262
definitions
263 POonoff.ParRegb[0] = nondet bool();
264 POonoff.ParRegb[1] = nondet bool();
266 POonoff.ParRegb[3] = nondet bool();
270 POonoff.ParRegb[7] = nondet bool();
271 POonoff.ParRegb[8] = nondet bool();
272 POonoff.ParRegb[9] = nondet bool();
273 POonoff.ParRegb[10] = nondet bool();
275 POonoff.ParRegb[12] = nondet bool();
276 POonoff.ParRegb[13] = nondet bool();
277 POonoff.ParRegb[14] = nondet bool();
279 POonoff.PPulseLeb = nondet int();
280 POonoff.PWDb = nondet int();
281
282 while (true) {
283     // extra assertion variables
284     bool pOutOnOV = OutOnOV;
285     bool pTStopI = TStopI;
286     bool pFuStopI = FuStopI;
287     bool pStartI = StartI;
288     bool pOutOnOVSt = OutOnOVSt;
289     bool pMEmSt = MEmSt;
290     bool pManreg01b12 = Manreg01b[12];
291     bool pManreg01b13 = Manreg01b[13];
292     // nondet input
293     HFOn = nondet bool();
294     HFOff = nondet bool();
295     HLD = nondet bool();
296     IOSimu = nondet bool();
297     AIb = nondet bool();
298     Manreg01b[0] = nondet bool();
299     Manreg01b[1] = nondet bool();
300     Manreg01b[2] = nondet bool();
301     Manreg01b[3] = nondet bool();
302     Manreg01b[4] = nondet bool();
303     Manreg01b[5] = nondet bool();
304     Manreg01b[6] = nondet bool();
305     Manreg01b[7] = nondet bool();
306     Manreg01b[8] = nondet bool();
307     Manreg01b[9] = nondet bool();
308     Manreg01b[10] = nondet bool();
309     Manreg01b[11] = nondet bool();
310     Manreg01b[12] = nondet bool();
311     Manreg01b[13] = nondet bool();
312     Manreg01b[14] = nondet bool();
313     Manreg01b[15] = nondet bool();
314 }
HOnR = nondet_bool();
HOffR = nondet_bool();
StartI = nondet_bool();
TStopI = nondet_bool();
Al = nondet_bool();
AuOnR = nondet_bool();
AuOffR = nondet_bool();
AuMoR = nondet_bool();
AuMMo = nondet_bool();
AuFoMo = nondet_bool();
AuIhMoR = nondet_bool();
AuIhMMo = nondet_bool();
AuIhFoMo = nondet_bool();
AuAlAck = nondet_bool();
IhAuMR = 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_MMoR = R_EDGE( Manreg01b[9] ,&MMoR_old);
E_MSoftLDR = R_EDGE( Manreg01b[10] ,&MSoftLDR_old);
E_MOhR = R_EDGE( Manreg01b[11] ,&MOhR_old);
E_MOHIR = R_EDGE( Manreg01b[12] ,&MOHIR_old);
E_MEnRstartR = R_EDGE( Manreg01b[1] ,&MEnRstartR_old);
E_MAIAKRF = R_EDGE( Manreg01b[7] ,&MAIAKRF); 

PFsPosOn = POnOffb. ParRegb[8];
PHFOn = POnOffb. ParRegb[9];
PHFOff = POnOffb. ParRegb[10];

//做强制模式
if ((E HLDD && PHLD) ) {
    //forced mode
    if ((AuMoSt_aux && MMoSt_aux && SoftLDSt_aux)) & E_MPoMoR & !(AuFeMo) {
        AuMoSt_aux = false;
        MMoSt_aux = false;
        FoMoSt_aux = true;
        SoftLDSt_aux = false;
    }
    InterlockR = TStopISt || !FuStopISt | | fullNotAcknowledged | | !EnRstartSt | |
                (StartSt && !POutOff && !OutOV) ||
                (StartSt && POutOff && ((PFsPosOn && OutOVSt_aux) | | !(PFsPosOn && !
                              OutOVSt_aux) ));
    FE_InterlockR = F_EDGE (InterlockR, &FE_InterlockR_old);
}

//mode manager
if ( !(HLDD && PHLD) ) {
    //forced mode
    if ((AuMoSt_aux || MMoSt_aux || SoftLDSt_aux)) & E_MPoMoR & !(AuFeMo) {
        AuMoSt_aux = false;
        MMoSt_aux = false;
        FoMoSt_aux = true;
        SoftLDSt_aux = false;
    }
if ((AuMoSt \_aux || FoMoSt \_aux || SoftLDSt \_aux) && MMoSt \_aux && !(AuIhMMo) ) {
    AuMoSt \_aux = false;
    MMoSt \_aux = true;
    FoMoSt \_aux = false;
    SoftLDSt \_aux = false;
}

// auto mode
if ( (MMoSt \_aux && (E\_MAnMoR || F\_EAnMoR) ) || (FoMoSt \_aux && E\_MAnMoR) ) ||
    (SoftLDSt \_aux && (H\_AnMoR || E\_MAnMoR) ) ) {
    AuMoSt \_aux = false;
    MMoSt \_aux = true;
    FoMoSt \_aux = false;
    SoftLDSt \_aux = false;
}

// Software Local Mode
if ((AuMoSt \_aux || MMoSt \_aux) && E\_MSoLDR && !(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) || (!PHFOn && !PHFOff && !OutOVSt \_aux);

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

// Manual On/Off Request
if ((!E\_MOHR && (MMoSt || SoftLDSt) ) || (AuOFRSt && AuMoSt) || (LDSt && PHLDCmd && !HOnRSt) || (!F\_FuStopI && !P\_False && !POutOff) && EnRstartSt ) {
    MOnRSt = false;
} else if ( ((E\_MOHR && (MMoSt || SoftLDSt) ) || (AuOnRSt && AuMoSt) || (LDSt && PHLDCmd && HOnRSt) && EnRstartSt ) || (E\_FuStopI && PFsPosOn) ){
    MOnRSt = true;
}
else {
    MOnRSt = !MOnRSt;
}

// Local Drive Request
if (H\_OHR) {
    HOnRSt = false;
} else if (H\_OHR) {
    HOnRSt = true;
}
else {
    HOnRSt = !(H\_OHR);  
}
// PULSE REQUEST MANAGER
if (Pulse) {
    if (InterlockR) {
        PulseOnR = (PFsPosOn && ! PFsPosOn2) || (PFsPosOn && PFsPosOn2);
        PulseOffR = (! PFsPosOn && ! PFsPosOn2) || (PFsPosOn && ! PFsPosOn2);
    } else if (FEInterlockR) {
        PulseOnR = false;
        PulseOffR = false;
        updateTP(&TimerPulseOn.Q, &TimerPulseOn.ET, &TimerPulseOn.old.in, &TimerPulseOff.ET, &TimerPulseOff.old.in, &TimerPulseOff.due, false, 0);
    } else if (MOIRSt && (MModSt || FoMoSt || SoftLDSt)) || (AuOnRSt && AuMoSt) ||
        (HOffR && LDSt && PHLDCmd) ) { // Off Request
        PulseOnR = false;
        PulseOffR = true;
        updateTP(&TimerPulseOff.Q, &TimerPulseOff.ET, &TimerPulseOff.old.in, &TimerPulseOn.ET, &TimerPulseOn.due, false, 0);
    }
    if (MOIRSt && (MModSt || FoMoSt || SoftLDSt)) || (AuOnRSt && AuMoSt) ||
        (RHoR && LDSt && PHLDCmd) ) { // On Request
        PulseOnR = true;
        PulseOffR = false;
    }
}
else {
    PulseOnR = false;
    PulseOffR = false;
}

// Pulse functions
updateTP(&TimerPulseOn.Q, &TimerPulseOn.ET, &TimerPulseOn.old.in, &TimerPulseOff.ET, &TimerPulseOff.due, false, 0);
updateTP(&TimerPulseOff.Q, &TimerPulseOff.ET, &TimerPulseOff.old.in, &TimerPulseOn.ET, &TimerPulseOn.due, false, 0);
if (REPulseOn) {
    updateTP(&TimerPulseOff.Q, &TimerPulseOff.ET, &TimerPulseOff.old.in, &TimerPulseOn.ET, &TimerPulseOn.due, false, 0);
}
if (REPulseOff) {
    updateTP(&TimerPulseOn.Q, &TimerPulseOn.ET, &TimerPulseOn.old.in, &TimerPulseOff.ET, &TimerPulseOff.due, false, 0);
}
if (PPulseOn) {
    PulseOn = TimerPulseOn.Q && ! PulseOffR;
    PulseOff = TimerPulseOff.Q && ! PulseOnR;
} else {
    PulseOn = TimerPulseOn.Q && ! PulseOffR && (! PHFOn || (PHFOn && ! HFOff));
    PulseOff = TimerPulseOff.Q && ! PulseOnR && (! PHFOff || (PHFOff && ! HFOff));
}

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

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

// Interlocks / FailSafe
if (InterlockR) {
    if (PPulse && ! PFsPosOn2) {
        OutOnOVSt = PulseOn;
        OutOffOVSt = false;
    } else {
        OutOnOVSt = false;
        OutOffOVSt = PulseOff;
    }
}
if (OutOffOVSt = (PFsPosOn && ! PFsPosOn2) || (PFsPosOn && PFsPosOn2);
    OutOffOVSt = (PFsPosOn && ! PFsPosOn2) || (PFsPosOn && PFsPosOn2);
)
else {
    OutOnOVSt = PulseOn;
    OutOffOVSt = false;
}
else {
    if (InterlockR) {
        OutOnOVSt = PulseOn;
    }
}
// Ready to Start Status
RdyStartSt = ! InterlockR;
578  // Alarms
579  AlSt = Al;
580
581  // SURVEILLANCE
582  // I/O Warning
583  IOErrorW = IOError;
584  IOSimuW = IOSimu;
585
586  // Auto<> Manual Warning
587  AuM = (MMoSt || FmMoSt || SoftLDSt) &&
588                ((AuOnRsSt || MoOnRsSt) && (AuOnRsSt && MoOnRsSt)) || ((AuOffRsSt && MOHRS) && (AuOffRsSt && MOHRS))) && !AaBW;
589
590  // OUTPUT MANAGER & OUTPUT REGISTER
591  if( !POutOff ){
592    if( PPosOn ){
593      OutOnOV = !OutOnOVSt;
594    } else {
595      OutOnOV = OutOnOVSt;
596    }
597    POutOff = OutOnOVSt;
598    POutOn = OutOnOVSt;
599  }
600
601  // Position warning
602  // Reset of the OutOnOVSt
603  if( OutOnOVSt || (PPulse && PulseOnR) ){
604    OutOnOVSt = true;
605  }
606  if( !POutOff ){
607    if( !POutOff ){
608      if( !OutOnOVSt && !POutOff ) || (OutOnOVSt && !POutOff) || (PPulse && PulseOnR) ){
609        OutOnOVSt = false;
610      }
611      REOutOnOVSt = REDGE(OutOnOVSt, REOutOnOVSt, REOutOnOVSt, old);
612      OutOnOVSt = FEDGE(OutOnOVSt, FEOutOnOVSt, FEOutOnOVSt, old);
613      if( !OutOnOVSt || ((PPOff && OffSt) == (PPOff && !OffSt)) || (PPOff && OnSt) )&
614        (PPOff && !POutOff) )
615      }
616    PosWaux = true;
617    if( !POutOff ){
618      if( !POutOff ){
619        if( !POutOff ){
620          if( !OutOnOVSt && !POutOff ) || (OutOnOVSt && !POutOff) || (PPulse && PulseOnR) ){
621            OutOnOVSt = false;
622          }
623          updateTON(&Timer Warning.Q, &Timer Warning.ET, &Timer Warning.running, & Timer Warning.start, PosWaux, POutOn, POutOff, PWDth);
624          PosW = Timer Warning.Q;
625          Time Warning = Timer Warning.ET;
626        }
627      }
628    }
629  // Alarm Blocked Warning
630  ABW = AlB;
631  // Maintain Interlock status 1.5s in Stsreg for PVSS
632  PulseWidth = 1500/T_CYC;
633  if( FuStopISt || FSInc > 0 ){
634    FSInc = FSInc + 1;
635    WFuStopISt = true;
636  }
637  if( TStopISt || TSInc > 0 ){
638    TSInc = TSInc + 1;
639    WTStopISt = true;
640  }
641  if( StartISt || SInc > 0 ){
642    SInc = SInc + 1;
643    WStartISt = true;
644  }
645  if( AlSt || Ainc > 0 ){
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:** $(\neg (s_{FoMoSt} \text{aux} \land \neg s_{AuAuMoR} \land \neg s_{Manreg01b8}) \lor \neg AuMoSt)$
  
  **CTL:** $\forall loc = \text{end} \rightarrow (\neg (s_{FoMoSt} \text{aux} \land \neg s_{AuAuMoR} \land \neg s_{Manreg01b8}) \lor \neg AuMoSt)$
  
  **Result:** false

- **R1-2**
  
  **Assertion:** $(\neg (s_{FoMoSt} \text{aux} \land \neg s_{AuAuMoR} \land \neg s_{Manreg01b8}) \lor \neg s_{AuIhFoMo} \lor \neg s_{AuIhMMo}) \lor \neg AuMoSt)$
  
  **CTL:** $\forall loc = \text{end} \rightarrow (\neg (s_{FoMoSt} \text{aux} \land \neg s_{AuAuMoR} \land \neg s_{Manreg01b8}) \lor \neg s_{AuIhFoMo} \lor \neg s_{AuIhMMo}) \lor \neg AuMoSt)$
  
  **Result:** true

- **R1-3**
  
  **Assertion:** $(\neg (s_{FoMoSt} \text{aux} \land \neg s_{AuAuMoR} \land \neg s_{Manreg01b8}) \lor \neg AuMoSt)$
  
  **CTL:** $\forall loc = \text{end} \rightarrow (\neg (s_{FoMoSt} \text{aux} \land \neg s_{AuAuMoR} \land \neg s_{Manreg01b8}) \lor \neg AuMoSt)$
  
  **Result:** false

- **R1-4**
  
  **Assertion:** $(\neg (s_{FoMoSt} \text{aux} \land \neg s_{AuAuMoR} \land \neg s_{Manreg01b8} \land \neg s_{AuIhFoMo} \land \neg s_{AuIhMMo}) \lor \neg AuMoSt)$
CTL: AG(loc = end -> (!(sFoMoSt_aux & sAuAuMoR & !sManreg01b8 & !sAuIhFoMo & !sAuIhMMo) | !AuMoSt))
Result: false

• R1-5
  Assertion (!sMMoSt_aux & sAuIhMMo & !(sManreg01b8) || !AuMoSt)
  CTL: AG(loc = end -> (!(sMMoSt_aux & !sAuAuMoR & !sManreg01b8) || !AuMoSt))
  Result: false

• R1-6
  Assertion (!AuMoSt) || (sAuMoR || sManreg01b8 || sAuIhMMo || !sMMoSt_aux))
  CTL: AG(loc = end -> (!(AuMoSt) || (sAuMoR | 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.
  Result: false

• R1-11b
  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
  Result: true

• R2-4
  Result: true

• R2-5
  Assertion (!pManreg01b12 & Manreg01b12) & & Manreg01b13) & & MMoSt & & !InterlockR & & !PFsPosOn)
  Result: true

  CTL: AG(loc = end -> (!pManreg01b12 & Manreg01b12) & & Manreg01b13) & & MMoSt & & !InterlockR & & !PFsPosOn)
  Result: true

  CTL: AG(loc = end -> (!pManreg01b12 & Manreg01b12) & & Manreg01b13) & & MMoSt & & !InterlockR & & !PFsPosOn)
  Result: true

  CTL: AG(loc = end -> (!pManreg01b12 & Manreg01b12) & & Manreg01b13) & & MMoSt & & !InterlockR & & !PFsPosOn)
  Result: true

  CTL: AG(loc = end -> (!pManreg01b12 & Manreg01b12) & & Manreg01b13) & & MMoSt & & !InterlockR & & !PFsPosOn)
  Result: true
R2-6

Result: true

  Result: true

  Result: true

R2-7


R2-8


R3-1

• Assert: (!(!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) || !OutOnOV)

R3-2

• Assert: (!(!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) || !OutOnOV)

R3-3

• Assert: (!(!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) || !OutOnOV)

R3-4

• Assert: (!(!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) || !OutOnOV)

R3-5

• Assert: (!(!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) || !OutOnOV)

R3-6

• Assert: (!(!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) || !OutOnOV)

R4-1

• Assert: (!(!(HOnR & !HOffR & LDSt & !InterlockR & !PFsPosOn) || !OutOnOV)

Result: true

129