



Department of Mathematics and Computer Science
Formal System Analysis Research Group

2IM91 – Master's Thesis

Verification of PLC code used at CERN

Petra van den Helder, BSc.

Supervisors:

dr.ir. T.A.C. Willemse

dr. D. Bošnački

May 18, 2016

Abstract

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

Preface

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

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

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

Petra van den Helder
Eindhoven, May 2016

Contents

1	Introduction	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 not be proved by the tools.

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

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

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

2 Preliminaries

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

2.1 Programmable Logic Controller (PLC)

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

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

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

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

2.2 Structured Control Language (SCL)

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

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

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

Additionally, there are some functions integrated in the Step 7 software. These are typically functions that are widely used in SCL programming. They are part of the operating system

and are not loaded as part of the program.

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

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

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

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

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

Tab. 1: Data types in SCL

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

There are a couple of rules for the structure of an SCL program. Called blocks must precede the calling blocks. In a block, the variables must be defined first. Each variable type

gets a subsection, which should contain all variables of that type. There is no fixed order for the subsections.

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

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

```

1 // The main program
2 FUNCTION_BLOCK SimpleExample
3 VAR_INPUT
4     error : BOOL;           // not used
5     toMode1 : BOOL;        // request to switch to model
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.

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

Listing 2.2: Function in SCL

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

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

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

```
1 TYPE ComplexSignal
2 STRUCT
3     out1 : BOOL;
4     out2 : BOOL;
5     remaining : INT;
6     elapsed : INT;
7 END_STRUCT
8 END_TYPE
```

Listing 2.3: User-defined data type in SCL

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

```
1 DATA_BLOCK ModeDB
2 STRUCT
3     mode : INT;
4 END_STRUCT
5 BEGIN
6     mode := -1;
7 END_DATA_BLOCK
```

Listing 2.4: Data block in SCL

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

3 Problem Description

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

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

3.1 Verification at CERN

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

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

An automaton is a tuple $a = (L, T, l_0, V_a, Val_0)$ where $L = \{l_0, l_1, \dots\}$ is a set of locations, T is a set of guarded transitions, l_0 is the initial location, V_a is a set of variables, and 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 `TRUE` to variable `a` in the function. It also has a synchronization `i1!`. This ensures that this transition is synchronized with the transition with `i1?` in the Function block. The other transition (from l_3 to l_4) with synchronization `i2?` has

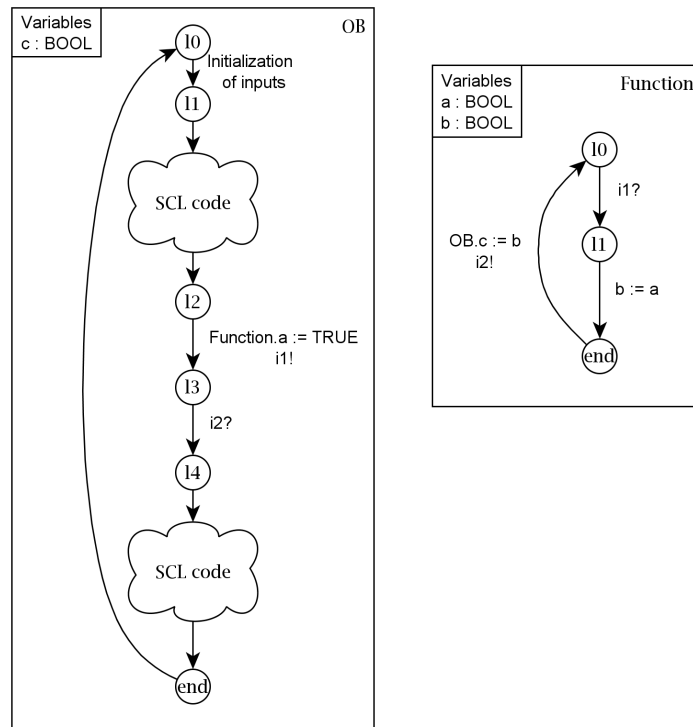


Fig. 1: Example of a function call in the IM

to wait until the function has reached the transition with *i2!*. The transition with *i2!* at the function also assigns a value to the variable *c* in the OB block. This is the return value of the function. The function has now returned to 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
    
```

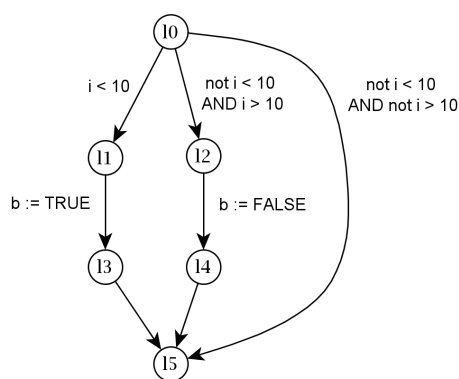


Fig. 2: Example of an IF-statement in the IM

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.

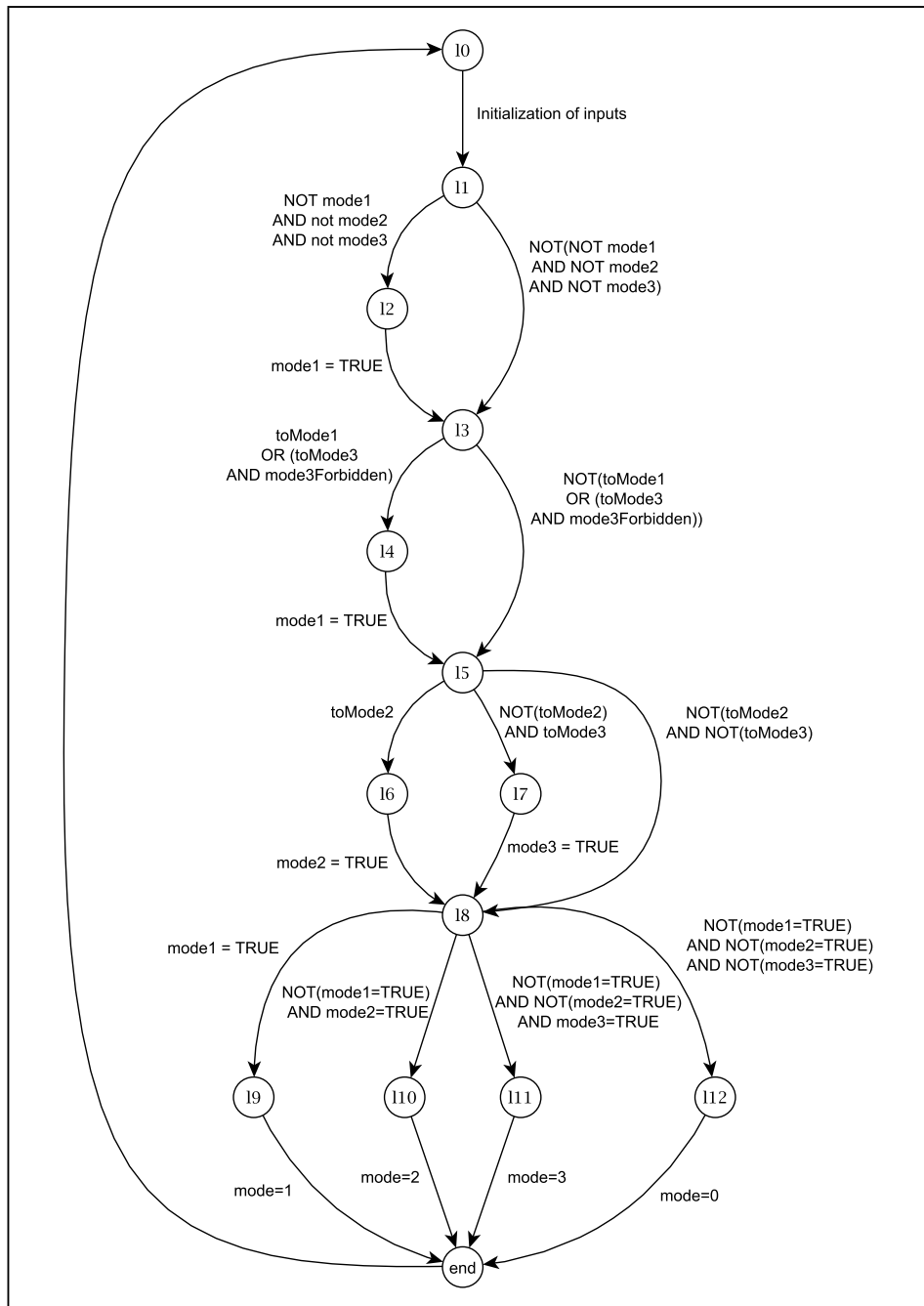


Fig. 3: Intermediate Model of the running example

3.2 Approach

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

conduct a number of experiments with example programs.

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

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

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

- If `out2` is true then `out1` should be true too.
Assertion: `(!out2 || out1)`
Expected result: `true`
- If `signal` is false then `out2` should be false too.
Assertion: `(signal || !out2)`
Expected result: `true`
- `out3.out1` equals `out1`.
Assertion: `(out3.out1 == out1)`
Expected result: `false`
- `out3.elapsed` is 0 when `out1` is false.
Assertion: `((out1 || (out3.elapsed == 0))`
Expected result: `false`

Variable	Data type	Variable type
<i>signal</i>	Boolean	Input variable
<i>out1</i>	Boolean	Output variable
<i>out2</i>	Boolean	Output variable
<i>out3.out1</i>	Boolean	Output variable
<i>out3.elapsed</i>	Integer	Output variable

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

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

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

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

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

- Ease of translation

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

- Results and Running time

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

- Counterexamples

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

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

4 Verification tools on Models

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

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

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

4.1 Spin

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

Spin can be run from the command line as well as from a graphical user interface, iSpin.

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

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

For the unconditional loop and the conditional repetition we use the repetition construct of PROMELA, which is as follows:

```
do
:: option1
:: option2
od
```

For the non-deterministic choices and the deterministic choices we use the select construct of PROMELA, which is as follows:

```
if
:: option1
:: option2
fi;
```

A construct can contain any number of options. An option consists of ':' followed by a sequence of statements. A statement can be an assignment, an assertion, a print statement, a communication statement, or a condition in the form of a logical expression. The first

statement of an option is called the guard. An option is executable if the guard is executable. If the guard is a logical expression it is executable if the expression is satisfied. A guard **else** can be used that is only satisfied when all other options are not executable. If no options are executable the program will block. The repetition construct chooses one of the executable options non-deterministically and executes it. This is repeated until no executable options are available.

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

```
do
  :: program
od
```

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

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

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

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

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

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

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

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

```
1 Variable declaration
2 active proctype go()
3 {
4     do
5         :: Non-deterministic input
6         Program body
7         Properties
8     od
9 }
```

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 mode1 = 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.

<pre>TYPE ComplexSignal STRUCT out1 : BOOL; out2 : BOOL; remaining : INT; elapsed : INT; END_STRUCT END_TYPE</pre>	<pre>typedef ComplexSignal{ bool out1; bool out2; short remaining; short elapsed; };</pre>
--	--

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.

```
1   if
2       :: error = 0
3       :: error = 1
4   fi ;
5   if
6       :: toMode1 = 0
7       :: toMode1 = 1
8   fi ;
```

```

9      if
10     :: toMode2 = 0
11     :: toMode2 = 1
12     fi ;
13     if
14     :: toMode3 = 0
15     :: toMode3 = 1
16     fi ;
17     if
18     :: mode3Forbidden = 0
19     :: mode3Forbidden = 1
20     fi ;

```

Listing 4.3: PROMELA non-deterministic assignments

Program body The following aspects of the SCL code will be discussed: IF-statements, WHILE-loops and function calls.

First we will look at the translation of an IF-statement. The select construct will be used as mentioned before. Consider the example below with on the left side the SCL code and on the right side the translation in PROMELA.

<pre> IF b1 THEN s1 ELSEIF b2 THEN s2 END_IF; </pre>	<pre> if ::b1 -> s1 ::!b1 && b2 -> s2 ::else fi; </pre>
--	---

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.

<pre> WHILE b DO s END_WHILE; </pre>	<pre> do ::b -> s ::else -> break od; </pre>
--	--

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

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

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

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

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

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

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

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

```
1 Variable declaration
2 active proctype go()
3 {
4     do
5     :: Non-deterministic input
6     d_step{
7     Program body
8     }
9     Properties
10    od
11 }
```

Listing 4.4: PROMELA structure with `d_step`

4.2 NuSMV and nuXmv

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

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

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

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

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

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

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

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

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

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

With this construction the variable `a` will get a non-deterministic value in every step, but we only want it to get a new value when the model is at a certain location. We use case distinction to give `a` a non-deterministic value at location 1. At any other location its value will remain the same, as we can see in the following construction:

```
next(a) :=
  case
    (loc = 1) : {TRUE, FALSE};
    TRUE : a;
  esac;
```

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

(11) IF a THEN (12) b := TRUE (11) ELSE (13) b := FALSE (14) END_IF	init(loc) := 11; next(loc) := case (loc = 11) & (a) : 12; (loc = 11) : 13; (loc = 12) : 14; (loc = 13) : 14; esac; next(b) := case (loc = 12) : TRUE; (loc = 13) : FALSE; TRUE : b; esac;
---	--

The `loc` variable starts at location 11. For the conditional choice we use the condition as a guard in the case distinction of the `loc` variable. If the condition holds we go to location 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.

<pre>(11) WHILE i < 10 DO (12) i := i + 1; (11) END_WHILE; (13)</pre>	<pre>init(loc) := 11; next(loc) := case (loc = 11) & (i<10) : 12; (loc = 11) : 13; (loc = 12) : 11; esac; next(i) := case (loc = 12) : i+1; TRUE : i; esac;</pre>
---	---

In the example we can see a while loop that increases the value of i with 1 if it is less than 10. The program starts with $loc = 11$. 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.

<pre>1 MODULE main 2 Variable declaration</pre>	
---	--

```

3 |         Location specification
4 |         Variable initialization
5 |         Non-deterministic input
6 |         Program body
7 | Properties

```

Listing 4.5: SMV structure

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

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

```

1 | VAR
2 |     error : boolean;
3 |     toMode1 : boolean;
4 |     toMode2 : boolean;
5 |     toMode3 : boolean;
6 |     mode3Forbidden : boolean;
7 |     mode1 : boolean;
8 |     mode2 : boolean;
9 |     mode3 : boolean;
10 |    mode : signed word[16];
11 |    loc : {start, step1, step2, step3, end};

```

Listing 4.6: SMV variable declaration

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

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

<pre> TYPE ComplexSignal STRUCT out1 : BOOL; out2 : BOOL; remaining : INT; elapsed : INT; END_STRUCT END_TYPE </pre>	<pre> out3.out1 : boolean; out3.out2 : boolean; out3.remaining : signed word[16]; out3.elapsed : signed word[16]; </pre>
--	--

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

```

1 | init(loc) := start;
2 | next(loc) :=
3 |     case

```

```

4 |         (loc = start)                                     : step 1;
5 |         (loc = step1) & (!mode1 & !mode2 & !mode3)      : step 2;
6 |         (loc = step1)                                     : step 3;
7 |         (loc = step2)                                     : step 3;
8 |         (loc = step3) & (toMode1) | (toMode3 & mode3Forbidden) : step 4;
9 |         (loc = step3)                                     : step 5;
10 |        (loc = step4)                                     : step 5;
11 |        (loc = step5) & (toMode2)                         : step 6;
12 |        (loc = step5) & (!toMode2) & (toMode3)           : step 7;
13 |        (loc = step5)                                     : step 8;
14 |        (loc = step6)                                     : step 8;
15 |        (loc = step7)                                     : step 8;
16 |        (loc = step8) & (mode1)                           : step 9;
17 |        (loc = step8) & (!mode1) & (mode2)               : step 10;
18 |        (loc = step8) & (!mode1) & (!mode2) & (mode3)    : step 11;
19 |        (loc = step8)                                     : step 12;
20 |        (loc = step9)                                     : end;
21 |        (loc = step10)                                    : end;
22 |        (loc = step11)                                    : end;
23 |        (loc = step12)                                    : end;
24 |        (loc = end)                                       : start;
25 |    esac;

```

Listing 4.7: SMV location specification

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

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

```

1 | init(mode1) := FALSE;
2 | init(mode2) := FALSE;
3 | init(mode3) := FALSE;
4 | init(mode) := 0;

```

Listing 4.8: SMV Variable initialization

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

```

1 | next(error) :=
2 |     case
3 |         (loc = start) : {TRUE, FALSE};
4 |         TRUE : error;

```

```

5   esac;
6   next(toMode1):=
7     case
8       (loc = start) : {TRUE, FALSE};
9       TRUE : toMode1;
10  esac;
11  next(toMode2):=
12    case
13      (loc = start) : {TRUE, FALSE};
14      TRUE : toMode2;
15    esac;
16  next(toMode3):=
17    case
18      (loc = start) : {TRUE, FALSE};
19      TRUE : toMode3;
20    esac;
21  next(mode3Forbidden):=
22    case
23      (loc = start) : {TRUE, FALSE};
24      TRUE : mode3Forbidden;
25    esac;

```

Listing 4.9: SMV non-deterministic assignments

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- If `out2` is true then `out1` should be true too.
Logical expression: `(loc = end -> (out2 -> out1))`
- If `signal` is false then `out2` should be false too.
Logical expression: `(loc = end -> (!signal -> !out2))`
- `out3.out1` equals `out1`.
Logical expression: `(loc = end -> (out3.out1 = out1))`
- `out3.elapsed` is 0 when `out1` is false.
Logical expression: `(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))`

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

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

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

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

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

For the invariant properties we change the last command to `check_invar`.

With `nuXmv` we use the new algorithm that uses `ic3` engines. To perform this verification, we use the interactive mode of `nuXmv` with the following sequence of commands:

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

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

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

- **-df** Disable the computation of the set of reachable states. This reduces the computation time because not all reachable states have to be created
- **-dynamic** Enables dynamic reordering of variables, this can reduce the size of the BDDs
- **-coi** Enables cone of influence reduction, this is similar to the reduction CERN implemented. A difference is that this reduction is on the SMV code. This reduction is less

effective than the reduction by CERN. A more detailed description can be found in Section 3.1.

4.3 Experiments and results

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

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

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

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

model improve the running time for most programs. Strangely it made the running time for nuXmv of `Example_while` with TRUE-property 1 worse.

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

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

5 Verification tools on Code

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

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

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

5.1 C

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

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

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

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

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

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

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

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

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

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

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

```
1 #include <stdbool.h>
2 function declarations
3 int main(){
4     Variable declaration and initialization
5     while(true){
6         Non-deterministic input
7         program body
8         properties
9     }
10 }
```

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.

```
1 bool error , toMode1, toMode2, toMode3, mode3Forbidden;
2 bool mode1 = false;
3 bool mode2 = false;
4 bool mode3 = false;
5 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.

<pre> TYPE ComplexSignal STRUCT out1 : BOOL; out2 : BOOL; remaining : INT; elapsed : INT; END_STRUCT END_TYPE </pre>	<pre> struct ComplexSignal{ bool out1; bool out2; short remaining; short elapsed; }; </pre>
--	---

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: `assert(property);`

Verification To perform verification with CBMC we use the following command in the command line:

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

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

5.3 K-Inductor

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

We will take a short look at k-induction, a more complete description can be found in the paper by De Moura et al. [21].

First we will look at the traditional induction. To prove a property p with induction we have to prove the base case and the step case. Let us say that if a property p holds in iteration i then p_i holds. We have to prove the following:

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

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

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

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

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

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

Example 5.1

```
1  int main() {
2      int a = 1;
3      int b = 2;
4      int c = 3;
5      int temp = 0;
6      while(true) {
7          temp = a;
8          a = b;
9          b = c;
10         c = temp;
11         assert(a != b);
12     }
13 }
```

Listing 5.4: Example k-induction

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

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

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

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

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

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

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

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

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

Verification Properties are added in the same way as with the BMC tool of CBMC. To perform verification we use the following command in the command line:

`kinductor --max-k k Example.c` where k is the maximum k for the k-induction.

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

5.4 CBMC Incremental

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

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

```

1  int nondet_int();
2
3  int main(){
4      int a, b, c, temp;
5  #ifdef BASE
6      a = 1;
7      b = 2;
8      c = 3;
9      temp = 0;
10 #endif
11
12     while(true){
13
14 #ifdef STEP
15         a = nondet_int();
16         b = nondet_int();
17         c = nondet_int();
18         temp = nondet_int();

```



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

Listing 5.5: Base case and Step case in CBMC Incremental

- **Base case** The code for the base case is placed between `#ifdef BASE` and `#endif`, see lines 3-8 in the example above. In this part we place the initialization of all variables in the code, which results in a separate declaration and initialization of the variable. This part of the code is placed directly after the declaration of the variables;
- **Step case** The code for the step case is placed between `#ifdef STEP` and `#endif`, see lines 12-18 in the example above. In this part we assign every variable in the code a non-deterministic value. Next we create an assumption using the property with the following statement: `__CPROVER_assume(property);`. This part of the code is placed immediately after the beginning of the main loop of the program;
- **Assertion** The property is also added as assertion at the end of the code, in the same way as with the BMC tool and the K-Inductor tool of CBMC, see line 24 in the example above.

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

```
cbmc Example.c --incremental -DBASE
```

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

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

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

5.5 2LS

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

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

Verification Properties are added in the same way as with CBMC and K-Inductor. This is done with `assert(property);`.

To run a verification with 2LS we use the following command in the command line.

```
2ls Example.c --k-induction --havoc
```

The option `--havoc` removes the loops and function calls from the model. To run the verification with the *kIkI* 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:

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

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

```

1  #include <stdbool.h>
2  function declarations
3  int main(){
4      Variable declaration
5      while(true){
6          Non-deterministic input
7          program body
8          properties
9      }
10     ERROR:
11     return (-1);
12 }
```

Listing 5.6: Structure of a program in CPA-checker

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

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

5.7 SATABS

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

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

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

Verification Properties are added in the same way as with CBMC, K-Inductor, and 2LS. To do this we add the following line of code to the program: `assert(property);`. To run a verification with SATABS, we use the following commands in the command line.

```
SATABS Example.c --modelchecker nusmv
```

The SATABS tool uses NuSMV with the option `-dynamic`. The running times of SATABS might improve if it would use NuSMV with the options `-df -dynamic -coi`, but it seems unable to use these options for NuSMV with SATABS.

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

5.8 Experiments and results

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

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

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

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

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

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

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

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

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

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

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

```

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

```

Listing 5.7: small example

```

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

```

Listing 5.8: the same example with a few changes

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

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

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

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

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

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

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

$$y_{i+1} = x_{i+1} \quad (3)$$

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

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

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

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

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

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

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

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

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 $(\neg \text{false} \vee b_1)$ which is *true* regardless of the value of b_1 .

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

$$a_2 = \begin{cases} \text{if } \neg x_2 & \text{false} \\ \text{if } x_2 \wedge \neg x_1 & \text{false} \\ \text{if } x_2 \wedge x_1 & \text{true} \end{cases} \quad (6)$$

$$b_2 = \begin{cases} \text{if } \neg x_2 & \text{false} \\ \text{if } x_2 \wedge \neg x_1 & \text{true} \\ \text{if } x_2 \wedge x_1 & \text{true} \end{cases} \quad (7)$$

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

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

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

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

$$y_{i+2} = x_{i+2} \quad (10)$$

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

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

$$y_{i+3} = x_{i+3} \quad (13)$$

We can write the equations for values of a and b in formulas. From equations 8, 9, 11, and 12 we get the following formulas.

$$a_{i+2} \iff (x_{i+2} \wedge x_{i+1})$$

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

$$a_{i+3} \iff (x_{i+3} \wedge x_{i+2})$$

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

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

$$\begin{aligned} & (\neg a_{i+2} \vee b_{i+2}) \\ \iff & \neg((x_{i+2} \wedge x_{i+1}) \vee ((x_{i+2} \wedge \neg x_{i+1}) \vee (x_{i+2} \wedge x_{i+1} \wedge \neg y_i \wedge \neg b_i) \vee (x_{i+2} \wedge x_{i+1} \wedge y_i \wedge b_i))) \\ \iff & (\neg x_{i+2} \vee \neg x_{i+1} \vee \neg x_{i+1} \vee (\neg y_i \wedge \neg b_i) \vee (y_i \wedge b_i)) \end{aligned}$$

Likewise for $(\neg a_{i+3} \vee b_{i+3})$.

$$\begin{aligned} & (\neg a_{i+3} \vee b_{i+3}) \\ \iff & \neg((x_{i+3} \wedge x_{i+2}) \vee ((x_{i+3} \wedge \neg x_{i+2}) \vee (x_{i+3} \wedge x_{i+2} \wedge \neg x_{i+1}) \vee (x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge \neg y_i \wedge \neg b_i) \vee (x_{i+3} \wedge x_{i+2} \wedge x_{i+1} \wedge y_i \wedge b_i))) \\ \iff & \end{aligned}$$

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

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

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

```

1 #include <stdbool.h>
2
3 int main(){
4     int x = 0;
5
6     while(true){
7         if(x > 1000) {
8             x = 1000;
9         }
10        x = x + 2;
11
12        assert(x != 3);
13    }
14 }
```

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 *kIkI* algorithm. We have strengthened the property to $x \geq 0 \wedge x \neq 3$. 2LS is able to prove this property with k-induction, while K-Inductor and CBMC Incremental still cannot. When we replace the property by $x \geq 0$ CBMC Incremental can also prove the property, but K-Inductor still is not able to do this.

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

6 Case Study

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

6.1 The CPC program

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

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

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

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

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

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

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

```
1 FUNCTION_BLOCK TP  
2 VAR_INPUT  
3     PT : TIME;  
4 END_VAR  
5  
6 VAR_IN_OUT  
7     IN : BOOL;  
8 END_VAR
```

```

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

```

Listing 6.1: FUNCTION BLOCK in SCL

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

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

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

6.2 Translation into SMV

We have made a translation of the CPC program into SMV in the same way as described in Section 4.2. The full program can be found in the Appendix F. This translation contains 1912 lines of code and 224 values of the location variable. The types that we have not yet seen in

that translation are translated as follows. The combination of a variable of type `WORD` and a variable of type `ARRAY` in SCL, as seen earlier in this section, can be translated to a variable of type `array` in SMV. To make an array of 16 boolean variables `array 0..15 of boolean` is used. Variables of the type `TIME` and type `REAL` in SCL are represented by variables of the type `signed word[32]` in SMV and the variable of the type `UINT` in SCL is represented by a variable of the type `unsigned word[16]` in SMV.

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

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

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

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

6.3 Translation into C

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

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

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

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

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

```
struct TP Timer_PulseOn;
```

```

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

```

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

6.4 Properties

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

To keep track of these values we have chosen to add some additional variables. Variables that are used to model a variable at the beginning of the current cycle or at the end of the previous cycle have a prefix. We have chosen not to use `old` because this has already been used in the other variables. Instead we have chosen to use an `'s'` as a prefix for the 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 $kI\bar{k}I$ algorithm has a significant larger running time than the k -induction algorithm for a number of FALSE-properties. While NuSMV and nuXmv had better running times on the Example programs, only the Invariant verification of NuSMV could refute a few properties within the given time.

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

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

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

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

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

7 Conclusion

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

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

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

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

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

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

7.1 Future work

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

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

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

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

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

References

- [1] CPC program. <http://cern.ch/unicos/>. Accessed: 2016-05-13.
- [2] C. Baier and J. Katoen. *Principles of model checking*. MIT Press, 2008.
- [3] H. Barbosa and D. Déharbe. Formal verification of PLC programs using the B method. In *Abstract State Machines, Alloy, B, VDM, and Z - Third International Conference, ABZ 2012, Pisa, Italy, June 18-21, 2012. Proceedings*, pages 353–356, 2012.
- [4] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, pages 504–518. Springer, 2007.
- [5] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Computer Aided Verification*, pages 184–190. Springer, 2011.
- [6] D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design*, pages 189–198. FMCAD Inc, 2010.
- [7] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic Model Checking Using SAT Procedures Instead of BDDs. In *Proceedings of the 36th Annual ACM/IEEE Design Automation Conference, DAC '99*, pages 317–320, New York, NY, USA, 1999. ACM.
- [8] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, pages 193–207, 1999.
- [9] W. Bolton. *Programmable logic controllers*. Newnes, 2015.
- [10] M. Brain, S. Joshi, D. Kroening, and P. Schrammel. *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, chapter Safety Verification and Refutation by k-Invariants and k-Induction, pages 145–161. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.
- [11] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.
- [12] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta. The nuXmv Symbolic Model Checker. In A. Biere and R. Bloem, editors, *CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.
- [13] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Computer Aided Verification*, pages 495–499. Springer, 1999.
- [14] E. M. Clarke, E. A. Emerson, and J. Sifakis. Model checking: algorithmic verification and debugging. *Communications of the ACM*, 52(11):74–84, 2009.

-
- [15] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
- [16] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.
- [17] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 168–176, 2004.
- [18] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.
- [19] S. Cranen, J. F. Groote, J. J. A. Keiren, F. P. M. Stappers, E. P. de Vink, W. Wesselink, and T. A. C. Willemse. *Tools and Algorithms for the Construction and Analysis of Systems: 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, chapter An Overview of the mCRL2 Toolset and Its Recent Advances, pages 199–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.
- [20] D. Darvas. Personal e-mail communication, 2015.
- [21] L. De Moura, H. Rueß, and M. Sorea. Bounded model checking and induction: From refutation to verification. *Lecture notes in computer science*, pages 14–26, 2003.
- [22] A. F. Donaldson, L. Haller, D. Kroening, and P. Rümmer. Software verification using k-induction. In *Static Analysis*, pages 351–368. Springer, 2011.
- [23] B. Fernández Adiego. *Bringing Automated Formal Verification to PLC Program Development*. PhD thesis, University of Oviedo, 2014.
- [24] B. Fernandez Adiego, V. Gonzalez Suarez, J. Blech, E. Blanco Vinuela, D. Darvas, and J. Tournier. Modelling and formal verification of timing aspects in large PLC programs. Technical report, 2014.
- [25] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003. Addison-Wesley Reading, 2004.
- [26] F. Lerda and W. Visser. Addressing Dynamic Issues of Program Model Checking. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 80–102, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
- [27] M. Meulen. Verification of PLC source code using propositional logic. Master's thesis, Technical University of Eindhoven, 2010.
- [28] S. C. Park, C. M. Park, G. Wang, J. Kwak, and S. Yeo. Plcstudio: Simulation based PLC code verification. In *Proceedings of the 2008 Winter Simulation Conference, Global Gateway to Discovery, WSC 2008, InterContinental Hotel, Miami, Florida, USA, December 7-10, 2008*, pages 222–228, 2008.

- [29] O. Pavlovic, R. Pinger, and M. Kollmann. Automated formal verification of PLC programs written in IL. In *Conference on Automated Deduction (CADE)*, pages 152–163, 2007.
- [30] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351. Springer, 1982.
- [31] M. Rausch and B. H. Krogh. Formal verification of PLC programs. In *American Control Conference, 1998. Proceedings of the 1998*, volume 1, pages 234–238. IEEE, 1998.
- [32] P. Schrammel, D. Kroening, M. Brain, R. Martins, T. Teige, and T. Bienmüller. Incremental Bounded Model Checking for Embedded Software (extended version). *CoRR*, abs/1409.5872, 2014.
- [33] S. SIMATIC. Structured Control Language (SCL) for S7-300/S7-400 Programming Manual, 1998.
- [34] F. Somenzi and A. R. Bradley. IC3: where monolithic and incremental meet. In *FMCAD*, pages 3–8, 2011.
- [35] M. Stokely, S. Chaki, and J. Ouaknine. Parallel assignments in software model checking. *Electronic Notes in Theoretical Computer Science*, 157(1):77–94, 2006.

Appendices

A Example programs in SCL

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

```

1 // Type definition
2 TYPE ComplexSignal
3 STRUCT
4     out1 : BOOL;
5     out2 : BOOL;
6     remaining : INT;
7     elapsed : INT;
8 END_STRUCT
9 END_TYPE
10
11 // The function block to be verified
12 FUNCTION_BLOCK ComplexExample
13 CONST
14     cntr_max := 5;
15 END_CONST
16 VAR_INPUT
17     signal : BOOL;           // input signal
18     error : BOOL;           // not used (but it happens that we have non-used variables)
19
20     toMode1 : BOOL;         // request to switch to mode1
21     toMode2 : BOOL;         // request to switch to mode2
22     toMode3 : BOOL;         // request to switch to mode3
23     mode3Forbidden : BOOL; // if it is true, it is forbidden to be in mode3
24 END_VAR
25 VAR
26     signal_old : BOOL := FALSE; // signal value from the last cycle
27     cntr : INT := 0;           // counter to delay the out2 signal
28
29     mode1 : BOOL;           // true if the block is in mode1
30     mode2 : BOOL;           // true if the block is in mode2
31     mode3 : BOOL;           // true if the block is in mode3
32 END_VAR
33 VAR_TEMP
34     edge_signal : BOOL;     // rising edge of out1
35 END_VAR
36 VAR_OUTPUT
37     out1 : BOOL;           // out1 is true if the signal is true
38     out2 : BOOL;           // out2 is true if the signal is true AND out1 was true for
39                             // 'cntr_max' cycles
40     out3 : ComplexSignal;
41 END_VAR
42 BEGIN
43     (* Signal handling *)
44     edge_signal := R_EDGE(new := signal, old := signal_old);
45
46     IF NOT signal THEN
47         // outputs are false if the signal is false
48         out1 := FALSE;
49         out2 := FALSE;
50         cntr := 0;
51     ELSIF edge_signal THEN
52         // if the signal has a rising edge, out1 should be true
53         out1 := NOT out1;
54         out3.out1 := out1;
55     ELSE
56         cntr := cntr + 1;
57         IF cntr > cntr_max AND signal THEN
58             out2 := TRUE;
59         ELSE
60             out2 := FALSE;
61         END_IF;
62         out3.out1 := out1;
63         out3.out2 := out2;
64         out3.remaining := cntr_max - cntr;
65         out3.elapsed := cntr;
66     END_IF;
67
68     (* ----- *)
69
70     (* Operation mode handling *)
71     IF not mode1 and not mode2 and not mode3 THEN
72         mode1 := TRUE;
73     END_IF;
74
75     IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
76         mode1 := TRUE;
77     END_IF;
78     IF toMode2 THEN
79         mode2 := TRUE;

```

```

80     ELSIF toMode3 THEN
81         mode3 := TRUE;
82     END_IF;
83
84     IF model THEN
85         ModeDB.mode := 1;
86     ELSIF mode2 THEN
87         ModeDB.mode := 2;
88     ELSIF mode3 THEN
89         ModeDB.mode := 3;
90     ELSE
91         ModeDB.mode := 0;
92     END_IF;
93 END_FUNCTION_BLOCK
94
95 // Global data storage
96 DATA_BLOCK ModeDB
97 STRUCT
98     mode : INT;
99 END_STRUCT
100 BEGIN
101     mode := -1; // default value for the mode variable in the data block
102 END_DATA_BLOCK
103
104 // Helper function to determine the rising edge on a signal.
105 FUNCTION R_EDGE : BOOL
106     VAR_INPUT
107         new : BOOL;
108     END_VAR
109     VAR_IN_OUT
110         old : BOOL;
111     END_VAR
112
113     BEGIN
114         IF (new = true AND old = false) THEN
115             R_EDGE := true;
116             old := true;
117         ELSE R_EDGE := false;
118             old := new;
119         END_IF;
120 END_FUNCTION

```

Listing A.1: Example.SCL

```

1 // Type definition
2 TYPE ComplexSignal
3 STRUCT
4     out1 : BOOL;
5     out2 : BOOL;
6     remaining : INT;
7     elapsed : INT;
8 END_STRUCT
9 END_TYPE
10
11 // The function block to be verified
12 FUNCTION_BLOCK ComplexExample
13 CONST
14     cntr_max := 5;
15 END_CONST
16 VAR_INPUT
17     signal : BOOL; // input signal
18     error : BOOL; // not used (but it happens that we have non-used variables)
19
20     toMode1 : BOOL; // request to switch to mode1
21     toMode2 : BOOL; // request to switch to mode2
22     toMode3 : BOOL; // request to switch to mode3
23     mode3Forbidden : BOOL; // if it is true, it is forbidden to be in mode3
24 END_VAR
25 VAR
26     signal_old : BOOL := FALSE; // signal value from the last cycle
27     cntr : INT := 0; // counter to delay the out2 signal
28     cntr2 : INT := 0;
29
30     mode1 : BOOL; // true if the block is in mode1
31     mode2 : BOOL; // true if the block is in mode2
32     mode3 : BOOL; // true if the block is in mode3
33 END_VAR
34 VAR_TEMP
35     edge_signal : BOOL; // rising edge of out1
36 END_VAR
37 VAR_OUTPUT
38     out1 : BOOL; // out1 is true if the signal is true
39     out2 : BOOL; // out2 is true if the signal is true AND out1 was true for
40         'cntr_max' cycles
41     out3 : ComplexSignal;
42 END_VAR
43 BEGIN

```

```

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      ELSIF edge_signal THEN
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
65          IF cntr2 > cntr_max AND signal THEN
66              out2 := TRUE;
67          ELSE
68              out2 := FALSE;
69          END_IF;
70          out3.out1 := out1;
71          out3.out2 := out2;
72          out3.remaining := cntr_max - cntr;
73          out3.elapsed := cntr;
74      END_IF;
75
76      (* ----- *)
77
78      (* Operation mode handling *)
79      IF not model and not mode2 and not mode3 THEN
80          model := TRUE;
81      END_IF;
82
83      IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
84          model := TRUE;
85      END_IF;
86      IF toMode2 THEN
87          mode2 := TRUE;
88      ELSIF toMode3 THEN
89          mode3 := TRUE;
90      END_IF;
91
92      IF model THEN
93          ModeDB.mode := 1;
94      ELSIF mode2 THEN
95          ModeDB.mode := 2;
96      ELSIF mode3 THEN
97          ModeDB.mode := 3;
98      ELSE
99          ModeDB.mode := 0;
100     END_IF;
101 END_FUNCTION_BLOCK
102
103 // Global data storage
104 DATA_BLOCK ModeDB
105 STRUCT
106     mode : INT;
107 END_STRUCT
108 BEGIN
109     mode := -1; // default value for the mode variable in the data block
110 END_DATA_BLOCK
111
112 // Helper function to determine the rising edge on a signal.
113 FUNCTION R_EDGE : BOOL
114     VAR_INPUT
115         new : BOOL;
116     END_VAR
117     VAR_IN_OUT
118         old : BOOL;
119     END_VAR
120
121     BEGIN
122         IF (new = true AND old = false) THEN
123             R_EDGE := true;
124             old := true;
125         ELSE R_EDGE := false;
126             old := new;
127         END_IF;
128     END_FUNCTION

```

Listing A.2: Example_int.SCL


```

1 // Type definition
2 TYPE ComplexSignal
3 STRUCT
4     out1 : BOOL;
5     out2 : BOOL;
6     remaining : INT;
7     elapsed : INT;
8 END_STRUCT
9 END_TYPE
10
11 // The function block to be verified
12 FUNCTION_BLOCK ComplexExample
13 CONST
14     ctr_max := 5;
15 END_CONST
16 VAR_INPUT
17     signal : BOOL;           // input signal
18     error : BOOL;           // not used (but it happens that we have non-used variables)
19
20     toMode1 : BOOL;         // request to switch to mode1
21     toMode2 : BOOL;         // request to switch to mode2
22     toMode3 : BOOL;         // request to switch to mode3
23     mode3Forbidden : BOOL; // if it is true, it is forbidden to be in mode3
24 END_VAR
25 VAR
26     signal_old : BOOL := FALSE; // signal value from the last cycle
27     ctr : INT := 0;             // counter to delay the out2 signal
28     ctr2 : INT := 0;
29
30     mode1 : BOOL;           // true if the block is in mode1
31     mode2 : BOOL;           // true if the block is in mode2
32     mode3 : BOOL;           // true if the block is in mode3
33 END_VAR
34 VAR_TEMP
35     edge_signal : BOOL;     // rising edge of out1
36 END_VAR
37 VAR_OUTPUT
38     out1 : BOOL;            // out1 is true if the signal is true
39     out2 : BOOL;            // out2 is true if the signal is true AND out1 was true for
40                             // 'ctr_max' cycles
41     out3 : ComplexSignal;
42 END_VAR
43 BEGIN
44     (* Signal handling *)
45     edge_signal := REDGE(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         ctr := 0;
52     ELSIF edge_signal THEN
53         // if the signal has a rising edge, out1 should be true
54         out1 := NOT out1;
55         out3.out1 := out1;
56     ELSE
57         ctr := ctr + 1;
58         // --- ADDITIONAL PART
59         WHILE ctr > ctr_max DO
60             ctr := ctr - 1;
61             ctr2 := ctr2 + 1;
62         END_WHILE;
63         // ---
64
65         IF ctr2 > ctr_max AND signal THEN
66             out2 := TRUE;
67         ELSE
68             out2 := FALSE;
69         END_IF;
70         out3.out1 := out1;
71         out3.out2 := out2;
72         out3.remaining := ctr_max - ctr;
73         out3.elapsed := ctr;
74     END_IF;
75
76     (* ----- *)
77
78     (* Operation mode handling *)
79     IF not mode1 and not mode2 and not mode3 THEN
80         mode1 := TRUE;
81     END_IF;
82
83     IF toMode1 OR (toMode3 AND mode3Forbidden) THEN
84         mode1 := TRUE;
85     END_IF;
86     IF toMode2 THEN
87         mode2 := TRUE;
88     ELSIF toMode3 THEN

```

```

89         mode3 := TRUE;
90     END_IF;
91
92     IF mode1 THEN
93         ModeDB.mode := 1;
94     ELSIF mode2 THEN
95         ModeDB.mode := 2;
96     ELSIF mode3 THEN
97         ModeDB.mode := 3;
98     ELSE
99         ModeDB.mode := 0;
100    END_IF;
101 END_FUNCTION_BLOCK
102
103 // Global data storage
104 DATA_BLOCK ModeDB
105 STRUCT
106     mode : INT;
107 END_STRUCT
108 BEGIN
109     mode := -1; // default value for the mode variable in the data block
110 END_DATA_BLOCK
111
112 // Helper function to determine the rising edge on a signal.
113 FUNCTION R_EDGE : BOOL
114     VAR_INPUT
115         new : BOOL;
116     END_VAR
117     VAR_IN_OUT
118         old : BOOL;
119     END_VAR
120
121     BEGIN
122         IF (new = true AND old = false) THEN
123             R_EDGE := true;
124             old := true;
125         ELSE R_EDGE := false;
126             old := new;
127         END_IF;
128 END_FUNCTION

```

Listing A.3: Example_while.SCL

B Example programs in PROMELA

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

```

1  typedef ComplexSignal{
2      bool out1;
3      bool out2;
4      short remaining;
5      short elapsed;
6  };
7  bool error, toModel, toMode2, toMode3, mode3Forbidden;
8  bool mode1 = 0;
9  bool mode2 = 0;
10 bool mode3 = 0;
11 bool signal;
12 bool signal_old = 0;
13 bool edge_signal;
14 bool out1, out2;
15 int mode = 0;
16 int cntr_max = 5;
17 short cntr = 0;
18 ComplexSignal out3;
19
20 active proctype go()
21 {
22     do
23     :: if
24         :: error = 0
25         :: error = 1
26     fi;
27     if
28         :: toModel = 0
29         :: toModel = 1
30     fi;
31     if
32         :: toMode2 = 0
33         :: toMode2 = 1
34     fi;
35     if

```

```

36     :: toMode3 = 0
37     :: toMode3 = 1
38   fi;
39   if
40     :: mode3Forbidden = 0
41     :: mode3Forbidden = 1
42   fi;
43   if
44     :: signal = 0
45     :: signal = 1
46   fi;
47
48   d_step{ if
49     :: signal && !signal_old ->      edge_signal = 1;
50                                     signal_old = 1;
51     :: else ->                       edge_signal = 0;
52                                     signal_old = signal;
53   fi;
54   if
55     :: !signal ->      out1 = 0;
56                       out2 = 0;
57                       cntr = 0
58     :: signal && edge_signal -> out1 = !out1;
59                               out3.out1 = out1
60     :: else -> cntr = cntr + 1; if
61                                     :: (cntr > cntr_max) && signal -> out2 = 1
62                                     :: else -> out2 = 0
63                                     fi;
64                                     out3.out1 = out1;
65                                     out3.out2 = out2;
66                                     out3.remaining = cntr_max - cntr;
67                                     out3.elapsed = cntr
68   fi;
69   if
70     :: !mode1 && !mode2 && !mode3 -> mode1 = 1
71     :: else
72   fi;
73   if
74     :: toMode1 || (toMode3 && mode3Forbidden) -> mode1 = 1
75     :: else
76   fi;
77   if
78     :: toMode2 -> mode2 = 1
79     :: toMode3 -> mode3 = 1
80     :: else
81   fi;
82   if
83     :: mode1 -> mode = 1
84     :: !mode1 && mode2 -> mode = 2
85     :: !mode1 && !mode2 && mode3 -> mode = 3
86     :: else -> mode = 0
87   fi;}
88
89   assert (!out2 || out1)
90   assert (signal || !out2)
91   assert (out3.out1 == out1)
92   assert (out1 || (out3.elapsed == 0))
93
94   od
95 }

```

Listing B.1: Example.pml

```

1  typedef ComplexSignal{
2     bool out1;
3     short elapsed;
4 };
5  bool signal;
6  bool signal_old = 0;
7  bool edge_signal;
8  bool out1, out2;
9  int cntr_max = 5;
10 short cntr = 0;
11 ComplexSignal out3;
12
13 active proctype go()
14 {
15     do
16     :: if
17       :: signal = 0
18       :: signal = 1
19     fi;
20
21     d_step{ if
22       :: (signal==1) && (signal_old==0) ->      edge_signal = 1;
23                                               signal_old = 1;
24       :: else ->                               edge_signal = 0;
25                                               signal_old = signal;

```

```

26     fi;
27     if
28         :: !signal ->   out1 = 0;
29                       out2 = 0;
30                       cntr = 0
31         :: signal && edge_signal -> out1 = !out1;
32                                       out3.out1 = out1
33         :: else -> cntr = cntr + 1; if
34                                       :: (cntr > cntr_max) && signal -> out2 = 1
35                                       :: else -> out2 = 0
36                                       fi;
37                                       out3.out1 = out1;
38                                       out3.elapsed = cntr
39     fi;}
40
41     assert (!out2 || out1)
42     assert (signal || !out2)
43     assert (out3.out1 == out1)
44     assert (out1 || (out3.elapsed == 0))
45
46     od
47 }

```

Listing B.2: Reduced version of Example.pml

```

1  typedef ComplexSignal{
2      bool out1;
3      bool out2;
4      short remaining;
5      short elapsed;
6  };
7  bool error, toMode1, toMode2, toMode3, mode3Forbidden;
8  bool model = 0;
9  bool mode2 = 0;
10 bool mode3 = 0;
11 bool signal;
12 bool signal_old = 0;
13 bool edge_signal;
14 bool out1, out2;
15 int mode = 0;
16 int cntr_max = 5;
17 short cntr = 0;
18 short cntr2 = 0;
19 ComplexSignal out3;
20
21 active proctype go()
22 {
23     do
24     :: if
25     :: error = 0
26     :: error = 1
27     fi;
28     if
29     :: toMode1 = 0
30     :: toMode1 = 1
31     fi;
32     if
33     :: toMode2 = 0
34     :: toMode2 = 1
35     fi;
36     if
37     :: toMode3 = 0
38     :: toMode3 = 1
39     fi;
40     if
41     :: mode3Forbidden = 0
42     :: mode3Forbidden = 1
43     fi;
44     if
45     :: signal = 0
46     :: signal = 1
47     fi;
48
49     d_step{if
50     :: (signal==1) && (signal_old==0) ->   edge_signal = 1;
51                                       signal_old = 1;
52     :: else ->                           edge_signal = 0;
53                                       signal_old = signal;
54     fi;
55     if
56     :: !signal ->   out1 = 0;
57                       out2 = 0;
58                       cntr = 0
59     :: signal && edge_signal -> out1 = !out1;
60                                       out3.out1 = out1
61     :: else -> cntr = cntr + 1; if
62                                       :: cntr > cntr_max -> cntr = 0; cntr2 = cntr2 + 1
63                                       :: else

```

```

64         fi;
65         if
66             :: (cntr2 > cntr_max) && signal -> out2 = 1
67             :: else -> out2 = 0
68         fi;
69         out3.out1 = out1;
70         out3.out2 = out2;
71         out3.remaining = cntr_max - cntr;
72         out3.elapsed = cntr
73     fi;
74     if
75         :: !mode1 && !mode2 && !mode3 -> mode1 = 1
76         :: else
77     fi;
78     if
79         :: toMode1 || (toMode3 && mode3Forbidden) -> mode1 = 1
80         :: else
81     fi;
82     if
83         :: toMode2 -> mode2 = 1
84         :: toMode3 -> mode3 = 1
85         :: else
86     fi;
87     if
88         :: mode1 -> mode = 1
89         :: !mode1 && mode2 -> mode = 2
90         :: !mode1 && !mode2 && mode3 -> mode = 3
91         :: else -> mode = 0
92     fi;}
93
94     assert (!out2 || out1)
95     assert (signal || !out2)
96     assert (out3.out1 == out1)
97     assert (out1 || (out3.elapsed == 0))
98
99     od
100 }

```

Listing B.3: Example_int.pml

```

1  typedef ComplexSignal{
2      bool out1;
3      short elapsed;
4  };
5  bool signal;
6  bool signal_old = 0;
7  bool edge_signal;
8  bool out1, out2;
9  int cntr_max = 5;
10 short cntr = 0;
11 short cntr2 = 0;
12 ComplexSignal out3;
13
14 active proctype go()
15 {
16     do
17     :: if
18         :: signal = 0
19         :: signal = 1
20     fi;
21
22     d_step{if
23         :: (signal==1) && (signal_old==0) -> edge_signal = 1;
24         :: else -> signal_old = 1;
25         :: else -> edge_signal = 0;
26         :: else -> signal_old = signal;
27     fi;
28     if
29         :: !signal -> out1 = 0;
30         :: !signal -> out2 = 0;
31         :: !signal -> cntr = 0
32         :: signal && edge_signal -> out1 = !out1;
33         :: signal && edge_signal -> out3.out1 = out1
34         :: else -> cntr = cntr + 1; if
35             :: cntr > cntr_max -> cntr = 0; cntr2 = cntr2 + 1
36             :: else
37         fi;
38         if
39             :: (cntr2 > cntr_max) && signal -> out2 = 1
40             :: else -> out2 = 0
41         fi;
42         out3.out1 = out1;
43         out3.elapsed = cntr
44     fi;}
45
46     assert (!out2 || out1)
47     assert (signal || !out2)
48     assert (out3.out1 == out1)

```

```

49     assert (out1 || (out3.elapsed == 0))
50
51     od
52 }

```

Listing B.4: Reduced version of Example_int.pml

```

1  typedef ComplexSignal{
2      bool out1;
3      bool out2;
4      short remaining;
5      short elapsed;
6  };
7  bool error, toMode1, toMode2, toMode3, mode3Forbidden;
8  bool mode1 = 0;
9  bool mode2 = 0;
10 bool mode3 = 0;
11 bool signal;
12 bool signal_old = 0;
13 bool edge_signal;
14 bool out1, out2;
15 int mode = 0;
16 int cntr_max = 5;
17 short cntr = 0;
18 short cntr2 = 0;
19 ComplexSignal out3;
20
21 active proctype go()
22 {
23     do
24     :: if
25         :: error = 0
26         :: error = 1
27     fi;
28     if
29         :: toMode1 = 0
30         :: toMode1 = 1
31     fi;
32     if
33         :: toMode2 = 0
34         :: toMode2 = 1
35     fi;
36     if
37         :: toMode3 = 0
38         :: toMode3 = 1
39     fi;
40     if
41         :: mode3Forbidden = 0
42         :: mode3Forbidden = 1
43     fi;
44     if
45         :: signal = 0
46         :: signal = 1
47     fi;
48
49     d_step{if
50         :: (signal==1) && (signal_old==0) -> edge_signal = 1;
51         signal_old = 1;
52         :: !signal || signal_old -> edge_signal = 0;
53         signal_old = signal;
54     fi;
55     if
56         :: !signal -> out1 = 0;
57         out2 = 0;
58         cntr = 0;
59         :: signal && edge_signal -> out1 = !out1;
60         out3.out1 = out1;
61         :: else -> cntr = cntr + 1;
62         do
63             :: cntr > cntr_max -> cntr = cntr - 1; cntr2 = cntr2
64                 + 1;
65             :: else -> break
66         od;
67         if
68             :: (cntr2 > cntr_max) && signal -> out2 = 1
69             :: else -> out2 = 0
70         fi;
71         out3.out1 = out1;
72         out3.out2 = out2;
73         out3.remaining = cntr_max - cntr;
74         out3.elapsed = cntr;
75     fi;
76     if
77         :: !mode1 && !mode2 && !mode3 -> mode1 = 1
78         :: else
79     fi;
80     if
81         :: toMode1 || (toMode3 && mode3Forbidden) -> mode1 = 1

```

```

81     :: else
82     fi;
83     if
84         :: toMode2 -> mode2 = 1
85         :: toMode3 -> mode3 = 1
86         :: else
87     fi;
88     if
89         :: mode1 -> mode = 1
90         :: !mode1 && mode2 -> mode = 2
91         :: !mode1 && !mode2 && mode3 -> mode = 3
92         :: else -> mode = 0
93     fi;}
94
95     assert (!out2 || out1)
96     assert (signal || !out2)
97     assert (out3.out1 == out1)
98     assert (out1 || (out3.elapsed == 0))
99
100     od
101 }

```

Listing B.5: Example_while.pml

```

1  typedef ComplexSignal{
2      bool out1;
3      short elapsed;
4  };
5  bool signal;
6  bool signal_old = 0;
7  bool edge_signal;
8  bool out1, out2;
9  int cntr_max = 5;
10 short cntr = 0;
11 short cntr2 = 0;
12 ComplexSignal out3;
13
14 active proctype go()
15 {
16     do
17     ::
18     if
19         :: signal = 0
20         :: signal = 1
21     fi;
22
23     d_step{if
24         :: (signal==1) && (signal_old==0) -> edge_signal = 1;
25         signal_old = 1;
26         :: !signal || signal_old -> edge_signal = 0;
27         signal_old = signal;
28     fi;
29     if
30         :: !signal -> out1 = 0;
31         out2 = 0;
32         cntr = 0;
33         :: signal && edge_signal -> out1 = !out1;
34         out3.out1 = out1
35         :: else -> cntr = cntr + 1;
36         do
37             :: cntr > cntr_max -> cntr = cntr - 1; cntr2 = cntr2
38             + 1
39             :: else -> break
40         od;
41         if
42             :: (cntr2 > cntr_max) && signal -> out2 = 1
43             :: else -> out2 = 0
44         fi;
45         out3.out1 = out1;
46         out3.elapsed = cntr
47     fi;}
48
49     assert (!out2 || out1)
50     assert (signal || !out2)
51     assert (out3.out1 == out1)
52     assert (out1 || (out3.elapsed == 0))
53
54     od
55 }

```

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.

```

1  MODULE main
2  VAR
3      signal : boolean;
4      error : boolean;
5      toMode1 : boolean;
6      toMode2 : boolean;
7      toMode3 : boolean;
8      mode3Forbidden : 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     mode1 : boolean;
19     mode2 : boolean;
20     mode3 : boolean;
21     mode : -1..3;
22     loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
           step11, step12, step13, step14, step15, step16, step17, step18, step19, step20,
           step21, step22, end};

23
24  ASSIGN
25     init(signal_old) := FALSE;
26     init(edge_signal) := FALSE;
27     init(cntr) := 0sd16_0;
28     init(out1) := FALSE;
29     init(out2) := FALSE;
30     init(out3.out1) := FALSE;
31     init(out3.out2) := FALSE;
32     init(out3.remaining) := 0sd16_0;
33     init(out3.elapsed) := 0sd16_0;
34     init(mode1) := FALSE;
35     init(mode2) := FALSE;
36     init(mode3) := FALSE;
37     init(mode) := -1;
38     init(loc) := start;
39
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) & (!signal) : step5;
48         (loc = step4) & (signal) & (edge_signal) : step6;
49         (loc = step4) : step7;
50         (loc = step5) : step11;
51         (loc = step6) : step11;
52         (loc = step7) & (cntr > 0sd16_5) & (signal) : step8;
53         (loc = step7) : step9;
54         (loc = step8) : step10;
55         (loc = step9) : step10;
56         (loc = step10) : step11;
57         (loc = step11) & (!mode1 & !mode2 & !mode3) : step12;
58         (loc = step11) : step13;
59         (loc = step12) : step13;
60         (loc = step13) & (toMode1) | (toMode3 & mode3Forbidden) : 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) & (mode1) : step19;
69         (loc = step18) & (!mode1) & (mode2) : step20;
70         (loc = step18) & (!mode1) & (!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
79     next(signal) :=
80     case

```



```

81         (loc = start) : {TRUE, FALSE};
82         TRUE : signal;
83     esac;
84     next(error) :=
85     case
86         (loc = start) : {TRUE, FALSE};
87         TRUE : error;
88     esac;
89     next(toModel) :=
90     case
91         (loc = start) : {TRUE, FALSE};
92         TRUE : toModel;
93     esac;
94     next(toMode2) :=
95     case
96         (loc = start) : {TRUE, FALSE};
97         TRUE : toMode2;
98     esac;
99     next(toMode3) :=
100    case
101        (loc = start) : {TRUE, FALSE};
102        TRUE : toMode3;
103    esac;
104    next(mode3Forbidden) :=
105    case
106        (loc = start) : {TRUE, FALSE};
107        TRUE : mode3Forbidden;
108    esac;
109
110    next(signal_old) :=
111    case
112        (loc = step2) : TRUE;
113        (loc = step3) : signal;
114        TRUE : signal_old;
115    esac;
116    next(edge_signal) :=
117    case
118        (loc = step2) : TRUE;
119        (loc = step3) : FALSE;
120        TRUE : edge_signal;
121    esac;
122    next(cntr) :=
123    case
124        (loc = step5) : 0sd16_0;
125        (loc = step7) : cntr + 0sd16_1;
126        TRUE : cntr;
127    esac;
128    next(out1) :=
129    case
130        (loc = step5) : FALSE;
131        (loc = step6) : !out1;
132        TRUE : out1;
133    esac;
134    next(out2) :=
135    case
136        (loc = step5) : FALSE;
137        (loc = step8) : TRUE;
138        (loc = step9) : FALSE;
139        TRUE : out2;
140    esac;
141    next(out3.out1) :=
142    case
143        (loc = step6) : !out1;
144        (loc = step10) : out1;
145        TRUE : out3.out1;
146    esac;
147    next(out3.out2) :=
148    case
149        (loc = step10) : out2;
150        TRUE : out3.out2;
151    esac;
152    next(out3.remaining) :=
153    case
154        (loc = step10) : 0sd16_5 - cntr;
155        TRUE : out3.remaining;
156    esac;
157    next(out3.elapsed) :=
158    case
159        (loc = step10) : cntr;
160        TRUE : out3.elapsed;
161    esac;
162    next(model) :=
163    case
164        (loc = step12) : TRUE;
165        (loc = step14) : TRUE;
166        TRUE : model;
167    esac;
168    next(mode2) :=
169    case
170        (loc = step16) : TRUE;

```

```

171         TRUE : mode2;
172     esac;
173     next(mode3) :=
174     case
175         (loc = step17) : TRUE;
176         TRUE : mode3;
177     esac;
178     next(mode) :=
179     case
180         (loc = step19) : 1;
181         (loc = step20) : 2;
182         (loc = step21) : 3;
183         (loc = step22) : 0;
184         TRUE : mode;
185     esac;
186
187 --CTL properties
188 SPEC AG(loc = end -> (out2 -> out1))
189 SPEC AG(loc = end -> (!signal -> !out2))
190 SPEC AG(loc = end -> (out3.out1 = out1))
191 SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
192
193 --INVARIANT properties
194 INVARSPEC(loc = end -> (out2 -> out1))
195 INVARSPEC(loc = end -> (!signal -> !out2))
196 INVARSPEC(loc = end -> (out3.out1 = out1))
197 INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

```

Listing C.1: Example.smv

```

1  MODULE main
2  VAR
3      signal : boolean;
4      signal_old : boolean;
5      edge_signal : boolean;
6      cntr : signed word[16];
7      out1 : boolean;
8      out2 : boolean;
9      out3.out1 : boolean;
10     out3.elapsed : signed word[16];
11     loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
12         end};
13
14     ASSIGN
15     init(signal_old) := FALSE;
16     init(edge_signal) := FALSE;
17     init(cntr) := 0sd16_0;
18     init(out1) := FALSE;
19     init(out2) := FALSE;
20     init(out3.out1) := FALSE;
21     init(out3.elapsed) := 0sd16_0;
22     init(loc) := start;
23
24     next(loc) :=
25     case
26         (loc = start) : step1;
27         (loc = step1) & (signal & !signal_old) : step2;
28         (loc = step1) : step3;
29         (loc = step2) : step4;
30         (loc = step3) : step4;
31         (loc = step4) & (!signal) : step5;
32         (loc = step4) & (signal) & (edge_signal) : step6;
33         (loc = step4) : step7;
34         (loc = step5) : end;
35         (loc = step6) : end;
36         (loc = step7) & (cntr > 0sd16_5) & (signal) : step8;
37         (loc = step7) : step9;
38         (loc = step8) : step10;
39         (loc = step9) : step10;
40         (loc = step10) : end;
41         (loc = end) : start;
42     esac;
43
44     next(signal) :=
45     case
46         (loc = start) : {TRUE, FALSE};
47         TRUE : signal;
48     esac;
49
50     next(signal_old) :=
51     case
52         (loc = step2) : TRUE;
53         (loc = step3) : signal;
54         TRUE : signal_old;
55     esac;
56
57     next(edge_signal) :=
58     case
59         (loc = step2) : TRUE;

```

```

58         (loc = step3) : FALSE;
59         TRUE : edge_signal;
60     esac;
61     next(cntnr) :=
62     case
63         (loc = step5) : 0sd16_0;
64         (loc = step7) : cntnr + 0sd16_1;
65         TRUE : cntnr;
66     esac;
67     next(out1) :=
68     case
69         (loc = step5) : FALSE;
70         (loc = step6) : !out1;
71         TRUE : out1;
72     esac;
73     next(out2) :=
74     case
75         (loc = step5) : FALSE;
76         (loc = step8) : TRUE;
77         (loc = step9) : FALSE;
78         TRUE : out2;
79     esac;
80     next(out3.out1) :=
81     case
82         (loc = step6) : out1;
83         (loc = step10) : out1;
84         TRUE : out3.out1;
85     esac;
86     next(out3.elapsed) :=
87     case
88         (loc = step10) : cntnr;
89         TRUE : out3.elapsed;
90     esac;
91
92 --CTL properties
93 SPEC AG(loc = end -> (out2 -> out1))
94 SPEC AG(loc = end -> (!signal -> !out2))
95 SPEC AG(loc = end -> (out3.out1 = out1))
96 SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
97
98 --INVARIANT properties
99 INVARSPEC(loc = end -> (out2 -> out1))
100 INVARSPEC(loc = end -> (!signal -> !out2))
101 INVARSPEC(loc = end -> (out3.out1 = out1))
102 INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

```

Listing C.2: Reduced version of Example.smv

```

1  MODULE main
2      VAR
3          signal : boolean;
4          error : boolean;
5          toModel : boolean;
6          toMode2 : boolean;
7          toMode3 : boolean;
8          mode3Forbidden : boolean;
9          signal_old : boolean;
10         edge_signal : boolean;
11         cntnr : signed word[16];
12         cntnr2 : signed word[16];
13         out1 : boolean;
14         out2 : boolean;
15         out3.out1 : boolean;
16         out3.out2 : boolean;
17         out3.remaining : signed word[16];
18         out3.elapsed : signed word[16];
19         model : boolean;
20         mode2 : boolean;
21         mode3 : boolean;
22         mode : -1..3;
23         loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
                step11, step12, step13, step14, step15, step16, step17, step18, step19, step20,
                step21, step22, step23, step24, end};
24
25     ASSIGN
26         init(signal_old) := FALSE;
27         init(edge_signal) := FALSE;
28         init(cntnr) := 0sd16_0;
29         init(cntnr2) := 0sd16_0;
30         init(out1) := FALSE;
31         init(out2) := FALSE;
32         init(out3.out1) := FALSE;
33         init(out3.out2) := FALSE;
34         init(out3.remaining) := 0sd16_0;
35         init(out3.elapsed) := 0sd16_0;
36         init(model) := FALSE;
37         init(mode2) := FALSE;
38         init(mode3) := FALSE;

```

```

39     init(mode) := -1;
40     init(loc) := start;
41
42     next(loc) :=
43     case
44         (loc = start) : step1;
45         (loc = step1) & (signal & !signal_old) : step2;
46         (loc = step1) : step3;
47         (loc = step2) : step4;
48         (loc = step3) : step4;
49         (loc = step4) & (!signal) : step5;
50         (loc = step4) & (signal) & (edge_signal) : step6;
51         (loc = step4) : step7;
52         (loc = step5) : step13;
53         (loc = step6) : step13;
54         (loc = step7) & (cntr > 0sd16_5) : step8;
55         (loc = step7) : step9;
56         (loc = step8) : step9;
57         (loc = step9) & (cntr2 > 0sd16_5) & (signal) : step10;
58         (loc = step9) : step11;
59         (loc = step10) : step12;
60         (loc = step11) : step12;
61         (loc = step12) : step13;
62         (loc = step13) & (!mode1 & !mode2 & !mode3) : step14;
63         (loc = step13) : step15;
64         (loc = step14) : step15;
65         (loc = step15) & (toMode1) | (toMode3 & mode3Forbidden) : step16;
66         (loc = step15) : step17;
67         (loc = step16) : step17;
68         (loc = step17) & (toMode2) : step18;
69         (loc = step17) & (!toMode2) & (toMode3) : step19;
70         (loc = step17) : step20;
71         (loc = step18) : step20;
72         (loc = step19) : step20;
73         (loc = step20) & (mode1) : step21;
74         (loc = step20) & (!mode1) & (mode2) : step22;
75         (loc = step20) & (!mode1) & (!mode2) & (mode3) : step23;
76         (loc = step20) : step24;
77         (loc = step21) : end;
78         (loc = step22) : end;
79         (loc = step23) : end;
80         (loc = step24) : end;
81         (loc = end) : start;
82     esac;
83
84     next(signal) :=
85     case
86         (loc = start) : {TRUE, FALSE};
87         TRUE : signal;
88     esac;
89     next(error) :=
90     case
91         (loc = start) : {TRUE, FALSE};
92         TRUE : error;
93     esac;
94     next(toMode1) :=
95     case
96         (loc = start) : {TRUE, FALSE};
97         TRUE : toMode1;
98     esac;
99     next(toMode2) :=
100    case
101        (loc = start) : {TRUE, FALSE};
102        TRUE : toMode2;
103    esac;
104    next(toMode3) :=
105    case
106        (loc = start) : {TRUE, FALSE};
107        TRUE : toMode3;
108    esac;
109    next(mode3Forbidden) :=
110    case
111        (loc = start) : {TRUE, FALSE};
112        TRUE : mode3Forbidden;
113    esac;
114
115    next(signal_old) :=
116    case
117        (loc = step2) : TRUE;
118        (loc = step3) : signal;
119        TRUE : signal_old;
120    esac;
121    next(edge_signal) :=
122    case
123        (loc = step2) : TRUE;
124        (loc = step3) : FALSE;
125        TRUE : edge_signal;
126    esac;
127    next(cntr) :=
128    case

```

```

129         (loc = step5) : 0sd16_0;
130         (loc = step7) : cntr + 0sd16_1;
131         (loc = step8) : 0sd16_0;
132         TRUE : cntr;
133     esac;
134     next(cntr2) :=
135     case
136         (loc = step8) : cntr2 + 0sd16_1;
137         TRUE : cntr2;
138     esac;
139     next(out1) :=
140     case
141         (loc = step5) : FALSE;
142         (loc = step6) : !out1;
143         TRUE : out1;
144     esac;
145     next(out2) :=
146     case
147         (loc = step5) : FALSE;
148         (loc = step10) : TRUE;
149         (loc = step11) : FALSE;
150         TRUE : out2;
151     esac;
152     next(out3.out1) :=
153     case
154         (loc = step6) : !out1;
155         (loc = step12) : out1;
156         TRUE : out3.out1;
157     esac;
158     next(out3.out2) :=
159     case
160         (loc = step12) : out2;
161         TRUE : out3.out2;
162     esac;
163     next(out3.remaining) :=
164     case
165         (loc = step12) : 0sd16_5 - cntr;
166         TRUE : out3.remaining;
167     esac;
168     next(out3.elapsed) :=
169     case
170         (loc = step12) : cntr;
171         TRUE : out3.elapsed;
172     esac;
173     next(mode1) :=
174     case
175         (loc = step14) : TRUE;
176         (loc = step16) : TRUE;
177         TRUE : mode1;
178     esac;
179     next(mode2) :=
180     case
181         (loc = step18) : TRUE;
182         TRUE : mode2;
183     esac;
184     next(mode3) :=
185     case
186         (loc = step19) : TRUE;
187         TRUE : mode3;
188     esac;
189     next(mode) :=
190     case
191         (loc = step21) : 1;
192         (loc = step22) : 2;
193         (loc = step23) : 3;
194         (loc = step24) : 0;
195         TRUE : mode;
196     esac;
197
198 --CTL properties
199 SPEC AG(loc = end -> (out2 -> out1))
200 SPEC AG(loc = end -> (!signal -> !out2))
201 SPEC AG(loc = end -> (out3.out1 = out1))
202 SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
203
204 --INVARIANT properties
205 INVARSPEC(loc = end -> (out2 -> out1))
206 INVARSPEC(loc = end -> (!signal -> !out2))
207 INVARSPEC(loc = end -> (out3.out1 = out1))
208 INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

```

Listing C.3: Example_int.smv

```

1 MODULE main
2     VAR
3         signal : boolean;
4         signal_old : boolean;
5         edge_signal : boolean;

```

```

6      cntnr : signed word[16];
7      cntnr2 : signed word[16];
8      out1 : boolean;
9      out2 : boolean;
10     out3.out1 : boolean;
11     out3.elapsed : signed word[16];
12     loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
13         step11, step12, end};
14
15     ASSIGN
16     init(signal_old) := FALSE;
17     init(edge_signal) := FALSE;
18     init(cntnr) := 0sd16_0;
19     init(cntnr2) := 0sd16_0;
20     init(out1) := FALSE;
21     init(out2) := FALSE;
22     init(out3.out1) := FALSE;
23     init(out3.elapsed) := 0sd16_0;
24     init(loc) := start;
25
26     next(loc) :=
27     case
28         (loc = start) : step1;
29         (loc = step1) & (signal & !signal_old) : step2;
30         (loc = step1) : step3;
31         (loc = step2) : step4;
32         (loc = step3) : step4;
33         (loc = step4) & (!signal) : step5;
34         (loc = step4) & (signal) & (edge_signal) : step6;
35         (loc = step4) : step7;
36         (loc = step5) : end;
37         (loc = step6) : end;
38         (loc = step7) & (cntnr > 0sd16_5) : step8;
39         (loc = step7) : step9;
40         (loc = step8) : step9;
41         (loc = step9) & (cntnr2 > 0sd16_5) & (signal) : step10;
42         (loc = step9) : step11;
43         (loc = step10) : step12;
44         (loc = step11) : step12;
45         (loc = step12) : end;
46         (loc = end) : start;
47     esac;
48
49     next(signal) :=
50     case
51         (loc = start) : {TRUE, FALSE};
52         TRUE : signal;
53     esac;
54
55     next(signal_old) :=
56     case
57         (loc = step2) : TRUE;
58         (loc = step3) : signal;
59         TRUE : signal_old;
60     esac;
61
62     next(edge_signal) :=
63     case
64         (loc = step2) : TRUE;
65         (loc = step3) : FALSE;
66         TRUE : edge_signal;
67     esac;
68
69     next(cntnr) :=
70     case
71         (loc = step5) : 0sd16_0;
72         (loc = step7) : cntnr + 0sd16_1;
73         (loc = step8) : 0sd16_0;
74         TRUE : cntnr;
75     esac;
76
77     next(cntnr2) :=
78     case
79         (loc = step8) : cntnr2 + 0sd16_1;
80         TRUE : cntnr2;
81     esac;
82
83     next(out1) :=
84     case
85         (loc = step5) : FALSE;
86         (loc = step6) : !out1;
87         TRUE : out1;
88     esac;
89
90     next(out2) :=
91     case
92         (loc = step5) : FALSE;
93         (loc = step10) : TRUE;
94         (loc = step11) : FALSE;
95         TRUE : out2;
96     esac;
97
98     next(out3.out1) :=
99     case
100        (loc = step6) : !out1;
101        (loc = step12) : out1;

```

```

95         TRUE : out3.out1;
96     esac;
97     next(out3.elapsed) :=
98     case
99         (loc = step12) : cntr;
100        TRUE : out3.elapsed;
101    esac;
102
103 --CTL properties
104 SPEC AG(loc = end -> (out2 -> out1))
105 SPEC AG(loc = end -> (!signal -> !out2))
106 SPEC AG(loc = end -> (out3.out1 = out1))
107 SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
108
109 --INVARIANT properties
110 INVARSPEC(loc = end -> (out2 -> out1))
111 INVARSPEC(loc = end -> (!signal -> !out2))
112 INVARSPEC(loc = end -> (out3.out1 = out1))
113 INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

```

Listing C.4: Reduced version of Example_int.smv

```

1  MODULE main
2  VAR
3      signal : boolean;
4      error : boolean;
5      toModel : boolean;
6      toMode2 : boolean;
7      toMode3 : boolean;
8      mode3Forbidden : boolean;
9      signal_old : boolean;
10     edge_signal : boolean;
11     cntr : signed word[16];
12     cntr2 : signed word[16];
13     out1 : boolean;
14     out2 : boolean;
15     out3.out1 : boolean;
16     out3.out2 : boolean;
17     out3.remaining : signed word[16];
18     out3.elapsed : signed word[16];
19     model : boolean;
20     mode2 : boolean;
21     mode3 : boolean;
22     mode : -1..3;
23     loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
24           step11, step12, step13, step14, step15, step16, step17, step18, step19, step20,
25           step21, step22, step23, step24, step25, end};
26
27 ASSIGN
28     init(signal_old) := FALSE;
29     init(edge_signal) := FALSE;
30     init(cntr) := 0sd16_0;
31     init(cntr2) := 0sd16_0;
32     init(out1) := FALSE;
33     init(out2) := FALSE;
34     init(out3.out1) := FALSE;
35     init(out3.out2) := FALSE;
36     init(out3.remaining) := 0sd16_0;
37     init(out3.elapsed) := 0sd16_0;
38     init(model) := FALSE;
39     init(mode2) := FALSE;
40     init(mode3) := FALSE;
41     init(mode) := -1;
42     init(loc) := start;
43
44     next(loc) :=
45     case
46         (loc = start) : step1;
47         (loc = step1) & (signal & !signal_old) : step2;
48         (loc = step1) : step3;
49         (loc = step2) : step4;
50         (loc = step3) : step4;
51         (loc = step4) & (!signal) : step5;
52         (loc = step4) & (signal) & (edge_signal) : step6;
53         (loc = step4) : step7;
54         (loc = step5) : step14;
55         (loc = step6) : step14;
56         (loc = step7) : step8;
57         (loc = step8) & (cntr > 0sd16_5) : step9;
58         (loc = step8) : step10;
59         (loc = step9) : step8;
60         (loc = step10) & (cntr2 > 0sd16_5) & (signal) : step11;
61         (loc = step10) : step12;
62         (loc = step11) : step13;
63         (loc = step12) : step13;
64         (loc = step13) : step14;
65         (loc = step14) & (!model & !mode2 & !mode3) : step15;
66         (loc = step14) : step16;

```

```

65         (loc = step15) : step16;
66         (loc = step16) & (toMode1) | (toMode3 & mode3Forbidden) : step17;
67         (loc = step16) : step18;
68         (loc = step17) : step18;
69         (loc = step18) & (toMode2) : step19;
70         (loc = step18) & (!toMode2) & (toMode3) : step20;
71         (loc = step18) : step21;
72         (loc = step19) : step21;
73         (loc = step20) : step21;
74         (loc = step21) & (mode1) : step22;
75         (loc = step21) & (!mode1) & (mode2) : step23;
76         (loc = step21) & (!mode1) & (!mode2) & (mode3) : step24;
77         (loc = step21) : step25;
78         (loc = step22) : end;
79         (loc = step23) : end;
80         (loc = step24) : end;
81         (loc = step25) : end;
82         (loc = end) : start;
83     esac;
84
85 next(signal) :=
86     case
87         (loc = start) : {TRUE, FALSE};
88         TRUE : signal;
89     esac;
90 next(error) :=
91     case
92         (loc = start) : {TRUE, FALSE};
93         TRUE : error;
94     esac;
95 next(toMode1) :=
96     case
97         (loc = start) : {TRUE, FALSE};
98         TRUE : toMode1;
99     esac;
100 next(toMode2) :=
101     case
102         (loc = start) : {TRUE, FALSE};
103         TRUE : toMode2;
104     esac;
105 next(toMode3) :=
106     case
107         (loc = start) : {TRUE, FALSE};
108         TRUE : toMode3;
109     esac;
110 next(mode3Forbidden) :=
111     case
112         (loc = start) : {TRUE, FALSE};
113         TRUE : mode3Forbidden;
114     esac;
115
116 next(signal_old) :=
117     case
118         (loc = step2) : TRUE;
119         (loc = step3) : signal;
120         TRUE : signal_old;
121     esac;
122 next(edge_signal) :=
123     case
124         (loc = step2) : TRUE;
125         (loc = step3) : FALSE;
126         TRUE : edge_signal;
127     esac;
128 next(cnt1) :=
129     case
130         (loc = step5) : 0sd16_0;
131         (loc = step7) : cnt1 + 0sd16_1;
132         (loc = step9) : cnt1 - 0sd16_1;
133         TRUE : cnt1;
134     esac;
135 next(cnt2) :=
136     case
137         (loc = step9) : cnt2 + 0sd16_1;
138         TRUE : cnt2;
139     esac;
140 next(out1) :=
141     case
142         (loc = step5) : FALSE;
143         (loc = step6) : !out1;
144         TRUE : out1;
145     esac;
146 next(out2) :=
147     case
148         (loc = step5) : FALSE;
149         (loc = step11) : TRUE;
150         (loc = step12) : FALSE;
151         TRUE : out2;
152     esac;
153 next(out3.out1) :=
154     case

```



```

155         (loc = step6) : !out1;
156         (loc = step13) : out1;
157         TRUE : out3.out1;
158     esac;
159     next(out3.out2) :=
160     case
161         (loc = step13) : out2;
162         TRUE : out3.out2;
163     esac;
164     next(out3.remaining) :=
165     case
166         (loc = step13) : 0sd16_5 - cntr;
167         TRUE : out3.remaining;
168     esac;
169     next(out3.elapsed) :=
170     case
171         (loc = step13) : cntr;
172         TRUE : out3.elapsed;
173     esac;
174     next(mode1) :=
175     case
176         (loc = step15) : TRUE;
177         (loc = step17) : TRUE;
178         TRUE : mode1;
179     esac;
180     next(mode2) :=
181     case
182         (loc = step19) : TRUE;
183         TRUE : mode2;
184     esac;
185     next(mode3) :=
186     case
187         (loc = step20) : TRUE;
188         TRUE : mode3;
189     esac;
190     next(mode) :=
191     case
192         (loc = step22) : 1;
193         (loc = step23) : 2;
194         (loc = step24) : 3;
195         (loc = step25) : 0;
196         TRUE : mode;
197     esac;
198
199 --CTL properties
200 SPEC AG(loc = end -> (out2 -> out1))
201 SPEC AG(loc = end -> (!signal -> !out2))
202 SPEC AG(loc = end -> (out3.out1 = out1))
203 SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
204
205 --INVARIANT properties
206 INVARSPEC(loc = end -> (out2 -> out1))
207 INVARSPEC(loc = end -> (!signal -> !out2))
208 INVARSPEC(loc = end -> (out3.out1 = out1))
209 INVARSPEC(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))

```

Listing C.5: Example_while.smv

```

1  MODULE main
2  VAR
3      signal : boolean;
4      signal_old : boolean;
5      edge_signal : boolean;
6      cntr : signed word[16];
7      cntr2 : signed word[16];
8      out1 : boolean;
9      out2 : boolean;
10     out3.out1 : boolean;
11     out3.elapsed : signed word[16];
12     loc : {start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10,
13           step11, step12, step13, end};
14
15  ASSIGN
16     init(signal_old) := FALSE;
17     init(edge_signal) := FALSE;
18     init(cntr) := 0sd16_0;
19     init(cntr2) := 0sd16_0;
20     init(out1) := FALSE;
21     init(out2) := FALSE;
22     init(out3.out1) := FALSE;
23     init(out3.elapsed) := 0sd16_0;
24     init(loc) := start;
25
26     next(loc) :=
27     case
28         (loc = start) : step1;
29         (loc = step1) & (signal & !signal_old) : step2;
30         (loc = step1) : step3;

```

```

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) : end;
37         (loc = step7) : step8;
38         (loc = step8) & (cntr > 0sd16_5) : step9;
39         (loc = step8) : step10;
40         (loc = step9) : step8;
41         (loc = step10) & (cntr2 > 0sd16_5) & (signal) : step11;
42         (loc = step10) : step12;
43         (loc = step11) : step13;
44         (loc = step12) : step13;
45         (loc = step13) : end;
46         (loc = end) : start;
47     esac;
48
49     next(signal) :=
50     case
51         (loc = start) : {TRUE, FALSE};
52         TRUE : signal;
53     esac;
54
55     next(signal_old) :=
56     case
57         (loc = step2) : TRUE;
58         (loc = step3) : signal;
59         TRUE : signal_old;
60     esac;
61     next(edge_signal) :=
62     case
63         (loc = step2) : TRUE;
64         (loc = step3) : FALSE;
65         TRUE : edge_signal;
66     esac;
67     next(cntr) :=
68     case
69         (loc = step5) : 0sd16_0;
70         (loc = step7) : cntr + 0sd16_1;
71         (loc = step9) : cntr - 0sd16_1;
72         TRUE : cntr;
73     esac;
74     next(cntr2) :=
75     case
76         (loc = step9) : cntr2 + 0sd16_1;
77         TRUE : cntr2;
78     esac;
79     next(out1) :=
80     case
81         (loc = step5) : FALSE;
82         (loc = step6) : !out1;
83         TRUE : out1;
84     esac;
85     next(out2) :=
86     case
87         (loc = step5) : FALSE;
88         (loc = step11) : TRUE;
89         (loc = step12) : FALSE;
90         TRUE : out2;
91     esac;
92     next(out3.out1) :=
93     case
94         (loc = step6) : !out1;
95         (loc = step13) : out1;
96         TRUE : out3.out1;
97     esac;
98     next(out3.elapsed) :=
99     case
100         (loc = step13) : cntr;
101         TRUE : out3.elapsed;
102     esac;
103
104     --CTL properties
105     SPEC AG(loc = end -> (out2 -> out1))
106     SPEC AG(loc = end -> (!signal -> !out2))
107     SPEC AG(loc = end -> (out3.out1 = out1))
108     SPEC AG(loc = end -> (!out1 -> out3.elapsed = 0sd16_0))
109
110     --INVARIANT properties
111     INVARSPEC(loc = end -> (out2 -> out1))
112     INVARSPEC(loc = end -> (!signal -> !out2))
113     INVARSPEC(loc = end -> (out3.out1 = out1))
114     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.

```

1 #include <stdbool.h>
2 bool nondet_bool();
3
4 bool R_edge(bool new, bool *old){
5     if(new &&!*old){
6         *old = true;
7         return true;
8     } else{
9         *old = new;
10        return false;
11    }
12 }
13
14 int main(){
15     struct ComplexSignal{
16         bool out1;
17         bool out2;
18         short remaining;
19         short elapsed;
20     };
21     bool error, signal, toMode1, toMode2, toMode3, mode3Forbidden;
22     bool signal_old = false;
23     bool edge_signal = false;
24     bool mode1 = false;
25     bool mode2 = false;
26     bool mode3 = false;
27     short mode = 0;
28     short cntr = 0;
29     short cntr_max = 5;
30     bool out1 = false;
31     bool out2 = false;
32     struct ComplexSignal out3;
33     out3.out1 = false;
34     out3.out2 = false;
35     out3.elapsed = 0;
36     out3.remaining = 0;
37
38     while(true){
39         error = nondet_bool();
40         toMode1 = nondet_bool();
41         toMode2 = nondet_bool();
42         toMode3 = nondet_bool();
43         mode3Forbidden = nondet_bool();
44         signal = nondet_bool();
45
46         edge_signal = R_edge(signal,&signal_old);
47         if(!signal){
48             out1 = false;
49             out2 = false;
50             cntr = 0;
51         } else if(edge_signal){
52             out1 = !out1;
53             out3.out1 = out1;
54         } else{
55             cntr = cntr + 1;
56             if(cntr > cntr_max && signal){
57                 out2 = true;
58             } else{
59                 out2 = false;
60             }
61             out3.out1 = out1;
62             out3.out2 = out2;
63             out3.remaining = cntr_max - cntr;
64             out3.elapsed = cntr;
65         }
66         if (!mode1 && !mode2 && !mode3){
67             mode1 = true;
68         }
69         if(toMode1 || (toMode3 && mode3Forbidden)){
70             mode1 = true;
71         }
72         if(toMode2){
73             mode2 = true;
74         } else if(toMode3){
75             mode3 = true;
76         }
77         if(mode1){
78             mode = 1;
79         } else if(mode2){
80             mode = 2;
81         } else if(mode3){
82             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

```

1  #include <stdbool.h>
2  bool nondet_bool();
3
4  bool R_edge(bool new, bool *old){
5      if(new &&!*old){
6          *old = true;
7          return true;
8      } else{
9          *old = new;
10         return false;
11     }
12 }
13
14 int main(){
15     struct ComplexSignal{
16         bool out1;
17         short elapsed;
18     };
19     bool signal;
20     bool signal_old = false;
21     bool edge_signal = false;
22     short cntr = 0;
23     short cntr_max = 5;
24     bool out1 = false;
25     bool out2 = false;
26     struct ComplexSignal out3;
27     out3.out1 = false;
28     out3.elapsed = 0;
29
30     while(true){
31         signal = nondet_bool();
32
33         edge_signal = R_edge(signal,&signal_old);
34
35         if(!signal){
36             out1 = false;
37             out2 = false;
38             cntr = 0;
39         } else if(edge_signal){
40             out1 = !out1;
41             out3.out1 = out1;
42         } else{
43             cntr = cntr + 1;
44             if(cntr > cntr_max && signal){
45                 out2 = true;
46             } else{
47                 out2 = false;
48             }
49             out3.out1 = out1;
50             out3.elapsed = cntr;
51         }
52
53         assert(!out2 || out1);
54         assert(signal || !out2);
55         assert(out3.out1 == out1);
56         assert(out1 || (out3.elapsed == 0));
57     }
58 }

```

Listing D.2: Reduced version of Example.c

```

1  #include <stdbool.h>
2  bool nondet_bool();
3
4  bool R_edge(bool new, bool *old){
5      if(new &&!*old){
6          *old = true;
7          return true;
8      } else{
9          *old = new;
10         return false;
11     }
12 }

```

```
13
14 int main(){
15     struct ComplexSignal{
16         bool out1;
17         bool out2;
18         short remaining;
19         short elapsed;
20     };
21     bool error, signal, toMode1, toMode2, toMode3, mode3Forbidden;
22     bool signal_old = false;
23     bool edge_signal = false;
24     bool mode1 = false;
25     bool mode2 = false;
26     bool mode3 = false;
27     short mode = 0;
28     short cntr = 0;
29     short cntr2 = 0;
30     short cntr_max = 5;
31     bool out1 = false;
32     bool out2 = false;
33     struct ComplexSignal out3;
34     out3.out1 = false;
35     out3.out2 = false;
36     out3.elapsed = 0;
37     out3.remaining = 0;
38
39     while(true){
40         error = nondet_bool();
41         toMode1 = nondet_bool();
42         toMode2 = nondet_bool();
43         toMode3 = nondet_bool();
44         mode3Forbidden = nondet_bool();
45         signal = nondet_bool();
46
47         edge_signal = R.Edge(signal,&signal_old);
48         if(!signal){
49             out1 = false;
50             out2 = false;
51             cntr = 0;
52         } else if(edge_signal){
53             out1 = !out1;
54             out3.out1 = out1;
55         } else{
56             cntr = cntr + 1;
57             if(cntr > cntr_max){
58                 cntr = 0;
59                 cntr2 = cntr2 + 1;
60             }
61             if(cntr2 > cntr_max && signal){
62                 out2 = true;
63             } else{
64                 out2 = false;
65             }
66             out3.out1 = out1;
67             out3.out2 = out2;
68             out3.remaining = cntr_max - cntr;
69             out3.elapsed = cntr;
70         }
71         if (!mode1 && !mode2 && !mode3){
72             mode1 = true;
73         }
74         if(toMode1 || (toMode3 && mode3Forbidden)){
75             mode1 = true;
76         }
77         if(toMode2){
78             mode2 = true;
79         } else if(toMode3){
80             mode3 = true;
81         }
82         if(mode1){
83             mode = 1;
84         } else if(mode2){
85             mode = 2;
86         } else if(mode3){
87             mode = 3;
88         } else{
89             mode = 0;
90         }
91
92         assert(!out2 || out1);
93         assert(signal || !out2);
94         assert(out3.out1 == out1);
95         assert(out1 || (out3.elapsed == 0));
96     }
97 }
```

Listing D.3: Example_int.c

```

1 #include <stdbool.h>
2 bool nondet_bool();
3
4 bool R_edge(bool new, bool *old){
5     if(new &&!*old){
6         *old = true;
7         return true;
8     } else{
9         *old = new;
10        return false;
11    }
12 }
13
14 int main(){
15     struct ComplexSignal{
16         bool out1;
17         short elapsed;
18     };
19     bool signal;
20     bool signal_old = false;
21     bool edge_signal = false;
22     short cntr = 0;
23     short cntr2 = 0;
24     short cntr_max = 5;
25     bool out1 = false;
26     bool out2 = false;
27     struct ComplexSignal out3;
28     out3.out1 = false;
29     out3.elapsed = 0;
30
31     while(true){
32         signal = nondet_bool();
33
34         edge_signal = R_edge(signal,&signal_old);
35         if(!signal){
36             out1 = false;
37             out2 = false;
38             cntr = 0;
39         } else if(edge_signal){
40             out1 = !out1;
41             out3.out1 = out1;
42         } else{
43             cntr = cntr + 1;
44             if(cntr > cntr_max){
45                 cntr = 0;
46                 cntr2 = cntr2 + 1;
47             }
48             if(cntr2 > cntr_max && signal){
49                 out2 = true;
50             } else{
51                 out2 = false;
52             }
53             out3.out1 = out1;
54             out3.elapsed = cntr;
55         }
56
57         assert(!out2 || out1);
58         assert(signal || !out2);
59         assert(out3.out1 == out1);
60         assert(out1 || (out3.elapsed == 0));
61     }
62 }

```

Listing D.4: Reduced version of Example_int.c

```

1 #include <stdbool.h>
2 bool nondet_bool();
3
4 bool R_edge(bool new, bool *old){
5     if(new &&!*old){
6         *old = true;
7         return true;
8     } else{
9         *old = new;
10        return false;
11    }
12 }
13
14 int main(){
15     struct ComplexSignal{
16         bool out1;
17         bool out2;
18         short remaining;
19         short elapsed;
20     };
21     bool error, signal, toMode1, toMode2, toMode3, mode3Forbidden;
22     bool signal_old = false;

```

```

23  bool edge_signal = false;;
24  bool mode1 = false;
25  bool mode2 = false;
26  bool mode3 = false;
27  short mode = 0;
28  short cntr = 0;
29  short cntr2 = 0;
30  short cntr_max = 5;
31  bool out1 = false;
32  bool out2 = false;
33  struct ComplexSignal out3;
34  out3.out1 = false;
35  out3.out2 = false;
36  out3.elapsed = 0;
37  out3.remaining = 0;
38
39  while(true){
40      error = nondet_bool();
41      toMode1 = nondet_bool();
42      toMode2 = nondet_bool();
43      toMode3 = nondet_bool();
44      mode3Forbidden = nondet_bool();
45      signal = nondet_bool();
46
47      edge_signal = R.edge(signal,&signal_old);
48      if(!signal){
49          out1 = false;
50          out2 = false;
51          cntr = 0;
52      } else if(edge_signal){
53          out1 = !out1;
54          out3.out1 = out1;
55      } else{
56          cntr = cntr + 1;
57          while(cntr > cntr_max){
58              cntr = cntr - 1;;
59              cntr2 = cntr2 + 1;
60          }
61          if(cntr2 > cntr_max && signal){
62              out2 = true;
63          } else{
64              out2 = false;
65          }
66          out3.out1 = out1;
67          out3.out2 = out2;
68          out3.remaining = cntr_max - cntr;
69          out3.elapsed = cntr;
70      }
71      if (!mode1 && !mode2 && !mode3){
72          mode1 = true;
73      }
74      if(toMode1 || (toMode3 && mode3Forbidden)){
75          mode1 = true;
76      }
77      if(toMode2){
78          mode2 = true;
79      } else if(toMode3){
80          mode3 = true;
81      }
82      if(mode1){
83          mode = 1;
84      } else if(mode2){
85          mode = 2;
86      } else if(mode3){
87          mode = 3;
88      } else{
89          mode = 0;
90      }
91
92      assert(!out2 || out1);
93      assert(signal || !out2);
94      assert(out3.out1 == out1);
95      assert(out1 || (out3.elapsed == 0));
96  }
97 }

```

Listing D.5: Example_while.c

```

1  #include <stdbool.h>
2  bool nondet_bool();
3
4  bool R.edge(bool new, bool *old){
5      if(new && !*old){
6          *old = true;
7          return true;
8      } else{
9          *old = new;
10         return false;

```

```

11     }
12 }
13
14 int main(){
15     struct ComplexSignal{
16         bool out1;
17         short elapsed;
18     };
19     bool signal;
20     bool signal_old = false;
21     bool edge_signal = false;
22     short cntr = 0;
23     short cntr2 = 0;
24     short cntr_max = 5;
25     bool out1 = false;
26     bool out2 = false;
27     struct ComplexSignal out3;
28     out3.out1 = false;
29     out3.elapsed = 0;
30
31     while(true){
32         signal = nondet_bool();
33
34         edge_signal = R.edge(signal,&signal_old);
35         if(!signal){
36             out1 = false;
37             out2 = false;
38             cntr = 0;
39         } else if(edge_signal){
40             out1 = !out1;
41             out3.out1 = out1;
42         } else{
43             cntr = cntr + 1;
44             while(cntr > cntr_max){
45                 cntr = cntr - 1;;
46                 cntr2 = cntr2 + 1;
47             }
48             if(cntr2 > cntr_max && signal){
49                 out2 = true;
50             } else{
51                 out2 = false;
52             }
53             out3.out1 = out1;
54             out3.elapsed = cntr;
55         }
56
57         assert(!out2 || out1);
58         assert(signal || !out2);
59         assert(out3.out1 == out1);
60         assert(out1 || (out3.elapsed == 0));
61     }
62 }

```

Listing D.6: Reduced version of Example_while.smv

E CPC program in SCL

```

1 //UNICOS
2 //Copyright CERN 2013 all rights reserved
3
4 (* ON/OFF OBJECT FUNCTION BLOCK ***** )
5
6 FUNCTION_BLOCK CPC.FB.ONOFF
7 TITLE = 'CPC.FB.ONOFF'
8 //
9 // ONOFF Object
10 //
11 VERSION: '6.5'
12 AUTHOR: 'EN/ICE'
13 NAME: 'OBJECT'
14 FAMILY: 'FO'
15
16 VAR_INPUT
17
18     HFOn:          BOOL;
19     HFOff:         BOOL;
20     HLD:           BOOL;
21     IOError:      BOOL;
22     IOSimu:       BOOL;
23     AIB:          BOOL;
24     Manreg01:     WORD;
25     Manreg01b AT Manreg01: ARRAY [0..15] OF BOOL;
26     HOnR:         BOOL;

```



```

27   HOffR:          BOOL;
28   StartI:        BOOL;
29   TStopI:        BOOL;
30   FuStopI:       BOOL;
31   AI:            BOOL;
32   AuOnR:         BOOL;
33   AuOffR:        BOOL;
34   AuAuMoR:       BOOL;
35   AuIhMMo:       BOOL;
36   AuIhFoMo:      BOOL;
37   AuAIack:       BOOL;
38   IhAuMRW:       BOOL;
39   AuRstart:      BOOL;
40   POnOff:        CPC_ONOFF_PARAM;
41   POnOffb AT POnOff:  STRUCT
42   ParRegb:       ARRAY [0..15] OF BOOL;
43   PPulseLeb:    TIME;
44   PWDtb:        TIME;
45   END_STRUCT;
46
47
48 END_VAR
49
50 VAR_OUTPUT
51
52   Stsreg01:      WORD;
53   Stsreg01b AT Stsreg01:  ARRAY [0..15] OF BOOL;
54   Stsreg02:      WORD;
55   Stsreg02b AT Stsreg02:  ARRAY [0..15] OF BOOL;
56   OutOnOV:      BOOL;
57   OutOffOV:     BOOL;
58   OnSt:         BOOL;
59   OffSt:        BOOL;
60   AuMoSt:       BOOL;
61   MMoSt:        BOOL;
62   LDSt:         BOOL;
63   SoftLDSt:     BOOL;
64   FoMoSt:       BOOL;
65   AuOnRSt:      BOOL;
66   AuOffRSt:     BOOL;
67   MOnRSt:       BOOL;
68   MOffRSt:      BOOL;
69   HOnRSt:       BOOL;
70   HOffRSt:      BOOL;
71   IOErrorW:     BOOL;
72   IOSimuW:      BOOL;
73   AuMRW:        BOOL;
74   AIUnAck:      BOOL;
75   PosW:         BOOL;
76   StartISt:     BOOL;
77   TStopISt:    BOOL;
78   FuStopISt:   BOOL;
79   AISt:         BOOL;
80   AIBW:         BOOL;
81   EnRstartSt:   BOOL := TRUE;
82   RdyStartSt:   BOOL;
83
84
85 END_VAR
86
87 VAR //Internal Variables
88
89   //Variables for Edge detection
90   E_MAuMoR:     BOOL;
91   E_MMMoR:      BOOL;
92   E_MFoMoR:     BOOL;
93   E_MOnR:       BOOL;
94   E_MOffR:      BOOL;
95   E_MAIackR:    BOOL;
96   E_StartI:     BOOL;
97   E_TStopI:     BOOL;
98   E_FuStopI:    BOOL;
99   E_AI:         BOOL;
100  E_AuAuMoR:    BOOL;
101  E_AuAIack:    BOOL;
102  E_MSoftLDR:   BOOL;
103  E_MEnRstartR:  BOOL;
104  RE_AIUnAck:   BOOL;
105  FE_AIUnAck:   BOOL;
106  RE_PulseOn:   BOOL;
107  FE_PulseOn:   BOOL;
108  RE_PulseOff:  BOOL;
109  RE_OutOVSt_aux:  BOOL;
110  FE_OutOVSt_aux:  BOOL;
111  FE_InterlockR:  BOOL;
112
113   //Variables for old values
114  MAuMoR_old:   BOOL;
115  MMMoR_old:    BOOL;
116  MFoMoR_old:   BOOL;

```

```

117 MOnR_old:          BOOL;
118 MOffR_old:         BOOL;
119 MAIAckR_old:       BOOL;
120 AuAuMoR_old:       BOOL;
121 AuAlAck_old:       BOOL;
122 StartI_old:        BOOL;
123 TStopI_old:        BOOL;
124 FuStopI_old:       BOOL;
125 Al_old:             BOOL;
126 AlUnAck_old:       BOOL;
127 MSoftLDR_old:      BOOL;
128 MEnRstartR_old:    BOOL;
129 RE_PulseOn_old:    BOOL;
130 FE_PulseOn_old:    BOOL;
131 RE_PulseOff_old:   BOOL;
132 RE_OutOVSt_aux_old:  BOOL;
133 FE_OutOVSt_aux_old:  BOOL;
134 FE_InterlockR_old:  BOOL;
135
136 //General internal variables
137 PFsPosOn:          BOOL;
138 PFsPosOn2:         BOOL;
139 PHFOn:             BOOL;
140 PHFOff:            BOOL;
141 PPulse:            BOOL;
142 PPulseCste:        BOOL;
143 PHLD:              BOOL;
144 PHLDCmd:           BOOL;
145 PAnim:             BOOL;
146 POutOff:           BOOL;
147 PEnRstart:         BOOL;
148 PRstartFS:         BOOL;
149 OutOnOVSt:         BOOL;
150 OutOffOVSt:        BOOL;
151 AuMoSt_aux:        BOOL;
152 MMoSt_aux:         BOOL;
153 FoMoSt_aux:        BOOL;
154 SoftLDSt_aux:      BOOL;
155 PulseOn:           BOOL;
156 PulseOff:          BOOL;
157 PosW_aux:          BOOL;
158 OutOVSt_aux:       BOOL;
159 fullNotAcknowledged:  BOOL;
160 PulseOnR:          BOOL;
161 PulseOffR:         BOOL;
162 InterlockR:        BOOL;
163
164 //Variables for IEC Timers
165 Time_Warning:      TIME;
166 Timer_PulseOn:    TP;
167 Timer_PulseOff:   TP;
168 Timer_Warning:    TON;
169
170 //Variables for interlock Ststus delay handling
171 PulseWidth:       REAL;
172 FSIinc:           INT;
173 TSIinc:           INT;
174 SIinc:            INT;
175 Alinc:            INT;
176 WTStopISt:       BOOL;
177 WStartISt:        BOOL;
178 WAIS:             BOOL;
179 WFuStopISt:       BOOL;
180
181 END_VAR
182
183 BEGIN
184
185 (* INPUT MANAGER *)
186
187   E_MAuMoR      := R_EDGE(new:=ManReg01b[8], old:=MAuMoR_old);
188   (* Manual Auto Mode Request *)
189   E_MMMoR       := R_EDGE(new:=ManReg01b[9], old:=MMMoR_old);
190   (* Manual Manual Mode Request *)
191   E_MFoMoR      := R_EDGE(new:=ManReg01b[10], old:=MFoMoR_old);
192   (* Manual Forced Mode Request *)
193   E_MSoftLDR    := R_EDGE(new:=ManReg01b[11], old:=MSoftLDR_old);
194   (* Manual Software Local Drive Request *)
195   E_MOnR        := R_EDGE(new:=ManReg01b[12], old:=MOnR_old);
196   (* Manual On/Open Request *)
197   E_MOffR       := R_EDGE(new:=ManReg01b[13], old:=MOffR_old);
198   (* Manual Off/close Request *)
199   E_MEnRstartR := R_EDGE(new:=ManReg01b[1], old:=MEnRstartR_old);
200   (* Manual Restart after full stop Request *)
201   E_MAlAckR     := R_EDGE(new:=ManReg01b[7], old:=MAIAckR_old);
202   (* Manual Alarm Ack. Request *)
203
204   PFsPosOn := POnOffb.ParRegb[8];
205   (* 1st Parameter bit to define Fail safe position behaviour *)
206   PHFOn    := POnOffb.ParRegb[9];

```

```

207 (* Hardware feedback On present *)
208 PHFOff := POnOffb.ParRegb[10];
209 (* Hardware feedback Off present *)
210 PPulse := POnOffb.ParRegb[11];
211 (* Object is pulsed pulse duration : POnOff.PulseLe *)
212 PHLD := POnOffb.ParRegb[12];
213 (* Local Drive mode Allowed *)
214 PHLDCmd := POnOffb.ParRegb[13];
215 (* Local Drive Command allowed *)
216 PAnim := POnOffb.ParRegb[14];
217 (* Inverted Output *)
218 POutOff := POnOffb.ParRegb[15];
219 PEnRstart := POnOffb.ParRegb[0];
220 (* Enable Restart after Full Stop *)
221 PRstartFS := POnOffb.ParRegb[1];
222 (* Enable Restart when Full Stop still active *)
223 PFsPosOn2 := POnOffb.ParRegb[2];
224 (* 2nd Parameter bit to define Fail safe position behaviour *)
225 PPulseCste := POnOffb.ParRegb[3];
226 (* Pulse Constant duration irrespective of the feedback status *)
227
228 E_AuAuMoR := R_EDGE(new:=AuAuMoR, old:=AuAuMoR_old);
229 (* Auto Auto Mode Request *)
230 E_AuAlAck := R_EDGE(new:=AuAlAck, old:=AuAlAck_old);
231 (* Auto Alarm Ack. Request *)
232
233 E_StartI := R_EDGE(new:=StartI, old:=StartI_old);
234 E_TStopI := R_EDGE(new:=TStopI, old:=TStopI_old);
235 E_FuStopI := R_EDGE(new:=FuStopI, old:=FuStopI_old);
236 E_Al := R_EDGE(new:=Al, old:=Al_old);
237
238 StartISt := StartI;
239 (* Start Interlock present *)
240 TStopISt := TStopI;
241 (* Temporary Stop Interlock present *)
242 FuStopISt := FuStopI;
243 (* Full Stop Interlock present *)
244
245 (* INTERLOCK & ACKNOWLEDGE *)
246
247 IF (E_MAIAckR OR E_AuAlAck) THEN
248     fullNotAcknowledged := FALSE;
249     AUnAck := FALSE;
250 ELSEIF (E_TStopI OR E_StartI OR E_FuStopI OR E_Al) THEN
251     AUnAck := TRUE;
252 END_IF;
253
254 IF ((PEnRstart AND (E_MEnRstartR OR AuRstart)) AND NOT FuStopISt) OR (PEnRstart AND
    PRstartFS AND (E_MEnRstartR OR AuRstart))) AND NOT fullNotAcknowledged THEN
255     EnRstartSt := TRUE;
256 END_IF;
257
258 IF E_FuStopI THEN
259     fullNotAcknowledged := TRUE;
260     IF PEnRstart THEN
261         EnRstartSt := FALSE;
262     END_IF;
263 END_IF;
264
265 InterlockR := (TStopISt OR FuStopISt OR FullNotAcknowledged OR NOT EnRstartSt OR
    (StartISt AND NOT POutOff AND NOT OutOnOV) OR
    (StartISt AND POutOff AND ((PFsPosOn AND OutOVSt_aux) OR (NOT PFsPosOn
    AND NOT OutOVSt_aux))));
266
267 FE_InterlockR := F_EDGE (new:=InterlockR, old:=FE_InterlockR_old);
268
269 (* MODE MANAGER *)
270
271 IF NOT (HLD AND PHLD) THEN
272
273     (* Forced Mode *)
274     IF (AuMoSt_aux OR MMoSt_aux OR SoftLDSt_aux) AND
275         EMFoMoR AND NOT(AuIhFoMo) THEN
276         AuMoSt_aux := FALSE;
277         MMoSt_aux := FALSE;
278         FoMoSt_aux := TRUE;
279         SoftLDSt_aux := FALSE;
280     END_IF;
281
282     (* Manual Mode *)
283     IF (AuMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux) AND
284         EMMMoR AND NOT(AuIhMMo) THEN
285         AuMoSt_aux := FALSE;
286         MMoSt_aux := TRUE;
287         FoMoSt_aux := FALSE;
288         SoftLDSt_aux := FALSE;
289     END_IF;
290
291     (* Auto Mode *)
292     IF (MMoSt_aux AND (E_MAuMoR OR E_AuAuMoR )) OR

```

```

295         (FoMoSt_aux AND E_MAuMoR) OR
296         (SoftLDSt_aux AND E_MAuMoR) OR
297         (MMoSt_aux AND AuIhMMo) OR
298         (FoMoSt_aux AND AuIhFoMo)OR
299         (SoftLDSt_aux AND AuIhFoMo) OR
300         NOT(AuMoSt_aux OR MMoSt_aux OR FoMoSt_aux OR SoftLDSt_aux) THEN
301             AuMoSt_aux := TRUE;
302             MMoSt_aux := FALSE;
303             FoMoSt_aux := FALSE;
304             SoftLDSt_aux := FALSE;
305         END_IF;
306
307     (* Software Local Mode *)
308     IF (AuMoSt_aux OR MMoSt_aux) AND E_MSoftLDR AND NOT AuIhFoMo THEN
309         AuMoSt_aux := FALSE;
310         MMoSt_aux := FALSE;
311         FoMoSt_aux := FALSE;
312         SoftLDSt_aux:= TRUE;
313     END_IF;
314
315     (* Status setting *)
316     LDSt := FALSE;
317     AuMoSt := AuMoSt_aux;
318     MMoSt := MMoSt_aux;
319     FoMoSt := FoMoSt_aux;
320     SoftLDSt := SoftLDSt_aux;
321
322     ELSE
323     (* Local Drive Mode *)
324     AuMoSt := FALSE;
325     MMoSt := FALSE;
326     FoMoSt := FALSE;
327     LDSt := TRUE;
328     SoftLDSt:= FALSE;
329     END_IF;
330
331     (* LIMIT MANAGER *)
332     (* On/Open Evaluation *)
333     OnSt:= (HFOOn AND PHFOOn) OR
334     (*Feedback ON present*)
335     (NOT PHFOOn AND PHFOff AND PAnim AND NOT HFOff) OR
336     (*Feedback ON not present and PAnim = TRUE*)
337     (NOT PHFOOn AND NOT PHFOff AND OutOVSt_aux);
338
339
340
341     (* Off/Closed Evaluation *)
342     OffSt:=(HFOff AND PHFOff) OR
343     (*Feedback OFF present*)
344     (NOT PHFOff AND PHFOOn AND PAnim AND NOT HFOOn) OR
345     (*Feedback OFF not present and PAnim = TRUE*)
346     (NOT PHFOOn AND NOT PHFOff AND NOT OutOVSt_aux);
347
348     (* REQUEST MANAGER *)
349
350     (* Auto On/Off Request*)
351
352     IF AuOffR THEN
353         AuOnRSt := FALSE;
354     ELSIF AuOnR THEN
355         AuOnRSt := TRUE;
356     ELSIF fullNotAcknowledged OR FuStopISt OR NOT EnRstartSt THEN
357         AuOnRSt := PFsPosOn;
358     END_IF;
359     AuOffRSt:= NOT AuOnRSt;
360
361     (* Manual On/Off Request*)
362
363     IF (((E_MOffR AND (MMoSt OR FoMoSt OR SoftLDSt))
364     OR (AuOffRSt AND AuMoSt)
365     OR (LDSt AND PHLDCmd AND HOffRSt)
366     OR (FE_PulseOn AND PPulse AND NOT POutOff) AND EnRstartSt)
367     OR (E_FuStopI AND NOT PFsPosOn)) THEN
368
369         MonRSt := FALSE;
370
371     ELSIF (((E_MOnR AND (MMoSt OR FoMoSt OR SoftLDSt))
372     OR (AuOnRSt AND AuMoSt)
373     OR (LDSt AND PHLDCmd AND HOnRSt) AND EnRstartSt)
374     OR (E_FuStopI AND PFsPosOn)) THEN
375
376         MonRSt := TRUE;
377     END_IF;
378
379     MOffRSt:= NOT MonRSt;
380
381     (* Local Drive Request *)
382
383     IF HOffR THEN
384         HOnRSt := FALSE;

```

```

385     ELSE IF HOnR THEN
386         HOnRSt := TRUE;
387         END_IF;
388     END_IF;
389     HOffRSt := NOT(HOnRSt);
390
391 (* PULSE REQUEST MANAGER*)
392 IF PPulse THEN
393     IF InterlockR THEN
394         PulseOnR:= (PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2);
395         PulseOffR:= (NOT PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2);
396     ELSIF FE_InterlockR THEN
397         (* Clear PulseOnR/PulseOffR to be sure you get a new pulse after InterlockR *)
398         PulseOnR:= FALSE;
399         PulseOffR:= FALSE;
400         Timer_PulseOn (IN:=FALSE,PT:=T#0s);
401         Timer_PulseOff (IN:=FALSE,PT:=T#0s);
402     ELSIF (MOffRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuOffRSt AND AuMoSt) OR (HOffR
403         AND LDSt AND PHLDCmd) THEN //Off Request
404         PulseOnR:= FALSE;
405         PulseOffR:= TRUE;
406     ELSIF (MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (AuOnRSt AND AuMoSt) OR (HOnR
407         AND LDSt AND PHLDCmd) THEN //On Request
408         PulseOnR:= TRUE;
409         PulseOffR:= FALSE;
410     ELSE
411         PulseOnR:= FALSE;
412         PulseOffR:= FALSE;
413     END_IF;
414
415 //Pulse functions
416 Timer_PulseOn (IN:= PulseOnR,PT:=POnOffb.PPulseLeb);
417 Timer_PulseOff (IN:=PulseOffR,PT:=POnOffb.PPulseLeb);
418
419 RE_PulseOn := R_EDGE(new:=PulseOn,old:=RE_PulseOn_old);
420 FE_PulseOn := F_EDGE(new:=PulseOn,old:=FE_PulseOn_old);
421 RE_PulseOff := R_EDGE(new:=PulseOff,old:=RE_PulseOff_old);
422
423 //The pulse functions have to be reset when changing from On to Off
424 IF RE_PulseOn THEN
425     Timer_PulseOff (IN:=FALSE,PT:=T#0s);
426 END_IF;
427
428 IF RE_PulseOff THEN
429     Timer_PulseOn (IN:=FALSE,PT:=T#0s);
430 END_IF;
431
432 IF PPulseCste THEN
433     (* Pulse constant duration irrespective of feedback status *)
434     PulseOn := Timer_PulseOn.Q AND NOT PulseOffR;
435     PulseOff := Timer_PulseOff.Q AND NOT PulseOnR;
436 ELSE
437     PulseOn := Timer_PulseOn.Q AND NOT PulseOffR AND (NOT PHFOn OR (PHFOn AND NOT
438         HFOn));
439     PulseOff := Timer_PulseOff.Q AND NOT PulseOnR AND (NOT PHFOff OR (PHFOff AND NOT
440         HFOff));
441 END_IF;
442 END_IF;
443
444 (* Output On Request *)
445 OutOnOVSt := (PPulse AND PulseOn) OR
446     (NOT PPulse AND ((MOnRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR
447     (AuOnRSt AND AuMoSt) OR
448     (HOnRSt AND LDSt AND PHLDCmd)));
449
450 (* Output Off Request *)
451 IF POutOff THEN
452     OutOffOVSt := (PulseOff AND PPulse) OR
453     (NOT(PPulse) AND ((MOffRSt AND (MMoSt OR FoMoSt OR SoftLDSt)) OR (
454     AuOffRSt AND AuMoSt) OR (HOffRSt AND LDSt AND PHLDCmd)));
455 END_IF;
456
457 (* Interlocks / FailSafe *)
458 IF POutOff THEN
459     IF InterlockR THEN
460         IF PPulse AND NOT PFsPosOn2 THEN
461             IF PFsPosOn THEN
462                 OutOnOVSt := PulseOn;
463                 OutOffOVSt := FALSE;
464             ELSE
465                 OutOnOVSt := FALSE;
466                 OutOffOVSt := PulseOff;
467             END_IF;
468         ELSE
469             OutOnOVSt := (PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2);
470             OutOffOVSt:= (NOT PFsPosOn AND NOT PFsPosOn2) OR (PFsPosOn AND PFsPosOn2
471 );
472         END_IF;
473     END_IF;
474 END_IF;

```

```

469         END_IF;
470     END_IF;
471     ELSE
472         IF InterlockR THEN
473             OutOnOVSt:= PFsPosOn;
474         END_IF;
475     END_IF;
476
477 (* Ready to Start Status *)
478     RdyStartSt := NOT InterlockR;
479
480 (*Alarms*)
481     AlSt := Al;
482
483 (* SURVEILLANCE *)
484
485 (* I/O Warning *)
486     IOErrorW := IOError;
487     IOSimuW := IOSimu;
488
489 (* Auto↔ Manual Warning *)
490     AuMRW := (MMoSt OR FoMoSt OR SoftLDSt) AND
491             ((AuOnRSt XOR MOnRSt) OR (AuOffRSt XOR MOffRSt)) AND NOT lhAuMRW;
492
493 (* OUTPUTMANAGER AND OUTPUT REGISTER *)
494     IF NOT POutOff THEN
495         IF PFsPosOn THEN
496             OutOnOV := NOT OutOnOVSt;
497         ELSE
498             OutOnOV := OutOnOVSt;
499         END_IF;
500     ELSE
501         OutOnOV := OutOnOVSt;
502         OutOffOV := OutOffOVSt;
503     END_IF;
504
505 (* Position warning *)
506
507 (* Set reset of the OutOnOVSt *)
508     IF OutOnOVSt OR (PPulse AND PulseOnR) THEN
509         OutOVSt_aux := TRUE;
510     END_IF;
511     IF (OutOffOVSt AND POutOff) OR (NOT OutOnOVSt AND NOT POutOff) OR (PPulse AND
512     PulseOffR) THEN
513         OutOVSt_aux := FALSE;
514     END_IF;
515
516     RE_OutOVSt_aux := R_EDGE(new:=OutOVSt_aux, old:=RE_OutOVSt_aux_old);
517     FE_OutOVSt_aux := F_EDGE(new:=OutOVSt_aux, old:=FE_OutOVSt_aux_old);
518
519     IF ((OutOVSt_aux AND ((PHFOn AND NOT OnSt) OR (PHFOff AND OffSt)))
520     OR (NOT OutOVSt_aux AND ((PHFOff AND NOT OffSt) OR (PHFON AND OnSt)))
521     OR (OffSt AND OnSt))
522     AND (NOT PPulse OR (POutOff AND PPulse AND NOT OutOnOV AND NOT OutOffOV))
523     THEN
524         PosW_aux:= TRUE;
525     END_IF;
526
527     IF NOT ((OutOVSt_aux AND ((PHFOn AND NOT OnSt) OR (PHFOff AND OffSt)))
528     OR (NOT OutOVSt_aux AND ((PHFOff AND NOT OffSt) OR (PHFON AND OnSt)))
529     OR (OffSt AND OnSt))
530     OR RE_OutOVSt_aux
531     OR FE_OutOVSt_aux
532     OR (PPulse AND POutOff AND OutOnOV)
533     OR (PPulse AND POutOff AND OutOffOV)
534     THEN
535         PosW_aux := FALSE;
536     END_IF;
537
538     Timer_Warning(IN := PosW_aux,
539     PT := POnOffb.PWDtb);
540
541     PosW := Timer_Warning.Q;
542     Time_Warning := Timer_Warning.ET;
543
544 (* Alarm Blocked Warning*)
545     AIBW := AIB;
546
547 (* Maintain Interlock status 1.5s in Stsreg for PVSS *)
548     PulseWidth := 1500 (* msec*) / DINT.TO_REAL(TIME.TO_DINT(T.CYCLE));
549
550     IF FuStopISt OR FSIinc > 0 THEN
551         FSIinc := FSIinc + 1;
552     END_IF;

```

```

558   WFuStopISt := TRUE;
559 END_IF;
560
561 IF FSIinc > PulseWidth OR (NOT FuStopISt AND FSIinc = 0) THEN
562   FSIinc := 0;
563   WFuStopISt := FuStopISt;
564 END_IF;
565
566 IF TStopISt OR TSIinc > 0 THEN
567   TSIinc := TSIinc + 1;
568   WTStopISt := TRUE;
569 END_IF;
570
571 IF TSIinc > PulseWidth OR (NOT TStopISt AND TSIinc = 0) THEN
572   TSIinc := 0;
573   WTStopISt := TStopISt;
574 END_IF;
575
576 IF StartISt OR SIinc > 0 THEN
577   SIinc := SIinc + 1;
578   WStartISt:= TRUE;
579 END_IF;
580
581 IF SIinc > PulseWidth OR (NOT StartISt AND SIinc = 0) THEN
582   SIinc := 0;
583   WStartISt := StartISt;
584 END_IF;
585
586 IF AISt OR Alinc > 0 THEN
587   Alinc := Alinc + 1;
588   WASt := TRUE;
589 END_IF;
590
591 IF Alinc > PulseWidth OR (NOT AISt AND Alinc = 0) THEN
592   Alinc := 0;
593   WASt := AISt;
594 END_IF;
595
596
597 (* STATUS REGISTER *)
598
599   Stsreg01b[8] := OnSt;           //StsReg01 Bit 00
600   Stsreg01b[9] := OffSt;        //StsReg01 Bit 01
601   Stsreg01b[10] := AuMoSt;      //StsReg01 Bit 02
602   Stsreg01b[11] := MMoSt;      //StsReg01 Bit 03
603   Stsreg01b[12] := FoMoSt;     //StsReg01 Bit 04
604   Stsreg01b[13] := LDSt;       //StsReg01 Bit 05
605   Stsreg01b[14] := IOErrorW;   //StsReg01 Bit 06
606   Stsreg01b[15] := IOSimuW;    //StsReg01 Bit 07
607   stsreg01b[0] := AuMRW;        //StsReg01 Bit 08
608   Stsreg01b[1] := PosW;        //StsReg01 Bit 09
609   Stsreg01b[2] := WStartISt;   //StsReg01 Bit 10
610   Stsreg01b[3] := WTStopISt;   //StsReg01 Bit 11
611   Stsreg01b[4] := AIUnAck;     //StsReg01 Bit 12
612   Stsreg01b[5] := AulhFoMo;    //StsReg01 Bit 13
613   Stsreg01b[6] := WASt;        //StsReg01 Bit 14
614   Stsreg01b[7] := AulhMMo;     //StsReg01 Bit 15
615
616   Stsreg02b[8] := OutOnOVSt;    //StsReg02 Bit 00
617   Stsreg02b[9] := AuOnRSt;     //StsReg02 Bit 01
618   Stsreg02b[10] := MOnRSt;    //StsReg02 Bit 02
619   Stsreg02b[11] := AuOffRSt;  //StsReg02 Bit 03
620   Stsreg02b[12] := MOffRSt;   //StsReg02 Bit 04
621   Stsreg02b[13] := HOOnRSt;   //StsReg02 Bit 05
622   Stsreg02b[14] := HOFFRSt;   //StsReg02 Bit 06
623   Stsreg02b[15] := 0;         //StsReg02 Bit 07
624   stsreg02b[0] := 0;          //StsReg02 Bit 08
625   Stsreg02b[1] := 0;          //StsReg02 Bit 09
626   Stsreg02b[2] := WFuStopISt ; //StsReg02 Bit 10
627   Stsreg02b[3] := EnRstartSt;  //StsReg02 Bit 11
628   Stsreg02b[4] := SoftLDSt;    //StsReg02 Bit 12
629   Stsreg02b[5] := AIBW;        //StsReg02 Bit 13
630   Stsreg02b[6] := OutOffOVSt;  //StsReg02 Bit 14
631   Stsreg02b[7] := 0;          //StsReg02 Bit 15
632
633 (* Edges *)
634
635   DETECT_EDGE(new:=AIUnAck, old:=AIUnAck_old, re:=RE_AIUnAck, fe:=FE_AIUnAck);
636
637
638 END_FUNCTION_BLOCK
639
640
641 //////////////////////////////////////
642 //////////////////////////////////////
643 //
644 //   Common functions for UNICOS applications + implementation of platform FBs.
645 //
646 //////////////////////////////////////
647 //////////////////////////////////////

```

```

648
649
650
651 //UNICOS
652 // Copyright CERN 2013 all rights reserved
653
654 (* DATA STRUCTURES *****)
655
656 TYPE CPC_ONOFF_PARAM
657 TITLE = 'CPC_ONOFF_PARAM'
658 //
659 // Parameters of ONOFF
660 //
661 AUTHOR : 'EN/ICE'
662 NAME : 'DataType'
663 FAMILY : 'Base'
664 STRUCT
665     ParReg : WORD;
666     PPulseLe : TIME;
667     PWDt : TIME;
668 END_STRUCT
669 END_TYPE
670
671 (* OTHER FUNCTIONS *****)
672
673 (* Rising Edge*)
674 FUNCTION R_EDGE : BOOL
675 TITLE = 'R_EDGE'
676 //
677 // Detect a Rising Edge on a signal
678 //
679 AUTHOR : 'EN/ICE'
680 NAME : 'Function'
681 FAMILY : 'Base'
682 VAR_INPUT
683     new : BOOL;
684 END_VAR
685 VAR_IN_OUT
686     old : BOOL;
687 END_VAR
688 BEGIN
689
690     IF (new = 1 AND old = 0) THEN //Raising edge detected
691         R_EDGE := 1;
692         old := 1;
693     ELSE R_EDGE := 0;
694         old := new;
695     END_IF;
696 END_FUNCTION
697
698 (* Falling Edge*)
699 FUNCTION F_EDGE : BOOL
700 TITLE = 'F_EDGE'
701 //
702 // Detect a Falling Edge on a signal
703 //
704 AUTHOR : 'EN/ICE'
705 NAME : 'Function'
706 FAMILY : 'Base'
707 VAR_INPUT
708     new : BOOL;
709 END_VAR
710 VAR_IN_OUT
711     old : BOOL;
712 END_VAR
713 BEGIN
714
715     IF (new = 0 AND old = 1) THEN //Falling edge detected
716         F_EDGE := 1;
717         old := 0;
718     ELSE F_EDGE := 0;
719         old := new;
720     END_IF;
721 END_FUNCTION
722
723 (* Rising and Falling Edge*)
724 FUNCTION DETECT_EDGE : VOID
725 TITLE = 'DETECT_EDGE'
726 //
727 // Detect a Rising and Falling Edge of a signal
728 //
729 AUTHOR : 'EN/ICE'
730 NAME : 'Function'
731 FAMILY : 'Base'
732 VAR_INPUT
733     new : BOOL;
734 END_VAR
735 VAR_IN_OUT
736     old : BOOL;
737 END_VAR

```



```
738 VAR_OUTPUT
739     re : BOOL;
740     fe : BOOL;
741 END_VAR
742
743 BEGIN
744     IF new <> old THEN
745         IF new = true THEN // Raising edge
746             re := true;
747             fe := false;
748         ELSE // Falling edge
749             re := false;
750             fe := true;
751         END_IF;
752         old := new; // shift new to old
753
754     ELSE re := false; // reset edge detection
755         fe := false;
756     END_IF;
757 END_FUNCTION
758
759 // TIMERS
760 //# GLOBALVAR _GLOBAL_TIME : TIME;
761 //# GLOBALVAR T_CYCLE : UINT;
762
763 // Pulse timer
764 FUNCTION_BLOCK TP
765 // updated on 10 Oct 2013
766 VAR_INPUT
767     PT : TIME;
768 END_VAR
769
770 VAR_IN_OUT
771     IN : BOOL;
772 END_VAR
773
774 VAR_OUTPUT
775     Q : BOOL := FALSE;
776     ET : TIME; // elapsed time
777
778 END_VAR
779
780 VAR
781     old_in : BOOL := FALSE;
782     due : TIME := T#0ms;
783 END_VAR
784
785 BEGIN
786     if (in and not old_in) and not Q then
787         due := _GLOBAL_TIME + pt;
788     end_if;
789     if _GLOBAL_TIME <= due then
790         Q := true;
791         ET := PT - (due - _GLOBAL_TIME);
792     else Q := false;
793         if in then
794             ET := PT;
795         else ET := 0;
796         end_if;
797     end_if;
798     old_in := in;
799 END_FUNCTION_BLOCK
800
801 // On-delay timer
802 FUNCTION_BLOCK TON
803 // !!! It is assumed that PT>0, if IN=true. !!!
804 // updated on 20/05/2014
805 VAR_INPUT
806     PT : TIME; // pulse time
807     IN : BOOL;
808 END_VAR
809
810 VAR_OUTPUT
811     Q : BOOL := FALSE;
812     ET : TIME := T#0s; // elapsed time
813
814 END_VAR
815
816 VAR
817     running : BOOL := FALSE;
818     start : TIME := T#0ms; //STIME in Siemens implementation (?)
819 END_VAR
820
821 BEGIN
822     if IN = false then
823         Q := false;
824         ET := T#0s;
825         running := false;
826     else
827         // in this case IN == TRUE
```

```

828     if running = false then
829         // just started
830         start := _GLOBAL_TIME;
831         running := true;
832         ET := 0;
833         // Q = false, ET = 0
834
835     else
836         if not (_GLOBAL_TIME - (start + pt) >= T#0s) then
837             // running, but no timeout
838             // do not modify Q
839             if not Q then
840                 ET := _GLOBAL_TIME - start;
841             end_if;
842             // ET should be good even if GT>0 & due < 0
843
844         else
845             // timeout
846             Q := true;
847             ET := PT;
848         end_if;
849     end_if;
850 end_if;
851 END_FUNCTION_BLOCK

```

Listing E.1: CPC.scl

F CPC program in SMV

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

```

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     FuStopI : boolean;
18     Al : boolean;
19     AuOnR : boolean;
20     AuOffR : boolean;
21     AuAuMoR : boolean;
22     AulhMMo : boolean;
23     AulhFoMo : boolean;
24     AuAlAck : boolean;
25     lhAuMRW : boolean;
26     AuRstart : boolean;
27     POnOffb.ParRegb : array 0..15 of boolean;
28     POnOffb.PPulseLeb : signed word[32];
29     POnOffb.PWDtb : signed word[32];
30
31     --VAR_OUTPUT
32     Stsreg01b : array 0..15 of boolean;
33     Stsreg02b : array 0..15 of boolean;
34     OutOnOV : boolean;
35     OutOffOV : boolean;
36     OnSt : boolean;
37     OffSt : boolean;
38     AuMoSt : boolean;
39     MMoSt : boolean;
40     LDSt : boolean;
41     SoftLDSt : boolean;
42     FoMoSt : boolean;
43     AuOnRSt : boolean;
44     AuOffRSt : boolean;
45     MOnRSt : boolean;
46     MOffRSt : boolean;
47     HOnRSt : boolean;
48     HOffRSt : boolean;
49     IOErrorW : boolean;
50     IOSimuW : boolean;
51     AuMRW : boolean;

```

```

52  AIUnAck : boolean;
53  PosW : boolean;
54  StartISt : boolean;
55  TStopISt : boolean;
56  FuStopISt : boolean;
57  AIST : boolean;
58  AIBW : boolean;
59  EnRstartSt : boolean;
60  RdyStartSt : boolean;
61
62  --Internal Variables
63  --Variables for Edge detection
64  E_MAuMoR : boolean;
65  E_MMMoR : boolean;
66  E_MFoMoR : boolean;
67  E_MOnR : boolean;
68  E_MOffR : boolean;
69  E_MAlAckR : boolean;
70  E_StartI : boolean;
71  E_TStopI : boolean;
72  E_FuStopI : boolean;
73  E_Al : boolean;
74  E_AuAuMoR : boolean;
75  E_AuAlAck : boolean;
76  E_MSoftLDR : boolean;
77  E_MEnRstartR : boolean;
78  RE_AIUnAck : boolean;
79  FE_AIUnAck : boolean;
80  RE_PulseOn : boolean;
81  FE_PulseOn : boolean;
82  RE_PulseOff : boolean;
83  RE_OutOVSt_aux : boolean;
84  FE_OutOVSt_aux : boolean;
85  FE_InterlockR : boolean;
86
87  --Variables for old values
88  MAuMoR_old : boolean;
89  MMMoR_old : boolean;
90  MFoMoR_old : boolean;
91  MOnR_old : boolean;
92  MOffR_old : boolean;
93  MAlAckR_old : boolean;
94  AuAuMoR_old : boolean;
95  AuAlAck_old : boolean;
96  StartI_old : boolean;
97  TStopI_old : boolean;
98  FuStopI_old : boolean;
99  Al_old : boolean;
100 AIUnAck_old : boolean;
101 MSoftLDR_old : boolean;
102 MEnRstartR_old : boolean;
103 RE_PulseOn_old : boolean;
104 FE_PulseOn_old : boolean;
105 RE_PulseOff_old : boolean;
106 RE_OutOVSt_aux_old : boolean;
107 FE_OutOVSt_aux_old : boolean;
108 FE_InterlockR_old : boolean;
109
110 --General internal variables
111 PFsPosOn : boolean;
112 PFsPosOn2 : boolean;
113 PHFOn : boolean;
114 PHFOff : boolean;
115 PPulse : boolean;
116 PPulseCste : boolean;
117 PHLD : boolean;
118 PHLDCmd : boolean;
119 PAnim : boolean;
120 POutOff : boolean;
121 PEnRstart : boolean;
122 PRstartFS : boolean;
123 OutOnOVSt : boolean;
124 OutOffOVSt : boolean;
125 AuMoSt_aux : boolean;
126 MMoSt_aux : boolean;
127 FoMoSt_aux : boolean;
128 SoftLDSt_aux : boolean;
129 PulseOn : boolean;
130 PulseOff : boolean;
131 PosW_aux : boolean;
132 OutOVSt_aux : boolean;
133 fullNotAcknowledged : boolean;
134 PulseOnR : boolean;
135 PulseOffR : boolean;
136 InterlockR : boolean;
137
138 --Variables for IEC Timers
139 Time_Warning : signed word[32];
140 Timer_PulseOn.Q : boolean;
141 Timer_PulseOn.ET : signed word[32];

```

```

142 Timer_PulseOn.old_in : boolean;
143 Timer_PulseOn.due : signed word[32];
144 Timer_PulseOff.Q : boolean;
145 Timer_PulseOff.ET : signed word[32];
146 Timer_PulseOff.old_in : boolean;
147 Timer_PulseOff.due : signed word[32];
148 Timer_Warning.Q : boolean;
149 Timer_Warning.ET : signed word[32];
150 Timer_Warning.running : boolean;
151 Timer_Warning.Tstart : signed word[32];
152
153 --Variables for interlock Ststus delay handling
154 PulseWidth : signed word[32];
155 FSIinc : signed word[16];
156 TSIinc : signed word[16];
157 SIinc : signed word[16];
158 Alinc : signed word[16];
159 WTStopISt : boolean;
160 WStartISt : boolean;
161 WAIS : boolean;
162 WFuStopISt : boolean;
163 _GLOBAL_TIME : signed word[32];
164 T_CYCLE : unsigned word[16];
165 random_t_cycle : unsigned word[8];
166
167 --extra assertion variables
168 sFoMoSt_aux : boolean;
169 sAuAuMoR : boolean;
170 sManreg01b8 : boolean;
171 sAulhFoMo : boolean;
172 sAulhMMo : boolean;
173 sMMoSt_aux : boolean;
174 pOutOnOV : boolean;
175 pTStopI : boolean;
176 pFuStopI : boolean;
177 pStartI : boolean;
178 pOutOnOVSt : boolean;
179 pMMoSt : boolean;
180 pManreg01b12 : boolean;
181 pManreg01b13 : boolean;
182 first : boolean;
183
184 loc:{start, step1, step2, step3, step4, step5, step6, step7, step8, step9, step10, step
185 11, step12, step13, step14, step15,
186 step16, step17, step18, step19, step20, step21, step22, step23, step24, step25, step26,
187 step27, step28, step29, step30,
188 step31, step32, step33, step34, step35, step36, step37, step38, step39, step40, step41,
189 step42, step43, step44, step45,
190 step46, step47, step48, step49, step50, step51, step52, step53, step54, step55, step56,
191 step57, step58, step59, step60,
192 step61, step62, step63, step64, step65, step66, step67, step68, step69, step70, step71,
193 step72, step73, step74, step75,
194 step76, step77, step78, step79, step80, step81, step82, step83, step84, step85, step86,
195 step87, step88, step89, step90,
196 step91, step92, step93, step94, step95, step96, step97, step98, step99, step100, step
197 101, step102, step103, step104, step105,
198 step106, step107, step108, step109, step110, step111, step112, step113, step114, step
199 115, step116, step117, step118, step119,
200 step120, step121, step122, step123, step124, step125, step126, step127, step128, step
201 129, step130, step131, step132, step133,
202 step134, step135, step136, step137, step138, step139, step140, step141, step142, step
203 143, step144, step145, step146, step147,
204 step148, step149, step150, step151, step152, step153, step154, step155, step156, step
205 157, step158, step159, step160, step161,
206 step162, step163, step164, step165, step166, step167, step168, step169, step170, step
207 171, step172, step173, step174, step175,
208 step176, step177, step178, step179, step180, step181, step182, step183, step184, step
209 185, step186, step187, step188, step189,
210 step190, step191, step192, step193, step194, step195, step196, step197, step198, step
211 199, step200, step201, step202, step203,
212 step204, step205, step206, step207, step208, step209, step210, step211, step212, step
213 213, step214, step215, step216, step217,
214 step218, step219, step220, step221, end, nvar};
215
216 ASSIGN
217 --Location specification
218 init(loc) := start;
219 next(loc) :=
220 case
221 (loc = start) : step1;
222 (loc = step1) & (Manreg01b[8] & !MAuMoR_old) : step2;
223 (loc = step1) : step3;
224 (loc = step2) : step4;
225 (loc = step3) : step4;
226 (loc = step4) & (Manreg01b[9] & !MMoR_old) : step5;
227 (loc = step4) : step6;
228 (loc = step5) : step7;
229 (loc = step6) : step7;
230 (loc = step7) & (Manreg01b[10] & !MFoMoR_old) : step8;
231 (loc = step7) : step9;

```

```

217 (loc = step8) : step10;
218 (loc = step9) : step10;
219 (loc = step10) & (Manreg01b[11] & !MSoftLDR_old) : step11;
220 (loc = step10) : step12;
221 (loc = step11) : step13;
222 (loc = step12) : step13;
223 (loc = step13) & (Manreg01b[12] & !MOnR_old) : step14;
224 (loc = step13) : step15;
225 (loc = step14) : step16;
226 (loc = step15) : step16;
227 (loc = step16) & (Manreg01b[13] & !MOffR_old) : step17;
228 (loc = step16) : step18;
229 (loc = step17) : step19;
230 (loc = step18) : step19;
231 (loc = step19) & (Manreg01b[1] & !MEnRstartR_old) : step20;
232 (loc = step19) : step21;
233 (loc = step20) : step22;
234 (loc = step21) : step22;
235 (loc = step22) & (Manreg01b[7] & !MAIAckR_old) : step23;
236 (loc = step22) : step24;
237 (loc = step23) : step25;
238 (loc = step24) : step25;
239 (loc = step25) : step26;
240 (loc = step26) & (AuAuMoR & !AuAuMoR_old) : step27;
241 (loc = step26) : step28;
242 (loc = step27) : step29;
243 (loc = step28) : step29;
244 (loc = step29) & (AuAlAck & !AuAlAck_old) : step30;
245 (loc = step29) : step31;
246 (loc = step30) : step32;
247 (loc = step31) : step32;
248 (loc = step32) & (StartI & !StartI_old) : step33;
249 (loc = step32) : step34;
250 (loc = step33) : step35;
251 (loc = step34) : step35;
252 (loc = step35) & (TStopI & !TStopI_old) : step36;
253 (loc = step35) : step37;
254 (loc = step36) : step38;
255 (loc = step37) : step38;
256 (loc = step38) & (FuStopI & !FuStopI_old) : step39;
257 (loc = step38) : step40;
258 (loc = step39) : step41;
259 (loc = step40) : step41;
260 (loc = step41) & (Al & !Al_old) : step42;
261 (loc = step41) : step43;
262 (loc = step42) : step44;
263 (loc = step43) : step44;
264 (loc = step44) : step45;
265 (loc = step45) & (E_MAlAckR | E_AuAlAck) : step46;
266 (loc = step45) & (E_TStopI | E_StartI | E_FuStopI | E_Al) : step47;
267 (loc = step45) : step48;
268 (loc = step46) : step48;
269 (loc = step47) : step48;
270 (loc = step48) & (((PEnrstart & (E_MEnRstartR | AuRstart) & !FuStopISt) | (
PEnrstart & PRstartFS & (E_MEnRstartR | AuRstart))) & !fullNotAcknowledged)
: step49;
271 (loc = step48) : step50;
272 (loc = step49) : step50;
273 (loc = step50) & (E_FuStopI) : step51;
274 (loc = step50) : step54;
275 (loc = step51) : step52;
276 (loc = step52) & (PEnrstart) : step53;
277 (loc = step52) : step54;
278 (loc = step53) : step54;
279 (loc = step54) : step55;
280 (loc = step55) & (!InterlockR & FE_InterlockR_old) : step56;
281 (loc = step55) : step57;
282 (loc = step56) : step58;
283 (loc = step57) : step58;
284 (loc = step58) & (!(HLD & PHL)) : step59;
285 (loc = step58) : step68;
286 (loc = step59) & ((AuMoSt_aux | MMoSt_aux | SoftLDSt_aux) & E_MFoMoR & !(
AulhFoMo)) : step60;
287 (loc = step59) : step61;
288 (loc = step60) : step61;
289 (loc = step61) & ((AuMoSt_aux | FoMoSt_aux | SoftLDSt_aux) & E_MMMoR & !(AulhMMo
)) : step61;
290 (loc = step61) : step63;
291 (loc = step62) : step63;
292 (loc = step63) & ((MMoSt_aux & (E_MAuMoR | E_AuAuMoR)) | (FoMoSt_aux & E_MAuMoR
) | (SoftLDSt_aux & E_MAuMoR) | (MMoSt_aux & AulhMMo) | (FoMoSt_aux &
AulhFoMo) | (SoftLDSt_aux & AulhFoMo) | !(AuMoSt_aux | MMoSt_aux | FoMoSt_
aux | SoftLDSt_aux)) : step64;
293 (loc = step63) : step65;
294 (loc = step64) : step65;
295 (loc = step65) & ((AuMoSt_aux | MMoSt_aux) & E_MSoftLDR & !AulhFoMo) : step66;
296 (loc = step65) : step67;
297 (loc = step66) : step67;
298 (loc = step67) : step69;
299 (loc = step68) : step69;

```

```

300 (loc = step69) : step70;
301 (loc = step70) & (AuOffR) : step71;
302 (loc = step70) & (AuOnR) : step72;
303 (loc = step70) & (fullNotAcknowledged | FuStopISt | !EnRstartSt) : step73;
304 (loc = step70) : step74;
305 (loc = step71) : step74;
306 (loc = step72) : step74;
307 (loc = step73) : step74;
308 (loc = step74) : step75;
309 (loc = step75) & (((E_MOffR & (MMoSt | FoMoSt | SoftLDSt)) | (AuOffRSt & AuMoSt)
| (LDSt & PHLDcmd & HOffRSt) | (FE_PulseOn & PPulse & !POutOff) &
EnRstartSt) | (E_FuStopI & !PFsPosOn)) : step76;
310 (loc = step75) & (((E_MOnR & (MMoSt | FoMoSt | SoftLDSt)) | (AuOnRSt & AuMoSt)
| (LDSt & PHLDcmd & HOnRSt) & EnRstartSt) | (E_FuStopI & PFsPosOn)) : step
77;
311 (loc = step75) : step78;
312 (loc = step76) : step78;
313 (loc = step77) : step78;
314 (loc = step78) : step79;
315 (loc = step79) & (HOffR) : step80;
316 (loc = step79) & (HOnR) : step81;
317 (loc = step79) : step82;
318 (loc = step80) : step82;
319 (loc = step81) : step82;
320 (loc = step82) : step83;
321 (loc = step83) & (PPulse) : step84;
322 (loc = step83) : step158;
323 (loc = step84) & (InterlockR) : step85;
324 (loc = step84) & (FE_InterlockR) : step86;
325 (loc = step84) & ((MOffRSt & (MMoSt | FoMoSt | SoftLDSt)) | (AuOffRSt & AuMoSt)
| (HOffR & LDSt & PHLDcmd)) : step105;
326 (loc = step84) & ((MOnRSt & (MMoSt | FoMoSt | SoftLDSt)) | (AuOnRSt & AuMoSt) |
(HOnR & LDSt & PHLDcmd)) : step106;
327 (loc = step84) : step107;
328 (loc = step85) : step108;
329 (loc = step86) : step87;
330 (loc = step87) & ((FALSE & !Timer_PulseOn.old_in) & !Timer_PulseOn.Q) : step88;
331 (loc = step87) : step89;
332 (loc = step88) : step89;
333 (loc = step89) & ( _GLOBAL_TIME <= Timer_PulseOn.due ) : step90;
334 (loc = step89) : step91;
335 (loc = step90) : step95;
336 (loc = step91) : step92;
337 (loc = step92) & (FALSE) : step93;
338 (loc = step92) : step94;
339 (loc = step93) : step95;
340 (loc = step94) : step95;
341 (loc = step95) : step96;
342 (loc = step96) & ((FALSE & !Timer_PulseOff.old_in) & !Timer_PulseOff.Q) : step
97;
343 (loc = step96) : step98;
344 (loc = step97) : step98;
345 (loc = step98) & ( _GLOBAL_TIME <= Timer_PulseOff.due ) : step99;
346 (loc = step98) : step100;
347 (loc = step99) : step104;
348 (loc = step100) : step101;
349 (loc = step101) & (FALSE) : step102;
350 (loc = step101) : step103;
351 (loc = step102) : step104;
352 (loc = step103) : step104;
353 (loc = step104) : step105;
354 (loc = step105) : step108;
355 (loc = step106) : step108;
356 (loc = step107) : step108;
357 (loc = step108) & ((PulseOnR & !Timer_PulseOn.old_in) & !Timer_PulseOn.Q) : step
109;
358 (loc = step108) : step110;
359 (loc = step109) : step110;
360 (loc = step110) & ( _GLOBAL_TIME <= Timer_PulseOn.due ) : step111;
361 (loc = step110) : step112;
362 (loc = step111) : step116;
363 (loc = step112) : step113;
364 (loc = step113) & (PulseOnR) : step114;
365 (loc = step113) : step115;
366 (loc = step114) : step116;
367 (loc = step115) : step116;
368 (loc = step116) : step117;
369 (loc = step117) & ((PulseOffR & !Timer_PulseOff.old_in) & !Timer_PulseOff.Q) :
step118;
370 (loc = step117) : step119;
371 (loc = step118) : step119;
372 (loc = step119) & ( _GLOBAL_TIME <= Timer_PulseOff.due ) : step120;
373 (loc = step119) : step121;
374 (loc = step120) : step125;
375 (loc = step121) : step122;
376 (loc = step122) & (PulseOffR) : step123;
377 (loc = step122) : step124;
378 (loc = step123) : step125;
379 (loc = step124) : step125;
380 (loc = step125) : step126;

```

```

381      (loc = step126) & (PulseOn & !RE_PulseOn_old) : step127;
382      (loc = step126) : step128;
383      (loc = step127) : step129;
384      (loc = step128) : step129;
385      (loc = step129) & (!PulseOn & FE_PulseOn_old) : step130;
386      (loc = step129) : step131;
387      (loc = step130) : step132;
388      (loc = step131) : step132;
389      (loc = step132) & (PulseOff & !RE_PulseOff_old) : step133;
390      (loc = step132) : step134;
391      (loc = step133) : step135;
392      (loc = step134) : step135;
393      (loc = step135) & (RE_PulseOn) : step136;
394      (loc = step135) : step145;
395      (loc = step136) & ((FALSE & !Timer_PulseOff_old_in) & !Timer_PulseOff.Q) : step
137;
396      (loc = step136) : step138;
397      (loc = step137) : step138;
398      (loc = step138) & (_GLOBAL_TIME <= Timer_PulseOff.due) : step139;
399      (loc = step138) : step140;
400      (loc = step139) : step144;
401      (loc = step140) : step141;
402      (loc = step141) & (FALSE) : step142;
403      (loc = step141) : step143;
404      (loc = step142) : step144;
405      (loc = step143) : step144;
406      (loc = step144) : step145;
407      (loc = step145) & (RE_PulseOff) : step146;
408      (loc = step145) : step155;
409      (loc = step146) & ((FALSE & !Timer_PulseOn_old_in) & !Timer_PulseOn.Q) : step147;
410      (loc = step146) : step148;
411      (loc = step147) : step149;
412      (loc = step148) & (_GLOBAL_TIME <= Timer_PulseOn.due) : step149;
413      (loc = step148) : step150;
414      (loc = step149) : step154;
415      (loc = step150) : step151;
416      (loc = step151) & (FALSE) : step152;
417      (loc = step151) : step153;
418      (loc = step152) : step154;
419      (loc = step153) : step154;
420      (loc = step154) : step155;
421      (loc = step155) & (PPulseCste) : step156;
422      (loc = step155) : step157;
423      (loc = step156) : step158;
424      (loc = step157) : step158;
425      (loc = step158) : step159;
426      (loc = step159) & (POutOff) : step160;
427      (loc = step159) : step161;
428      (loc = step160) : step161;
429      (loc = step161) & (POutOff) : step162;
430      (loc = step161) : step168;
431      (loc = step162) & (InterlockR) : step163;
432      (loc = step162) : step170;
433      (loc = step163) & (PPulse & !PFsPosOn2) : step164;
434      (loc = step163) : step167;
435      (loc = step164) & (PFsPosOn) : step165;
436      (loc = step164) : step166;
437      (loc = step165) : step170;
438      (loc = step166) : step170;
439      (loc = step167) : step170;
440      (loc = step168) & (InterlockR) : step169;
441      (loc = step168) : step170;
442      (loc = step169) : step170;
443      (loc = step170) : step171;
444      (loc = step171) & (!POutOff) : step172;
445      (loc = step171) : step175;
446      (loc = step172) & (PFsPosOn) : step173;
447      (loc = step172) : step174;
448      (loc = step173) : step176;
449      (loc = step174) : step176;
450      (loc = step175) : step176;
451      (loc = step176) & (OutOnOVSt | (PPulse & PulseOnR)) : step177;
452      (loc = step176) : step178;
453      (loc = step177) : step178;
454      (loc = step178) & ((OutOffOVSt & POutOff) | (!OutOnOVSt & !POutOff) | (PPulse &
PulseOffR)) : step179;
455      (loc = step178) : step180;
456      (loc = step179) : step180;
457      (loc = step180) & (OutOVSt_aux & !RE_OutOVSt_aux_old) : step181;
458      (loc = step180) : step182;
459      (loc = step181) : step183;
460      (loc = step182) : step183;
461      (loc = step183) & (!OutOVSt_aux & FE_OutOVSt_aux_old) : step184;
462      (loc = step183) : step185;
463      (loc = step184) : step186;
464      (loc = step185) : step186;
465      (loc = step186) & (((OutOVSt_aux & ((PHFOn & !OnSt) | (PHFOff & OffSt))) | (!
OutOVSt_aux & ((PHFOff & !OffSt) | (PHFOn & OnSt))) | (OffSt & OnSt)) & (!
PPulse | (POutOff & PPulse & !OutOnOV & !OutOffOV))) : step187;
466      (loc = step186) : step188;

```

```

467         (loc = step187) : step188;
468         (loc = step188) & (!((OutOVSt_aux & ((PHFOn & !OnSt) | (PHFOff & OffSt))) | (!
OutOVSt_aux & ((PHFOff & !OffSt) | (PHFOn & OnSt))) | (OffSt & OnSt)) | RE-
OutOVSt_aux | FE_OutOVSt_aux | (PPulse & POutOff & OutOnOV) | (PPulse &
POutOff & OutOffOV)) : step189;
469         (loc = step188) : step190;
470         (loc = step189) : step190;
471         (loc = step190) & (!PosW_aux) : step191;
472         (loc = step190) : step192;
473         (loc = step191) : step198;
474         (loc = step192) & (!Timer_Warning.running) : step193;
475         (loc = step192) : step194;
476         (loc = step193) : step198;
477         (loc = step194) & (!(_GLOBAL_TIME - (Timer_Warning.Tstart + POnOffb.PWDtb) >= 0
sd32_0)) : step195;
478         (loc = step194) : step197;
479         (loc = step195) & (!Timer_Warning.Q) : step196;
480         (loc = step195) : step198;
481         (loc = step196) : step198;
482         (loc = step197) : step198;
483         (loc = step198) : step199;
484         (loc = step199) & (FuStopISt | FSIinc > 0sd16_0) : step200;
485         (loc = step199) : step201;
486         (loc = step200) : step201;
487         (loc = step201) & (extend(FSIinc,16) > PulseWidth | (!FuStopISt & FSIinc = 0sd
16_0)) : step202;
488         (loc = step201) : step203;
489         (loc = step202) : step203;
490         (loc = step203) & (TStopISt | TSIinc > 0sd16_0) : step204;
491         (loc = step203) : step205;
492         (loc = step204) : step205;
493         (loc = step205) & (extend(TSIinc,16) > PulseWidth | (!TStopISt & TSIinc = 0sd
16_0)) : step206;
494         (loc = step205) : step207;
495         (loc = step206) : step207;
496         (loc = step207) & (StartISt | SIinc > 0sd16_0) : step208;
497         (loc = step207) : step209;
498         (loc = step208) : step209;
499         (loc = step209) & (extend(SIinc,16) > PulseWidth | (!StartISt & SIinc = 0sd16_0)
) : step210;
500         (loc = step209) : step211;
501         (loc = step210) : step211;
502         (loc = step211) & (AISt | AInc > 0sd16_0) : step212;
503         (loc = step211) : step213;
504         (loc = step212) : step213;
505         (loc = step213) & (extend(AInc,16) > PulseWidth | (!AISt & AInc = 0sd16_0)) :
step214;
506         (loc = step213) : step215;
507         (loc = step214) : step215;
508         (loc = step215) : step216;
509         (loc = step216) & (AIUnAck != AIUnAck_old) : step217;
510         (loc = step216) : step221;
511         (loc = step217) & (AIUnAck) : step218;
512         (loc = step217) : step219;
513         (loc = step218) : step220;
514         (loc = step219) : step220;
515         (loc = step220) : end;
516         (loc = step221) : end;
517         (loc = end) : nvar;
518         (loc = nvar) : start;
519     esac;
520
521     --Variable initialization
522     init(Stsreg01b[0]) := FALSE;
523     init(Stsreg01b[1]) := FALSE;
524     init(Stsreg01b[2]) := FALSE;
525     init(Stsreg01b[3]) := FALSE;
526     init(Stsreg01b[4]) := FALSE;
527     init(Stsreg01b[5]) := FALSE;
528     init(Stsreg01b[6]) := FALSE;
529     init(Stsreg01b[7]) := FALSE;
530     init(Stsreg01b[8]) := FALSE;
531     init(Stsreg01b[9]) := FALSE;
532     init(Stsreg01b[10]) := FALSE;
533     init(Stsreg01b[11]) := FALSE;
534     init(Stsreg01b[12]) := FALSE;
535     init(Stsreg01b[13]) := FALSE;
536     init(Stsreg01b[14]) := FALSE;
537     init(Stsreg01b[15]) := FALSE;
538     init(Stsreg02b[0]) := FALSE;
539     init(Stsreg02b[1]) := FALSE;
540     init(Stsreg02b[2]) := FALSE;
541     init(Stsreg02b[3]) := FALSE;
542     init(Stsreg02b[4]) := FALSE;
543     init(Stsreg02b[5]) := FALSE;
544     init(Stsreg02b[6]) := FALSE;
545     init(Stsreg02b[7]) := FALSE;
546     init(Stsreg02b[8]) := FALSE;
547     init(Stsreg02b[9]) := FALSE;
548     init(Stsreg02b[10]) := FALSE;

```



```

549   init (Stsreg02b[11]) := FALSE;
550   init (Stsreg02b[12]) := FALSE;
551   init (Stsreg02b[13]) := FALSE;
552   init (Stsreg02b[14]) := FALSE;
553   init (Stsreg02b[15]) := FALSE;
554
555   init (OutOnOV) := FALSE;
556   init (OutOffOV) := FALSE;
557   init (OnSt) := FALSE;
558   init (OffSt) := FALSE;
559   init (AuMoSt) := FALSE;
560   init (MMoSt) := FALSE;
561   init (LDSt) := FALSE;
562   init (SoftLDSt) := FALSE;
563   init (FoMoSt) := FALSE;
564   init (AuOnRSt) := FALSE;
565   init (AuOffRSt) := FALSE;
566   init (MOnRSt) := FALSE;
567   init (MOffRSt) := FALSE;
568   init (HOnRSt) := FALSE;
569   init (HOffRSt) := FALSE;
570   init (IOErrorW) := FALSE;
571   init (IOSimuW) := FALSE;
572   init (AuMRW) := FALSE;
573   init (AlUnAck) := FALSE;
574   init (PosW) := FALSE;
575   init (StartISt) := FALSE;
576   init (TStopISt) := FALSE;
577   init (FuStopISt) := FALSE;
578   init (AlSt) := FALSE;
579   init (AlBW) := FALSE;
580   init (EnRstartSt) := TRUE;
581   init (RdyStartSt) := FALSE;
582
583   --Internal Variables
584   init (E_MAuMoR) := FALSE;
585   init (E_MMMoR) := FALSE;
586   init (E_MFoMoR) := FALSE;
587   init (E_MOnR) := FALSE;
588   init (E_MOffR) := FALSE;
589   init (E_MAlAckR) := FALSE;
590   init (E_StartI) := FALSE;
591   init (E_TStopI) := FALSE;
592   init (E_FuStopI) := FALSE;
593   init (E_Al) := FALSE;
594   init (E_AuAuMoR) := FALSE;
595   init (E_AuAlAck) := FALSE;
596   init (E_MSoftLDR) := FALSE;
597   init (E_MEnRstartR) := FALSE;
598   init (RE_AlUnAck) := FALSE;
599   init (FE_AlUnAck) := FALSE;
600   init (RE_PulseOn) := FALSE;
601   init (FE_PulseOn) := FALSE;
602   init (RE_PulseOff) := FALSE;
603   init (RE_OutOVSt_aux) := FALSE;
604   init (FE_OutOVSt_aux) := FALSE;
605   init (FE_InterlockR) := FALSE;
606
607   init (MAuMoR_old) := FALSE;
608   init (MMMoR_old) := FALSE;
609   init (MFoMoR_old) := FALSE;
610   init (MOnR_old) := FALSE;
611   init (MOffR_old) := FALSE;
612   init (MAlAckR_old) := FALSE;
613   init (AuAuMoR_old) := FALSE;
614   init (AuAlAck_old) := FALSE;
615   init (StartI_old) := FALSE;
616   init (TStopI_old) := FALSE;
617   init (FuStopI_old) := FALSE;
618   init (Al_old) := FALSE;
619   init (AlUnAck_old) := FALSE;
620   init (MSoftLDR_old) := FALSE;
621   init (MEnRstartR_old) := FALSE;
622   init (RE_PulseOn_old) := FALSE;
623   init (FE_PulseOn_old) := FALSE;
624   init (RE_PulseOff_old) := FALSE;
625   init (RE_OutOVSt_aux_old) := FALSE;
626   init (FE_OutOVSt_aux_old) := FALSE;
627   init (FE_InterlockR_old) := FALSE;
628
629   init (PFsPosOn) := FALSE;
630   init (PFsPosOn2) := FALSE;
631   init (PHFOn) := FALSE;
632   init (PHFOff) := FALSE;
633   init (PPulse) := FALSE;
634   init (PPulseCste) := FALSE;
635   init (PHLD) := FALSE;
636   init (PHLDCmd) := FALSE;
637   init (PAnim) := FALSE;
638   init (POutOff) := FALSE;

```

```

639   init (PEnrstart) := FALSE;
640   init (PRstartFS) := FALSE;
641   init (OutOnOVSt) := FALSE;
642   init (OutOffOVSt) := 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 (Time_Warning) := 0sd32_0;
658   init (Timer_PulseOn.Q) := FALSE;
659   init (Timer_PulseOn.ET) := 0sd32_0;
660   init (Timer_PulseOn.old_in) := FALSE;
661   init (Timer_PulseOn.due) := 0sd32_0;
662   init (Timer_PulseOff.Q) := FALSE;
663   init (Timer_PulseOff.ET) := 0sd32_0;
664   init (Timer_PulseOff.old_in) := FALSE;
665   init (Timer_PulseOff.due) := 0sd32_0;
666   init (Timer_Warning.Q) := FALSE;
667   init (Timer_Warning.ET) := 0sd32_0;
668   init (Timer_Warning.running) := FALSE;
669   init (Timer_Warning.Tstart) := 0sd32_0;
670
671   --Variables for interlock Ststus delay handling
672   init (PulseWidth) := 0sd32_0;
673   init (FSIinc) := 0sd16_0;
674   init (TSIinc) := 0sd16_0;
675   init (SIinc) := 0sd16_0;
676   init (Alinc) := 0sd16_0;
677   init (WTstopIst) := FALSE;
678   init (WStartIst) := FALSE;
679   init (WAIst) := FALSE;
680   init (WFuStopIst) := FALSE;
681   init (_GLOBAL_TIME) := 0sd32_0;
682   init (T_CYCLE) := 0ud16_0;
683   init (random_t_cycle) := 0ud8_0;
684
685   --Non-deterministic input
686   next (HFOn) :=
687     case
688       (loc = start) : {TRUE,FALSE};
689       TRUE          : HFOn ;
690     esac;
691   next (HFOff) :=
692     case
693       (loc = start) : {TRUE,FALSE};
694       TRUE          : HFOff ;
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};

```

```

729         TRUE : Manreg01b[2];
730     esac;
731 next (Manreg01b[3]) :=
732     case
733         (loc = start) : {TRUE,FALSE};
734         TRUE : Manreg01b[3];
735     esac;
736 next (Manreg01b[4]) :=
737     case
738         (loc = start) : {TRUE,FALSE};
739         TRUE : Manreg01b[4];
740     esac;
741 next (Manreg01b[5]) :=
742     case
743         (loc = start) : {TRUE,FALSE};
744         TRUE : Manreg01b[5];
745     esac;
746 next (Manreg01b[6]) :=
747     case
748         (loc = start) : {TRUE,FALSE};
749         TRUE : Manreg01b[6];
750     esac;
751 next (Manreg01b[7]) :=
752     case
753         (loc = start) : {TRUE,FALSE};
754         TRUE : Manreg01b[7];
755     esac;
756 next (Manreg01b[8]) :=
757     case
758         (loc = start) : {TRUE,FALSE};
759         TRUE : Manreg01b[8];
760     esac;
761 next (Manreg01b[9]) :=
762     case
763         (loc = start) : {TRUE,FALSE};
764         TRUE : Manreg01b[0];
765     esac;
766 next (Manreg01b[10]) :=
767     case
768         (loc = start) : {TRUE,FALSE};
769         TRUE : Manreg01b[10];
770     esac;
771 next (Manreg01b[11]) :=
772     case
773         (loc = start) : {TRUE,FALSE};
774         TRUE : Manreg01b[11];
775     esac;
776 next (Manreg01b[12]) :=
777     case
778         (loc = start) : {TRUE,FALSE};
779         TRUE : Manreg01b[12];
780     esac;
781 next (Manreg01b[13]) :=
782     case
783         (loc = start) : {TRUE,FALSE};
784         TRUE : Manreg01b[13];
785     esac;
786 next (Manreg01b[14]) :=
787     case
788         (loc = start) : {TRUE,FALSE};
789         TRUE : Manreg01b[14];
790     esac;
791 next (Manreg01b[15]) :=
792     case
793         (loc = start) : {TRUE,FALSE};
794         TRUE : Manreg01b[15];
795     esac;
796 next (HOnR) :=
797     case
798         (loc = start) : {TRUE,FALSE};
799         TRUE : HOnR;
800     esac;
801 next (HOffR) :=
802     case
803         (loc = start) : {TRUE,FALSE};
804         TRUE : HOffR;
805     esac;
806 next (StartI) :=
807     case
808         (loc = start) : {TRUE,FALSE};
809         TRUE : StartI;
810     esac;
811 next (TStopI) :=
812     case
813         (loc = start) : {TRUE,FALSE};
814         TRUE : TStopI;
815     esac;
816 next (FuStopI) :=
817     case
818         (loc = start) : {TRUE,FALSE};

```

```

819         TRUE : FuStopI;
820     esac;
821 next (A1) :=
822     case
823         (loc = start) : {TRUE,FALSE};
824         TRUE : A1;
825     esac;
826 next (AuOnR) :=
827     case
828         (loc = start) : {TRUE,FALSE};
829         TRUE : AuOnR;
830     esac;
831 next (AuOffR) :=
832     case
833         (loc = start) : {TRUE,FALSE};
834         TRUE : AuOffR;
835     esac;
836 next (AuAuMoR) :=
837     case
838         (loc = start) : {TRUE,FALSE};
839         TRUE : AuAuMoR;
840     esac;
841 next (AulhMMo) :=
842     case
843         (loc = start) : {TRUE,FALSE};
844         TRUE : AulhMMo;
845     esac;
846 next (AulhFoMo) :=
847     case
848         (loc = start) : {TRUE,FALSE};
849         TRUE : AulhFoMo;
850     esac;
851 next (AuAlAck) :=
852     case
853         (loc = start) : {TRUE,FALSE};
854         TRUE : AuAlAck;
855     esac;
856 next (IhAuMRW) :=
857     case
858         (loc = start) : {TRUE,FALSE};
859         TRUE : IhAuMRW;
860     esac;
861 next (AuRstart) :=
862     case
863         (loc = start) : {TRUE,FALSE};
864         TRUE : AuRstart;
865     esac;
866
867 --Values that are non-deterministic, but do not got a new value every iteration
868 next (POnOffb.ParRegb[0]) :=
869     case
870         TRUE : POnOffb.ParRegb[0];
871     esac;
872 next (POnOffb.ParRegb[1]) :=
873     case
874         TRUE : POnOffb.ParRegb[1];
875     esac;
876 next (POnOffb.ParRegb[2]) :=
877     case
878         TRUE : POnOffb.ParRegb[2];
879     esac;
880 next (POnOffb.ParRegb[3]) :=
881     case
882         TRUE : POnOffb.ParRegb[3];
883     esac;
884 next (POnOffb.ParRegb[4]) :=
885     case
886         TRUE : POnOffb.ParRegb[4];
887     esac;
888 next (POnOffb.ParRegb[5]) :=
889     case
890         TRUE : POnOffb.ParRegb[5];
891     esac;
892 next (POnOffb.ParRegb[6]) :=
893     case
894         TRUE : POnOffb.ParRegb[6];
895     esac;
896 next (POnOffb.ParRegb[7]) :=
897     case
898         TRUE : POnOffb.ParRegb[7];
899     esac;
900 next (POnOffb.ParRegb[8]) :=
901     case
902         TRUE : POnOffb.ParRegb[8];
903     esac;
904 next (POnOffb.ParRegb[9]) :=
905     case
906         TRUE : POnOffb.ParRegb[9];
907     esac;
908 next (POnOffb.ParRegb[10]) :=

```

```

909     case
910         TRUE : POnOffb.ParRegb[10];
911     esac;
912 next (POnOffb.ParRegb[11]) :=
913     case
914         TRUE : POnOffb.ParRegb[11];
915     esac;
916 next (POnOffb.ParRegb[12]) :=
917     case
918         TRUE : POnOffb.ParRegb[12];
919     esac;
920 next (POnOffb.ParRegb[13]) :=
921     case
922         TRUE : POnOffb.ParRegb[13];
923     esac;
924 next (POnOffb.ParRegb[14]) :=
925     case
926         TRUE : POnOffb.ParRegb[14];
927     esac;
928 next (POnOffb.ParRegb[15]) :=
929     case
930         TRUE : POnOffb.ParRegb[15];
931     esac;
932 next (POnOffb.PPulseLeb) :=
933     case
934         TRUE : POnOffb.PPulseLeb;
935     esac;
936 next (POnOffb.PWDtb) :=
937     case
938         TRUE : POnOffb.PWDtb;
939     esac;
940
941 --Program body
942 next (E_MAuMoR) :=
943     case
944         (loc = step2) : TRUE;
945         (loc = step3) : FALSE;
946         TRUE : E_MAuMoR;
947     esac;
948 next (MAuMoR_old) :=
949     case
950         (loc = step2) : TRUE;
951         (loc = step3) : Manreg01b[8];
952         TRUE : MAuMoR_old;
953     esac;
954 next (E_MMMoR) :=
955     case
956         (loc = step5) : TRUE;
957         (loc = step6) : FALSE;
958         TRUE : E_MMMoR;
959     esac;
960 next (MMMoR_old) :=
961     case
962         (loc = step5) : TRUE;
963         (loc = step6) : Manreg01b[9];
964         TRUE : MMMoR_old;
965     esac;
966 next (E_MFoMoR) :=
967     case
968         (loc = step8) : TRUE;
969         (loc = step9) : FALSE;
970         TRUE : E_MFoMoR;
971     esac;
972 next (MFoMoR_old) :=
973     case
974         (loc = step8) : TRUE;
975         (loc = step9) : Manreg01b[10];
976         TRUE : MFoMoR_old;
977     esac;
978 next (E_MSoftLDR) :=
979     case
980         (loc = step11) : TRUE;
981         (loc = step12) : FALSE;
982         TRUE : E_MSoftLDR;
983     esac;
984 next (MSoftLDR_old) :=
985     case
986         (loc = step11) : TRUE;
987         (loc = step12) : Manreg01b[11];
988         TRUE : MSoftLDR_old;
989     esac;
990 next (E_MOnR) :=
991     case
992         (loc = step14) : TRUE;
993         (loc = step15) : FALSE;
994         TRUE : E_MOnR;
995     esac;
996 next (MOnR_old) :=
997     case
998         (loc = step14) : TRUE;

```

```

999         (loc = step15) : Manreg01b[12];
1000         TRUE : MOnR_old;
1001     esac;
1002 next(E_MOffR) :=
1003     case
1004         (loc = step17) : TRUE;
1005         (loc = step18) : FALSE;
1006         TRUE : E_MOffR;
1007     esac;
1008 next(MOffR_old) :=
1009     case
1010         (loc = step17) : TRUE;
1011         (loc = step18) : Manreg01b[13];
1012         TRUE : MOffR_old;
1013     esac;
1014 next(E_MEnRstartR) :=
1015     case
1016         (loc = step20) : TRUE;
1017         (loc = step21) : FALSE;
1018         TRUE : E_MEnRstartR;
1019     esac;
1020 next(MEnRstartR_old) :=
1021     case
1022         (loc = step20) : TRUE;
1023         (loc = step21) : Manreg01b[1];
1024         TRUE : MEnRstartR_old;
1025     esac;
1026 next(E_MAIAckR) :=
1027     case
1028         (loc = step23) : TRUE;
1029         (loc = step24) : FALSE;
1030         TRUE : E_MAIAckR;
1031     esac;
1032 next(MAIAckR_old) :=
1033     case
1034         (loc = step23) : TRUE;
1035         (loc = step24) : Manreg01b[7];
1036         TRUE : MAIAckR_old;
1037     esac;
1038 next(PFsPosOn) :=
1039     case
1040         (loc = step25) : POnOffb.ParRegb[8];
1041         TRUE : PFsPosOn;
1042     esac;
1043 next(PHFOn) :=
1044     case
1045         (loc = step25) : POnOffb.ParRegb[9];
1046         TRUE : PHFOn;
1047     esac;
1048 next(PHFOff) :=
1049     case
1050         (loc = step25) : POnOffb.ParRegb[10];
1051         TRUE : PHFOff;
1052     esac;
1053 next(PPulse) :=
1054     case
1055         (loc = step25) : FALSE;
1056         TRUE : PPulse;
1057     esac;
1058 next(PHLD) :=
1059     case
1060         (loc = step25) : POnOffb.ParRegb[12];
1061         TRUE : PHLD;
1062     esac;
1063 next(PHLDcmd) :=
1064     case
1065         (loc = step25) : POnOffb.ParRegb[13];
1066         TRUE : PHLDcmd;
1067     esac;
1068 next(PAnim) :=
1069     case
1070         (loc = step25) : POnOffb.ParRegb[14];
1071         TRUE : PAnim;
1072     esac;
1073 next(POutOff) :=
1074     case
1075         (loc = step25) : POnOffb.ParRegb[15];
1076         TRUE : POutOff;
1077     esac;
1078 next(PEnRstart) :=
1079     case
1080         (loc = step25) : POnOffb.ParRegb[0];
1081         TRUE : PEnRstart;
1082     esac;
1083 next(PRstartFS) :=
1084     case
1085         (loc = step25) : POnOffb.ParRegb[1];
1086         TRUE : PRstartFS;
1087     esac;
1088 next(PFsPosOn2) :=

```

```

1089     case
1090         (loc = step25) : POnOffb.ParRegb[2];
1091         TRUE : PFsPosOn2;
1092     esac;
1093 next(PPulseCste) :=
1094     case
1095         (loc = step25) : POnOffb.ParRegb[3];
1096         TRUE : PPulseCste;
1097     esac;
1098 next(E_AuAuMoR) :=
1099     case
1100         (loc = step27) : TRUE;
1101         (loc = step28) : FALSE;
1102         TRUE : E_AuAuMoR;
1103     esac;
1104 next(AuAuMoR_old) :=
1105     case
1106         (loc = step27) : TRUE;
1107         (loc = step28) : AuAuMoR;
1108         TRUE : AuAuMoR_old;
1109     esac;
1110 next(E_AuAlAck) :=
1111     case
1112         (loc = step30) : TRUE;
1113         (loc = step31) : FALSE;
1114         TRUE : E_AuAlAck;
1115     esac;
1116 next(AuAlAck_old) :=
1117     case
1118         (loc = step30) : TRUE;
1119         (loc = step31) : AuAlAck;
1120         TRUE : AuAlAck_old;
1121     esac;
1122 next(E_StartI) :=
1123     case
1124         (loc = step33) : TRUE;
1125         (loc = step34) : FALSE;
1126         TRUE : E_StartI;
1127     esac;
1128 next(StartI_old) :=
1129     case
1130         (loc = step33) : TRUE;
1131         (loc = step34) : StartI;
1132         TRUE : StartI_old;
1133     esac;
1134 next(E_TStopI) :=
1135     case
1136         (loc = step36) : TRUE;
1137         (loc = step37) : FALSE;
1138         TRUE : E_TStopI;
1139     esac;
1140 next(TStopI_old) :=
1141     case
1142         (loc = step36) : TRUE;
1143         (loc = step37) : TStopI;
1144         TRUE : TStopI_old;
1145     esac;
1146 next(E_FuStopI) :=
1147     case
1148         (loc = step39) : TRUE;
1149         (loc = step40) : FALSE;
1150         TRUE : E_FuStopI;
1151     esac;
1152 next(FuStopI_old) :=
1153     case
1154         (loc = step39) : TRUE;
1155         (loc = step40) : FuStopI;
1156         TRUE : FuStopI_old;
1157     esac;
1158 next(E_Al) :=
1159     case
1160         (loc = step42) : TRUE;
1161         (loc = step43) : FALSE;
1162         TRUE : E_Al;
1163     esac;
1164 next(Al_old) :=
1165     case
1166         (loc = step42) : TRUE;
1167         (loc = step43) : Al;
1168         TRUE : Al_old;
1169     esac;
1170 next(StartISt) :=
1171     case
1172         (loc = step44) : StartI;
1173         TRUE : StartISt;
1174     esac;
1175 next(TStopISt) :=
1176     case
1177         (loc = step44) : TStopI;
1178         TRUE : TStopISt;

```

```

1179     esac;
1180 next(FuStopISt) :=
1181     case
1182         (loc = step44) : FuStopI;
1183         TRUE : FuStopISt;
1184     esac;
1185 next(fullNotAcknowledged) :=
1186     case
1187         (loc = step46) : FALSE;
1188         (loc = step51) : TRUE;
1189         TRUE : fullNotAcknowledged;
1190     esac;
1191 next(AIUnAck) :=
1192     case
1193         (loc = step46) : FALSE;
1194         (loc = step47) : TRUE;
1195         TRUE : AIUnAck;
1196     esac;
1197 next(EnRstartSt) :=
1198     case
1199         (loc = step49) : TRUE;
1200         (loc = step53) : FALSE;
1201         TRUE : EnRstartSt;
1202     esac;
1203 next(InterlockR) :=
1204     case
1205         (loc = step54) : (TStopISt | FuStopISt | fullNotAcknowledged | !EnRstartSt | (
1206             StartISt & !POutOff & !OutOnOV) | (StartISt & POutOff & ((PFsPosOn & OutOVSt
1207             _aux) | (!PFsPosOn & !OutOVSt_aux))));
1208         TRUE : InterlockR;
1209     esac;
1210 next(FE_InterlockR) :=
1211     case
1212         (loc = step56) : TRUE;
1213         (loc = step57) : FALSE;
1214         TRUE : FE_InterlockR;
1215     esac;
1216 next(FE_InterlockR_old) :=
1217     case
1218         (loc = step56) : FALSE;
1219         (loc = step57) : InterlockR;
1220         TRUE : FE_InterlockR_old;
1221     esac;
1222 next(AuMoSt_aux) :=
1223     case
1224         (loc = step60) : FALSE;
1225         (loc = step62) : FALSE;
1226         (loc = step64) : TRUE;
1227         (loc = step66) : FALSE;
1228         TRUE : AuMoSt_aux;
1229     esac;
1230 next(MMoSt_aux) :=
1231     case
1232         (loc = step60) : FALSE;
1233         (loc = step62) : TRUE;
1234         (loc = step64) : FALSE;
1235         (loc = step66) : FALSE;
1236         TRUE : MMoSt_aux;
1237     esac;
1238 next(FoMoSt_aux) :=
1239     case
1240         (loc = step60) : TRUE;
1241         (loc = step62) : FALSE;
1242         (loc = step64) : FALSE;
1243         (loc = step66) : FALSE;
1244         TRUE : FoMoSt_aux;
1245     esac;
1246 next(SoftLDSt_aux) :=
1247     case
1248         (loc = step60) : FALSE;
1249         (loc = step62) : FALSE;
1250         (loc = step64) : FALSE;
1251         (loc = step66) : TRUE;
1252         TRUE : SoftLDSt_aux;
1253     esac;
1254 next(LDSt) :=
1255     case
1256         (loc = step67) : FALSE;
1257         (loc = step68) : TRUE;
1258         TRUE : LDSt;
1259     esac;
1260 next(AuMoSt) :=
1261     case
1262         (loc = step67) : AuMoSt_aux;
1263         (loc = step68) : FALSE;
1264         TRUE : AuMoSt;
1265     esac;
1266 next(MMoSt) :=
1267     case
1268         (loc = step67) : 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) : SoftLDSt_aux;
1279         (loc = step 68) : FALSE;
1280         TRUE : SoftLDSt;
1281     esac;
1282 next (OnSt) :=
1283     case
1284         (loc = step 69) : (HFOOn & PHFOOn) | (!PHFOOn & PHFOff & PAnim & !HFOff) | (!PHFOOn &
1285             !PHFOff & OutOVSt_aux);
1286         TRUE : OnSt;
1287     esac;
1288 next (OffSt) :=
1289     case
1290         (loc = step 69) : (HFOff & PHFOff) | (!PHFOff & PHFOOn & PAnim & !HFOOn) | (!PHFOOn
1291             & !PHFOff & !OutOVSt_aux);
1292         TRUE : OffSt;
1293     esac;
1294 next (AuOnRSt) :=
1295     case
1296         (loc = step 71) : FALSE;
1297         (loc = step 72) : TRUE;
1298         (loc = step 73) : PFsPosOn;
1299         TRUE : AuOnRSt;
1300     esac;
1301 next (AuOffRSt) :=
1302     case
1303         (loc = step 74) : !AuOnRSt;
1304         TRUE : AuOffRSt;
1305     esac;
1306 next (MOnRSt) :=
1307     case
1308         (loc = step 76) : FALSE;
1309         (loc = step 77) : TRUE;
1310         TRUE : MOnRSt;
1311     esac;
1312 next (HOnRSt) :=
1313     case
1314         (loc = step 80) : FALSE;
1315         (loc = step 81) : TRUE;
1316         TRUE : HOnRSt;
1317     esac;
1318 next (HOffRSt) :=
1319     case
1320         (loc = step 82) : !HOnRSt;
1321         TRUE : HOffRSt;
1322     esac;
1323 next (PulseOnR) :=
1324     case
1325         (loc = step 85) : (PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
1326         (loc = step 86) : FALSE;
1327         (loc = step 105) : FALSE;
1328         (loc = step 106) : TRUE;
1329         (loc = step 107) : FALSE;
1330         TRUE : PulseOnR;
1331     esac;
1332 next (PulseOffR) :=
1333     case
1334         (loc = step 85) : (!PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
1335         (loc = step 86) : FALSE;
1336         (loc = step 105) : TRUE;
1337         (loc = step 106) : FALSE;
1338         (loc = step 107) : FALSE;
1339         TRUE : PulseOffR;
1340     esac;
1341 next (Timer_PulseOn.due) :=
1342     case
1343         (loc = step 88) : --GLOBAL_TIME + 0sd 32_0;
1344         (loc = step 109) : --GLOBAL_TIME + POnOffb.PPulseLeb;
1345         (loc = step 147) : --GLOBAL_TIME + 0sd 32_0;
1346         TRUE : Timer_PulseOn.due;
1347     esac;
1348 next (Timer_PulseOn.Q) :=
1349     case
1350         (loc = step 90) : TRUE;
1351         (loc = step 91) : FALSE;
1352         (loc = step 110) : TRUE;
1353         (loc = step 111) : FALSE;
1354         (loc = step 149) : TRUE;
1355         (loc = step 150) : FALSE;
1356         TRUE : Timer_PulseOn.Q;

```

```

1355     esac;
1356 next(Timer_PulseOn.ET) :=
1357     case
1358         (loc = step90) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
1359         (loc = step93) : 0sd32_0;
1360         (loc = step94) : 0sd32_0;
1361         (loc = step111) : (POnOffb.PPulseLeb - (Timer_PulseOn.due - __GLOBAL_TIME));
1362         (loc = step114) : POnOffb.PPulseLeb;
1363         (loc = step115) : 0sd32_0;
1364         (loc = step149) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
1365         (loc = step152) : 0sd32_0;
1366         (loc = step153) : 0sd32_0;
1367         TRUE : Timer_PulseOn.ET;
1368     esac;
1369 next(Timer_PulseOn.old_in) :=
1370     case
1371         (loc = step95) : FALSE;
1372         (loc = step116) : PulseOnR;
1373         (loc = step154) : FALSE;
1374         TRUE : Timer_PulseOn.old_in;
1375     esac;
1376 next(Timer_PulseOff.due) :=
1377     case
1378         (loc = step97) : __GLOBAL_TIME + 0sd32_0;
1379         (loc = step118) : __GLOBAL_TIME + POnOffb.PPulseLeb;
1380         (loc = step137) : __GLOBAL_TIME + 0sd32_0;
1381         TRUE : Timer_PulseOff.due;
1382     esac;
1383 next(Timer_PulseOff.Q) :=
1384     case
1385         (loc = step99) : TRUE;
1386         (loc = step100) : FALSE;
1387         (loc = step120) : TRUE;
1388         (loc = step121) : FALSE;
1389         (loc = step139) : TRUE;
1390         (loc = step140) : FALSE;
1391         TRUE : Timer_PulseOff.Q;
1392     esac;
1393 next(Timer_PulseOff.ET) :=
1394     case
1395         (loc = step99) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
1396         (loc = step102) : 0sd32_0;
1397         (loc = step103) : 0sd32_0;
1398         (loc = step120) : (POnOffb.PPulseLeb - (Timer_PulseOn.due - __GLOBAL_TIME));
1399         (loc = step123) : POnOffb.PPulseLeb;
1400         (loc = step124) : 0sd32_0;
1401         (loc = step139) : (0sd32_0 - (Timer_PulseOn.due - __GLOBAL_TIME));
1402         (loc = step142) : 0sd32_0;
1403         (loc = step143) : 0sd32_0;
1404         TRUE : Timer_PulseOff.ET;
1405     esac;
1406 next(Timer_PulseOff.old_in) :=
1407     case
1408         (loc = step104) : FALSE;
1409         (loc = step125) : PulseOffR;
1410         (loc = step144) : FALSE;
1411         TRUE : Timer_PulseOff.old_in;
1412     esac;
1413 next(RE_PulseOn) :=
1414     case
1415         (loc = step127) : TRUE;
1416         (loc = step128) : FALSE;
1417         TRUE : RE_PulseOn;
1418     esac;
1419 next(RE_PulseOn_old) :=
1420     case
1421         (loc = step127) : TRUE;
1422         (loc = step128) : PulseOn;
1423         TRUE : RE_PulseOn_old;
1424     esac;
1425 next(FE_PulseOn) :=
1426     case
1427         (loc = step130) : TRUE;
1428         (loc = step131) : FALSE;
1429         TRUE : FE_PulseOn;
1430     esac;
1431 next(FE_PulseOn_old) :=
1432     case
1433         (loc = step130) : FALSE;
1434         (loc = step131) : PulseOn;
1435         TRUE : FE_PulseOn_old;
1436     esac;
1437 next(RE_PulseOff) :=
1438     case
1439         (loc = step133) : TRUE;
1440         (loc = step134) : FALSE;
1441         TRUE : RE_PulseOff;
1442     esac;
1443 next(RE_PulseOff_old) :=
1444     case

```

```

1445         (loc = step 133) : TRUE;
1446         (loc = step 134) : PulseOff;
1447         TRUE : RE_PulseOff_old;
1448     esac;
1449 next(PulseOn) :=
1450     case
1451         (loc = step 156) : Timer_PulseOn.Q & !PulseOffR;
1452         (loc = step 157) : Timer_PulseOn.Q & !PulseOffR & (!PHFOn | (PHFOn & !HFOn));
1453         TRUE : PulseOn;
1454     esac;
1455 next(PulseOff) :=
1456     case
1457         (loc = step 156) : Timer_PulseOff.Q & !PulseOnR;
1458         (loc = step 157) : Timer_PulseOff.Q & !PulseOnR & (!PHFOff | (PHFOff & !HFOff));
1459         TRUE : PulseOff;
1460     esac;
1461 next(OutOnOVSt) :=
1462     case
1463         (loc = step 158) : (PPulse & PulseOn) | (!PPulse & ((MOnRSt & (MMoSt | FoMoSt |
1464             SoftLDSt)) | (AuOnRSt & AuMoSt) | (HOnRSt & LDSt & PHLDCmd)));
1465         (loc = step 165) : PulseOn;
1466         (loc = step 166) : FALSE;
1467         (loc = step 167) : (PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
1468         (loc = step 169) : PFsPosOn;
1469         TRUE : OutOnOVSt;
1470     esac;
1471 next(OutOffOVSt) :=
1472     case
1473         (loc = step 160) : (PulseOff & PPulse) | (!PPulse & ((MOffRSt & (MMoSt | FoMoSt
1474             | SoftLDSt)) | (AuOffRSt & AuMoSt) | (HOffRSt & LDSt & PHLDCmd)));
1475         (loc = step 165) : FALSE;
1476         (loc = step 166) : PulseOff;
1477         (loc = step 167) : (!PFsPosOn & !PFsPosOn2) | (PFsPosOn & PFsPosOn2);
1478         TRUE : OutOffOVSt;
1479     esac;
1480 next(RdyStartSt) :=
1481     case
1482         (loc = step 170) : !InterlockR;
1483         TRUE : RdyStartSt;
1484     esac;
1485 next(A1St) :=
1486     case
1487         (loc = step 170) : A1;
1488         TRUE : A1St;
1489     esac;
1490 next(IOErrorW) :=
1491     case
1492         (loc = step 170) : IOError;
1493         TRUE : IOErrorW;
1494     esac;
1495 next(IOSimuW) :=
1496     case
1497         (loc = step 170) : IOSimu;
1498         TRUE : IOSimuW;
1499     esac;
1500 next(AuMRW) :=
1501     case
1502         (loc = step 170) : (MMoSt | FoMoSt | SoftLDSt) & ((AuOnRSt xor MOnRSt) | (
1503             AuOffRSt xor MOffRSt)) & !IhAuMRW;
1504         TRUE : AuMRW;
1505     esac;
1506 next(OutOnOV) :=
1507     case
1508         (loc = step 173) : !OutOnOVSt;
1509         (loc = step 174) : OutOnOVSt;
1510         (loc = step 175) : OutOnOVSt;
1511         TRUE : OutOnOV;
1512     esac;
1513 next(OutOffOV) :=
1514     case
1515         (loc = step 175) : OutOffOVSt;
1516         TRUE : OutOffOV;
1517     esac;
1518 next(OutOVSt_aux) :=
1519     case
1520         (loc = step 177) : TRUE;
1521         (loc = step 179) : FALSE;
1522         TRUE : OutOVSt_aux;
1523     esac;
1524 next(RE_OutOVSt_aux) :=
1525     case
1526         (loc = step 181) : TRUE;
1527         (loc = step 182) : FALSE;
1528         TRUE : RE_OutOVSt_aux;
1529     esac;
1530 next(RE_OutOVSt_aux_old) :=
1531     case
1532         (loc = step 181) : TRUE;
1533         (loc = step 182) : OutOVSt_aux;
1534         TRUE : RE_OutOVSt_aux_old;

```

```

1532     esac;
1533 next (FE_OutOVSt_aux) :=
1534     case
1535         (loc = step184) : TRUE;
1536         (loc = step185) : FALSE;
1537         TRUE : FE_OutOVSt_aux;
1538     esac;
1539 next (FE_OutOVSt_aux_old) :=
1540     case
1541         (loc = step184) : FALSE;
1542         (loc = step185) : OutOVSt_aux;
1543         TRUE : FE_OutOVSt_aux_old;
1544     esac;
1545 next (PosW_aux) :=
1546     case
1547         (loc = step187) : TRUE;
1548         (loc = step189) : FALSE;
1549         TRUE : PosW_aux;
1550     esac;
1551 next (Timer_Warning.Q) :=
1552     case
1553         (loc = step191) : FALSE;
1554         (loc = step197) : TRUE;
1555         TRUE : Timer_Warning.Q;
1556     esac;
1557 next (Timer_Warning.ET) :=
1558     case
1559         (loc = step191) : 0sd32_0;
1560         (loc = step193) : 0sd32_0;
1561         (loc = step196) : !_GLOBAL.TIME - Timer_Warning.Tstart;
1562         (loc = step197) : POnOffb.PWDtb;
1563         TRUE : Timer_Warning.ET;
1564     esac;
1565 next (Timer_Warning.running) :=
1566     case
1567         (loc = step191) : FALSE;
1568         (loc = step193) : TRUE;
1569         TRUE : Timer_Warning.running;
1570     esac;
1571 next (Timer_Warning.Tstart) :=
1572     case
1573         (loc = step193) : !_GLOBAL.TIME;
1574         TRUE : Timer_Warning.Tstart;
1575     esac;
1576 next (PosW) :=
1577     case
1578         (loc = step198) : Timer_Warning.Q;
1579         TRUE : PosW;
1580     esac;
1581 next (Time_Warning) :=
1582     case
1583         (loc = step198) : Timer_Warning.ET;
1584         TRUE : Time_Warning;
1585     esac;
1586 next (AIBW) :=
1587     case
1588         (loc = step198) : AIB;
1589         TRUE : AIBW;
1590     esac;
1591 next (PulseWidth) :=
1592     case
1593         (loc = step198) : (0sd32_150000 * 0sd32_100) / (signed(extend(T_CYCLE, 16)) * 0
1594             sd32_100 + 0sd32_1);
1595         TRUE : PulseWidth;
1596     esac;
1597 next (FSIinc) :=
1598     case
1599         (loc = step200) : FSIinc + 0sd16_1;
1600         (loc = step202) : 0sd16_0;
1601         TRUE : FSIinc;
1602     esac;
1603 next (WFuStopISt) :=
1604     case
1605         (loc = step200) : TRUE;
1606         (loc = step202) : FuStopISt;
1607         TRUE : WFuStopISt;
1608     esac;
1609 next (TSIinc) :=
1610     case
1611         (loc = step204) : TSIinc + 0sd16_1;
1612         (loc = step206) : 0sd16_0;
1613         TRUE : TSIinc;
1614     esac;
1615 next (WTStopISt) :=
1616     case
1617         (loc = step204) : TRUE;
1618         (loc = step206) : TStopISt;
1619         TRUE : WTStopISt;
1620     esac;
1621 next (SIinc) :=

```

```

1621     case
1622         (loc = step208) : SIinc + 0sd16_1;
1623         (loc = step210) : 0sd16_0;
1624         TRUE : SIinc;
1625     esac;
1626 next(WStartISt) :=
1627     case
1628         (loc = step208) : TRUE;
1629         (loc = step210) : StartISt;
1630         TRUE : WStartISt;
1631     esac;
1632 next(AIinc) :=
1633     case
1634         (loc = step212) : AIinc + 0sd16_1;
1635         (loc = step214) : 0sd16_0;
1636         TRUE : AIinc;
1637     esac;
1638 next(WAISt) :=
1639     case
1640         (loc = step212) : TRUE;
1641         (loc = step214) : AISt;
1642         TRUE : WAISt;
1643     esac;
1644 next(Stsreg01b[8]) :=
1645     case
1646         (loc = step215) : OnSt;
1647         TRUE : Stsreg01b[8];
1648     esac;
1649 next(Stsreg01b[9]) :=
1650     case
1651         (loc = step215) : OffSt;
1652         TRUE : Stsreg01b[9];
1653     esac;
1654 next(Stsreg01b[10]) :=
1655     case
1656         (loc = step215) : AuMoSt;
1657         TRUE : Stsreg01b[10];
1658     esac;
1659 next(Stsreg01b[11]) :=
1660     case
1661         (loc = step215) : MMoSt;
1662         TRUE : Stsreg01b[11];
1663     esac;
1664 next(Stsreg01b[12]) :=
1665     case
1666         (loc = step215) : FoMoSt;
1667         TRUE : Stsreg01b[12];
1668     esac;
1669 next(Stsreg01b[13]) :=
1670     case
1671         (loc = step215) : LDSt;
1672         TRUE : Stsreg01b[13];
1673     esac;
1674 next(Stsreg01b[14]) :=
1675     case
1676         (loc = step215) : IOErrorW;
1677         TRUE : Stsreg01b[14];
1678     esac;
1679 next(Stsreg01b[15]) :=
1680     case
1681         (loc = step215) : IOSimuW;
1682         TRUE : Stsreg01b[15];
1683     esac;
1684 next(Stsreg01b[0]) :=
1685     case
1686         (loc = step215) : AuMRW;
1687         TRUE : Stsreg01b[0];
1688     esac;
1689 next(Stsreg01b[1]) :=
1690     case
1691         (loc = step215) : PosW;
1692         TRUE : Stsreg01b[1];
1693     esac;
1694 next(Stsreg01b[2]) :=
1695     case
1696         (loc = step215) : WStartISt;
1697         TRUE : Stsreg01b[2];
1698     esac;
1699 next(Stsreg01b[3]) :=
1700     case
1701         (loc = step215) : WTStopISt;
1702         TRUE : Stsreg01b[3];
1703     esac;
1704 next(Stsreg01b[4]) :=
1705     case
1706         (loc = step215) : AIUnAck;
1707         TRUE : Stsreg01b[4];
1708     esac;
1709 next(Stsreg01b[5]) :=
1710     case

```

```

1711         (loc = step215) : AuIhFoMo;
1712         TRUE : Stsreg01b[5];
1713     esac;
1714 next(Stsreg01b[6]) :=
1715     case
1716         (loc = step215) : WA1St;
1717         TRUE : Stsreg01b[6];
1718     esac;
1719 next(Stsreg01b[7]) :=
1720     case
1721         (loc = step215) : AuIhMMo;
1722         TRUE : Stsreg01b[7];
1723     esac;
1724 next(Stsreg02b[8]) :=
1725     case
1726         (loc = step215) : OutOnOVSt;
1727         TRUE : Stsreg02b[8];
1728     esac;
1729 next(Stsreg02b[9]) :=
1730     case
1731         (loc = step215) : AuOnRSt;
1732         TRUE : Stsreg02b[9];
1733     esac;
1734 next(Stsreg02b[10]) :=
1735     case
1736         (loc = step215) : MOnRSt;
1737         TRUE : Stsreg02b[10];
1738     esac;
1739 next(Stsreg02b[11]) :=
1740     case
1741         (loc = step215) : AuOffRSt;
1742         TRUE : Stsreg02b[11];
1743     esac;
1744 next(Stsreg02b[12]) :=
1745     case
1746         (loc = step215) : MOffRSt;
1747         TRUE : Stsreg02b[12];
1748     esac;
1749 next(Stsreg02b[13]) :=
1750     case
1751         (loc = step215) : HOnRSt;
1752         TRUE : Stsreg02b[13];
1753     esac;
1754 next(Stsreg02b[14]) :=
1755     case
1756         (loc = step215) : HOffRSt;
1757         TRUE : Stsreg02b[14];
1758     esac;
1759 next(Stsreg02b[15]) :=
1760     case
1761         (loc = step215) : FALSE;
1762         TRUE : Stsreg02b[15];
1763     esac;
1764 next(Stsreg02b[0]) :=
1765     case
1766         (loc = step215) : FALSE;
1767         TRUE : Stsreg02b[0];
1768     esac;
1769 next(Stsreg02b[1]) :=
1770     case
1771         (loc = step215) : FALSE;
1772         TRUE : Stsreg02b[1];
1773     esac;
1774 next(Stsreg02b[2]) :=
1775     case
1776         (loc = step215) : WFuStop1St;
1777         TRUE : Stsreg02b[2];
1778     esac;
1779 next(Stsreg02b[3]) :=
1780     case
1781         (loc = step215) : EnRstartSt;
1782         TRUE : Stsreg02b[3];
1783     esac;
1784 next(Stsreg02b[4]) :=
1785     case
1786         (loc = step215) : SoftLDSt;
1787         TRUE : Stsreg02b[4];
1788     esac;
1789 next(Stsreg02b[5]) :=
1790     case
1791         (loc = step215) : AlBW;
1792         TRUE : Stsreg02b[5];
1793     esac;
1794 next(Stsreg02b[6]) :=
1795     case
1796         (loc = step215) : OutOffOVSt;
1797         TRUE : Stsreg02b[6];
1798     esac;
1799 next(Stsreg02b[7]) :=
1800     case

```

```

1801         (loc = step215) : FALSE;
1802         TRUE : Stsreg02b[7];
1803     esac;
1804 next(RE_AlUnAck) :=
1805     case
1806         (loc = step218) : TRUE;
1807         (loc = step219) : FALSE;
1808         (loc = step221) : FALSE;
1809         TRUE : RE_AlUnAck;
1810     esac;
1811 next(FE_AlUnAck) :=
1812     case
1813         (loc = step218) : FALSE;
1814         (loc = step219) : TRUE;
1815         (loc = step221) : FALSE;
1816         TRUE : FE_AlUnAck;
1817     esac;
1818 next(AlUnAck_old) :=
1819     case
1820         (loc = step220) : AlUnAck;
1821         TRUE : AlUnAck_old;
1822     esac;
1823
1824 next(_GLOBAL_TIME) :=
1825     case
1826         (loc = end) : _GLOBAL_TIME + signed(extend(T_CYCLE, 16));
1827         TRUE : _GLOBAL_TIME;
1828     esac;
1829
1830 next(T_CYCLE) :=
1831     case
1832         (loc = start) : ((extend(random_t_cycle, 8)) mod 0ud16_95 + 0ud16_5);
1833         TRUE : T_CYCLE;
1834     esac;
1835
1836 --extra assertion variables
1837 next(sFoMoSt_aux) :=
1838     case
1839         (loc = step1) : FoMoSt_aux;
1840         TRUE : sFoMoSt_aux;
1841     esac;
1842 next(sAuAuMoR) :=
1843     case
1844         (loc = step1) : AuAuMoR;
1845         TRUE : sAuAuMoR;
1846     esac;
1847 next(sManreg01b8) :=
1848     case
1849         (loc = step1) : Manreg01b[8];
1850         TRUE : sManreg01b8;
1851     esac;
1852 next(sAuIhFoMo) :=
1853     case
1854         (loc = step1) : AuIhFoMo;
1855         TRUE : sAuIhFoMo;
1856     esac;
1857 next(sAuIhMMo) :=
1858     case
1859         (loc = step1) : AuIhMMo;
1860         TRUE : sAuIhMMo;
1861     esac;
1862 next(sMMoSt_aux) :=
1863     case
1864         (loc = step1) : MMoSt_aux;
1865         TRUE : sMMoSt_aux;
1866     esac;
1867 next(pOutOnOV) :=
1868     case
1869         (loc = nvar) : OutOnOV;
1870         TRUE : pOutOnOV;
1871     esac;
1872 next(pTStopI) :=
1873     case
1874         (loc = nvar) : TStopI;
1875         TRUE : pTStopI;
1876     esac;
1877 next(pFuStopI) :=
1878     case
1879         (loc = nvar) : FuStopI;
1880         TRUE : pFuStopI;
1881     esac;
1882 next(pStartI) :=
1883     case
1884         (loc = nvar) : StartI;
1885         TRUE : pStartI;
1886     esac;
1887 next(pOutOnOVSt) :=
1888     case
1889         (loc = nvar) : OutOnOVSt;
1890         TRUE : pOutOnOVSt;

```

```

1891     esac;
1892     next(pMMoSt) :=
1893     case
1894         (loc = nvar) : MMoSt;
1895         TRUE : pMMoSt;
1896     esac;
1897     next(pManreg01b12) :=
1898     case
1899         (loc = nvar) : Manreg01b[12];
1900         TRUE : pManreg01b12;
1901     esac;
1902     next(pManreg01b13) :=
1903     case
1904         (loc = nvar) : Manreg01b[13];
1905         TRUE : pManreg01b13;
1906     esac;
1907     init(first) := TRUE;
1908     next(first) :=
1909     case
1910         (loc = end) : FALSE;
1911         TRUE : first;
1912     esac;
1913
1914 --Properties

```

Listing F.1: CPC.smv

G CPC program in C

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

```

1  #include <stdbool.h>
2  _Bool nondet_bool();
3  int nondet_int();
4  unsigned short nondet_unsignedshort();
5
6  struct TP{
7      bool Q;
8      int ET;
9      bool old-in;
10     int due;
11 };
12
13 struct TON{
14     bool Q;
15     int ET;
16     bool running;
17     int start;
18 };
19
20 //VAR_INPUT
21 bool HFOn, HFOff, HLD, IOError, IOSimu, AIB;
22 bool Manreg01b[16];
23 bool HOnR, HOffR, StartI, TStopI, FuStopI, Al, AuOnR, AuOffR, AuAuMoR, AuIhMMo, AuIhFoMo,
24     AuAlAck, IhAuMRW, AuRstart;
25 struct CPC_ONOFF_PARAM{
26     bool ParRegb[16];
27     int PPulseLeb, PWDtb;
28 };
29
30 //VAR_OUTPUT
31 bool Stsreg01b[16];
32 bool Stsreg02b[16];
33 bool OutOnOV = false;
34 bool OutOffOV = false;
35 bool OnSt = false;
36 bool OffSt = false;
37 bool AuMoSt = false;
38 bool MMoSt = false;
39 bool LDSt = false;
40 bool SoftLDSt = false;
41 bool FoMoSt = false;
42 bool AuOnRSt = false;
43 bool AuOffRSt = false;
44 bool MOnRSt = false;
45 bool MOffRSt = false;
46 bool HOnRSt = false;
47 bool HOffRSt = false;
48 bool IOErrorW = false;
49 bool IOSimuW = false;
50 bool AuMRW = false;

```



```
51 | bool AIUnAck = false;
52 | bool PosW = false;
53 | bool StartISt = false;
54 | bool TStopISt = false;
55 | bool FuStopISt = false;
56 | bool AISt = false;
57 | bool AIBW = false;
58 | bool EnRstartSt = true;
59 | bool RdyStartSt = false;
60 |
61 | //Internal Variables
62 | bool E_MAuMoR = false;
63 | bool E_MMMoR = false;
64 | bool E_MFoMoR = false;
65 | bool E_MOnR = false;
66 | bool E_MOffR = false;
67 | bool E_MAIAckR = false;
68 | bool E_StartI = false;
69 | bool E_TStopI = false;
70 | bool E_FuStopI = false;
71 | bool E_AI = false;
72 | bool E_AuAuMoR = false;
73 | bool E_AuAIAck = false;
74 | bool E_MSoftLDR = false;
75 | bool E_MEnRstartR = false;
76 | bool RE_AIUnAck = false;
77 | bool FE_AIUnAck = false;
78 | bool RE_PulseOn = false;
79 | bool FE_PulseOn = false;
80 | bool RE_PulseOff = false;
81 | bool RE_OutOVSt_aux = false;
82 | bool FE_OutOVSt_aux = false;
83 | bool FE_InterlockR = false;
84 |
85 | bool MAuMoR_old = false;
86 | bool MMMoR_old = false;
87 | bool MFoMoR_old = false;
88 | bool MOnR_old = false;
89 | bool MOffR_old = false;
90 | bool MAIAckR_old = false;
91 | bool AuAuMoR_old = false;
92 | bool AuAIAck_old = false;
93 | bool StartI_old = false;
94 | bool TStopI_old = false;
95 | bool FuStopI_old = false;
96 | bool AI_old = false;
97 | bool AIUnAck_old = false;
98 | bool MSoftLDR_old = false;
99 | bool MEnRstartR_old = false;
100 | bool RE_PulseOn_old = false;
101 | bool FE_PulseOn_old = false;
102 | bool RE_PulseOff_old = false;
103 | bool RE_OutOVSt_aux_old = false;
104 | bool FE_OutOVSt_aux_old = false;
105 | bool FE_InterlockR_old = false;
106 |
107 | bool PFsPosOn = false;
108 | bool PFsPosOn2 = false;
109 | bool PHFOn = false;
110 | bool PHFOff = false;
111 | bool PPulse = false;
112 | bool PPulseCste = false;
113 | bool PHLD = false;
114 | bool PHLDCmd = false;
115 | bool PAnim = false;
116 | bool POutOff = false;
117 | bool PEnRstart = false;
118 | bool PRstartFS = false;
119 | bool OutOnOVSt = false;
120 | bool OutOffOVSt = false;
121 | bool AuMoSt_aux = false;
122 | bool MMoSt_aux = false;
123 | bool FoMoSt_aux = false;
124 | bool SoftLDSt_aux = false;
125 | bool PulseOn = false;
126 | bool PulseOff = false;
127 | bool PosW_aux = false;
128 | bool OutOVSt_aux = false;
129 | bool fullNotAcknowledged = false;
130 | bool PulseOnR = false;
131 | bool PulseOffR = false;
132 | bool InterlockR = false;
133 |
134 | //time Time.Warning;
135 | int Time.Warning;
136 | struct TP Timer_PulseOn;
137 | struct TP Timer_PulseOff;
138 | struct TON Timer_Warning;
139 |
140 | double PulseWidth;
```

```

141 short FSIinc, TSIinc, SIinc, Ainc;
142 bool WTStopISt, WStartISt, WAISt, WFuStopISt;
143 int _GLOBAL_TIME = 0;
144 unsigned short T_CYCLE;
145
146 bool R_EDGE(bool new, bool *old){
147     if(new &&!*old){
148         *old = true;
149         return true;
150     } else{
151         *old = new;
152         return false;
153     }
154 }
155
156 bool F_EDGE(bool new, bool *old){
157     if(!new && *old){
158         *old = false;
159         return true;
160     } else{
161         *old = new;
162         return false;
163     }
164 }
165
166 void DETECT_EDGE(bool new, bool old, bool *re, bool *fe){
167     if(new!=old){
168         if(new){
169             *re = true;
170             *fe = false;
171         } else{
172             *re = false;
173             *fe = true;
174         }
175     } else{
176         *re = false;
177         *fe = false;
178     }
179 }
180
181 void updateTP(bool *tpQ, int *tpET, bool *tpold_in, int *tpdue, bool in, int PT){
182     if(in && ! *tpold_in && ! *tpQ){
183         *tpdue = _GLOBAL_TIME + PT;
184     }
185     if(_GLOBAL_TIME <= *tpdue){
186         *tpQ = true;
187         *tpET = PT - (*tpdue - _GLOBAL_TIME);
188     } else{
189         *tpQ = false;
190         if(in){
191             *tpET = PT;
192         } else{
193             *tpET = 0;
194         }
195     }
196     *tpold_in = in;
197 }
198
199 void updateTON(bool *tonQ, int *tonET, bool *tonrunning, int *tonstart, int PT, bool in){
200     if(!in){
201         *tonQ = false;
202         *tonET = 0;
203         *tonrunning = false;
204     } else if(!*tonrunning){
205         *tonstart = _GLOBAL_TIME;
206         *tonrunning = true;
207         *tonET = 0;
208     } else if(!((_GLOBAL_TIME - (*tonstart + PT)) >= 0)){
209         if(!*tonQ){
210             *tonET = _GLOBAL_TIME - *tonstart;
211         }
212     } else{
213         *tonQ = true;
214         *tonET = PT;
215     }
216 }
217
218 int main(){
219
220 Stsreg01b[0] = false;
221 Stsreg02b[0] = false;
222 Stsreg01b[1] = false;
223 Stsreg02b[1] = false;
224 Stsreg01b[2] = false;
225 Stsreg02b[2] = false;
226 Stsreg01b[3] = false;
227 Stsreg02b[3] = false;
228 Stsreg01b[4] = false;
229 Stsreg02b[4] = false;
230 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;
242 Stsreg01b[11] = false;
243 Stsreg02b[11] = 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
253 Timer_PulseOn.Q = false;
254 Timer_PulseOn.old_in = false;
255 Timer_PulseOn.due = 0;
256 Timer_PulseOff.Q = false;
257 Timer_PulseOff.old_in = false;
258 Timer_PulseOff.due = 0;
259 Timer_Warning.Q = false;
260 Timer_Warning.ET = 0;
261 Timer_Warning.running = false;
262 Timer_Warning.start = 0;
263 int first = true;
264
265 POnOffb.ParRegb[0] = nondet_bool();
266 POnOffb.ParRegb[1] = nondet_bool();
267 POnOffb.ParRegb[2] = nondet_bool();
268 POnOffb.ParRegb[3] = nondet_bool();
269 POnOffb.ParRegb[4] = nondet_bool();
270 POnOffb.ParRegb[5] = nondet_bool();
271 POnOffb.ParRegb[6] = nondet_bool();
272 POnOffb.ParRegb[7] = nondet_bool();
273 POnOffb.ParRegb[8] = nondet_bool();
274 POnOffb.ParRegb[9] = nondet_bool();
275 POnOffb.ParRegb[10] = nondet_bool();
276 POnOffb.ParRegb[11] = nondet_bool();
277 POnOffb.ParRegb[12] = nondet_bool();
278 POnOffb.ParRegb[13] = nondet_bool();
279 POnOffb.ParRegb[14] = nondet_bool();
280 POnOffb.ParRegb[15] = nondet_bool();
281
282 POnOffb.PPulseLeb = nondet_int();
283 POnOffb.PWDtb = nondet_int();
284
285 while(true){
286     //extra assertion variables
287     bool pOutOnOV = OutOnOV;
288     bool pTStopI = TStopI;
289     bool pFuStopI = FuStopI;
290     bool pStartI = StartI;
291     bool pOutOnOVSt = OutOnOVSt;
292     bool pMMoSt = MMoSt;
293     bool pManreg01b12 = Manreg01b[12];
294     bool pManreg01b13 = Manreg01b[13];
295
296     //nondet-input
297     HFOn = nondet_bool();
298     HFOff = nondet_bool();
299     HLD = nondet_bool();
300     IOError = nondet_bool();
301     IOSimu = nondet_bool();
302     AIB = nondet_bool();
303
304     Manreg01b[0] = nondet_bool();
305     Manreg01b[1] = nondet_bool();
306     Manreg01b[2] = nondet_bool();
307     Manreg01b[3] = nondet_bool();
308     Manreg01b[4] = nondet_bool();
309     Manreg01b[5] = nondet_bool();
310     Manreg01b[6] = nondet_bool();
311     Manreg01b[7] = nondet_bool();
312     Manreg01b[8] = nondet_bool();
313     Manreg01b[9] = nondet_bool();
314     Manreg01b[10] = nondet_bool();
315     Manreg01b[11] = nondet_bool();
316     Manreg01b[12] = nondet_bool();
317     Manreg01b[13] = nondet_bool();
318     Manreg01b[14] = nondet_bool();
319     Manreg01b[15] = nondet_bool();
320

```

```

321     HOnR = nondet_bool();
322     HOffR = nondet_bool();
323     StartI = nondet_bool();
324     TStopI = nondet_bool();
325     FuStopI = nondet_bool();
326     A1 = nondet_bool();
327     AuOnR = nondet_bool();
328     AuOffR = nondet_bool();
329     AuAuMoR = nondet_bool();
330     AulhMMo = nondet_bool();
331     AulhFoMo = nondet_bool();
332     AuAlAck = nondet_bool();
333     IhAuMRW = nondet_bool();
334     AuRstart = nondet_bool();
335     T_CYCLE = 5 + (nondet_unsignedshort() % 95);
336
337     //extra assertion variables
338     bool sFoMoSt_aux = FoMoSt_aux;
339     bool sAuAuMoR = AuAuMoR;
340     bool sManreg01b8 = Manreg01b[8];
341     bool sAuIhFoMo = AulhFoMo;
342     bool sAulhMMo = AulhMMo;
343     bool sMMoSt_aux = MMoSt_aux;
344
345     //input manager
346     E_MAuMoR = R_EDGE(Manreg01b[8], &MAuMoR_old);
347     E_MMMoR = R_EDGE(Manreg01b[9], &MMMoR_old);
348     E_MFoMoR = R_EDGE(Manreg01b[10], &MFoMoR_old);
349     E_MSoftLDR = R_EDGE(Manreg01b[11], &MSoftLDR_old);
350     E_MOnR = R_EDGE(Manreg01b[12], &MOnR_old);
351     E_MOffR = R_EDGE(Manreg01b[13], &MOffR_old);
352     E_MEnRstartR = R_EDGE(Manreg01b[1], &MEnRstartR_old);
353     E_MAlAckR = R_EDGE(Manreg01b[7], &MAIAckR_old);
354
355     PFSPosOn = POnOffb.ParRegb[8];
356     PHFOn = POnOffb.ParRegb[9];
357     PHFOff = POnOffb.ParRegb[10];
358     //PPulse = POnOffb.ParRegb[11];
359     PPulse = false;
360     PHLD = POnOffb.ParRegb[12];
361     PHLDCmd = POnOffb.ParRegb[13];
362     PAnim = POnOffb.ParRegb[14];
363     POutOff = POnOffb.ParRegb[15];
364     PEnRstart = POnOffb.ParRegb[0];
365     PRstartFS = POnOffb.ParRegb[1];
366     PFSPosOn2 = POnOffb.ParRegb[2];
367     PPulseCste = POnOffb.ParRegb[3];
368
369     E_AuAuMoR = R_EDGE(AuAuMoR, &AuAuMoR_old);
370     E_AuAlAck = R_EDGE(AuAlAck, &AuAlAck_old);
371     E_StartI = R_EDGE(StartI, &StartI_old);
372     E_TStopI = R_EDGE(TStopI, &TStopI_old);
373     E_FuStopI = R_EDGE(FuStopI, &FuStopI_old);
374     E_A1 = R_EDGE(A1, &A1_old);
375
376     StartISt = StartI;
377     TStopISt = TStopI;
378     FuStopISt = FuStopI;
379
380     //interlock & acknowledge
381     if (E_MAlAckR || E_AuAlAck){
382         fullNotAcknowledged = false;
383         AIUnAck = false;
384     } else if (E_TStopI || E_StartI || E_FuStopI || E_A1){
385         AIUnAck = true;
386     }
387     if (((PEnRstart && (E_MEnRstartR || AuRstart) && !FuStopISt) || (PEnRstart &&
388         PRstartFS && (E_MEnRstartR || AuRstart))) && !fullNotAcknowledged){
389         EnRstartSt = true;
390     }
391     if (E_FuStopI){
392         fullNotAcknowledged = true;
393         if (PEnRstart){
394             EnRstartSt = false;
395         }
396     }
397     InterlockR = TStopISt || FuStopISt || fullNotAcknowledged || !EnRstartSt ||
398         (StartISt && !POutOff && !OutOnOV) ||
399         (StartISt && POutOff && ((PFSPosOn && OutOVSt_aux) || (!PFSPosOn && !
400             OutOVSt_aux)));
401     FE_InterlockR = F_EDGE(InterlockR, &FE_InterlockR_old);
402
403     //mode manager
404     if (!(HLD && PHLD)){
405         //forced mode
406         if ((AuMoSt_aux || MMoSt_aux || SoftLDSt_aux) && E_MFoMoR && !(AulhFoMo)){
407             AuMoSt_aux = false;
408             MMoSt_aux = false;
409             FoMoSt_aux = true;
410             SoftLDSt_aux = false;

```

```

409     }
410     //manual mode
411     if((AuMoSt_aux || FoMoSt_aux || SoftLDSt_aux) && E_MMMoR && !(AuIhMMo)){
412         AuMoSt_aux = false;
413         MMoSt_aux = true;
414         FoMoSt_aux = false;
415         SoftLDSt_aux = false;
416     }
417     //auto mode
418     if((MMoSt_aux && (E_MAuMoR || E_AuAuMoR )) || (FoMoSt_aux && E_MAuMoR) ||
419     (SoftLDSt_aux && E_MAuMoR) || (MMoSt_aux && AuIhMMo) || (FoMoSt_aux && AuIhFoMo)
420     ||
421     (SoftLDSt_aux && AuIhFoMo) || !(AuMoSt_aux || MMoSt_aux ||
422     FoMoSt_aux || SoftLDSt_aux)){
423         AuMoSt_aux = true;
424         MMoSt_aux = false;
425         FoMoSt_aux = false;
426         SoftLDSt_aux = false;
427     }
428     // Software Local Mode
429     if((AuMoSt_aux || MMoSt_aux) && E_MSoftLDR && !(AuIhFoMo)){
430         AuMoSt_aux = false;
431         MMoSt_aux = false;
432         FoMoSt_aux = false;
433         SoftLDSt_aux = true;
434     }
435     // Status setting
436     LDSt = false;
437     AuMoSt = AuMoSt_aux;
438     MMoSt = MMoSt_aux;
439     FoMoSt = FoMoSt_aux;
440     SoftLDSt = SoftLDSt_aux;
441 }else{
442 // Local Drive Mode
443 AuMoSt = false;
444 MMoSt = false;
445 FoMoSt = false;
446 LDSt = true;
447 SoftLDSt= false;
448 }
449 // LIMIT MANAGER
450 // On/Open Evaluation
451 OnSt=(HFOn && PHFOn) ||
452 (!PHFOn && PHFOff && PAnim && !HFOff) ||
453 (!PHFOn && !PHFOff && OutOVSt_aux);
454 // Off/Closed Evaluation
455 OffSt=(HFOff && PHFOff) ||
456 (!PHFOff && PHFOn && PAnim && !HFOn) ||
457 (!PHFOn && !PHFOff && !OutOVSt_aux);
458 // REQUEST MANAGER
459 // Auto On/Off Request
460 if( AuOffR ){
461     AuOnRSt = false;
462 }else if( AuOnR ){
463     AuOnRSt = true;
464 }else if( fullNotAcknowledged || FuStopISt || !EnRstartSt ){
465     AuOnRSt = PFsPosOn;
466 }
467 AuOffRSt= !AuOnRSt;
468 // Manual On/Off Request
469 if( (((E_MOffR && (MMoSt || FoMoSt || SoftLDSt))
470 || (AuOffRSt && AuMoSt)
471 || (LDSt && PHLDCmd && HOffRSt)
472 || (FE_PulseOn && PPulse && !POutOff) && EnRstartSt)
473 || (E_FuStopI && !PFsPosOn) ){
474     MOnRSt = false;
475 }
476 }else if( (((E_MOnR && (MMoSt || FoMoSt || SoftLDSt))
477 || (AuOnRSt && AuMoSt)
478 || (LDSt && PHLDCmd && HOnRSt) && EnRstartSt)
479 || (E_FuStopI && PFsPosOn) ){
480     MOnRSt = true;
481 }
482 }
483 MOffRSt= !MOnRSt;
484 // Local Drive Request
485 if( HOffR ){
486     HOnRSt = false;
487 }else if( HOnR ){
488     HOnRSt = true;
489 }
490 HOffRSt = !(HOnRSt);
491

```

```

497 // PULSE REQUEST MANAGER
498 if( PPulse ){
499     if( InterlockR ){
500         PulseOnR= (PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
501         PulseOffR= (!PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
502     }else if( FE_InterlockR ){
503         PulseOnR= false;
504         PulseOffR= false;
505         updateTP(&Timer_PulseOn.Q, &Timer_PulseOn.ET, &Timer_PulseOn.old_in , &
506             Timer_PulseOn.due, false ,0);
507         updateTP(&Timer_PulseOff.Q, &Timer_PulseOff.ET, &Timer_PulseOff.old_in , &
508             Timer_PulseOff.due, false ,0);
509     }else if( (MOffRSt && (MMoSt || FoMoSt || SoftLDSt)) || (AuOffRSt && AuMoSt) || (
510         HOffR && LDSt && PHLDCmd) ){ //Off Request
511         PulseOnR= false;
512         PulseOffR= true;
513     }else if( (MOnRSt && (MMoSt || FoMoSt || SoftLDSt)) || (AuOnRSt && AuMoSt) || (
514         HOnR && LDSt && PHLDCmd) ){ //On Request
515         PulseOnR= true;
516         PulseOffR= false;
517     }else {
518         PulseOnR= false;
519         PulseOffR= false;
520     }
521     //Pulse functions
522     updateTP(&Timer_PulseOn.Q, &Timer_PulseOn.ET, &Timer_PulseOn.old_in , &
523         Timer_PulseOn.due, PulseOnR, POnOffb.PPulseLeb);
524     updateTP(&Timer_PulseOff.Q, &Timer_PulseOff.ET, &Timer_PulseOff.old_in , &
525         Timer_PulseOff.due, PulseOffR, POnOffb.PPulseLeb);
526     RE_PulseOn = R_EDGE(PulseOn, RE_PulseOn_old);
527     FE_PulseOn = F_EDGE(PulseOn, FE_PulseOn_old);
528     RE_PulseOff = R_EDGE(PulseOff, RE_PulseOff_old);
529     //The pulse functions have to be reset when changing from On to Off
530     if( RE_PulseOn ){
531         updateTP(&Timer_PulseOff.Q, &Timer_PulseOff.ET, &Timer_PulseOff.old_in , &
532             Timer_PulseOff.due, false ,0);
533     }
534     if( RE_PulseOff ){
535         updateTP(&Timer_PulseOn.Q, &Timer_PulseOn.ET, &Timer_PulseOn.old_in , &
536             Timer_PulseOn.due, false ,0);
537     }
538     if( PPulseCste ){
539         PulseOn = Timer_PulseOn.Q && !PulseOffR;
540         PulseOff = Timer_PulseOff.Q && !PulseOnR;
541     }else{
542         PulseOn = Timer_PulseOn.Q && !PulseOffR && (!PHFOn || (PHFOn && !HFOn));
543         PulseOff = Timer_PulseOff.Q && !PulseOnR && (!PHFOff || (PHFOff && !HFOff));
544     }
545 }
546 // Output On Request
547 OutOnOVSt = (PPulse && PulseOn) ||
548     (!PPulse && ((MOnRSt && (MMoSt || FoMoSt || SoftLDSt)) ||
549     (AuOnRSt && AuMoSt) ||
550     (HOnRSt && LDSt && PHLDCmd)));
551 // Output Off Request
552 if( POutOff ){
553     OutOffOVSt = (PulseOff && PPulse) ||
554     (!PPulse && ((MOffRSt && (MMoSt || FoMoSt || SoftLDSt)) || (AuOffRSt &&
555     AuMoSt) || (HOffRSt && LDSt && PHLDCmd)));
556 }
557 // Interlocks / FailSafe
558 if( POutOff ){
559     if( InterlockR ){
560         if( PPulse && !PFsPosOn2 ){
561             if( PFsPosOn ){
562                 OutOnOVSt = PulseOn;
563                 OutOffOVSt = false;
564             }else{
565                 OutOnOVSt = false;
566                 OutOffOVSt = PulseOff;
567             }
568         }else{
569             OutOnOVSt = (PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
570             OutOffOVSt= (!PFsPosOn && !PFsPosOn2) || (PFsPosOn && PFsPosOn2);
571         }
572     }else{
573         if( InterlockR ){
574             OutOnOVSt= PFsPosOn;
575         }
576     }
577 }
578 // Ready to Start Status
579 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 AuMRW = (MMoSt || FoMoSt || SoftLDSt) &&
588         ((AuOnRSt || MOnRSt) && !(AuOnRSt && MOnRSt)) || ((AuOffRSt || MOffRSt
589         ) && !(AuOffRSt && MOffRSt)) && !IhAuMRW;
590
591 // OUTPUT_MANAGER && OUTPUT REGISTER
592 if( !POutOff ){
593     if( PFsPosOn ){
594         OutOnOV = !OutOnOVSt;
595     }else{
596         OutOnOV = OutOnOVSt;
597     }
598 }else{
599     OutOnOV = OutOnOVSt;
600     OutOffOV = OutOffOVSt;
601 }
602
603 // Position warning
604 // Set reset of the OutOnOVSt
605 if( OutOnOVSt || (PPulse && PulseOnR) ){
606     OutOVSt_aux = true;
607 }
608 if( (OutOffOVSt && POutOff) || (!OutOnOVSt && !POutOff) || (PPulse && PulseOffR
609     ) ){
610     OutOVSt_aux = false;
611 }
612 RE_OutOVSt_aux = R_EDGE(OutOVSt_aux, RE_OutOVSt_aux_old);
613 FE_OutOVSt_aux = F_EDGE(OutOVSt_aux, FE_OutOVSt_aux_old);
614 if( ((OutOVSt_aux && ((PHFOn && !OnSt) || (PHFOff && OffSt)))
615     || (!OutOVSt_aux && ((PHFOff && !OffSt) || (PHFOn && OnSt)))
616     || (OffSt && OnSt))
617     && (!PPulse || (POutOff && PPulse && !OutOnOV && !OutOffOV))
618 ){
619     PosW_aux= true;
620 }
621 if( (!(OutOVSt_aux && ((PHFOn && !OnSt) || (PHFOff && OffSt)))
622     || (!OutOVSt_aux && ((PHFOff && !OffSt) || (PHFOn && OnSt)))
623     || (OffSt && OnSt))
624     || RE_OutOVSt_aux
625     || FE_OutOVSt_aux
626     || (PPulse && POutOff && OutOnOV)
627     || (PPulse && POutOff && OutOffOV)
628 ){
629     PosW_aux = false;
630 }
631 updateTON(&Timer_Warning.Q, &Timer_Warning.ET, &Timer_Warning.running, &
632     Timer_Warning.start, PosW_aux, POnOffb.PWDtb);
633
634 PosW = Timer_Warning.Q;
635 Time_Warning = Timer_Warning.ET;
636
637 // Alarm Blocked Warning
638 AlBW = AlB;
639
640 // Maintain Interlock status 1.5s in Stsreg for PVSS
641 PulseWidth = 1500/T_CYCLE;
642
643 if( FuStopISt || FSIinc > 0 ){
644     FSIinc = FSIinc + 1;
645     WFuStopISt = true;
646 }
647 if( FSIinc > PulseWidth || (!FuStopISt && FSIinc == 0) ){
648     FSIinc = 0;
649     WFuStopISt = FuStopISt;
650 }
651 if( TStopISt || TSIinc > 0 ){
652     TSIinc = TSIinc + 1;
653     WTStopISt = true;
654 }
655 if( TSIinc > PulseWidth || (!TStopISt && TSIinc == 0) ){
656     TSIinc = 0;
657     WTStopISt = TStopISt;
658 }
659 if( StartISt || SIinc > 0 ){
660     SIinc = SIinc + 1;
661     WStartISt= true;
662 }
663 if( SIinc > PulseWidth || (!StartISt && SIinc == 0) ){
664     SIinc = 0;
665     WStartISt = StartISt;
666 }
667 if( AlSt || Alinc > 0 ){

```

```

665     AInc = AInc + 1;
666     WA1St = true;
667 }
668 if( AInc > PulseWidth || (!A1St && AInc == 0) ){
669     AInc = 0;
670     WA1St = A1St;
671 }
672
673 // STATUS REGISTER
674 Stsreg01b[8] = OnSt;           //StsReg01 Bit 00
675 Stsreg01b[9] = OffSt;        //StsReg01 Bit 01
676 Stsreg01b[10] = AuMoSt;      //StsReg01 Bit 02
677 Stsreg01b[11] = MMoSt;      //StsReg01 Bit 03
678 Stsreg01b[12] = FoMoSt;     //StsReg01 Bit 04
679 Stsreg01b[13] = LDSt;       //StsReg01 Bit 05
680 Stsreg01b[14] = IOErrorW;   //StsReg01 Bit 06
681 Stsreg01b[15] = IOSimuW;    //StsReg01 Bit 07
682 Stsreg01b[0] = AuMRW;       //StsReg01 Bit 08
683 Stsreg01b[1] = PosW;        //StsReg01 Bit 09
684 Stsreg01b[2] = WStartISt;   //StsReg01 Bit 10
685 Stsreg01b[3] = WTStopISt;   //StsReg01 Bit 11
686 Stsreg01b[4] = AIUnAck;     //StsReg01 Bit 12
687 Stsreg01b[5] = AuIhFoMo;    //StsReg01 Bit 13
688 Stsreg01b[6] = WA1St;       //StsReg01 Bit 14
689 Stsreg01b[7] = AuIhMMo;     //StsReg01 Bit 15
690
691 Stsreg02b[8] = OutOnOVSt;    //StsReg02 Bit 00
692 Stsreg02b[9] = AuOnRSt;     //StsReg02 Bit 01
693 Stsreg02b[10] = MOnRSt;     //StsReg02 Bit 02
694 Stsreg02b[11] = AuOffRSt;   //StsReg02 Bit 03
695 Stsreg02b[12] = MOffRSt;    //StsReg02 Bit 04
696 Stsreg02b[13] = HOnRSt;     //StsReg02 Bit 05
697 Stsreg02b[14] = HOffRSt;    //StsReg02 Bit 06
698 Stsreg02b[15] = 0;          //StsReg02 Bit 07
699 Stsreg02b[0] = 0;           //StsReg02 Bit 08
700 Stsreg02b[1] = 0;           //StsReg02 Bit 09
701 Stsreg02b[2] = WFuStopISt;  //StsReg02 Bit 10
702 Stsreg02b[3] = EnRstartSt;  //StsReg02 Bit 11
703 Stsreg02b[4] = SoftLDSt;    //StsReg02 Bit 12
704 Stsreg02b[5] = AIBW;        //StsReg02 Bit 13
705 Stsreg02b[6] = OutOffOVSt;  //StsReg02 Bit 14
706 Stsreg02b[7] = 0;          //StsReg02 Bit 15
707
708 // Edges
709 DETECT_EDGE(AIUnAck, AIUnAck_old, &RE_AIUnAck, &FE_AIUnAck);
710 _GLOBAL_TIME = _GLOBAL_TIME + T_CYCLE;
711
712 // Properties
713 first = false;
714 }
715 }

```

Listing G.1: CPC.c

H Properties of the CPC program

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

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

- ```

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

```
- R1-5

```

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

```
  - R1-6

```

Assertion (!(AuMoSt) || (sAuAuMoR || sManreg01b8 || sAuIhMMo || !sMMoSt_aux))
CTL: AG(loc = end -> (!(AuMoSt) | (sAuAuMoR | sManreg01b8 | sAuIhMMo | !sMMoSt_aux)))
Result: true

```
  - R1-7

```

Assertion (!AuIhFoMo || !SoftLDSt)
CTL: AG(loc = end -> (!AuIhFoMo | !SoftLDSt))
Result: true

```
  - R1-8

```

Assertion (!AuIhFoMo || !FoMoSt)
CTL: AG(loc = end -> (!AuIhFoMo | !FoMoSt))
Result: true

```
  - R1-9

```

Assertion (!AuIhMMo || !MMoSt)
CTL: AG(loc = end -> (!AuIhMMo | !MMoSt))
Result: true

```
  - R1-11a

We have chosen to split property R1-11 in two properties.

```

Assertion (!(!pOutOnOV && !pTStopI && !pFuStopI && !pStartI && HLD && !HO nR && !HoffR && !TStopI
&& !FuStopI && !StartI) || !OutOnOV)
CTL: AG(loc = end -> (!(!pOutOnOV & !pTStopI & !pFuStopI & !pStartI & HLD & !HO nR & !HoffR
& !TStopI & !FuStopI & !StartI) | !OutOnOV | first))
Result: false

```
  - R1-11b

```

Assertion (!(pOutOnOV && !pTStopI && !pFuStopI && !pStartI && HLD && !HO nR && !HoffR && !TStopI
&& !FuStopI && !StartI) || OutOnOV)
CTL: AG(loc = end -> (!(pOutOnOV & !pTStopI & !pFuStopI & !pStartI & HLD & !HO nR & !HoffR
& !TStopI & !FuStopI & !StartI) | OutOnOV | first))
Result: false

```
  - R2-1

```

Assertion (!AuOffR || AuOffRSt)
CTL: AG(loc = end -> (!AuOffR | AuOffRSt))
Result: true

```
  - R2-2

```

Assertion (!AuOnR || AuOnRSt)
CTL: AG(loc = end -> (!AuOnR | AuOnRSt))
Result: false

```
  - R2-3

```

Assertion (!(AuOnR && !AuOffR && AuMoSt && !InterlockR && !PFsPosOn) || OutOnOV)
CTL: AG(loc = end -> (!(AuOnR & !AuOffR & AuMoSt & !InterlockR & !PFsPosOn) | OutOnOV))
Result: true

```
  - R2-4

```

Assertion (!(AuOffR && !AuOnR && AuMoSt && !InterlockR && !PFsPosOn) || !OutOnOV)
CTL: AG(loc = end -> (!(AuOffR & !AuOnR & AuMoSt & !InterlockR & !PFsPosOn) | !OutOnOV))
Result: true

```
  - R2-5

```

Assertion (!(!pManreg01b12 && Manreg01b[12] && !Manreg01b[13] && MMoSt && !InterlockR && !PFsPosOn)
|| OutOnOV)
CTL: AG(loc = end -> (!(!pManreg01b12 & Manreg01b[12] & !Manreg01b[13] & MMoSt & !InterlockR
& !PFsPosOn) | OutOnOV | first))

```

- Result: true
- R2-6
 

```
Assertion (!(pManreg01b13 && Manreg01b[13] && !Manreg01b[12] && MMoSt && !InterlockR && !PFsPosOn)
|| !OutOnOV)
CTL: AG(loc = end -> (!(pManreg01b13 & Manreg01b[13] & !Manreg01b[12] & MMoSt & !InterlockR
& !PFsPosOn) | !OutOnOV | first))
Result: true
```
  - R2-7
 

```
Assertion (!(HOnR && !HoffR && LDSt && !InterlockR && !PFsPosOn) || OutOnOV)
CTL: AG(loc = end -> (!(HOnR & !HoffR & LDSt & !InterlockR & !PFsPosOn) | OutOnOV))
Result: false
```
  - R2-8
 

```
Assertion (!(HoffR && !HOnR && LDSt && !InterlockR && !PFsPosOn) || !OutOnOV)
CTL: AG(loc = end -> (!(HoffR & !HOnR & LDSt & !InterlockR & !PFsPosOn) | !OutOnOV))
Result: true
```
  - R3-1
 

```
Assertion (!(InterlockR && PFsPosOn) || OutOnOVSt)
CTL: AG(loc = end -> (!(InterlockR & PFsPosOn) | OutOnOVSt))
Result: true
```
  - R3-2
 

```
Assertion (!(InterlockR && PFsPosOn) || !OutOffOVSt)
CTL: AG(loc = end -> (!(InterlockR & PFsPosOn) | !OutOffOVSt))
Result: false
```
  - R3-3
 

```
Assertion (!((TStopI || FuStopI || !EnRstartSt) && !PFsPosOn && PEnRstart && !PRstartFS) ||
!OutOnOVSt)
CTL: AG(loc = end -> (!((TStopI | FuStopI | !EnRstartSt) & !PFsPosOn & PEnRstart & !PRstartFS)
| !OutOnOVSt))
Result: true
```
  - R3-4
 

```
Assertion (!((FuStopI || fullNotAcknowledged) && PEnRstart && !PRstartFS) || EnRstartSt)
CTL: AG(loc = end -> (!((FuStopI | fullNotAcknowledged) & PEnRstart & !PRstartFS) | EnRstartSt))
Result: false
```
  - R3-5
 

```
Assertion (!(pOutOnOVSt && (StartI || FuStopI || TStopI || !EnRstartSt) && !PFsPosOn && PEnRstart
&& !PRstartFS) || !OutOnOVSt)
CTL: AG(loc = end -> (!(pOutOnOVSt & (StartI | FuStopI | TStopI | !EnRstartSt) & !PFsPosOn
& PEnRstart & !PRstartFS) | !OutOnOVSt | first))
Result: false
```
  - R3-6
 

```
Assertion (!(FuStopISt && !PFsPosOn && !POutOff && !PPulse) || !OutOnOV)
CTL: AG(loc = end -> (!(FuStopISt & !PFsPosOn & !POutOff & !PPulse) | !OutOnOV))
Result: true
```
  - R3-7
 

```
Assertion (!(pFuStopI && pMMoSt && !FuStopI && !Manreg01b[12] && MMoSt) || ((MOnRSt && PFsPosOn)
|| (!MOnRSt && !PFsPosOn)))
CTL: AG(loc = end -> (!(pFuStopI & pMMoSt & !FuStopI & !Manreg01b[12] & MMoSt) | ((MOnRSt
& PFsPosOn) | (!MOnRSt & !PFsPosOn)) | first))
Result: false
```
  - R4-1
 

```
Assertion (!(!HFOn || !HFOff) || (!OnSt || !OffSt))
CTL: AG(loc = end -> (!(!HFOn | !HFOff) | (!OnSt | !OffSt)))
Result: true
```