Espoo 2008 TKK-AS-R5

APPLYING MODEL CHECKING TO ANALYSING SAFETY INSTRUMENTED SYSTEMS

Matti Koskimies

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

Espoo 2008 TKK-ICS-R5

APPLYING MODEL CHECKING TO ANALYSING SAFETY INSTRUMENTED SYSTEMS

Matti Koskimies

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 451 1
Fax +358 9 451 3369
E-mail: series@ics.tkk.fi

ISBN 978-951-22-9477-0 (Print)
ISBN 978-951-22-9478-7 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://www.otalib.fi/tkk/edoc/

TKK ICS
Espoo 2008

Abstract: There is an ongoing change in the industry in which old analogue instrumentation and control (I&C) systems are replaced with new digital ones. New digital systems enable more complex control tasks and especially their application to safety instrumented systems (SIS) has created a need for new verification methods such as model checking.

Our goal is to study the applicability of model checking methods to a real safety instrumented system used in industry and to evaluate whether such a system can be modelled on a level which, on one hand, enables verification of relevant safety properties and, on the other hand, keeps the size of the model feasible. A central objective is also to create a general methodology for applying model checking to analysing safety instrumented systems.

As a case study we modelled an application of UTU Falcon arc protection system along with it's environment with NuSMV modelling language. Moreover, we used NuSMV to verify this model against the most relevant safety properties for the system.

Our results indicate that model checking seems to be a promising method for verification of safety instrumented systems.

Keywords: Model checking, safety instrumented systems

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

DCS Distributed Control System

FBD Function Block Diagram

I&C Instrumentation and Control

IL Instruction List

LD Ladder Diagram

NPP Nuclear Power Plant

(O)BDD (Ordered) Binary Decision Diagram

PLC Programmable Logic Controller

SAT Propositional Satisfiability

SFC Sequential Function Chart

SMV Simple Model Verifier

ST Structured Text

1 INTRODUCTION

Instrumentation and control (I&C) systems are an important part in the operation of nuclear power plants and many other industrial facilities. They are divided into basic process control systems (BPCS) and safety instrumented systems (SIS). The basic process control systems are related to the main functions of plants, e.g., to tasks related to energy production [14]. The safety instrumented systems are used to implement safety related functions such as emergency shutdown systems [14]. The nuclear power plants that are currently in active use in Finland have been built at the time when I&C systems were implemented by using analogue hardwired circuits and electromechanical relays. In many other countries the situation is the same. As these power plants are starting to be at the end of their life cycle, there is an ongoing process in which the plants are being renewed and modernised in order to extend their lifetime. An important part of this renewal is the replacement of the old analogue instrumentation and control (I&C) systems by new digitalised ones.

Traditionally, the verification of I&C systems (both analogue and digital ones) has been based on manual testing or simulation [32]. In manual testing an actual system is tested against human- or machine-generated test cases. In simulation a system is tested by simulating the behaviour of a model of the system with a simulator. However, as the implementation of I&C systems with digital programmable logic controllers (PLC) has enabled more and more complex control logic designs the traditional verification methods are becoming insufficient [32]. Especially, as new digital PLC based systems are used to replace old analogue safety instrumented system in highly safety critical domains such as nuclear power plants, new verification methods are needed.

Alternatives for new verification methods for I&C systems include formal verification methods [13]. Formal verification refers to the act of proving or disproving mathematically the correctness of design of a system. The benefit of this approach is that formal methods are at best fully automated and exhaustive, i.e., they analyse a given system with respect to its all possible behaviours. A promising formal method for the verification of I&C systems is model checking [9]. In model checking, the basic idea is to verify whether a model of a system fulfils the specification of the system. This is done by analysing all possible behaviours of the model. In practice, model checking is carried out by first modelling a given system by the input language of a model checker. Next the properties that the system is supposed to fulfil are specified by using a suitable specification language such as temporal logic. Finally, a model checker is used to verify that the model of the system fulfils the specified properties. The key difference compared to simulation method is that the verification is done against *all* possible executions of the system. Moreover, if a specified property is violated, model checker returns a counterexample of the violation, i.e., an execution of the model that violates the property. One of the key challenges for the model checking method is so-called *state explosion* problem [9] which, informally speaking, refers to the exponential growth of number of possible states of a system to the size of the system description. Thus, currently model checking is not a feasible verification method for arbitrary sized systems.

There has already been a notable number of studies on applying model checking to PLC based I&C systems [18] and this work is a continuation to that research area. Our goal is to study the applicability of model checking to safety instrumented systems by considering a real world case study. Moreover, we introduce a general methodology for applying model checking to similar systems as the case study that we have chosen. For the subject of research we have chosen UTU Falcon arc protection system. It can be considered as a typical industrial safety instrumented system implementing an emergency shutdown function. The purpose of the system is to cut down power feed from an electricity distribution system when an electric arc is detected.

I&C systems are a challenging target domain for model checking. This is because they typically form a closed control loop between their environment, i.e., the controller of an I&C system receives inputs through a feedback loop from its environment. Consequently, in order to verify properties of the system design one has to model also the physical environment to the appropriate extent. This can be difficult because the physical environment typically includes different kinds of continuous quantities and time delays. Especially in the case of safety instrumented systems, the most relevant properties to be verified are often dependent on different kinds of time delays. An option for dealing with these kinds of systems would be to use a real-time model checker which are specifically developed to handle continuous variables [4]. However, in case of safety instrumented systems, where the modelling of environment is often inevitable, the size of the model grows easily too large and model checking becomes infeasible. Therefore, our purpose is to study whether it is possible to model a SIS by using non-real-time model checker NuSMV so that the most relevant properties can be verified. An essential issue in this approach is to find a suitable level of abstraction for handling delays and timing issues.

1.1 Outline of the Report

This work is organised as follows. In Chapter 2 we discuss the existing research made in the field of applying model checking to automation systems. In Chapter 3 we give a concise overview on safety instrumented systems. Moreover, we present an abstract model for safety instrumented systems which captures the overall structure of the systems into which the modelling approach applied with the Falcon system is suitable. In Chapter 4 we describe the NuSMV model checker to the extent that is needed for the reader to be able to follow the rest of the work. Chapter 5 describes how the general parts of the abstract model of Chapter 3 can be modelled by using NuSMV. In Chapter 6 the actual case study is presented, and finally, Chapter 7 draws conclusions on the work.

2 MODELS AND ANALYSIS OF CONTROL SYSTEMS

2.1 Formal Methods for Digital Automation Systems

In this section we review existing work on applying formal methods to the analysis of digital automation systems. However, as the subject area is very wide, we had to make clear restrictions on the covered topics. Therefore we discuss only studies concerning automation systems based on programmable logic controllers (hereafter PLCs). Consequently, we have mostly excluded systems which are based, e.g., on softPLCs (PLCs based on standard PCs) and distributed control systems (DCSs). This restriction seems reasonable, since the conclusions made on PLCs can be extended to cover also softPLCs. On the other hand, covering also the subject of DCS would have required a survey of a much larger scale. However, the PLC domain can be seen as a logical stepping stone towards analysing of DCS based automation systems. On the method side we are focusing mainly on applying model checking methods to PLC applications.

The analysis of PLC based systems with formal methods can be done on different levels. In the most comprehensive approach the PLC program is verified against the specification of the entire system which consists of the combined specification of the controller and its environment. Another option is to restrict only to verifying the correctness of the controller. In this case the specification of the system as a whole is divided into the specifications of the environment and the controller part, and the program of the controller is analysed with respect to the specification of the controller.

Another classification of approaches on applying formal methods to PLC applications can be made based on the initial objective of the process. That is, the goal might be to analyse an existing application or to design a completely new one. In the first approach an existing PLC program is first transformed into some formal modelling language and then, based on the model, the validation of properties is carried out with a model checker. This approach is often referred to as modelling or formalising existing PLC programs [23]. The related studies are often — but not always — restricted to only validating the controller against its specification. In the second approach a system is designed from the beginning by using a formal modelling method. After the model (which in this case often consists of both, the model of the controller and the environment) is finished, it can be used, alongside of validating properties, to derive a PLC program automatically. This approach is usually referred to as program synthesis [13].

In this study we review studies on both approaches, the modelling of existing PLC programs and the program synthesis. We start by listing some earlier surveys made on the subject. From these we found especially the papers [23, 18] as a good starting point for our own review. After listing already existing surveys we proceed to present references to most relevant studies made on the field up to date. However, before going to the actual survey we give first an overview on PLCs in Section 2.2 and then describe some classification criteria for models of PLC programs in Section 2.3. We use the terminology of the classification framework while describing the references that we list in this work, though not all the referred studies are by no means classified according to all these criteria.

A different version of the survey presented in this work can be found from a technical report prepared for the MODSAFE project [37].

2.2 Overview on PLCs

Programmable Logic Controllers (PLCs) are self-contained microcomputers optimised for industrial control [32]. They were introduced in the 1970s as a replacement for control systems based on hardwired circuitry of electromechanical relays. A typical PLC hardware consists of a single microprocessor based CPU, a memory, and input/output-ports through which PLC is connected to sensors and actuators. Typical sources of input data might be e.g., light, current, or heat sensors where as actuators might be e.g., motors or valves. The key characteristic distinguishing PLCs from general microprocessor based systems is their cyclic operation mode. That is, PLC programs are always executed in a permanent loop. A single iteration of the loop is often referred to as a *scan cycle* and it consists of the following three phases: first the input values are read from the sensors, then the program computes a new internal state and output values, and finally the updated output values are passed to the actuators.

Initially, PLCs provided only very restricted functionality comparable to those tasks that could be achieved by using relays. However, since their first appearance to the market the functionality of PLCs has evolved to include many sophisticated features such as multi-tasking, interrupts, watchdogs, etc. Therefore, with respect to the hardware capabilities the difference between PLCs and ordinary PCs is diminishing all the time, and the most fundamental differences to general purpose programming systems lay in the operation mode and the areas of usage of PLCs.

However, with respect to the programming languages, there are huge differences between PLC based systems and general purpose programming systems. The following statements are characteristic to the variety of different PLC languages:

- PLC programming languages tend to be very low-level (this applies especially to languages most applied in the industry), and
- PLC languages have traditionally been vendor specific without conforming to any common standard.

Reasons for this situation are of historical nature. Initially as PLCs were introduced to the market their programming languages were designed to closely resemble the design of hardwired relay circuits. This was done to make the transition to PLC based systems as easy as possible for the control engineers in order to speed up the introduction of the new technique. On the other hand, because PLC based systems are typically designed for a specific industrial target domain, and moreover, the software has initially been only a small part of the whole PLC based system (especially of total design expenses), there has not been as strong demand for fast and constant development of the programming languages and tools as it has been in the case of the general purpose programmable systems [27]. Moreover, even though more evolved programming languages have appeared, the control engineering industry seems to have a tendency of sticking to the most traditional tools.

However, as systems keep getting larger and more complex, and the need for interconnecting industrial systems is increasing, IEC has come up with a common standard IEC 61131-3 for PLC languages. The standard introduces five different languages and it is intended that PLC manufacturers will gradually transform their programming languages to conform to one of languages of the standard. The IEC 61131-3 standard will be discussed more in Section 2.3.

2.3 Classification Criteria for PLC Models

Here we describe the three orthogonal criteria for classifying PLC models originally presented by Mader in [23]. The discussion is intentionally kept brief and an interested reader is advised to turn to [23] for more in-depth coverage on the issue.

Modelling of the Cyclic Operation Mode

The most fundamental characteristic of PLCs is their cyclic operation mode. Therefore the first logical choice in classifying PLC models can be made on the basis of how the scan cycle of the PLC is modelled. There are three possible choices: the scan cycle can be modelled either explicitly or implicitly, or then one can abstract entirely from the scan cycle.

In the explicit modelling of the scan cycle the exact duration of the cycle in actual time units is modelled. Instead, in the implicit modelling the existence of the cycle in itself is modelled, but the actual duration of it is not measured in time units, and moreover, it is considered to be constant. The third option is to abstract from the scan cycle entirely so that the time model of the PLC is considered to be continuous instead of discrete.

Modelling of Timers

The use of timers is a fundamental characteristic of PLC programs. However, not all of the PLC applications need timers and in many cases the use of them can be avoided, either by modelling techniques or by altering the system design. Therefore, it is justifiable that there are studies concerned with modelling of timers as well as those which abstract from them.

The Language Fragment Considered

By the language fragment criterion a choice is made on which parts of the PLC programming language are considered in the modelling process. The first obvious question is which of the five languages defined in the IEC 61131-3 standard is considered. The languages are: Instruction List (IL), Structured Text (ST), Ladder Diagrams (LDs), Function Block Diagrams (FBDs), and Sequential Function Charts (SFCs).

Moreover, usually only a restricted part of the features of the chosen target language is considered. This is because the IEC 61131-3 standard lacks the definitions of the formal semantics on the languages in standard. Therefore, from the viewpoint of academic research, it simply is not reasonable to consider all possible semantic interpretations of all parts of the languages.

Finally, the language fragment considered can also be restricted on the possible data types, i.e., which of the data types of booleans, integers and real numbers are allowed.

2.4 Survey of Studies on Model Checking PLCs

In this section a survey of studies on applying model checking to PLCs is presented. We start by presenting already existing surveys on the subject in Section 2.4.1 after which we go through studies on modelling of PLC programs in Section 2.4.2. Finally in Section 2.4.3 studies on PLC program synthesis are reviewed.

2.4.1 Previous Surveys

The paper [23] by Angelika Mader presents a classification of different PLC models. It first classifies an orthogonal set of criteria on which PLC models can be classified with. The paper also introduces an extensive list of publications which are classified against the presented criteria. The Doctoral Thesis of Ralf Huuck [18] presents a quite similar survey which includes also more recent studies and a bit extended list of the classification criteria.

The paper [24] also by Mader discusses on a general level applying formal methods to PLC applications. It presents a schema on the structure of a general PLC application and based on this framework analyses the possibilities for applying different formal methods on PLCs.

The paper [43] by Frey introduces four criteria on which studies considering formalisation of existing PLC programs can be categorised. It also presents references to studies falling in each of these categories. The study is by no means as thorough as the survey presented in the papers [23, 18] but contains some additional references and shows another way to classify the existing research.

The paper [13] by Frey presents a general framework on the different phases of verification and validation and discusses what formal methods can be used in these different phases. Therefore, it is not focused only on transforming existing PLC programs to models and it also discusses other formal methods than just model checking.

2.4.2 Modelling PLC Applications

In the following we present studies considering the modelling of PLC applications. The references are organised according to the different PLC programming languages.

Research on modelling SFC programs The Doctoral Thesis of Ralf Huuck [18] shows how Sequential Function Charts (SFC) programs can be given formal semantics, including a translation of the untimed semantics of

PLCs to the Cadence SMV model checker input language. In the paper [3] this research is extended by giving formal semantics also for timed SFCs and in the paper [2] it is shown how timed SFCs can be translated into timed automata. The latter study illustrates in both, timed and untimed cases, the complete verification procedure from model transformation to identifying errors with model checking tools UPPAAL and Cadence SMV.

Research on modelling IL programs The widely referred paper [26] by Mader and Wupper shows how a fragment of Instruction List (IL) programs can be translated into timed automata. Based on this study a tool is presented in the paper [39] by Willems which translates IL programs automatically into the timed automata format accepted by UPPAAL model checking tool. This toolchain allows model checking of real time properties with explicit modelling of the scan cycle. The Willems's tool also allows IL programs to contain bounded integer variables. Moreover, UPPAAL tool can be used to model the environment of the PLC as well. In an unpublished paper [22] Mader presents two examples on modelling IL programs and performing their verification with the UPPAAL tool.

In the paper [16] it is shown how IL programs can be transformed into Petri nets. The method allows usage of data structures up to length of 8-bits and it takes into account all standard instructions excluding commands from libraries. However, the real-time aspects are not modelled.

In the paper [15] it is shown how IL programs can be transformed into Timed Net Condition/Event systems. The scan cycle is modelled explicitly and timers are taken into account at some level. However, the possible data structures are restricted to boolean values and only **load**, **store**, **and**, and **or** instructions are considered.

The paper [6] deals with the same fragment of IL programs as the paper [15] described above. In addition, the loop operations are considered. However, [6] is concerned with translating IL programs directly to the input language of the SMV model checker. As SMV cannot directly handle real-time issues, these aspects of the IL program are not analysed.

The paper [21] uses BDD and BMC based symbolic model checkers to model check two small PLC based automation systems written in IL.

Research on modelling LD programs In paper [28] Ladder Diagrams (LD) programs are modelled with the SMV model checker without taking timing aspects into account. Based on this study there exists a research paper [34] presenting two comprehensive case studies on existing chemical processing systems. In these case studies also the model of the environment is presented. The verification process revealed numerous faults and the results could be used to improve the designs.

In the paper [31] a large fragment of LD programs are modelled with the SMV model checker. The scan cycle is modelled implicitly and it is shown how a particular type of timers can be modelled in non-real time manner so that certain liveness and safety properties can still be verified.

Research on modelling ST programs The paper [19] by Jiménez-Fraustro and Rutten considers of modelling a fragment of the Structured Text (ST) language with the synchronous language SIGNAL. The fragment includes at least assignments, conditionals and bounded loops. Scan cycles are modelled implicitly and real-time behaviour is not considered. The follow up study [20] considers also the FBD language. Unfortunately, there does not seem to exist any related studies on actual model checking based on the SIGNAL model.

2.4.3 Methods for Synthesising PLC Programs from Models

In the following we present studies considering the PLC program synthesis.

In the paper [10] Henning Dierks presents a new modelling formalism named PLC-automata especially designed for modelling PLC applications. PLC-automata allows explicit modelling of the scan cycle but it is possible to model only particular type of timers in which an input signal is ignored for a certain time. Dierks shows also how PLC-automata models can be transformed automatically into language of Structured Text.

In the paper [11] it is shown how PLC-automata models can be transformed into timed automata models which makes it possible to perform model checking with a real-time model checker such as KRONOS or UPPAAL. Moreover, in the paper [33] a tool MOBY/PLC is presented which can be used to modelling PLC-automata, validation, and code generation.

As a continuation for the research based on PLC-automata formalism Olderog presents in paper [29] an approach for designing valid PLC applications. His method is based on formulating design specifications with PLC-automata and specifying requirements in Constraint Diagrams. Olderog also presents a case study from industry for which he applies his approach. In the paper [30] Dierks and Olderog present a tool Moby/RT which is based on the design approach of [29].

A quite interesting research project is reported in the papers [25, 5] by Mader, Brinksma et al. In [25] a systematic design and validation of a PLC control program for a batch plant by using formal methods is reported. This plant was selected as a case study for the EC project on Verification of Hybrid Systems (VHS). In follow up paper [5] it is reported how the SPIN model checker was used for both the verification of a process control program and the derivation of optimal control schedules.

In the paper [41] a Korean research group presents a PLC-based safety critical software development technique for NPP domain. In the method a formal language NuSCR is used for writing software requirement specifications. After a requirement specification is written in NuSCR it can be transformed mechanically into a FBD program (the synthesis phase). The benefit of the method is that NuSCR specifications can be automatically analysed for completeness, consistency, and against the properties specified in temporal logic. Moreover, NuSCR specification language was developed during the research project together with NPP domain experts and it was designed to especially suit the needs of nuclear engineering domain. NuSCR specification language is explained more closely in the paper [42] and the paper [40] describes more thoroughly the process of synthesising FBD programs from NuSCR specifications. The paper [8] presents NuEditor tool which can be used for creating NuSCR specifications and performing consistency and completeness analysis on the specifications. NuEditor can also be used to translate a NuSCR specification into SMV input so that model checking of safety and liveness properties can be performed on the specification. However, NuEditor cannot be used for synthesising FBD programs from specifications and, apparently, for this task there does not yet seem to exist any tool support. The paper [8] presents also a case study in which a reactor protection system for Korean nuclear power plant was specified with NuEditor and verified against safety and liveness properties with SMV model checker.

2.5 Overall Status of the Research on Model Checking PLCs

On the study field of modelling PLC programs most research papers seem to be available on the language of Instruction List. On Ladder Diagram programs there is also quite a lot research but, still, on this area there seems to be many gaps to be filled, especially on covering timers and real time issues. Instead, in the case of the SFC programs the situation is quite opposite: there hasn't been that many research projects but the existing ones are actually quite covering.

Considering the last two languages of the IEC 61131-3 standard, i.e., on Structured Text and Function Block Diagrams, there seems to exist only very few studies. We presume that the reason for this in the case of the ST language might be that the relevant problems on the modelling issues are present also in the IL language which in its brevity suits better academic research. Instead, in the case of the FBD language, the similar but more evolved SFC language provides features not existing in FBD, and therefore, might be more appealing target for the research.

For the synthesising approach there has also clearly been quite a lot of research activity. The usefulness of this type of an approach has especially been shown in the Korean research project which was described above. This project has proven that the synthesis approach can be successfully applied in producing real application programs in the NPP I&C domain. Moreover, the project was also successful in producing a formal specification language which was readily accepted by domain experts.

3 SAFETY INSTRUMENTED SYSTEMS

3.1 Overview on Safety Instrumented Systems

Safety Instrumented System (SIS) is used to implement safety related functions on industrial systems or processes [14]. They consist of sensors, logic solvers, and final elements. Typical uses for a SIS are shutdown functions and permissive functions. In shutdown functions a protected system or a process is taken to a safe state if a hazardous event or condition is met. In case of a permissive function the purpose is the opposite: a process is permitted to move forward when specified conditions are met.

Regardless of the specific safety function implemented, a SIS operates by monitoring process variables of the protected system and it only initiates action if the variables reach specified threshold values. If the threshold values are reached, SIS performs the specified safety function by operating final elements such as switches, valves, or breakers of the protected system. It should be noted here that the final elements of a SIS are typically physical parts of the protected system, i.e., terminologically they belong to both of them.

3.2 An Abstract Model of Safety Instrumented Systems

In this section we present an abstract model of safety instrumented systems which characterises the systems which our model checking method can be applied to. Further in Section 5 we present the general overview of our model checking method with the aid of this model.

Our model is very general in a sense that it doesn't make any restrictions nor assumptions on the protected system or the specific safety function implemented by SIS. However, as our model does not allow the logic solver to be arbitrary, it cannot be seen as a generic model of all possible SISs.

The abstract model is shown in Figure 1 and described in the following.

Overall description of the model

The model describes a SIS with its environment and the closed control loop between them. Model is divided into two parts, into Controller and System environment. Controller models only the logic solver of a SIS. The model assumes that the logic solver is based on a controller with cyclic operation mode and constant length scan cycle (see Section 2.2 for description of scan cycle). System environment is an abstraction of the sensors and the final elements of a SIS as well as the protected system. It should include all the parts of the physical environment which might affect the operation of the SIS. The arrows from System environment to Controller and Controller to System environment represent the signals from sensors to the logic solver of a SIS and the signals from logic solver to the final elements of a SIS, respectively. The values of the signals are binary, i.e., at any given moment the value of each signal is either 0 or 1. The control loop is closed as Controller affects the state of the protected system by controlling the final elements and the sensor data received by Controller depends on the state of the protected system.

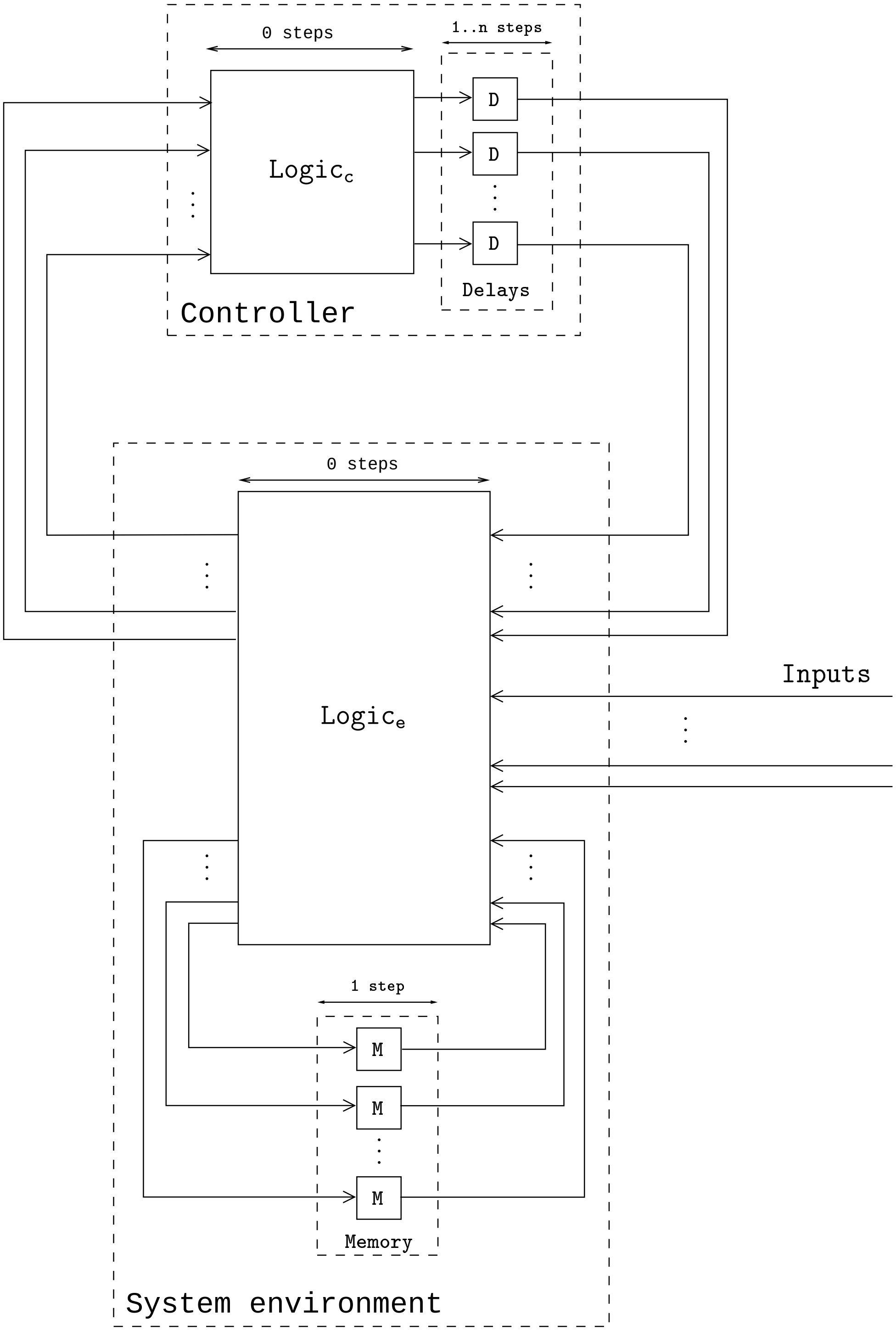


Figure 1: An abstract model of a SIS

The model abstracts physical devices with both discrete and continuous behaviour. However, as the operation mode of the logic solver of the SIS is assumed to be cyclic with constant scan cycle, it interacts with the environment only on discrete instants. Therefore, the transient states of the system, that is, the states between starting and ending points of a scan cycle, are irrelevant with respect to the control logic of the logic solver. For this reason the time model of the whole system can be considered discrete so that the values of signals and state variables are updated only on discrete time steps. The unit time step of the model represents a single scan cycle of the logic solver of the SIS.

Description of Controller

Controller abstracts logic solver of a SIS into Logic_c and Delays parts. Logic_c part represents the control logic of the modelled SIS, i.e., it specifies the logical rules on how the SIS reacts to each combination of input signals. In the model Logic_c part is modelled as a logical function which calculates outputs as a function of input values instantaneously without any time delay.

Delays of Controller represent a specific type of delays commonly used with SISs. The delay gates of Delays part are associated with a delay length parameter (number of time steps) and they operate in such a way that a delay gate only gives an output signal 1 if it has received a non-stop input signal for a time period specified with the parameter of that delay. Otherwise the output signal is 0. These kinds of delays can be used with SISs to eliminate too hasty reactions by ensuring that the alarm condition remains for a certain time before an action is taken. In our model the scan cycle of the controller of a SIS is also included into Delays. For this reason it takes always at least one time step for a signal to go through Delays part.

We define the state of Controller as the set of the input signals of Controller, the input signals of Delays (obtained as the results of Logic_c acting on input values), the internal state of the delay gates of Delays part, and the outputs of Controller. The internal state of a single delay gate can be seen as an integer number in the value range $0..delay$ where the $delay$ stands for the delay parameter of the gate. The purpose of the states of the delay gates is to keep track of the number of consecutive 1-valued signals up to number of scan cycles specified by the delay parameter.

Description of System environment

System environment abstracts the physical elements it represents into Logic_e, Inputs, and Memory parts. Inputs represent events which affect the system but whose cause or origin cannot be modelled. Most importantly this means events which are caused by emergency situations and perceived by sensors. Inputs can also be used to represent malfunctions in physical devices, for instance. As it is with the input and output signals of Controller, Inputs are expected to be binary valued.

Logic_e part represents the operational model of the physical devices included in System environment and the interaction between them. Basically Logic_e specifies how the control signals of Controller and Inputs of System environment affect the input signals of Controller. Part of the interactions between physical devices can be seen as instantaneous. For instance, depending on the situation an electricity feed might be considered to be cut down instantaneously at the moment it is switched off. On the other hand, some interactions might not be instantaneous but happen with a delay. For instance, for a valve it might take a considerable amount of time to close a pipe after it has received a launch signal. Moreover, effects of an interaction (either delayed or instantaneous) might also depend on the current state of System environment and/or they might change the state of System environment. For these reasons System environment contains Memory part whose memory elements can be used to store such variables whose values in a given time step are determined in the previous time step. We define the state of System environment in a given time step as a set of values which includes the predetermined variables stored in Memory, Inputs of System environment, the output signals from Controller, and the effects of instantaneous interactions determined by Logic$_e$ part.

System environment works so that on each time step Logic$_e$ part counts the effects of interactions as a logical function of the output signals of Controller, Input of System environment, and the outputs from Memory part. That is, the model is designed so that regardless of the type of the interaction (instantaneous or delayed) the result is calculated immediately without a time delay. Therefore, the effects of the instantaneous interactions happening in a certain time step can be seen at the same time step. However, in case of the delayed interactions, the result is not shown immediately but it is stored in the memory element, which shows the result only in the following time step. If an interaction happens with longer delay than one time step, it can be modelled simply by iteratively storing the result into a memory element until the corresponding time delay has elapsed.

3.3 Classification of the Abstract SIS Model

In Section 2.3 we presented a coarse framework with three orthogonal criteria for classification of PLC models. The first criterion specifies how the scan cycle of a PLC is modelled. In our case the modelling is done implicitly, i.e., the existence of scan cycle is modelled but its length is assumed to be constant. Moreover, the time length of the scan cycle is not modelled but it is taken to be the unit time step of the SIS model.

The second criterion simply divides PLC models to two classes based on whether they model timers or do not. Thus, our model falls into the class in which timers are modelled.

Finally, the third criterion makes distinction on the models on the basis of which programming language of the IEC 61131-3 standard is used. With this respect our model cannot be classified as it describes SIS on an abstract level. On behalf of our model, any of the five languages of the IEC 61131-3 standard could be used to implement the control logic of the logic solver of a SIS. However, the language used in the case study of Section 6 resembles closest to a very limited subset of the Function Block Diagrams language.

4 MODELLING AND ANALYSING SYSTEMS WITH NUSMV

In this section we describe the NuSMV model checker [1] which was used in the Falcon case study. Section 4.1 gives a general overview of NuSMV, Section 4.2 describes how models are build with the input language of NuSMV, and Section 4.3 describes how verified properties are specified with the input language of NuSMV. The discussion on the syntax and semantics of the input language of NuSMV covers only the parts of the language which are used in this study. For further information we advise the reader to see the NuSMV user manual [7].

4.1 General Overview

NuSMV is a symbolic model checker developed by ITC-IRST. NuSMV can be used to describe finite state systems that range from completely synchronous to completely asynchronous. The main reason for choosing NuSMV was that it is a state-of-the-art model checker which has proven to be capable of handling industrial-sized systems. Moreover, NuSMV supports both BDD (Binary Decision Diagram) and SAT[1] (propositional satisfiability) based model checking which are currently the main approaches in implementing symbolic model checking tools. Being distributed under an OpenSource licence, NuSMV also offers a promising platform for research purposes.

4.2 Modelling with NuSMV

4.2.1 General Structure of NuSMV Models

NuSMV models (also referred to as NuSMV programs) consist of one or more *module declarations*. A module declaration is an encapsulated collection of declarations, constraints, and specifications. Intuitively, the idea of the module concept is to encapsulate closely related state variables together in order clarify the structure of the whole model. Modules are used in such a way that a module declaration is used as a variable type to create *module instances*. Therefore, multiple realisations of a module can be created based on a single module declaration. A module declaration may contain instances of other modules so that the modules form a hierarchical structure. Each NuSMV model is built on a declaration of a special module which has to be named as `main`.

Next we describe the basic constructs needed for creating module declarations. The description is based on the NuSMV model shown in Figure 2, which has two module declarations. The model is complete and it introduces all structures used in our actual case study.

4.2.2 Structure of a Module Declaration

A description of a NuSMV module consists of several different segments containing different kinds of declarations, specifications, and constraints. In this

[1] Some of the SAT based model checking algorithms inside NuSMV have been developed at TKK/ICS [17].

```
MODULE exampleModule(param1,param2)
VAR
     var1 : boolean;
     var2 : -1 .. 10;
     var3 : -1 .. 2;
ASSIGN
     -- An example of direct assignment.
     var1 := (var2 > 1);

     -- An example of assignment with init/next-construct.
     init(var2) := 0;
     next(var2) := param1 + param2;

     -- An example of assignment with init/next-construct
     -- with a case expression.
     init(var3) := -1;
     next(var3) :=
          case
               (var3 < 2) : var3 + 1;
               (var3 = 2) : {0,2};
          esac;

MODULE main()
VAR
     moduleInstance : exampleModule(definedConstant, 5);
DEFINE
     definedConstant := 1;

---------------------------------
-- Specification of properties

LTLSPEC G (moduleInstance.var1 -> O (moduleInstance.var2 = 10))
LTLSPEC F (moduleInstance.var2 > moduleInstance.var3)
```

Figure 2: An example of a NuSMV model

case study, only the most central constructs were needed. These include the parameters of modules, the declaration and assignment of state variables, and define declarations. These are described in the following.

Parameters of a module. Parameters are defined as a list of identifiers which can be used for passing data to a module from other modules. The parameters of a module are specified with a parenthesised list of identifiers following the name of a module (see `param1` and `param2` in the example above). The `main` module is not allowed to have parameters.

State variables of a module. The state variables of a module are listed in a segment identified with the keyword `VAR`. A state variable declaration consists of an identifier which can be used to refer to the variable and a type specification which describes the data type and the range of possible values of the variable. As data types one can use either built-in data types or module declarations.

In our case study, only two built-in data types, boolean and integer, are used. The boolean data type comprises two integer values, 0 and 1 (or their symbolic counterparts `false` and `true` respectively.) The value range of the integer type consists of integer values from -2^{31} to $2^{31} - 1$. The integer type is specified by declaring a value range after the variable identifier (see declaration of `var2` in the `exampleModule`.)

If a module declaration is used as a data type in a variable declaration, the variable is said to be an instance of the module, and the variable declaration is said to be a module instantiation. The declaration is formed simply by referring to the module name (followed by a list of parameters) in the place of the variable type (see variable `moduleInstance` in the `main` module in Figure 2.)

Assignment of state variables. State variables are assigned in a segment identified with the keyword `ASSIGN`. A state variable can be assigned in two distinct ways, either directly or with an `init`/`next` construct. The variable `var1` in the `exampleModule` in Figure 2 shows an example of direct assignment. In this case, the value of the current value of the `var1` is set to the value of the expression (`var2` > `1`) by using the current value of the `var2` at all time steps.

In the case of the variable `var2` in the `exampleModule`, the assignment is done using the `init`/`next` construct. In this case, the assignment is done in two steps: first the *initial* value (i.e., the value of the state variable at the first time step, or in the initial state) of `var2` is set to zero. On the following line, it is stated that the value of `var2` in the *next* state will be the value of `param1` + `param2`.

The variable `var3` in the `exampleModule` is also assigned with an `init`/`next` construct but in its `next`-expression another two important constructs related to assignments are shown: the *case expression* and the *set expression*. The segment surrounded by keywords `case` and `esac` defines a `case` expression. It can be used to express how the value assigned to a state variable depends on the condition of other state variables. Each line of the case segment has on its left-hand side a boolean valued condition statement and on its right-hand side a value which is assigned to the state variable if the condition holds. The lines are evaluated sequentially one-by-one starting from the first line until the first line whose condition part is non-zero is reached. NuSMV requires the conditions of a case expression to be exhaustive, that is, there has to be always at least one line whose condition part evaluates to a non-zero value.

In the case of `var3`, the case statement increases its value in the next state by one if the current value is below the value 2 (which is the maximum value it can have). If the current value of `var3` is 2, its value in the next state is chosen randomly from the set expression $\{0, 2\}$. Set expression lists the possible values one by one, so in this case the possible values are 0 and 2. The purpose of the set expressions is discussed in Section 4.2.3.

DEFINE declarations. Define declarations are yet another basic construct used to build modules. Define declarations are added in a module declaration after the keyword `DEFINE` and they are used to associate a common expression with a symbol. That is, the define declarations are used to define shorthands for complex expressions or numeric values in order to make module descriptions more concise. The `defineConstant` of the `main` module in Figure 2 shows an example of define declaration in which the numeric value 1 is associated with the identifier in question.

4.2.3 Semantics of NuSMV Models

When restricted to the features of NuSMV that are used in this study, NuSMV models represent a finite state machine (FSM), i.e., a structure with a finite set of states, a total transition relation between the states, and a set of possible initial states. A *state* of a NuSMV model is an assignment of values to the state variables of the model. Thereby, the number of different potentially reachable states specified by a NuSMV model equals the number of the possible value assignments of the state variables of the model. The set of all states of a NuSMV model is referred to as the state space of the model. The updating of the values of the state variables of a NuSMV model according to their assignment specifications is referred to as a *transition* from a state to another. The possible transitions of the FSM can be specified by the transition relation that relates a set of states to each state. By an initial state we mean the first assignment of values to the state variables of the model when the model checking process is started. NuSMV allows defining different possible initial states by using the set expressions described in Section 4.2.2. If the model allows more than one possible initial state, we say that the model has a *nondeterministic* initial state or that the initial state of the model is chosen *nondeterministically*.

A finite state machine can be represented graphically with a state transition diagram. In Figure 3, a state transition diagram of the example NuSMV model of Figure 2 is shown. The boxes in the figure represent the possible states and the arrows between boxes represent the possible transitions between states. If a state belongs to the set of initial states, it is depicted with a double-stroke box.

For model checking, the possible executions of the finite state machine specified by a NuSMV model are of central interest. By an execution we mean an infinite path $\pi = s_0\,s_1\ldots$ of states where the states of the path belong to the state space of the NuSMV model, the initial state of the path belongs to the set of possible initial states of the model, and the transitions between consecutive states on the path are allowed by the transition relation of the model.

A NuSMV model allows more than one kind of an execution if the initial state of the model is nondeterministic or if the transition relation of the model is nondeterministic. The transition relation is nondeterministic if it relates a set of states with more than one state to at least one state of the model. In other words, the transition relation of a model is nondeterministic if it allows more than one transition from at least one of the states of the model. In our example model, the transition relation is nondeterministic as it allows transition from state S3 to S1 as well as a transition from the state S3 to the state S3 itself.

A state of a NuSMV model is *reachable* if the state belongs to any of the possible executions of the model. The state space of our example model consists of 96 states in total but from these only four are reachable. There might also be legal transitions starting from an unreachable state and ending to any of the states of the model but as long as an execution starts from one of the initial states of the system, such transitions cannot ever be taken.

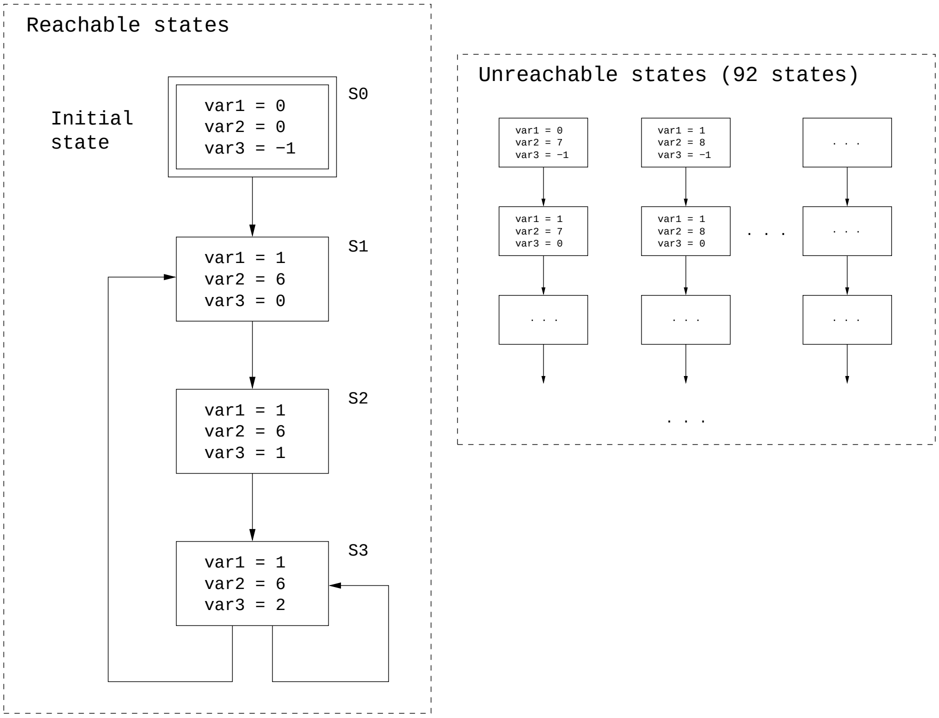


Figure 3: State diagram of the running example

4.3 Specifying Properties with NuSMV

The properties of this study are specified by using Linear Temporal Logic extended with past operators (hereafter PLTL). Also invariant specifications are used, but they can be formulated in PLTL as well. In this section we describe the syntax of PLTL in NuSMV.

In NuSMV, PLTL formulas are used to specify conditions or relations between *atomic propositions* of a NuSMV model. Atomic propositions are boolean valued expressions formed by connecting state variables with comparison operators $\{<, >, \leq, \geq, =\}$. An atomic proposition consisting of a single non-boolean variable is interpreted as a boolean variable so that the value 0 is interpreted as the boolean value `false` and all non zero values are interpreted as the boolean value `true`. PLTL formulas are formed by connecting atomic propositions with boolean operators and special *temporal* operators which can be used to specify time related statements. In this context, time is interpreted as the indexing of states of an execution. That is, the Nth state of an execution is considered as the state of the system in the *time step* N. Consequently, temporal operators are used, strictly speaking, to specify properties related to the executions of a NuSMV model.

The following list contains the logical operators used in this study[2] in the

[2] More extensive coverage of PLTL with past operators can be found in the NuSMV user

syntax of NuSMV and describes their semantics informally. In the following section, it is shown how the semantics of these operators is defined formally.

Boolean operators in the NuSMV syntax:

!x (**logical not**): !x is true if x is not true.

x & y (**logical** AND): x & y is true if x is true and y is true.

x | y (**logical** OR): x | y is true if x is true or y is true.

→ (**implication**): x → y is true if y is true whenever x is true.

↔ (**equivalence**): x ↔ y is true if the values of x and y are equal.

Temporal operators in the NuSMV syntax:

G (globally): G f is true if f is true at all time steps.

F (finally): F f is true at this time step if f will be true at some time step in the future.

O (once): O f is true if f is true at this time step or has been true at some previous time step.

Y (previous state): Y f is true if f was true at previous time step.

In the example model of Figure 2, two examples of PLTL property specifications are shown. The first property states that "in all time steps it holds that if the value of `var1` of `moduleInstance` is `true`, then there has to be a time step in the past in which the value of `var2` of `moduleInstance` was 10". The second property states that "there has to be some time step in which it holds that the value of `var2` of `moduleInstance` is bigger than the value of `var3` of `moduleInstance`."

4.3.1 Semantics of PLTL formulas

In the following, the formal semantics of PLTL formulas is given. The semantics is defined along the executions of a NuSMV model. Each state s_i of an execution $\pi = s_0\, s_1 \ldots$ is associated with a set of atomic propositions which evaluate to the value `true` in that state. This is expressed with a labelling function $L(s) \in 2^{AP}$ where AP is the set of all atomic propositions of the NuSMV model and s is a state of the NuSMV model.

Definition 1 (PLTL semantics.)**.** Let π be an execution of a NuSMV model and h, f and g be PLTL formulas so that f and g are subformulas of h. Moreover, let $\pi(i)$ denote the state s_i along a path π. Then $\pi \models h$ (h is valid along

manual [7] or in the paper [17] by Heljanko, Junttila, and Latvala.

π) iff $\pi^0 \models h$, where:

$$\begin{array}{ll}
\pi^i \models p & \textit{iff } p \in L(\pi(i)) \text{ for } p \in AP. \\
\pi^i \models \neg f & \textit{iff } \pi^i \not\models f. \\
\pi^i \models f \vee g & \textit{iff } \pi^i \models f \text{ or } \pi^i \models g. \\
\pi^i \models f \wedge g & \textit{iff } \pi^i \models f \text{ and } \pi^i \models g. \\
\pi^i \models \mathbf{G}f & \textit{iff } \pi^j \models f \text{ for all } j \geq i. \\
\pi^i \models \mathbf{F}f & \textit{iff } \pi^j \models f \text{ for some } j \geq i. \\
\pi^i \models \mathbf{O}f & \textit{iff } \pi^j \models f \text{ for some } 0 \leq j \leq i. \\
\pi^i \models \mathbf{Y}f & \textit{iff } i > 0 \text{ and } \pi^{i-1} \models f.
\end{array}$$

The definition above describes the meaning of a PLTL formula with respect to a single execution of a NuSMV model. In contrast, the following definition describes the meaning of a PLTL formula in the context of a whole NuSMV model.

Definition 2 (Validity of a PLTL formula.)**.** A PLTL formula f is valid in a NuSMV model M (denoted as $M \models f$) iff $\pi \models f$ for all executions π in M.

With these definitions, the model checking with NuSMV can be described as the process of deciding whether given PLTL formulas are valid in a given NuSMV model. Moreover, an essential function of the NuSMV model checker is to return a *counter example* for each PLTL formula that is not valid. A counter example for a PLTL formula is an execution of a NuSMV model along which the formula is not valid.

5 MODELLING SAFETY INSTRUMENTED SYSTEMS WITH NUSMV

In this chapter we give general guidelines on how a safety instrumented system compatible with the abstract model of Section 3.2 can be modelled using the NuSMV modelling language. Section 5.1 covers the Controller and Section 5.2 covers the System environment of the abstract model.

5.1 Modelling the Controller

In NuSMV, Controller of the abstract model (see Figure 1) can be modelled by defining two modules: one encodes Delays part and the other one encodes the whole Controller by using instances of the Delay module as building blocks. In the following we describe the overall structure of these modules.

5.1.1 Implementation of the Delay module

Figure 5 shows how the Delay module can be implemented. The module has two parameters: boolean valued input signal (named as `input_signal`) and a delay value (named as `delay`) whose type is a non-negative integer (it should be noted here that NuSMV does not allow explicit type declarations for module parameters but type checking is carried out implicitly). The module has a boolean valued variable named as `output` which represents the output signal of the abstract delay.

The operation logic of the module can be stated as follows. On a given time step t_n, the value of the `output` variable is set to 1 if and only if the value of the `input_signal` has been 1 on the previous consecutive time steps $t_{n-(delay+1)}, \ldots, t_{n-1}$ where $delay$ is the value of the `delay` parameter. That is, the value of the `input_signal` parameter in time step t_n does not affect the value of the `output` in the time step t_n. Moreover, if the value of the `delay` parameter is 0, the value of the `output` variable depends only on the variable of the `input_signal` parameter in the previous time step t_{n-1}. The operation logic of the Delay module is illustrated in Figure 4.

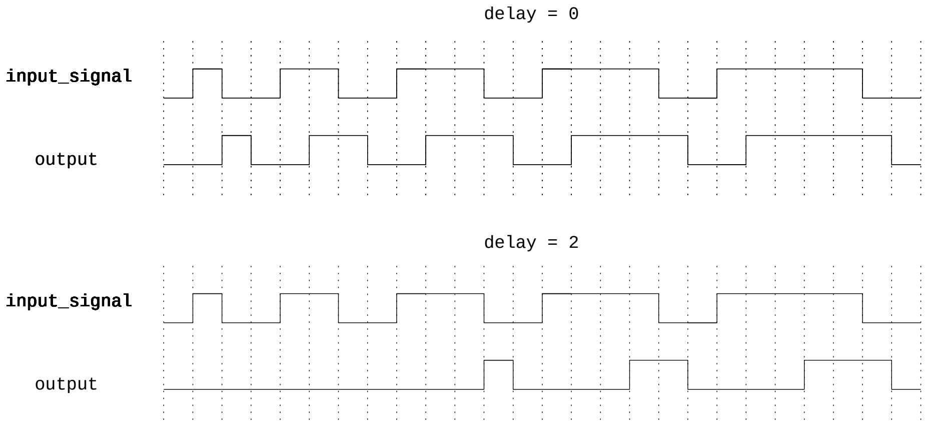


Figure 4: Input/output behaviour of the Delay module

The operation logic of the Delay module is implemented by using an integer valued counter variable (named as `count`) whose values may vary between 0 and the value of the sum delay + 1. The value of the count is set to 0 whenever the input_signal parameter is 0. If the value of the input_signal parameter is 1, the value of the count is increased until it reaches the value of the sum delay + 1 (here the sum delay + 1 is used instead of delay because the total delay corresponds to the delay caused by the delay gate plus the unit delay caused by the scan cycle of the Controller, see Section 3.2.) After this the value of the count remains unchanged until the input_signal value 0 is received.

The value assignment of the output variable is specified so that it will be 1 in a given time step if (and only if) the count has reached the value of the delay parameter in that time step (this is achieved by referring to the value next(count) instead of count in the assignment specification of the output variable.) Otherwise the value of the output is set to the value 0.

```
MODULE Delay(input_signal, delay)
VAR
  count : 0..DELAY_RANGE_UPPER_LIMIT;
  output : boolean;

DEFINE
  -- Total delay consist of the delay + scan cycle
  total_delay := delay + 1;

ASSIGN
  init(count) := 0;
  next(count) :=
    case
      input_signal = 0        : 0;
      count >= total_delay    : count;
      1                       : count + 1;
    esac;

  init(output) := 0;
  next(output) :=
    case
      -- At the step when count = delay, output has to be 1.
      next(count) >= total_delay   : 1;
      1                            : 0;
    esac;
```

Figure 5: Implementation of the Delay module

5.1.2 Implementation of the Controller module

Figure 6 shows an outline for the implementation of the Controller module which corresponds to the Controller part of the abstract model (referred to as the abstract Controller from here on). The Controller module has a boolean valued parameter for each input signal i of the abstract Controller (named as input_i). For each output signal j of the abstract Controller the Controller module has two define declarations named as logic_output_j and controller_output_j, and an instance of the Delay module named as delay_gate_j. Each define declaration logic_output_j is defined as a function of the parameters of the Controller module. Thus, these functions encode the Logic_{c} part of the abstract model. Each define declaration `logic_output_j` is passed as an input signal parameter to the Delay module instance `delay_gate_j` (see parameter `input_signal` in the implementation of the Delay module). Finally, each define declaration `controller_output_j` is set to the value of the `output` variable of the Delay module instance `delay_gate_j`. Consequently, the declaration `controller_output_j` corresponds to the output value j of the Controller.

The delay parameter values of the Delay module instances (see e.g., `delay_param_1` in Figure 6) are set to values $\lceil D/t \rceil$ where D is the delay in milliseconds of the corresponding delay gate in the modelled system and t is the length (also measured in milliseconds) of the scan cycle of the controller of the modelled system.

```
MODULE Controller(input_1, ... , input_i, ...)
VAR
  delay_gate_1 : Delay(logic_output_1, delay_param_1);
      ...
  delay_gate_j : Delay(logic_output_j, delay_param_j);
      ...

DEFINE
  -- Logic of the Controller is encoded in these define declarations:
  logic_output_1 := input_1 & input_2;
      ...
  logic_output_j := input_1 | input_i;
      ...

  -- Outputs of the Controller module.
  controller_output_1 := delay_gate_1.output;
      ...
  controller_output_j := delay_gate_j.output;
      ...
```

Figure 6: An outline for the implementation of the Controller module

5.2 Modelling the Environment

As it was described in Section 3.2, the purpose of the System environment of the abstract model (see Figure 1) is to capture all the relevant parts of the physical environment of a SIS and the interactions between them. In the abstract model the logical rules of the interactions are represented with Logic_{e} part. In NuSMV, Logic_{e} is implemented by specifying logical functions on the variables of the NuSMV model. As specified by the abstract model, the results of the instantaneous interactions are shown immediately at the same time step that they are calculated. This can be implemented in NuSMV by assigning the results of the corresponding functions to define declarations. For an example see the define declaration `electricity_feed_off` in Figure 7.

On the other hand, the abstract model specifies that in the case of the delayed interactions the results of the interactions are stored in the Memory and they are shown only after one scan cycle. In NuSMV this can be implemented by assigning the results of the corresponding functions to state variables by using the init/next-construct. For an example see the variable circuit_breaker1_cuts in Figure 7.

```
VAR
   water_boiler_is_empty : boolean;
   -- Timer for water_boiler_empty variable
   -- (using delay of 5 time steps.):
   timer : Timer(valve1_closed & valve2_closed, 5);
   circuit_breaker1_cuts : boolean;
     ...

DEFINE
   electricity_feed_off :=
              circuit_breaker1_cuts | circuit_breaker2_cuts;

ASSIGN
   init(circuit_breaker1_cuts) := 0
   next(circuit_breaker1_cuts) :=
     case
       (circuit_breaker1_launched = 1) : 1;
       (circuit_breaker1_launched = 0) : 0;
     esac;

   water_boiler_empty := timer.output;
```

Figure 7: Implementation of Logic$_e$ and Memory parts of the abstract SIS model

Often the delay of an interaction might be longer than one scan cycle. Longer delays can be handled by implementing a certain type of a timer module which, informally, holds a given value for the number of scan cycles specified with a parameter value. In practice, instead of assigning the result of a function implementing the logic of an interaction to a state variable, the result is passed to an instance of a timer module which, informally, holds the value for the specified time before making it available. This is demonstrated with the variable water_boiler_empty in Figure 7. In the end of this section we show how to implement this kind of a timer module.

The inputs of the system can be fully independent of the system or they can depend either on the current state of the system (e.g., a short-circuit cannot occur if the electricity feed is down) or the previous state of the system (e.g., a device cannot break down if it is already broken.) Either way, in NuSMV an unconstrained input can be implemented by assigning a set expression $\{0, 1\}$ to a boolean state variable. The value 0 of the set expression at a time step implies that the event corresponding to an input does not occur at that time step and the value 1 implies that the event does occur at the time step.

Figure 8 shows an example on how inputs can be implemented in NuSMV. If an input is fully independent of the system the assignment can be done directly (see the it_is_raining_outside variable.) If an input depends on the current state of the system it can be done by using a direct assignment with a `case` expression (see the `short_circuit` variable.) Finally, if an input depends on the state of the system in the previous time step then the assignment is done by using an *init/next* construct and the `case` expression (see the `device_is_broken` variable.)

With respect to implementing the inputs, one might want to use define declarations in the place of the variables and direct assigment. However, this is not possible because NuSMV does not allow to use set expressions in the define declarations.

```
VAR
   it_is_raining_outside : boolean;
   short_circuit : boolean;
   device_is_broken : boolean;
      ...

ASSIGN
   it_is_raining_outside := {0,1};

   short_circuit :=
      case
        (electricity_feed_on = 0) : 0;
        (electricity_feed_on = 1) : {0,1};
      esac;

   init(device_is_broken) := {0,1};
   next(device_is_broken) :=
      case
        (device_is_broken = 1) : 1;
        (device_is_broken = 0) : {0,1};
      esac;
```

Figure 8: Implementation of the Inputs of the abstract SIS model

5.2.1 Implementation of the Timer module

Figure 10 shows an implementation of the Timer module. The module has the same parameters as the Delay module described above: a boolean valued input signal and a delay value whose type is a non-negative integer. These are named as `input_signal` and `delay`, respectively. It also defines the output signal (named as `output`) as a boolean valued variable. The operation logic of the Timer module is described as follows. In the *basic state* the module is waiting for an input signal, i.e., it is waiting for that value 1 is passed through the `input_signal` parameter. When this happens the module will be in the *waiting state* for the number of time steps specified by the `delay` parameter. After this the module transfers automatically into the *launch state* in which it will stay only for one time step. In the launch state the value of the `output` variable is set to 1 (in the other states it is 0). After staying in the launch state for one time step, the module automatically transfers back into the waiting state. The value of the `input_signal` parameter is ignored whenever the module is not in the basic state. The operation logic of the Timer module is illustrated in Figure 9.

As in the case of the Delay module, the implementation of the Timer

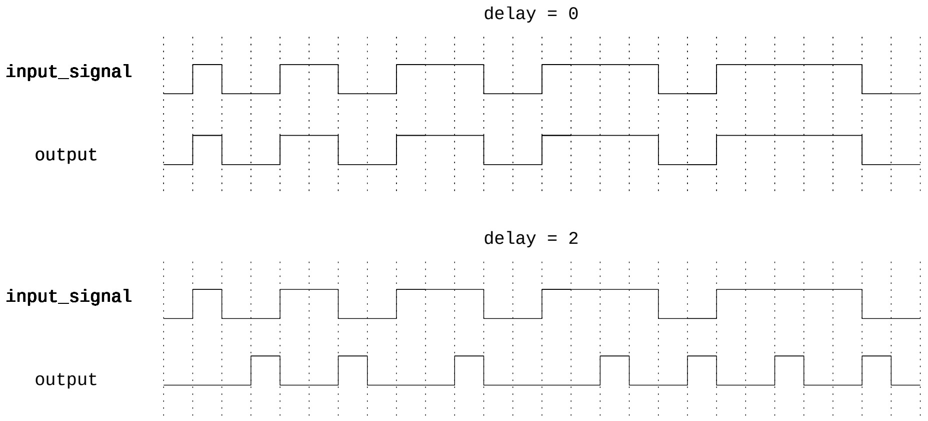


Figure 9: Input/output behaviour of the Timer module

module is also based on an integer valued counter (named as `counter`) which counts the number of time steps passed. Initially, the `counter` variable is set to the value of the `delay` parameter and it continues to hold that value until the `input_signal` parameter is 1. After this, regardless of the value of the `input_signal`, the `counter` is decreased by the value 1 until it reaches the value 0, after which it is set back to the value of the `delay` parameter. The value of the `output` variable is set to 1 only at the time when the value of the `counter` is 0.

```
MODULE Timer(input_signal,delay)
VAR
  -- TIMER_RANGE_UPPER_LIMIT is replaced with the
  -- value of the delay parameter.
  counter : 0..TIMER_RANGE_UPPER_LIMIT;

DEFINE
  output :=
    case
      (delay = 0) : input_signal;
      (counter = 0) : 1;
      1 : 0;
    esac;

ASSIGN
  init(counter) := delay;
  next(counter) :=
    case
      (delay = 0) : 0;
      (counter = 0) : delay;
      (counter < delay) : counter - 1;
      (counter = delay) & (input_signal = 1) : counter - 1;
      1 : counter;
    esac;
```

Figure 10: Implementation of the Timer module

5.2.2 Implementation of the OneShotTimer module

Figure 11 shows a modified version of the Timer module specified above. We have named this module as the ”OneShotTimer“ as it corresponds to the Timer module in all the other respects apart from the fact that it, informally speaking, can be used only once. That is, after being once in the waiting state the module will transfer into the launch state but in this case it will stay in the launch state forever instead of transfering back into the basic state.

The OneShotTimer module can be used to model the delays of interactions that occur only once (e.g., a final element of a SIS could be assumed to function only once so that after it is launched and become activated it will stay activated forever.) Such interactions could be modelled also by using the Timer module presented above but this would enlarge the state space of the overall model unnecessarily. Thus, the OneShotTimer can be seen as an optimization technique.

The implementation of the OneShotTimer module differs from the implementation of the Timer module only in a single place. In this case, in the assignment specification of the `counter` variable, the value of the `counter` is kept unchanged after it has reached the value 0.

```
MODULE OneShotTimer(input_signal,delay)
VAR
  -- TIMER_RANGE_UPPER_LIMIT is replaced with the
  -- value of the delay parameter.
  counter : 0..TIMER_RANGE_UPPER_LIMIT;

DEFINE
  output :=
    case
      (delay = 0) : input_signal;
      (counter = 0) : 1;
      1 : 0;
    esac;

ASSIGN
  init(counter) := delay;
  next(counter) :=
    case
      (delay = 0) : 0;
      (counter = 0) : 0;  -- When counter is zeroed it stays in zero.
      (counter < delay) : counter - 1;
      (counter = delay) & (input_signal = 1) : counter - 1;
      1 : counter;
    esac;
```

Figure 11: Implementation of the OneShotTimer module

6 CASE STUDY: ELECTRIC ARC PROTECTION SYSTEM

6.1 Overview of the Falcon System

Falcon protection system by Engineering Office Urho Tuominen (UTU) can be used to protect switchgear and electrical instrumentation from electric arcs. The system consists of a master unit, overcurrent sensor units, and light sensor units. Sensors are installed into the protected system and connected to the master unit via optical cables. The master unit collects the alarm signals from sensors, and when necessary, launches circuit breakers which close the power feed from the protected device leading to the termination of the electric arc. This basic setting is illustrated in Figure 12 [36].

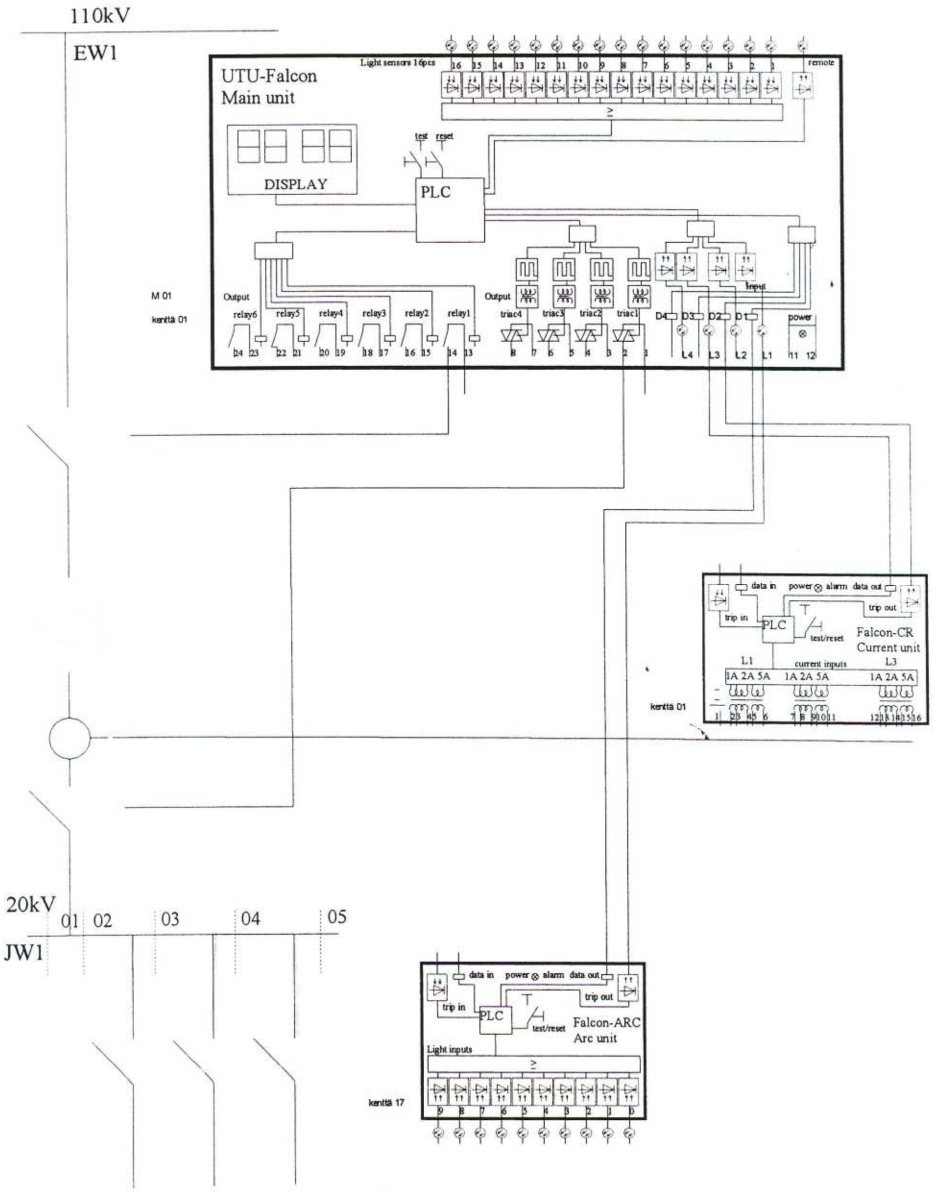


Figure 12: The Falcon Protection System

The master unit is based on a Programmable Logic Controller (PLC) so that one can freely design and program the tripping logic according to the protected system and the protection required for it. This provides the possibility for selective tripping: the protected system can be divided into several protection zones with different tripping conditions. The Falcon system also provides a possibility for controlling backup breakers which can be launched in the case of a malfunction in a primary breaker.

Figure 13 [35] shows an example of a tripping logic of the Falcon system. The figure also shows the input and output ports of the master unit. For attaching sensor units there are four regular input ports. In addition, there is also a so-called "extra light board" with additional 16 inputs which are meant for light sensors. However, the signals from these ports are combined optically before they are transmitted to the controller, so from the perspective of tripping logic these input ports correspond to a single input port.

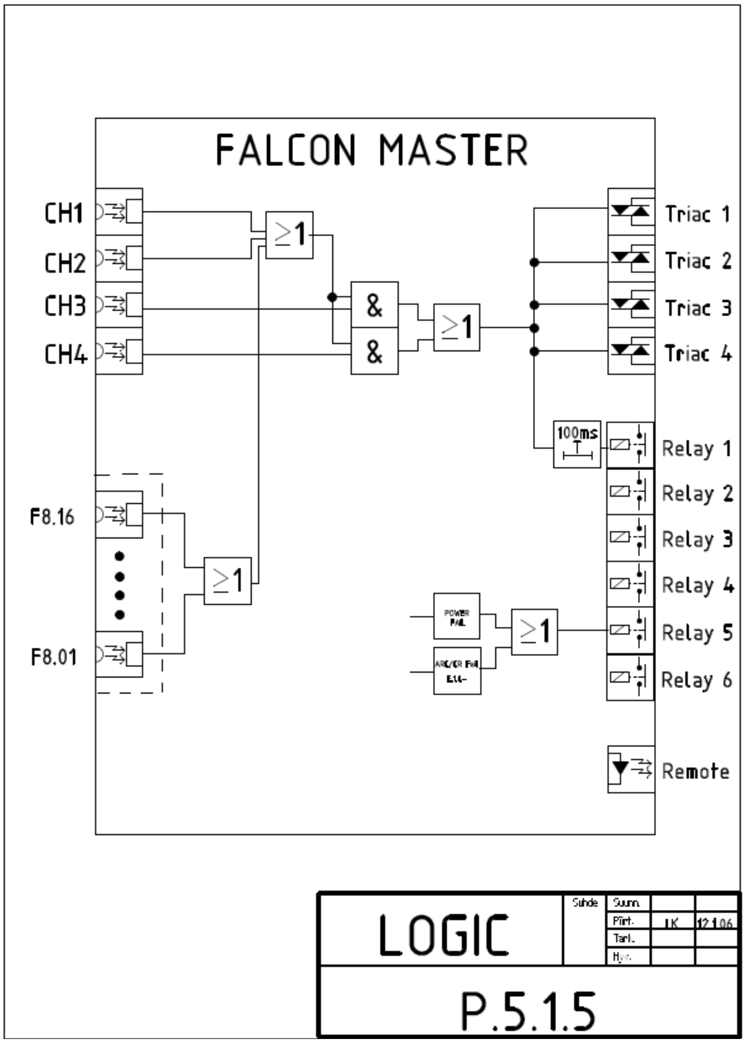


Figure 13: A tripping logic of the Falcon system

The number of output ports of the master unit is 10. Four of these are based on fast TRIAC semiconductors and are meant for launching the primary circuit breakers. The other six outputs are based on ordinary relays and are used for launching backup relays and alarm signals for operators.

The basic programming of a tripping logic is done simply by connecting signals with logical AND and OR gates. If backup breakers are used, delay gates of a certain type are also needed. This is because backup breakers might typically cover more than one protection zone and therefore they are supposed to be launched only after it is evident that the primary breakers covered by it have been broken. This is done by transmitting the launch signal of a backup breaker through such a delay gate which passes an output signal only once it has received an input signal continuously for a certain time period. Now, the delay of a delay gate corresponding to a certain backup breaker has to be longer than the physical activation time of the primary breakers protected by the backup breaker. In this way it is guaranteed that a backup breaker is not launched before the primary breakers protected by it have had enough time to have closed the power feed of the protection zone (and terminated the cause of the alarm, i.e., the electric arc) if they are not broken.

6.2 Verifying the Implementation of Control Logic

6.2.1 Overview on the Verification Task

Here we present an example of the task of verifying whether an implementation of control logic conforms to its specification. In this context, by a specification we mean a description of the input/output behaviour of the control logic. That is, a specification describes what output signals the controller should return for each possible input combination. The verification task introduced is to verify that an implementation built on a specification actually behaves precisely according to the specification. With respect to the abstract model presented in Section 3.2, only the Logic part of the Controller in Figure 1 is considered.

The verification was carried out on a set of real system descriptions provided by UTU. Figure 14 shows the implementation of the control logic which is based on the specification document shown in Figure 15.

6.2.2 Description of the NuSMV Model

The NuSMV model consist simply of two modules named `TruthTable` and `Falcon` which encode the specification and the implementation (respectively) of the control logic. The structure of both of the modules is very similar. Both have the inputs of the Falcon master unit as parameters and both include boolean state variables for the four output signals used in the control logic. In the case of the `Falcon` module, the logic is encoded conveniently by introducing a define declaration for each of the logical gates of the tripping logic and by using these declarations with the assignments of the state variables.

In the case of the `TruthTable` module, the state variable assignments were done by encoding the rows of the specification document directly into `case` expressions.

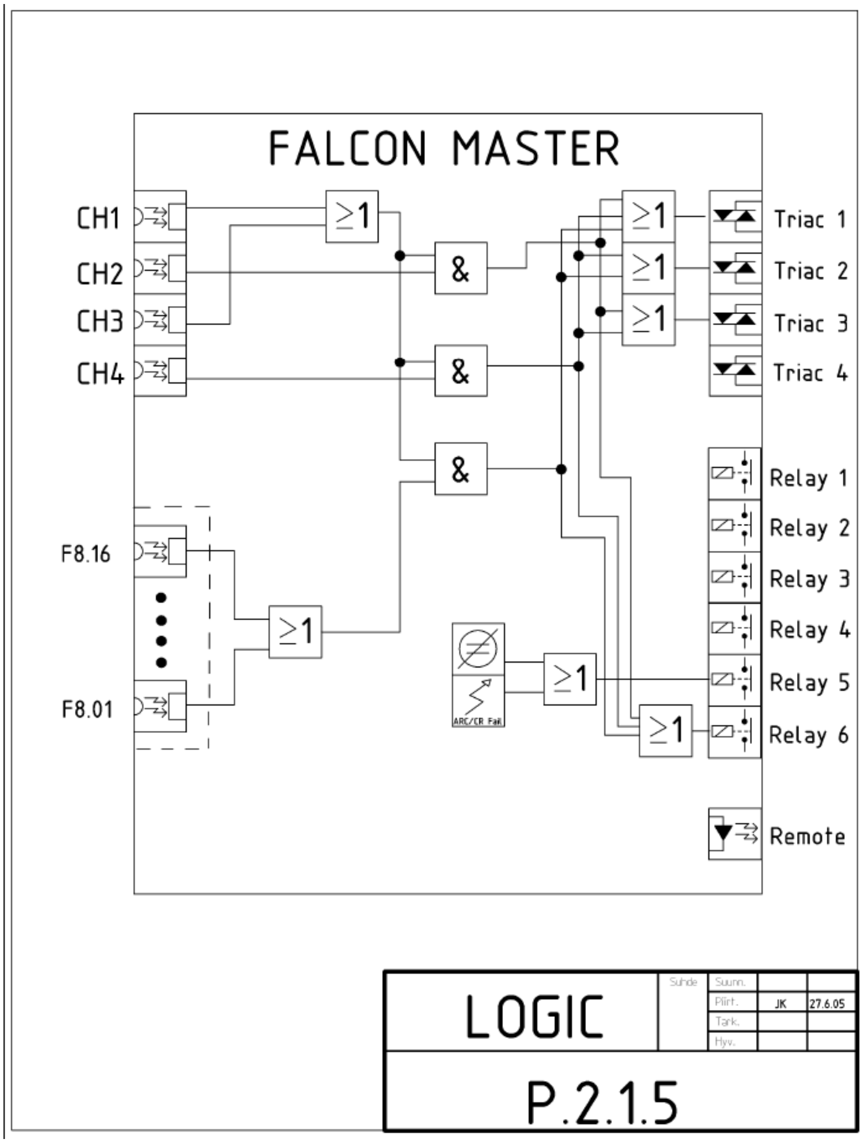
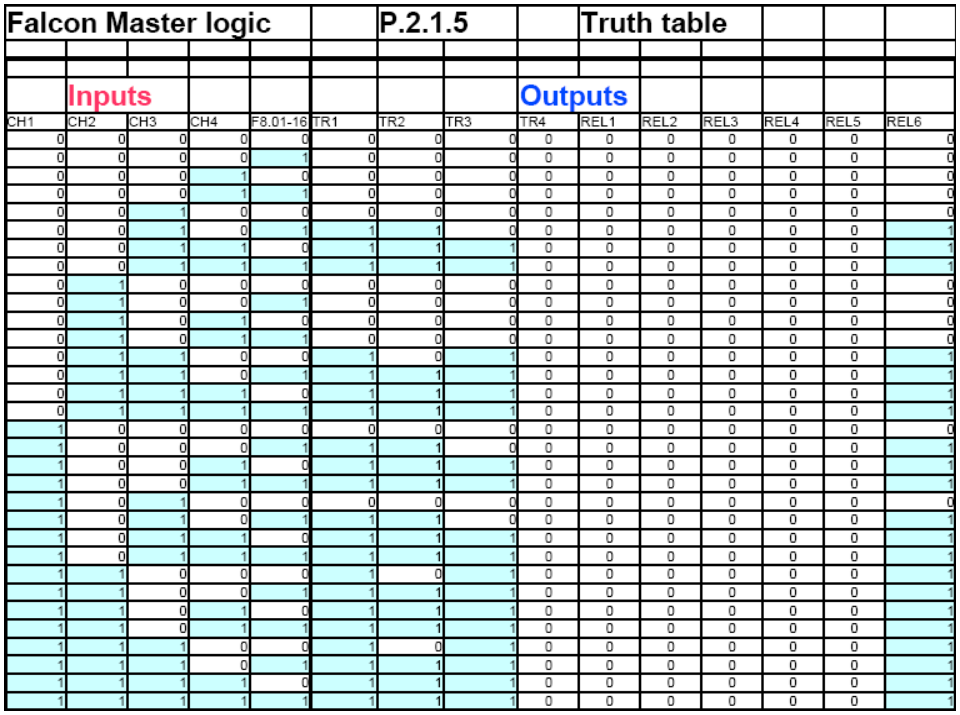


Figure 14: Tripping logic diagram of the example system



Falcon Master logic						P.2.1.5			Truth table					
	Inputs							Outputs						
CH1	CH2	CH3	CH4	F8.01-16	TR1	TR2	TR3	TR4	REL1	REL2	REL3	REL4	REL5	REL6
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	0	0	0	0	0	0	0	1
0	0	1	1	0	1	1	1	0	0	0	0	0	0	1
0	0	1	1	1	1	1	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	0	0	0	0	0	0	0	0	0	0
0	1	1	0	0	1	0	1	0	0	0	0	0	0	1
0	1	1	0	1	1	1	1	0	0	0	0	0	0	1
0	1	1	1	0	1	1	1	0	0	0	0	0	0	1
0	1	1	1	1	1	1	1	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	1	1	0	0	0	0	0	0	0	1
1	0	0	1	0	1	1	1	0	0	0	0	0	0	1
1	0	0	1	1	1	1	1	0	0	0	0	0	0	1
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	1	1	1	0	0	0	0	0	0	0	1
1	0	1	1	0	1	1	1	0	0	0	0	0	0	1
1	0	1	1	1	1	1	1	0	0	0	0	0	0	1
1	1	0	0	0	1	0	1	0	0	0	0	0	0	1
1	1	0	0	1	1	1	1	0	0	0	0	0	0	1
1	1	0	1	0	1	1	1	0	0	0	0	0	0	1
1	1	0	1	1	1	1	1	0	0	0	0	0	0	1
1	1	1	0	0	1	0	1	0	0	0	0	0	0	1
1	1	1	0	1	1	1	1	0	0	0	0	0	0	1
1	1	1	1	0	1	1	1	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	0	0	0	0	0	0	1

Figure 15: Truth table representation of the specification of the tripping logic of the example system

6.2.3 Specification of Properties with NuSMV

With this verification task only one property needed to be specified. It is an invariant specification which states that with all possible combinations of inputs, the outputs have to be the same. This property is specified in the input language of NuSMV in the following way:

```
LTLSPEC G ((falcon.triac1 <-> truth_table.triac1) &
           (falcon.triac2 <-> truth_table.triac2) &
           (falcon.triac3 <-> truth_table.triac3) &
           (falcon.relay6 <-> truth_table.relay6))
```

6.3 Verifying the Correctness of System Design

In Section 6.2 we showed how it can be verified that the control logic of the Falcon system conforms to its specification. In contrast, here we show how the correctness in the design of a whole system can be verified. That is, we want to verify that a protection system based on a certain control logic operates as intended with respect to the system it protects.

This section is organised as follows: Section 6.3.1 describes the properties which the system is required to fulfil in order that the design is considered to be correct. Section 6.3.2 describes the types of information required from the system that the model checking can carry out. Section 6.3.3 describes the specific application of the Falcon system which was used in the case study. Section 6.3.4 describes what kinds of assumptions one needs to make on the system so that it can be modelled. Section 6.3.5 gives an overview of the

NuSMV model of the case study and Section 6.3.6 explains how the verified properties are specified in the input language of NuSMV. Section 6.3.7 presents some experimental results of the running times of the model checking of the case study with different parameter values. Finally, in Appendix B, the full source code of the NuSMV model with the property specifications is presented.

An earlier version of the model presented in this section is reported in [38].

6.3.1 Verified Properties

In the case of the Falcon system, the most important property to be verified is that the system does not make unnecessary tripping decisions. This is because the system is often used to protect, for example, large manufacturing plants for which an unnecessary shutdown caused by an unnecessary tripping decision might cause very high expenses.

In order to avoid any false trips, the following properties have to hold:

p1: The couplings and the tripping logic have to conform to the specified tripping conditions.

p2: The backup breakers should not be tripped unless necessary.

The requirement of the absence of unnecessary tripping decisions falls into the category of *safety properties* as it states that the system should not do anything unwanted. Another type of properties called *liveness properties* informally state that the system should always perform the task that it is designed for. In the case of the Falcon system, this would be stated as the following requirement:

p3: Existence of an electric arc on the protected system leads eventually to shutting down the power feed for the protected system.

These properties are the most relevant requirements for the Falcon system. In the following section we list the types of information and documents needed in order to be able to verify these properties with the aid of model checking.

6.3.2 Information Required for Verification

Here we describe what sorts of information one needs in order to model check the properties of the Falcon system:

1. Description of the specific application:
   In case of verifying the correctness of the system design of a safety instrumented system, the question is of verifying whether the control logic of a controller is designed correctly with respect to the environment in which the controller is installed. Therefore, in this case it is not sufficient to model only the control logic of the controller, but one also has to build a model of the environment of the controller. For this reason, besides the control logic, we need now also a switch diagram and a system description with the following information:

   - What is the structure of the protected system (structure of the power-distribution network, location of the power feeds, transformers, circuit breakers)?
   - How are the sensor units installed into the protected system?
   - Into what kinds of protection zones is the protected system divided into?
   - What are the tripping conditions of the protection zones?
   - Which circuit breakers need to be launched in order to disable the power feed from the protection zones?
   - Are there any backup circuit breakers, and if so, what are their tripping conditions?

2. Assumptions about the whole system:
   The information listed in the previous item describes the architectural structure of the protected system and the installation and intended operation model of the protection system. However, for the modelling of the whole system, one also needs to clarify all relevant physical and functional properties on both the protection system and the protected system. A few examples of the things to be clarified in the case of the Falcon system are:

   - What kinds of delays are there with the devices of the system?
   - In which parts of the protected system can short circuits occur?
   - What are the failure modes of the associated devices?

   Because all aspects of the physical world cannot be modelled, one has to make *assumptions* on the physical system so that the physical model can be stated to conform to the model in case the assumptions hold.

   These kinds of detailed descriptions of the system might not be available in the existing documentation neither in the case of the protected or the protection system. Therefore, with critical applications, the modelling of the system should always be carried out in cooperation with domain specialists.

3. List of unambiguously defined requirements to be verified:
   In the previous section the verified properties of the Falcon system were listed on a general level. However, in order to perform model checking, the properties have to be described more precisely so that there are no questions about how the properties should be interpreted. In this case, for example, one needs to state precisely when a tripping decision is unnecessary. In Section 6.3.6, it is shown how the verified properties are refined so that they can be stated in the terms of the formal model of the system.

Unfortunately a complete set of all this information concerning a single specific application of the Falcon system was not available. Therefore we designed our own application on the basis of the documents we received from

UTU and which related to several different applications. Our model was reviewed by UTU representatives and it was considered to be fully realistic in all aspects.

6.3.3 Description of the Application

Architecture of the System

Our example application of the Falcon system is shown in Figure 16. The system consists of the protected system and the Falcon system. The protected system consists of the following things:

- main power feeds pf1 and pf2,
- transformers tr1, tr2, tr3, and tr4,
- primary circuit breakers A, B, C, and D,
- backup circuit breakers E, F, H, and G, and
- protection zones 1, 2, and 3.

The Falcon system introduces the following elements into the whole system:

- the Falcon master unit,
- overcurrent sensors Cr1, Cr2, Cr3a, and Cr3b, and
- light sensors L1, L2, and L3.

Operation of the System

The main power feeds pf1 and pf2 distribute electricity to the protected system. They are connected to each other by a switch operated by the circuit breaker C, and therefore, they act as each others backup systems. That is, both pf1 and pf2 can deliver power to the whole protected system alone if a malfunction occurs in one of them.

The protected system is divided into three distinct protection zones. For all of these there is a zone-specific tripping condition which causes tripping of circuit breakers that leads to the isolation of the protection zone from the power feed. The protection system is designed to operate with each protection zone so that there are two "levels" of backup breakers. That is, if the primary breakers are broken, the protection system first tries to cut down the power feed only from the main power feed which is closest to the alarming zone (the "first level"). If the alarm is still on (which might result, e.g., if the connecting breaker C was broken), then the power feed will be cut also from the other main power feed which will lead to the power feed from the whole system (presumed that the backup breakers are working correctly) being cut off.

The tripping conditions and related actions are listed in the Table 1 and in Figure 17 a tripping logic which is created based on this table is presented.

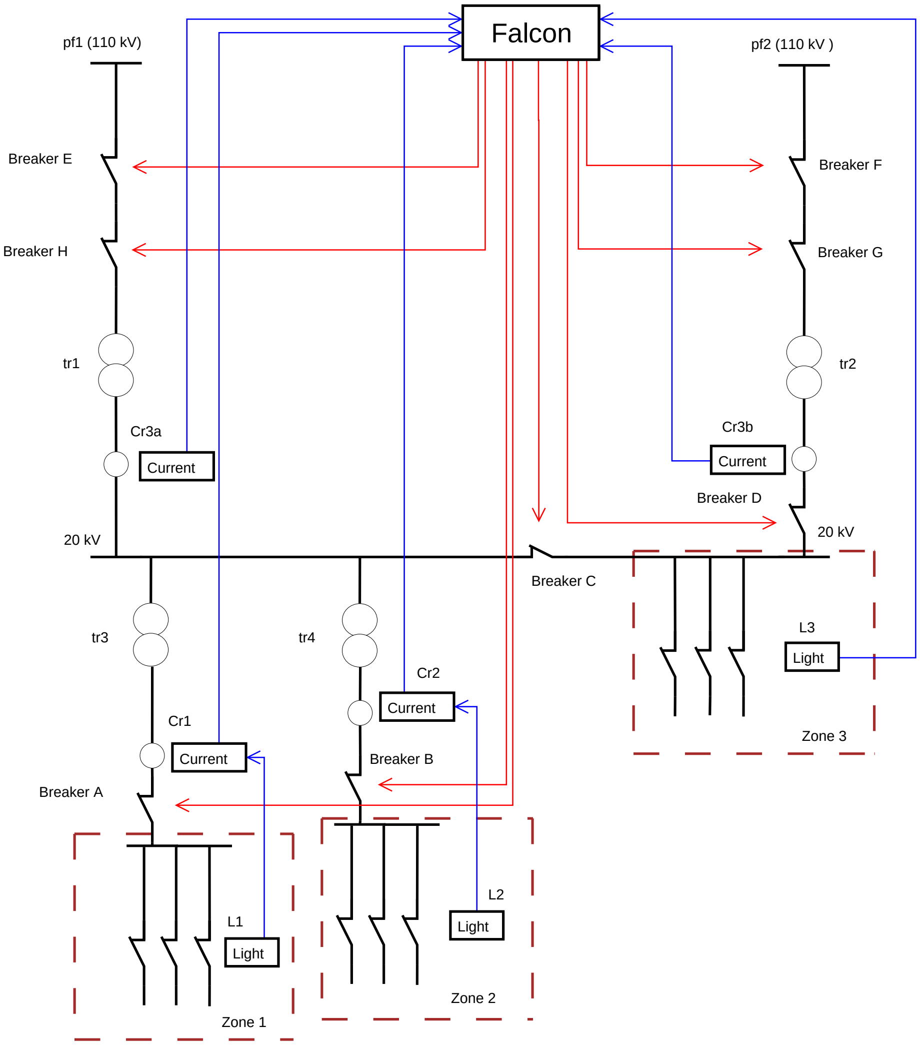


Figure 16: Switch diagram of the example system

The delays D1 and D3 are related to the backup breakers of the "first level" and delays D2 and D4 are related to the "second level". Therefore, it should be that D1 < D2 and D3 < D4.

Table 1: Actions caused by alarms on different protection zones

Alarm	First action	Second action	Third action
Cr1 AND L1 (alarm on zone 1)	Breakers A and C launched	Breaker E launched (after delay D1)	Breaker F launched (after delay D2)
Cr2 AND L2 (alarm on zone 2)	Breakers B and C launched	Breaker E launched (after delay D1)	Breaker F launched (after delay D2)
(Cr3a OR Cr3b) AND L3 (alarm on zone 3)	Breakers C and D launched	Breaker G launched (after delay D3)	Breaker H launched (after delay D4)

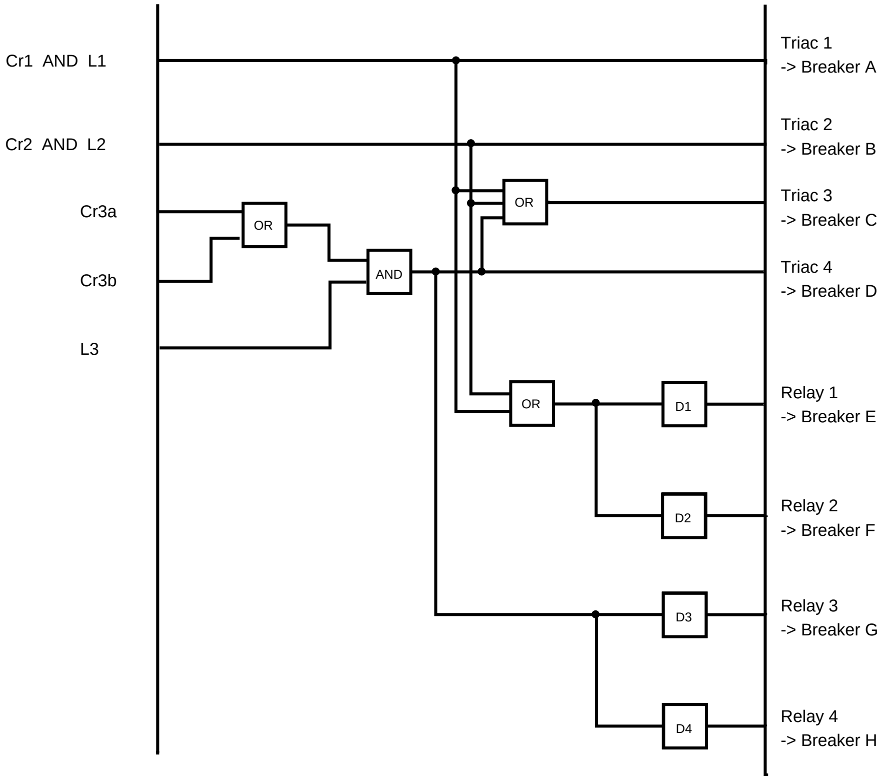


Figure 17: Tripping logic of the example system

6.3.4 Assumptions of the System

In the previous section the structure and operation of the example system was described. However, in order to be able to carry out the modelling process, we also need to make some assumptions about the functional and behavioural properties of the system. Here is the list of assumptions made on the example system.

General assumptions:

- The duration of one *operation cycle* of the controller of the Falcon master unit, i.e., time during which the Falcon system detects an alarm signal through a sensor and passes a launch signal to a circuit breaker is 1 millisecond. This time period will correspond to a single time step in the model of the system.

- Failures of the physical devices other than the primary circuit breakers are not considered.

Overcurrent alarms:

- Overcurrent peaks detected by the overcurrent sensors are caused by short circuits.
- Short circuits can arise only in the parts of the protected system which are defined as protection zones.
- Overcurrent peaks cannot move through the transformers, i.e., an overcurrent peak observed in a point A is not observed in a point B if there is a transformer between the points A and B.
- An overcurrent sensor can raise an alarm signal nondeterministically anytime as long as it is connected to the protection zone it is overseeing and the protection zone is still connected to a power feed. If these conditions are not met, the overcurrent sensor cannot raise an alarm.

Light alarms:

- A light sensor can raise an alarm signal nondeterministically at any given time instant, i.e., light alarms are independent of the rest of the system.

Circuit breakers:

- Once a circuit breaker has been *activated*, it opens the electric circuit and prevents the flowing of the current.
- An activated circuit breaker will remain activated forever.
- There is an *activation delay* associated with each circuit breaker, which is the time period between the moment when a breaker is launched and the moment when it has opened the circuit preventing the electric current flowing. (The model checking was carried out with different parameter values for the size of the activation delay, see Section 6.3.7.)
- A non-activated primary circuit breaker can break down at any given time.
- A broken circuit breaker cannot open a circuit.
- A broken circuit breaker will stay broken forever.

6.3.5 Description of the NuSMV Model

In this section we give an overall description on how the Falcon system and it's environment was modelled with NuSMV. The text is organised according to the abstract SIS model covered in Section 3.2. We describe for each part of the abstract model which parts of the Falcon system correspond to it. Moreover, we give an overall description on how these parts of the Falcon system were modelled with NuSMV. In the following text we will refer to the parts of the abstract model with the "abstract"-prefix to emphasise the distinction between corresponding parts of the Falcon system or the NuSMV model.

We begin the discussion from the Controller part of the abstract model and then proceed to the System environment. The NuSMV modules described in the following are also illustrated in Figure 18 which depicts the data flow between the modules.

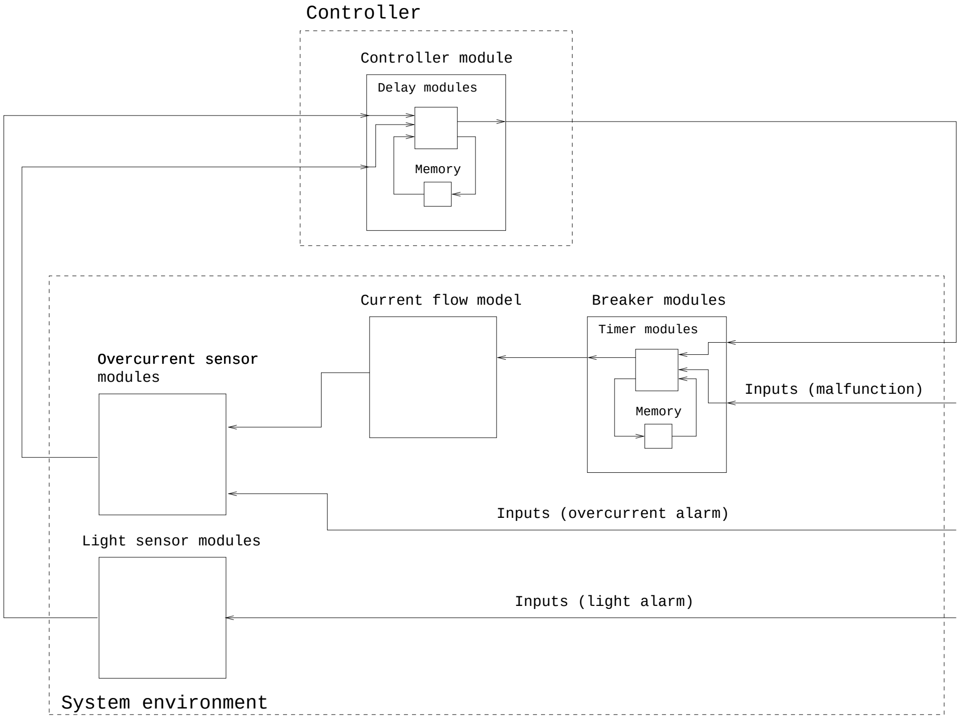


Figure 18: Data flow between NuSMV modules

Controller

In the case of the Falcon system, the master unit corresponds to the Controller part of the abstract SIS model. The Falcon counterpart for the logic part of the abstract Controller is the logical circuit of the tripping logic excluding the delay gates. The delay gates correspond to the delays of the controller in the abstract model. The delay gates are implemented exactly as it was described in Section 5.1 so their implementation is not repeated here. In Section 5.1 we showed also a general outline for NuSMV implementation of the abstract Controller. In the following we show how this implementation is done in the case of the Falcon system.

Implementation of the Controller module The Controller module implementing the controller of the Falcon system is shown in Figure 19. In the case of the Falcon system, the Controller module has five parameters named as `ch1` – `ch4` and `ch_light` (in the actual system the 16 inputs of the light board are combined into a single signal with an optical `OR` and this signal is represented by the parameter `ch_light`). The number of outputs is eight of which half are TRIAC outputs and other half are relay outputs. For each output there is an instance of the Delay module (named as `triacX_delay` and `relayX_delay`) and two constant definitions. The constants named `logic_output_trX` (for TRIAC outputs) and `logic_output_relayX` (for relay outputs) correspond to the constant definitions `logic_output_i` of the generic Controller implementation in Section 5.1. Similarly, the constant declarations `triacX_output` and `relayX_output` correspond to the constant definitions `controller_output_i` of the generic Controller implementation in Section 5.1.

The delay parameter values of the Delay module instances are set to values $\lceil D/t \rceil$ where D is the delay in milliseconds of the corresponding delay gate in the Falcon tripping logic and t is the length (also in milliseconds) of the operation cycle of the controller of the Falcon master unit. In practice, the parameter value is the physical delay in milliseconds since the operation cycle of the Falcon master unit is 1ms as stated in Section 6.3.4. In the case of the TRIAC outputs the delay parameters are set to the value 0 because in the tripping logic of the Falcon system there are no delay gates corresponding to the TRIAC outputs. With all the outputs, the parameter can be assigned by setting a numeric value directly to the parameter of the delay module instance or by specifying a define declaration for each parameter as we have chosen to do (see `TRIAC_DELAY` and `RELAYX_DELAY` declarations in Figure 19).

System environment

In section 5.2 we presented general instructions on implementing the System environment of the abstract model in terms of different kinds of interactions and inputs that can be captured by the System environment. In the following we describe first what kinds of things comprise the System environment in the case of the Falcon system. Then we describe into what kinds of modules the system environment is encoded and finally we describe the implementation of these modules one by one by using the methods described in Section 5.2.

In the case of the Falcon system, the system environment of the abstract model breaks down to the protected system (divided into one or more protection zones), primary and backup circuit breakers, and the sensor units of the Falcon system.

The logic of the system environment consists of the following things:

- operational and failure models of the breakers,
- operational model of the sensors, and
- reasoning of whether each protection zone is connected to a power feed.

```
MODULE Controller(ch1,ch2,ch3,ch4,ch_light)
VAR
  triac1_delay : Delay(logic_output_tr1,TRIAC_DELAY);
  triac2_delay : Delay(logic_output_tr2,TRIAC_DELAY);
  triac3_delay : Delay(logic_output_tr3,TRIAC_DELAY);
  triac4_delay : Delay(logic_output_tr4,TRIAC_DELAY);

  relay1_delay : Delay(logic_output_relay1,RELAY1_DELAY);
  relay2_delay : Delay(logic_output_relay2,RELAY2_DELAY);
  relay3_delay : Delay(logic_output_relay3,RELAY3_DELAY);
  relay4_delay : Delay(logic_output_relay4,RELAY4_DELAY);


DEFINE
  -- Delay values of the delay gates. These values should be set to
  -- the delay values (in milliseconds) of the corresponding delay
  -- gates D1-D4 in the modelled tripping logic.

  TRIAC_DELAY := 0;
  RELAY1_DELAY := D1;
  RELAY2_DELAY := D2;
  RELAY3_DELAY := D3;
  RELAY4_DELAY := D4;

  -- Logic of the circuits.
  OR1 := (ch3 | ch4);
  AND1 := (OR1 & ch_light);
  OR2 := (ch1 | ch2 | AND1);
  OR3 := (ch1 | ch2);

  -- Inputs to delay gates.
  logic_output_tr1 := ch1;
  logic_output_tr2 := ch2;
  logic_output_tr3 := OR2;
  logic_output_tr4 := AND1;

  logic_output_relay1 := OR3;
  logic_output_relay2 := OR3;
  logic_output_relay3 := AND1;
  logic_output_relay4 := AND1;

  -- Outputs of the controller module.
  triac1_output := triac1_delay.output;
  triac2_output := triac2_delay.output;
  triac3_output := triac3_delay.output;
  triac4_output := triac4_delay.output;

  relay1_output := relay1_delay.output;
  relay2_output := relay2_delay.output;
  relay3_output := relay3_delay.output;
  relay4_output := relay4_delay.output;
```

Figure 19: Controller module

The memory elements of the abstract system environment are used for holding the state of the system environment in the previous time step. In the case of the Falcon system, these states are related to the circuit breakers. That is, for each circuit breaker we need to know whether the following things held in the previous time step:

- is the breaker broken,
- has the breaker been launched, and
- is the breaker activated.

In the case of the Falcon system, the inputs of the system environment are:

- overcurrent and light signals, and
- the information of whether the primary breakers are broken.

The NuSMV model of the system environment consists of two distinct modules for light and overcurrent sensors (UTU_ARC and UTU_CR modules, respectively), a module for circuit breakers (the same module is used for both primary and backup breakers), a module for encoding a counter representing the activation delay of the breakers, and constant definitions for the current flow model. In the following we give an overview on how these entities were implemented.

Implementation of the Breaker module The implementation of the Breaker module is shown in Figure 20. The Breaker module models the physical circuit breakers of the protected system that are controlled by the Falcon master unit. In practice, the Breaker module models the delayed interaction in which a launch signal received from the Falcon master unit leads to the activation of a circuit breaker. The activation of the breaker is represented with the boolean valued variable `cuts` and the launch signal is represented with the boolean valued parameter `launch_signal`. The delay of the interaction, i.e., the activation time of the physical circuit breaker is represented with the parameter `setting_up_time`. This parameter should be set to the value $\lceil D/t \rceil$ where D is the physical activation delay (in milliseconds) of the corresponding real circuit breaker and t is the length (in milliseconds) of the operation cycle of the controller of the Falcon master unit. In practice, the parameter value is the physical delay in milliseconds since the operation cycle of the Falcon master unit is 1 ms as stated in Section 6.3.4.

The breaker module also introduces an input signal for the environment model which corresponds to a malfunction in a physical breaker. This input is represented with the boolean valued variable `is_broken` and it depends on the previous state of the system as the failure model of the breakers assumes that a broken breaker stays broken forever. Consequently, if a breaker has been broken in the previous time step, it has to be broken also in the current time step. However, as the malfunctions are only related to primary breakers the Breaker module has a boolean valued parameter `can_break` which specifies whether the breaker can get broken or not.

If the value of the `can_break` parameter is set to 0, the breaker is not able to break down and the value of the `is_broken` variable is set to 0 in every time step. In this case the operation logic of the Breaker module is identical to the OneShotTimer module with the identifiers `launch_signal`, setting_up_time, and cuts of the Breaker module corresponding to the identifiers signal, delay, and output of the OneShotTimer, respectively. Thus, the states *basic state*, *waiting state*, and *launch states* are defined in the same way for the Breaker module as they were defined for the Timer and OneShotTimer modules in Section 5.2.

On the other hand, if the value of the can_break parameter is set to 1, the Breaker is able to break down on any given time step. In terms of the variables, the operation logic of the Breaker module can be stated as follows. In the initial state, the is_broken variable is set to 0. After the initial state the value can be either 0 or 1 as long as the value of the variable is set to 1 for the first time. Once the value of the is_broken is set to 1 it will be 1 on all the succeeding time steps. If the value of the is_broken variable stays as 0 until the cuts variable has been set to 1 (i.e. the Breaker module is in the launch state) it has no effect on the operation of the Breaker module. However, if the is_broken is set to 1 while the Breaker module is in initial state or in the waiting state the state of the module will stay unchanged and the value of the cuts variable will continue to be 0 in all the succeeding time steps.

As the input is_break depends on its state in the previous time step it is implemented with direct assignment by using case and set expressions similarly as the variable device_is_broken in the example of Figure 8. The delayed interaction implemented by the Breaker module is implemented by using a case expression and the OneShotTimer (see Fig. 11) with the assignment of the cuts variable. OneShotTimer is used in similar manner as the Timer, for an example see the definition of the variable water_boiler_empty in Figure 7 and the related discussion in Section 5.2.

Implementation of the UTU_ARC module The UTU_ARC module models the light sensors of the Falcon system. The implementation of the module is shown in Figure 21. In practice, the UTU_ARC has only the function of implementing the light alarm inputs of the System environment. As the light alarms are fully independent of the state of the system, the module only contains a single boolean valued variable named as light. If the value of light is 1 in a given time step the sensor is observing a light alarm and if the value is 0 the sensor does not observe light alarm.

The light input is implemented simply by using a direct assignment and a set expression.

Implementation of the UTU_CR module The UTU_CR module models the overcurrent sensors of the Falcon system. The implementation of the module is shown in Figure 22. In practice, the UTU_CR has only the function of implementing the overcurrent alarm inputs of the System environment. These inputs are dependent on the current state of the system as it is assumed that overcurrent alarms can only occur while the sensor is connected to the protection zone it is overseeing and the protection zone is connected to the power feed (see Section 6.3.4). The boolean valued parameter has_voltage is used to pass the information whether the corresponding protection zone is still connected to the power feed. The parameter breaker is used to pass the Breaker module instance that is located between the pro-

```
MODULE Breaker(launch_signal, setting_up_time, can_break)
VAR
  is_broken : boolean;
  cuts : boolean;
  timer : OneShotTimer(launch_signal,setting_up_time);

DEFINE
  launched := launch_signal;

ASSIGN
  init(is_broken) := 0;
  next(is_broken) :=
    case
      can_break = 0 : 0;
      is_broken = 0 : {0,1};
      is_broken = 1 : 1;
      1             : 1;
    esac;

  init(cuts) := timer.output;
  next(cuts) :=
    case
      (cuts = 1) : 1;
      (is_broken = 1) : cuts;
      (next(timer.output) = 1) : 1;
      (next(timer.output) = 0) : 0;
    esac;
```

Figure 20: Breaker module

```
MODULE UTU_ARC()
VAR
  light : boolean;

ASSIGN
  light := {0,1};
```

Figure 21: UTU_ARC module

tection zone and the overcurrent sensor. It is assumed that the overcurrent sensor is always connected to the protection zone it is overseeing through a single route and therefore the state of the breaker in midway of the sensor and the zone is enough for reasoning whether the sensor and the zone are connected or not. The overcurrent alarm input is represented in the UTU_CR module with the boolean valued variable `overcurrent`. If the value of `overcurrent` is 1 in a given time step the sensor is observing an overcurrent alarm and if the value is 0 the sensor does not observe overcurrent alarm. The overcurrent input is implemented by using a direct assignment with case and set expressions.

Implementation of the Current flow model The purpose of the current flow model is to encode the logic of the interactions in which the activation of given circuit breakers leads to ending of the current flow in certain parts

```
MODULE UTU_CR(has_voltage,breaker)
VAR
  overcurrent : boolean;

ASSIGN
  overcurrent :=
    case
      !has_voltage | breaker.cuts  : 0;
      1                            : {0,1};
    esac;
```

Figure 22: UTU_CR module

of the power-distribution network. As it was described in Section 6.3.4, it is assumed that an activation of a circuit breaker ends the electric current through the breaker immediately so these interactions are instantaneous and can be implemented by using define declarations.

In practice The current flow model is implemented in such a way that there is a define declaration corresponding to each of the three protection zones which tell whether the zone is connected to a power feed or not. The declarations are named as `zoneX_hasvoltage` and their definition is shown in Figure 23. The value of the define constant corresponding to a certain protection zone is set to 1 if there is at least one closed circuit line connecting the protection zone to a power feed. Therefore, the zone constants are functions of the output values of the circuit breakers which tell whether the circuit breaker is active or not.

We also included one additional define declaration for each protection zone which tells whether the tripping condition of a zone is `true` at each time step (see the "alarm model" in Figure 23). These constants are not indispensable but with them the specification of properties becomes more convenient.

```
-- The alarm model
zone1_alarm := Cr_1.overcurrent & L_1.light;
zone2_alarm := Cr_2.overcurrent & L_2.light;
zone3_alarm := (Cr_3a.overcurrent | Cr_3b.overcurrent) & L_3.light;

-- The current flow model
zone1_hasvoltage :=
  !(breaker_A.cuts |
  ((breaker_E.cuts | breaker_H.cuts) &
  (breaker_C.cuts | breaker_D.cuts | breaker_F.cuts | breaker_G.cuts)));

zone2_hasvoltage :=
  !(breaker_B.cuts |
  ((breaker_E.cuts | breaker_H.cuts) &
  (breaker_C.cuts | breaker_D.cuts | breaker_F.cuts | breaker_G.cuts)));

zone3_hasvoltage :=
  !((breaker_C.cuts | breaker_E.cuts | breaker_H.cuts) &
  (breaker_D.cuts | breaker_F.cuts | breaker_G.cuts));
```

Figure 23: Implementation of the Current flow model

6.3.6 Specification of Properties with NuSMV

In this section it is shown how the properties described in Section 6.3.1 are specified with the input language of the NuSMV model checker. However, first we refine and specify each property in as specific form as is needed for the formal specification to be possible.

Safety properties

The first safety property **p1** of Section 6.3.1 states that the couplings of the system and tripping logic are done correctly. In the case of the primary breakers, this property is formulated specifically in the following way:

If a primary circuit breaker is launched at a certain time step, then the tripping condition of this breaker was realised in the previous time step.

With NuSMV this is specified as:

```
LTLSPEC G (LTLSPEC G (breaker_A.launched -> Y zone1_alarm))
```

In the case of the backup breakers, the property can be formulated more conveniently as follows:

If a backup breaker is launched at a given time step, then at the same time step one of the primary breakers covered by the backup breaker is receiving a launch signal.

With NuSMV this is specified like this:

```
LTLSPEC G (breaker_E.launched -> (breaker_A.launched | breaker_B.launched))
```

The second safety property **p2** of Section 6.3.1 states that the backup breakers should not be launched unless necessary. This requirement is formulated more precisely in this way:

If a backup breaker receives a launch signal, then at least one of the primary breakers covered by it has broken down.

With NuSMV this is specified as follows:

```
LTLSPEC G (breaker_E.launched -> (breaker_A.is_broken | breaker_B.is_broken))
```

Liveness properties

The liveness property **p3** of Section 6.3.1 is formulated more specifically like this:

If the protection system receives an alarm from a protection zone in a given instant of time, there will be a instant of time in the future, when the alarm has either disappeared from the protection zone or the protection zone *is disconnected from the power feed.*

With NuSMV this is specified like this:

```
LTLSPEC G (zone1_alarm -> F (!zone1_alarm | !zone1_hasvoltage))
```

6.3.7 Experimental Results

In the following we present some measurements on the running times of the model checking of our example system.

Test Equipment

The model checking was carried out with a PC with 1.8GHz Intel Core 2 Duo E63xx DualCore processor. Available virtual memory was limited to 1.5 GiB. The operating system used was Debian GNU/Linux and the model checking was carried out with NuSMV version 2.4.2 by using BDD–based LTL model checking with default options.

Measurements

The model checking was carried out on the model shown in Appendix B. The parameters altered were the delay parameters D1, D2, D3, and D4 of the tripping logic of the example system (see Figure 17 and Table 1) and the activation time A of the circuit breakers (with each distinct model checking process the same activation time was used with all the breakers). As explained in Section 6.3.5 (see the descriptions of the Controller and Breaker modules), these parameter values correspond to milliseconds in real-time.

For each value of A, valid values for the parameters D1–D4 have to be determined from the system design. For a given parameter A it is of interest to find as small values of parameters D1–D4 as possible for which the verified properties are still valid. Table 2 shows running times for the model checking process with different parameter values for which the properties p1–p3 described in Section 6.3.1 are satisfied.

For a complex system design, finding of smallest possible delay parameters corresponding to a given activation time A is a difficult and error-prone task. This is reflected by the fact that with the parameter values of Table 2, the state space of the system model of this case study varies between $3,0 * 10^{21}$ and $2,4 * 10^{29}$ states (size of the state space can be calculated as the product of the value ranges of all the state variables of the model). Consequently, the benefit of the automatic verification with model checking increases with the complexity of the verified system.

If the delay parameters for a certain activation delay A are chosen to be too small, all the properties are not valid anymore. In this case, the NuSMV model checker returns a counter example for each property that is violated. For example, if the value of the parameter D1 on the row A=3 in Table 2 is decreased from value 8 to value 7, a property of type p2 no longer holds and NuSMV returns a counter example, that is, an execution of the system along which the property is violated. Table 3 shows running times for parameters which are othervise same as in Table 2 but the parameter D1 on each row is decreased by 1 leading to unsatisfiability of one or more of the verified properties.

Analysis of results

The results of Table 2 show quite clearly the exponential behaviour of the running time with respect to the size of the parameters. As the opening times of real circuit breakers might be anything from 20 milliseconds to 50-100 milliseconds or even more, the applicability of the chosen modelling technique is, generally speaking, questionable. However, the system design studied here is advisedly made quite complex and for slightly smaller models the counter modelling technique might be quite practical.

A clear benefit of the model checking method with respect to this case study is that the optimal delay parameter values can be found reliably.

Table 2: Running times of the model checking process with different parameter values

A (Activation time of breakers)	D1	D2	D3	D4	Running Time
2	6	9	3	6	1 min
3	8	12	4	8	4 min
4	10	15	5	10	21 min
5	12	18	6	12	39 min
6	14	21	7	14	1h 48min
7	16	24	8	16	3h 33min
8	18	27	9	18	26min
9	20	30	10	20	10h 59min

Table 3: Running times of the model checking process with insufficient parameter values

Activation delay of breakers	D1	D2	D3	D4	Running Time
2	5	9	3	6	1 min
3	7	12	4	8	12 min
4	9	15	5	10	25 min
5	11	18	6	12	1h 19min
6	13	21	7	14	1h 48min
7	15	24	8	16	5h 4min
8	17	27	9	18	27 min
9	19	30	10	20	11h 38min

7 CONCLUSIONS

In this work, we have studied the applicability of the non-real-time model checker NuSMV to verification of safety and liveness properties of the UTU Falcon arc protection system. The central study objective was to find out whether the studied system could be modelled on the appropriate level of abstraction which is needed to guarantee reliable verification results and which, on the other hand, keeps the size of the model feasible.

The biggest challenge in the modelling of the arc protection system was the modelling of the time delays associated with both the protection system and its environment. In our approach the continuous time was discretised by using the scan cycle of the controller of the arc protection system as the basic unit of time. This way, the time delays could be modelled by using certain type of discrete counters. The main benefit of this technique is that it is very straightforward to implement. However, the scalability of the technique is a clear problem and, therefore, models based on counters have to be strongly restricted either in the number of counters or in the value ranges of the counters. The arc protection case study was shown to be at the limits for the applicability of the counter technique. The determining time delay, in this case, is the physical opening delay of the circuit breakers. We were able to carry out model checking with a basic desktop PC while using parameter values corresponding to the opening times of the circuit breakers of approximately up to 10ms. This result is promising but the question of the scalability of the modelling technique to parameter values closer to the average opening time of standard circuit breakers of high voltage power distribution networks was left open.

Generally speaking, the case study presented in this work shows that model checking can be both an applicable and a valuable tool in the verification of safety instrumented systems. As it was stated in Section 6.3.7, the size of the state space of the model built in the case of verifying the correctness of a system design varies between $3,0 * 10^{21}$ and $2,4 * 10^{29}$. This clearly shows the need for automatic verification. Moreover, model checking makes an exhaustive analysis over the system model which is not guaranteed by any other verification method, like simulation for example. The case study of this work also showed us, that besides using model checking for verifying an existing system design, it can be a valuable aid in the design phase of a new system. This was found out as we designed our own experimental environment model for studying the modelling of a system design. Finally, while model checking is used to verify a system consisting of a physical environment alongside a controller, the process of building a model of the whole system compels one to think very thoroughly of assumptions on the behaviour and features of the whole system. This issue can be quite crucial especially in situations where verification of a system is done by an external evaluator who lacks specific domain knowledge.

This work has presented also a general methodology for model checking safety instrumented systems. We have presented an abstract model which captures the structure of the systems into which our methodology can be applied to. With the aid of this model, we have specified what kinds of issues have to be considered while modelling a SIS. Most importantly, we have shown how the generic parts of the abstract model can be modelled by using NuSMV. We have also given general guidelines for modelling the parts which are application specific and cannot be abstracted to any single model.

Our methodology was created by generalising the modelling techniques that were used with the arc protection case study. Although the methodology is based only on a single case study, we believe that it can be a helpful starting point for model checking other safety instrumented systems since the basic structure of most safety instrumented systems seems to be quite similar.

7.1 Future Work

The work done in this report can be extended to many directions. First of all, the practical value of the model checking methodology presented in this work should be evaluated against other case studies. Especially the applicability of the counter technique that was used with the arc protection case should be tested with other safety instrumented systems. If the technique shows to be infeasible with large portion of real-world systems, different techniques for abstracting time delays are needed.

A further study related to modelling an UPS system is already planned. Through this new case study, the purpose is to study, among other things, a far more complicated control logic that the one encountered with the Falcon arc protection system.

Finally, with different industrial case studies being modelled, also comparative studies with real-time model checkers [4], such as UPPAAL, are welcomed for evaluating the differences in performance and applicability between the different modelling approaches.

Acknowledgements

This report is a reprint of my Master's Thesis. This work has been prepared under the research project Model-based safety evaluation of automation systems (MODSAFE) which is part of the Finnish Research Programme on Nuclear Power Plant Safety 2007–2010 (SAFIR2010). I want to express my gratitude to my supervisor Prof. Ilkka Niemelä for offering me the opportunity to work in this research project that has been educative and rewarding for me in numerous ways. Moreover, I thank warmly both Prof. Niemelä and my instructor Docent Keijo Heljanko for the excellent guidance they have given me during the preparation of this Thesis. I have also highly appreciated the friendly and competent working atmosphere at the former TCS laboratory which has recently merged into the new-found Department of Information and Computer Science. For this, my compliments go to all the members of the former TCS lab, and especially, to my closest colleagues Jussi Lahtinen and Jori Dubrovin.

Finally, I am deeply indebted to my beloved family as they have been extremely supportive throughout the whole long period I have studied at TKK.

[1] NuSMV Model Checker v.2.4.2, 2007. Available from `http://nusmv.irst.itc.it/`.

[2] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg. Verification of PLC programs given as sequential function charts. In Ehrig et al. [12], pages 517–540.

[3] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell. A unifying semantics for sequential function charts. In Ehrig et al. [12], pages 400–418.

[4] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer, 2003.

[5] H. Brinksma and A. H. Mader. Verification and optimization of a PLC control schedule. In K. Havelund, J. Penix, and W. Visser, editors, *7th Int. SPIN Workshop on Model Checking of Software, Stanford Univ., California*, volume 1885 of *Lecture Notes in Computer Science*, pages 73–92, Berlin, August 2000. Springer-Verlag.

[6] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and Ph. Schnoebelen. Towards the automatic verification of PLC programs written in Instruction List. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC 2000)*, pages 2449–2454, Nashville, Tennessee, USA, Oct. 2000. Argos Press.

[7] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev. *NuSMV 2.4 User Manual*. ITC-IRST, `http://nusmv.irst.itc.it/`.

[8] J. Cho, J. Yoo, and S. D. Cha. NuEditor - A tool suite for specification and verification of NuSCR. In W. Dosch, R. Y. Lee, and C. Wu, editors, *SERA*, volume 3647 of *Lecture Notes in Computer Science*, pages 19–28. Springer, 2004.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[10] H. Dierks. PLC-Automata: A new class of implementable real-time automata. In M. Bertran and T. Rus, editors, *4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, volume 1231 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 1997.

[11] H. Dierks, A. Fehnker, A. H. Mader, and F. W. Vaandrager. Operational and logical semantics for polling real-time systems. In P. R. Anders and H. Rischel, editors, *7th Int. Symp. on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT), Lyngby, Denmark*, volume 1486 of *Lecture Notes in Computer Science*, pages 29–40, Berlin, September 1998. Springer-Verlag.

[12] H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, editors. *Integration of Software Specification Techniques for Applications in Engineering, Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*, volume 3147 of *Lecture Notes in Computer Science*. Springer, 2004.

[13] G. Frey and L. Litz. Formal methods in PLC programming. In *IEEE International Conference on Systems, Man and Cybernetics (SMC 2000)*, pages 2431–2436, Nashville (TN), USA, Oct. 2000.

[14] W. M. Goble and H. Cheddie. *Safety Instrumented Systems Verification: Practical Probabilistic Calculation*. ISA, 2005.

[15] H.-M. Hanisch, J. Thieme, A. Lüder, and O. Wienhold. Modelling of PLC behaviour by means of timed net Condition/Event systems. In *Proc. of IEEE Int. Symposium on Emerging Technologies and Factory Automation (EFTA '97)*, pages 361–369, 1997.

[16] M. Heiner and T. Menzel. A Petri net semantics for the PLC language Instruction List. In *Workshop on Discrete Event Systems (WODES '98)*, pages 161–166. IEE Control, 1998.

[17] K. Heljanko, T. Junttila, and T. Latvala. Incremental and Complete Bounded Model Checking for Full PLTL. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 98–111. Springer, 2005.

[18] R. Huuck. *Software Verification for Programmable Logic Controllers*. PhD thesis, Christian-Albrechts-Universität zu Kiel, Germany, 2003.

[19] F. Jiménez-Fraustro and E. Rutten. A synchronous model of the PLC programming language ST. In *Proceedings of the Work In Progress session, 1st Euromicro Conference on Real-Time Systems, ERTS'99, York, England*, pages 21–24, 1999.

[20] F. Jiménez-Fraustro and Éric Rutten. A synchronous model of IEC 61131 PLC languages in SIGNAL. In *Euromicro Conference on Real-Time Systems*, pages 135–142. IEEE Computer Society, 2001.

[21] K. Loeis, M. Bani Younis, and G. Frey. Application of symbolic and bounded model checking to the verification of logic control systems. In *10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005)*, pages 247–250, Catania, Italy, Sept. 2005.

[22] A. Mader. Precise timing analysis of PLC applications two small examples. Unpublished manuscript. Available from `http://citeseer.ist.psu.edu/mader00precise.html`.

[23] A. H. Mader. A classification of PLC models and applications. In R. Boel and G. Stremersch, editors, *5th Int. Workshop on Discrete Event Systems, Analysis and Control, Ghent, Belgium*, pages 239–247, Boston, Massachusetts, August 2000. Kluwer Academic Publishers.

[24] A. H. Mader. What is the method in applying formal methods to PLC applications? In S. Engel, S. Kowalewski, and J. Zaytoon, editors, *4th Int. Conf. Automation of Mixed Processes: Hybrid Dynamic Systems, Dortmund, Germany*, pages 165–171, Aachen, Germany, 2000. Shaker Verlag.

[25] A. H. Mader, H. Brinksma, H. Wupper, and N. Bauer. Design of a PLC control program for a batch plant - VHS Case Study 1. *European Journal of Control*, 7(4):416–439, 2001.

[26] A. H. Mader and H. Wupper. Timed automaton models for simple programmable logic controllers. In *11th Euromicro Conf. on Real-Time Systems, York, UK*, pages 114–122, Los Alamitos, California, June 1999. IEEE Computer Society.

[27] O. Maler. On the programming of industrial computers. VHS deliverable in Workpackage IP.1, May 1999.

[28] I. Moon. Modeling programmable logic controllers for logic verification. *IEEE Control Systems Magazine*, 14(2):53–59, 1994.

[29] E.-R. Olderog. Correct Real-Time Software for Programmable Logic Controllers. In *Correct System Design - Recent Insights and Advances*, volume 1710 of *Lecture Notes in Computer Science*, pages 342–362. Springer, 1999.

[30] E.-R. Olderog and H. Dierks. Moby/RT: A Tool for Specification and Verification of Real-Time Systems. *Journal of Universal Computer Science*, 9(2):88–105, Feb. 2003.

[31] O. Rossi and Ph. Schnoebelen. Formal modeling of timed function blocks for the automatic verification of Ladder Diagram programs. In S. Engell, S. Kowalewski, and J. Zaytoon, editors, *Proceedings of the 4th International Conference on Automation of Mixed Processes: Hybrid Dynamic Systems (ADPM 2000)*, pages 177–182, Dortmund, Germany, Sept. 2000. Shaker Verlag.

[32] N. R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[33] J. Tapken and H. Dierks. MOBY/PLC - graphical development of PLC-Automata. In A. P. Ravn and H. Rischel, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1486 of *Lecture Notes in Computer Science*, pages 311–314. Springer, 1998.

[34] A. L. Turk, S. T. Probst, and G. J. Powers. Verification of real time chemical processing systems. In O. Maler, editor, *Hybrid and Real-Time Systems*, volume 1201 of *Lecture Notes in Computer Science*, pages 259–272. Springer, 1997.

[35] Urho Tuominen OY. Example diagrams provided by UTU.

[36] Urho Tuominen OY. UTU-Falcon valokaarisuojausjärjestelmä: asennus- ja käyttöohje.

[37] J. Valkonen, I. Karanta, M. Koskimies, K. Heljanko, I. Niemelä, D. Sheridan, and R. E. Bloomfield. VTT Working Papers 94. NPP Safety Automation Systems Analysis: State of the Art – MODSAFE 2007 Work Report. Technical report, 2008.

[38] J. Valkonen, V. Pettersson, K. Björkman, J.-E. Holmberg, M. Koskimies, K. Heljanko, and I. Niemelä. VTT Working Papers 93. Model-Based Analysis of an Arc Protection and an Emergency Cooling System – MODSAFE 2007 Work Report. Technical report, 2008.

[39] H. X. Willems. Compact timed automata for PLC programs. Technical report, University of Nijmegen, Computing Science Institute (Netherlands), 1999. Available from `ftp://ftp.inrialpes.fr/pub/vasy/publications/others/Willems-99.pdf`.

[40] J. Yoo, S. Cha, C. H. Kim, and D. Y. Song. Synthesis of FBD-based PLC design from NuSCR formal specification. *Reliability Engineering and System Safety*, 87(2):287–294, 2005.

[41] J. Yoo, S. Cha, H. S. Son, C. H. Kim, and J.-S. Lee. PLC-Based safety critical software development for nuclear power plants. In M. Heisel, P. Liggesmeyer, and S. Wittmann, editors, *Computer Safety, Reliability and Security*, volume 3219 of *Lecture Notes in Computer Science*, pages 155–165. Springer, 2004.

[42] J. Yoo, T. Kim, S. D. Cha, J.-S. Lee, and H. S. Son. A formal software requirements specification method for digital nuclear plant protection systems. *Journal of Systems and Software*, 74(1):73–83, 2005.

[43] M. B. Younis and G. Frey. Formalization of existing PLC programs: A survey. In *Proceedings of Computing Engineering in Systems Applications 2003. Paper No.S2-R-00-0239*, Lille, France, July 2003.

A FULL SOURCE CODE OF THE NUSMV MODEL — CASE 1

```
--------------------------------------------------------------------------
MODULE Falcon(ch1,ch2,ch3,ch4,lights)
VAR
  triac1 : boolean;
  triac2 : boolean;
  triac3 : boolean;
  relay6 : boolean;

DEFINE
  or_gate0 := ch1 | ch3;

  and_gate0 := or_gate0 & ch2;
  and_gate1 := or_gate0 & ch4;
  and_gate2 := or_gate0 & lights;

  or_gate1 := and_gate0 | and_gate1 | and_gate2;
  or_gate2 := and_gate1 | and_gate2;
  or_gate3 := and_gate0 | and_gate1;
  or_gate4 := and_gate0 | and_gate1 | and_gate2;


ASSIGN
  init(triac1) := 0;
  init(triac2) := 0;
  init(triac3) := 0;
  init(relay6) := 0;

  next(triac1) := or_gate1;
  next(triac2) := or_gate2;
  next(triac3) := or_gate3;
  next(relay6) := or_gate4;



--------------------------------------------------------------------------
MODULE TruthTable(ch1,ch2,ch3,ch4,lights)
VAR
  triac1 : boolean;
  triac2 : boolean;
  triac3 : boolean;
  relay6 : boolean;

ASSIGN
  init(triac1) := 0;
  init(triac2) := 0;
  init(triac3) := 0;
  init(relay6) := 0;

  next(triac1) :=
    case
      -- Truth table rows with output value 0.
      !ch1 & !ch2 & !ch3 & !ch4 & !lights : 0;   -- row 1
      !ch1 & !ch2 & !ch3 & !ch4 & lights  : 0;   -- row 2
      !ch1 & !ch2 & !ch3 & ch4  & !lights : 0;   -- row 3
      !ch1 & !ch2 & !ch3 & ch4  & lights  : 0;   -- row 4
      !ch1 & !ch2 & ch3  & !ch4 & !lights : 0;   -- row 5


      !ch1 & ch2  & !ch3 & !ch4 & !lights : 0;   -- row 9
      !ch1 & ch2  & !ch3 & !ch4 & lights  : 0;   -- row 10
      !ch1 & ch2  & !ch3 & ch4  & !lights : 0;   -- row 11
      !ch1 & ch2  & !ch3 & ch4  & lights  : 0;   -- row 12

      ch1  & !ch2 & !ch3 & !ch4 & !lights : 0;   -- row 17
      ch1  & !ch2 & ch3  & !ch4 & !lights : 0;   -- row 21
      -- Truth table rows with output value 1.
      1                                   : 1;
    esac;

  next(triac2) :=
    case
      -- Truth table rows with output value 0.
      !ch1 & !ch2 & !ch3 & !ch4 & !lights : 0;   -- row 1
      !ch1 & !ch2 & !ch3 & !ch4 & lights  : 0;   -- row 2
      !ch1 & !ch2 & !ch3 & ch4  & !lights : 0;   -- row 3
      !ch1 & !ch2 & !ch3 & ch4  & lights  : 0;   -- row 4
      !ch1 & !ch2 & ch3  & !ch4 & !lights : 0;   -- row 5

      !ch1 & ch2  & !ch3 & !ch4 & !lights : 0;   -- row 9
      !ch1 & ch2  & !ch3 & !ch4 & lights  : 0;   -- row 10
      !ch1 & ch2  & !ch3 & ch4  & !lights : 0;   -- row 11
      !ch1 & ch2  & !ch3 & ch4  & lights  : 0;   -- row 12
      !ch1 & ch2  & ch3  & !ch4 & !lights : 0;   -- row 13

      ch1 & !ch2 & !ch3 & !ch4 & !lights  : 0;   -- row 17
      ch1 & !ch2 & ch3  & !ch4 & !lights  : 0;   -- row 21
      ch1 & ch2  & !ch3 & !ch4 & !lights  : 0;   -- row 25
      ch1 & ch2  & ch3  & !ch4 & !lights  : 0;   -- row 29
      -- Truth table rows with output value 1.
      1     : 1;
    esac;


  next(triac3) :=
    case
      -- Truth table rows with output value 0.
      !ch1 & !ch2 & !ch3 & !ch4 & !lights : 0;   -- row 1
      !ch1 & !ch2 & !ch3 & !ch4 & lights  : 0;   -- row 2
      !ch1 & !ch2 & !ch3 & ch4  & !lights : 0;   -- row 3
      !ch1 & !ch2 & !ch3 & ch4  & lights  : 0;   -- row 4
      !ch1 & !ch2 & ch3  & !ch4 & !lights : 0;   -- row 5
      !ch1 & !ch2 & ch3  & !ch4 & lights  : 0;   -- row 6

      !ch1 & ch2  & !ch3 & !ch4 & !lights : 0;   -- row 9
      !ch1 & ch2  & !ch3 & !ch4 & lights  : 0;   -- row 10
      !ch1 & ch2  & !ch3 & ch4  & !lights : 0;   -- row 11
      !ch1 & ch2  & !ch3 & ch4  & lights  : 0;   -- row 12

      ch1  & !ch2 & !ch3 & !ch4 & !lights : 0;   -- row 17
      ch1  & !ch2 & !ch3 & !ch4 & lights  : 0;   -- row 18
      ch1  & !ch2 & ch3  & !ch4 & !lights : 0;   -- row 21
      ch1  & !ch2 & ch3  & !ch4 & lights  : 0;   -- row 22
      -- Truth table rows with output value 1.
      1                                   : 1;
    esac;

  next(relay6) :=
    case
      -- Truth table rows with output value 0.
      !ch1 & !ch2 & !ch3 & !ch4 & !lights : 0;   -- row 1
      !ch1 & !ch2 & !ch3 & !ch4 & lights  : 0;   -- row 2
      !ch1 & !ch2 & !ch3 & ch4  & !lights : 0;   -- row 3
      !ch1 & !ch2 & !ch3 & ch4  & lights  : 0;   -- row 4
      !ch1 & !ch2 & ch3  & !ch4 & !lights : 0;   -- row 5

      !ch1 & ch2  & !ch3 & !ch4 & !lights : 0;   -- row 9
      !ch1 & ch2  & !ch3 & !ch4 & lights  : 0;   -- row 10
      !ch1 & ch2  & !ch3 & ch4  & !lights : 0;   -- row 11
      !ch1 & ch2  & !ch3 & ch4  & lights  : 0;   -- row 12

      ch1  & !ch2 & !ch3 & !ch4 & !lights : 0;   -- row 17
      ch1  & !ch2 & ch3  & !ch4 & !lights : 0;   -- row 21
      -- Truth table rows with output value 1.
      1                                   : 1;
    esac;


--------------------------------------------------------------------------
MODULE main
VAR
  ch1 : boolean;
  ch2 : boolean;
  ch3 : boolean;
  ch4 : boolean;
  lights : boolean;
  falcon : Falcon(ch1,ch2,ch3,ch4,lights);
  truth_table : TruthTable(ch1,ch2,ch3,ch4,lights);


ASSIGN
  init(ch1) := {0,1};
  init(ch2) := {0,1};
  init(ch3) := {0,1};
  init(ch4) := {0,1};
  init(lights) := {0,1};

  next(ch1) := {0,1};
  next(ch2) := {0,1};
  next(ch3) := {0,1};
  next(ch4) := {0,1};
  next(lights) := {0,1};


--------------------------------------------------------------------------
-- Specification of properties

-- The outputs of the modules have to be equal with all inputs.
LTLSPEC G ((falcon.triac1 <-> truth_table.triac1) &
           (falcon.triac2 <-> truth_table.triac2) &
           (falcon.triac3 <-> truth_table.triac3) &
           (falcon.relay6 <-> truth_table.relay6))
```

B FULL SOURCE CODE OF THE NUSMV MODEL — CASE 2

```
-- M4 preprocessor macros:

define(‘AD’,‘6’)
define(‘D1’,‘14’)
define(‘D2’,‘21’)
define(‘D3’,‘7’)
define(‘D4’,‘14’)

define(‘max’, ‘ifelse(eval($1 > $2),‘1’,‘$1’,‘$2’)’)
define(‘DELAY_RANGE_UPPER_LIMIT’, ‘eval(max(max(max(D1,D2),D3),D4)+1)’)
define(‘TIMER_RANGE_UPPER_LIMIT’, ‘AD’)


---------------------------------------------------------------------------
-- Delay module is used to model the delay gates of the tripping logic
-- of the Falcon master unit.
-- With delay=0 the relay acts in one cycle. The delay
-- parameter specifies how many additional scan cycles the input has
-- to be TRUE before an output signal TRUE is given.

MODULE Delay(input_signal, delay)
VAR
  count : 0..DELAY_RANGE_UPPER_LIMIT;
  output : boolean;

DEFINE
  -- Total delay consist of the delay + scan cycle
  total_delay := delay + 1;

ASSIGN
  init(count) := 0;
  next(count) :=
    case
      input_signal = 0        : 0;
      count >= total_delay    : count;
      1                       : count + 1;
    esac;

  init(output) := 0;
  next(output) :=
    case
      -- At the step when count = delay, output has to be 1.
      next(count) >= total_delay   : 1;
      1                            : 0;
    esac;


---------------------------------------------------------------------------
-- OneShotTimer module is used by the Breaker module to model the physical
-- activation delay of a breaker. The module implements such timer which
-- can only run once through its value range. This restriction is made to
-- keep the state space of the model as small as possible. This behaviour
--- also corresponds to the operational model of breaker.


MODULE OneShotTimer(signal,delay)
VAR
  counter : 0..TIMER_RANGE_UPPER_LIMIT;

DEFINE
  output :=
    case
      (delay = 0) : signal;
      (counter = 0) : 1;
      1 : 0;
    esac;

ASSIGN
  init(counter) := delay;
  next(counter) :=
    case
```

```
      (delay = 0) : 0;
      (counter = 0) : 0;
      (counter < delay) : counter - 1;
      (counter = delay) & (signal = 1) : counter - 1;
      1 : counter;
    esac;


-------------------------------------------------------------------------
-- Breaker module is used to model the physical circuit breakers
-- controlled by the Falcon master unit.

MODULE Breaker(launch_signal, setting_up_time, can_break)
VAR
  is_broken : boolean;
  cuts : boolean;
  timer : OneShotTimer(launch_signal,setting_up_time);

DEFINE
  launched := launch_signal;

ASSIGN
  init(is_broken) := 0;
  next(is_broken) :=
    case
      can_break = 0 : 0;
      is_broken = 0 : {0,1};
      is_broken = 1 : 1;
      1             : 1;
    esac;

  init(cuts) := timer.output;
  next(cuts) :=
    case
      (cuts = 1) : 1;
      (is_broken = 1) : cuts;
      (next(timer.output) = 1) : 1;
      (next(timer.output) = 0) : 0;
    esac;





-------------------------------------------------------------------------
-- UTU_CR module is used to model the overcurrent sensors of the Falcon
-- system.

MODULE UTU_CR(has_voltage,breaker)
VAR
  overcurrent : boolean;

ASSIGN
  overcurrent :=
    case
      !has_voltage | breaker.cuts  : 0;
      1                            : {0,1};
    esac;



-------------------------------------------------------------------------
-- UTU_ARC module is used to model the light sensors of the Falcon
-- system.

MODULE UTU_ARC()
VAR
  light : boolean;

ASSIGN
  light := {0,1};
```

```
------------------------------------------------------------------------
-- Controller module models the Falcon master unit.

MODULE Controller(ch1,ch2,ch3,ch4,ch_light)
VAR
  triac1_delay : Delay(logic_output_tr1,TRIAC_DELAY);
  triac2_delay : Delay(logic_output_tr2,TRIAC_DELAY);
  triac3_delay : Delay(logic_output_tr3,TRIAC_DELAY);
  triac4_delay : Delay(logic_output_tr4,TRIAC_DELAY);

  relay1_delay : Delay(logic_output_relay1,RELAY1_DELAY);
  relay2_delay : Delay(logic_output_relay2,RELAY2_DELAY);
  relay3_delay : Delay(logic_output_relay3,RELAY3_DELAY);
  relay4_delay : Delay(logic_output_relay4,RELAY4_DELAY);


DEFINE
  -- Delay values of the delay gates. These values should be set to
  -- the delay values (in milliseconds) of the corresponding delay
  -- gates in the modelled tripping logic.

  TRIAC_DELAY := 0;
  RELAY1_DELAY := D1; --Values D1-D4 set with M4 preprocessor macro.
  RELAY2_DELAY := D2;
  RELAY3_DELAY := D3;
  RELAY4_DELAY := D4;

  -- Logic of the circuits.
  OR1 := (ch3 | ch4);
  AND1 := (OR1 & ch_light);
  OR2 := (ch1 | ch2 | AND1);
  OR3 := (ch1 | ch2);

  -- Inputs to delay gates.
  logic_output_tr1 := ch1;
  logic_output_tr2 := ch2;
  logic_output_tr3 := OR2;
  logic_output_tr4 := AND1;

  logic_output_relay1 := OR3;
  logic_output_relay2 := OR3;
  logic_output_relay3 := AND1;
  logic_output_relay4 := AND1;

  -- Outputs of the controller module.
  triac1_output := triac1_delay.output;
  triac2_output := triac2_delay.output;
  triac3_output := triac3_delay.output;
  triac4_output := triac4_delay.output;

  relay1_output := relay1_delay.output;
  relay2_output := relay2_delay.output;
  relay3_output := relay3_delay.output;
  relay4_output := relay4_delay.output;



------------------------------------------------------------------------
-- main module is the main program of the whole model and it encompasses
-- both, the model of the controller and its environment.

MODULE main
VAR
  -- The controller of Falcon master unit
  ctrl : Controller(zone1_alarm, zone2_alarm, Cr_3a.overcurrent,
                    Cr_3b.overcurrent, L_3.light);

  -- Overcurrent sensors
  Cr_1  : UTU_CR(zone1_hasvoltage,breaker_A);
  Cr_2  : UTU_CR(zone2_hasvoltage,breaker_B);
  Cr_3a : UTU_CR(zone3_hasvoltage,breaker_C);
  Cr_3b : UTU_CR(zone3_hasvoltage,breaker_D);
  -- Light sensors
  L_1 : UTU_ARC();
  L_2 : UTU_ARC();
  L_3 : UTU_ARC();

  -- Primary breakers
  breaker_A : Breaker(ctrl.triac1_output, BREAKER_OPENING_TIME, CAN_BREAK_DOWN);
  breaker_B : Breaker(ctrl.triac2_output, BREAKER_OPENING_TIME, CAN_BREAK_DOWN);
  breaker_C : Breaker(ctrl.triac3_output, BREAKER_OPENING_TIME, CAN_BREAK_DOWN);
  breaker_D : Breaker(ctrl.triac4_output, BREAKER_OPENING_TIME, CAN_BREAK_DOWN);

  -- Backup breakers
  breaker_E : Breaker(ctrl.relay1_output, BREAKER_OPENING_TIME, CAN_NOT_BREAK_DOWN);
  breaker_F : Breaker(ctrl.relay2_output, BREAKER_OPENING_TIME, CAN_NOT_BREAK_DOWN);
  breaker_G : Breaker(ctrl.relay3_output, BREAKER_OPENING_TIME, CAN_NOT_BREAK_DOWN);
  breaker_H : Breaker(ctrl.relay4_output, BREAKER_OPENING_TIME, CAN_NOT_BREAK_DOWN);

DEFINE
  -- The activation delay of the breakers
  -- (=the time passed from receiving a launch
  -- signal to cutting off the power.) With each breaker, the
  -- value should be set to the value of the activation delay in
  -- milliseconds of the corresponding real circuit breaker.
  BREAKER_OPENING_TIME := AD; --Value set with M4 preprocessor macro.

  -- Define constants for specifying whether
  -- a breaker can be malfunctioned.
  CAN_BREAK_DOWN := 1;
  CAN_NOT_BREAK_DOWN := 0;

  -- The alarm model
  zone1_alarm := Cr_1.overcurrent & L_1.light;
  zone2_alarm := Cr_2.overcurrent & L_2.light;
  zone3_alarm := (Cr_3a.overcurrent | Cr_3b.overcurrent) & L_3.light;

  -- The current flow model
  zone1_hasvoltage :=
    !(breaker_A.cuts |
    ((breaker_E.cuts | breaker_H.cuts) &
    (breaker_C.cuts | breaker_D.cuts | breaker_F.cuts | breaker_G.cuts)));

  zone2_hasvoltage :=
    !(breaker_B.cuts |
    ((breaker_E.cuts | breaker_H.cuts) &
    (breaker_C.cuts | breaker_D.cuts | breaker_F.cuts | breaker_G.cuts)));

  zone3_hasvoltage :=
    !((breaker_C.cuts | breaker_E.cuts | breaker_H.cuts) &
    (breaker_D.cuts | breaker_F.cuts | breaker_G.cuts));


--------------------------------------------------------------------------
-- Specification of properties:

-- 1. Connections of the primary breakers have to be correct.
LTLSPEC G (breaker_A.launched -> Y zone1_alarm)
LTLSPEC G (breaker_B.launched -> Y zone2_alarm)
LTLSPEC G (breaker_C.launched -> Y (zone1_alarm | zone2_alarm | zone3_alarm))
LTLSPEC G (breaker_D.launched -> Y zone3_alarm)

-- 2. Connections of the backup breakers have to be correct.
LTLSPEC G (breaker_E.launched -> (breaker_A.launched | breaker_B.launched))
LTLSPEC G (breaker_F.launched -> (breaker_E.launched & breaker_C.launched))
LTLSPEC G (breaker_G.launched -> (breaker_D.launched))
LTLSPEC G (breaker_H.launched -> (breaker_G.launched))

-- 3. Backup breakers must not be launched too easily.
LTLSPEC G (breaker_E.launched -> (breaker_A.is_broken | breaker_B.is_broken))
LTLSPEC G (breaker_F.launched -> (breaker_A.is_broken | breaker_B.is_broken))
LTLSPEC G (breaker_F.launched -> (breaker_C.is_broken))
LTLSPEC G (breaker_G.launched -> (breaker_C.is_broken | breaker_D.is_broken))
LTLSPEC G (breaker_H.launched -> (breaker_C.is_broken))
```

```
-- 4. The system has to terminate a continuous electric arc.
LTLSPEC G (zone1_alarm -> F (!zone1_alarm | !zone1_hasvoltage))
LTLSPEC G (zone2_alarm -> F (!zone2_alarm | !zone2_hasvoltage))
LTLSPEC G (zone3_alarm -> F (!zone3_alarm | !zone3_hasvoltage))
```

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-R1	Nikolaj Tatti, Hannes Heikinheimo Decomposable Families of Itemsets. May 2008.
TKK-ICS-R2	Ville Viitaniemi, Jorma Laaksonen Evaluation of Techniques for Image Classification, Object Detection and Object Segmentation. June 2008.
TKK-ICS-R3	Jussi Lahtinen Model Checking Timed Safety Instrumented Systems. June 2008.
TKK-ICS-R4	Jani Lampinen Interface Specification Methods for Software Components. June 2008.

ISBN 978-951-22-9477-0 (Print)
ISBN 978-951-22-9478-7 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)