"Safety Automata" – A new Specification Language for the Development of PLC Safety Applications

Georg Frey
Saarland University, Chair of Automation
Saarbrücken, Germany
georg.frey@aut.uni-saarland.de

Bastian Schlich
ABB Corporate Research Germany
Ladenburg, Germany
bastian.schlich@de.bb.com

Rainer Drath
ABB Corporate Research Germany
Ladenburg, Germany
rainer.drath@de.abb.com

Robert Eschbach
ITK Engineering AG
Herxheim, Germany
robert.eschbach@itk-engineering.de

Abstract

This contribution defines for the first time „safety automata", a specification language for safety control functions. This fills an important gap in the method tool box of the automation engineer. The definition has a profound potential to broaden the use of automata and their well-known advantages in industrial applications. The advantages and application of safety automata are explained by example via the function block SF_Equivalent of the PLCopen specification. Stepwise, the authors illustrate the specification of this function block by means of safety automata, the transformation rules to implement the automata in fully functional PLC code, and finally the generation of test cases that allow checking both the automata and the PLC code.

1 Motivation

In the academic world, automata are well explored. They offer manifold and attractive analysis possibilities. Unfortunately, automata quickly reach a high complexity, which is hard to control for real industrial applications. This is the main reason that automata have not been established in the automation industry yet. Particularly for safety functions with inherently low complexity, automata lend themselves as a specification language.

The application of automata requires a "state based thinking", which has already been established in the automation technology for decades, especially in PLC programming (SFC, StateCharts, StateFlow, etc.). PLCopen already utilizes automata for the specification of safety function blocks [1], but the underlying automata class is undefined. Furthermore, there are no hints to the implementation or for the test of the safety function blocks. The interpretation of the PLCopen automata and the method for implementation and test of the safety function blocks must be worked out by each user individually. Since automata offer promising means for automatic code generation, test, and verification, the authors already developed a method for the application of automata for developing safety function blocks in [2] and [3]. However, the basis is missing: there is no dedicated automata class available or defined in literature.

This paper presents for the first time a profound definition of "safety automata" as a systematic and reliable basis for using automata for the specification of safety-related function blocks in industrial applications. Furthermore, this paper provides transformation rules for the implementation of safety automata in PLC code using the PLC programming language SFC and illustrates this by means of a prototypical implementation of the SF_Equivalent function block. Finally, a method to generate test cases is presented.

2 Safety Automata

2.1 Requirements

The definition of safety automata fulfills the following four main goals:

1. Unambiguity: Precise definition of syntax and semantics.
2. Simplicity: Limitation to necessary elements in order to ease the communication between developers, end-users, and certification bodies.
3. Executability: Domain specific definition oriented at PLC languages to allow a simple, error-free, and

possibly automatic translation into an executable implementation language on PLCs.
4. Testability: Allowing application of algorithms for automated verification and test case generation.

Discussions with domain experts about the functionality necessary to implement safety functions yielded the following four main results:

1. A clear distinction between input and output signals is necessary to avoid hidden feedback structures.
2. The output assignment (current value of all output variables) has to be clearly described in each state of the automaton. In the visualization this assignment should be directly visible.
3. Hierarchical and modular structures are not strictly necessary since most safety functions are quite simple.
4. Concurrency, as present, e.g., in SFC, should be avoided as it leads to complex structures.

These requirements imply that the desired description means should be based on Moore automata where the outputs are associated with states and the inputs are associated with transitions. As extensions to the basic concept, priorities, timing conditions, and a predefined notation for activation and deactivation of the safety function are required. The use of transition priorities leads to more compact transition conditions and is a well-accepted and favored means by application engineers. As a recent example for this demand, see the currently revised version of IEC 61499, where the ECC will be extended with explicit priorities on transitions. Regarding executability, this extension is not critical since implementation languages like SFC also use priorities. Regarding testability, priorities can be removed systematically or even automatically by extending the firing conditions accordingly whenever an analysis tool does not support priorities: For two given transitions T1 und T2 from the same state with transition conditions B1 und B2 a higher priority of T1 is reached by changing the condition for T2 to B2new = B2 AND NOT B1. This procedure can be extended directly to any number of transitions.

In safety functions, transition conditions related to time are limited to waiting times, i.e., time elapsed since the current state has been activated or equivalently since the last transition has been fired. Here, it is important to provide a clear definition regarding the start and the reset of timers and especially regarding the allowed usage of time information in transition conditions. In addition, executability and testability should be considered carefully in this definition. Therefore, the time that the automaton already stayed in the current state could be used in conditions on transitions. This is equivalent to the step timer used in SFC (StepName.T). The corresponding clock is started at activation of the step and reset (and stopped) at deactivation. Querying of this timer is only allowed and useful in transitions originating at the corresponding state.

To provide uniform activation and deactivation special variables should be defined.

2.2 Specification

Based on the requirements given above, the safety automaton is defined as a Moore automaton in the form of an input/output automaton with Boolean input and output signals (I/O automaton). I/O automata provide input and output signals as they are used in logic control design instead of events used in classical automata theory, see, e.g., [10] for a comparison. The automaton has a finite number of **states** with one specified **initial state**. In a graphical representation, states are described by circles or boxes where the initial state is highlighted by a bold or double border.

Possible **transitions** between states are described in the graphical notion by arcs from the source state to the destination state. A **transition condition** is a Boolean expression on the set of input variables. Finally, the automaton specifies the values of output variables in each state.

The I/O automaton following the standard definition so far needs the following four extensions to fulfill all the stated requirements:

1. Assignment of Priorities in the form on non-negative integers to transition where 0 is the highest priority.

   This extension makes specification of transition conditions more compact and increases readability. If necessary for consistency checks, implementation or test-case generation priorities can be easily removed by extending the conditions as discussed above.

2. Optional use of the time that the current state is already active in the transition condition by comparing it to a given value.

   This extends the modeling power of I/O automata. We assume that in the activation of a new state, a clock τ is started. In outgoing transitions, the value of this clock can be compared to a pre-defined constant T_{min} in the form $\tau \geq T_{min}$. Other comparative operators are neither useful nor necessary in safety automata. Especially, the check whether the state timer reads a value smaller than a given constant can lead to a deadlock in the system, which should be avoided.

3. Definition of one special input variable to activate and deactivate the safety function and one special output variable to indicate the activity status in the interface. Furthermore, the definition of rules for the deactivation.

This extension does not change the functionality of a safety automaton as defined so far. However, it eases the handling of systems composed of different safety automata considerably especially if for activation variables and status variables the same names are used throughout a function block library.

4. The safety automaton allows only Boolean input and output variables.

   The inclusion of other types of input and output variables would extend the modeling power of safety automata. However, it would heavily decrease the possibilities for automatic testing and verification. In the case of inputs, the transition condition has to be Boolean anyway in the end. The test of real-valued process inputs against limits has to be modeled as a pre-processing step outside the safety automaton. In our view, this increases readability and reduces complexity. A corresponding post-processing of Boolean output signals allows non-Boolean values as well if necessary.

   The active state of a safety automaton is together with the state timer the only state information of the safety function. Hence, knowledge about the current state is sufficient to determine the output of the function block. However, the information about the current state is not available as an output signal in the block interface. PLCopen defines an additional output called Diag-Code for each state. In our view, and in all known examples, those codes are unique. Hence, they convey the same information as the current state. If it is necessary for visualization purposes to transfer this information, it can be done by implementing additional functionality at this point. For the automaton itself, it is not necessary.

2.3 Formal Definition

Syntax: Elements

A Safety Automaton is given by an 11-Tuple $SF = (Z, T, E, A, z_1, e_1, a_1, \tau, o, s, p)$
with

$Z = \{z_1, \ldots z_n\}$	a non-empty finite set of n states
$T = \{t_1, \ldots t_m\}$	a non-empty finite set of m transitions
$E = \{e_1, \ldots e_k\}$	a non-empty finite set of k Boolean input variables
$A = \{a_1, \ldots a_l\}$	a non-empty finite set of l Boolean output variables
$z_1 \in Z$	an initial state
$e_1 \in E$	an activation input
$a_1 \in A$	an activity output
τ	a state clock always indicating the time since the last state change
o	an output function assigning in every state z_i from Z a Boolean value to all output variables a_j from A: $o(z_i, a_j) \in \{True, False\}\ \forall\ i \in \{1, \ldots n\}, j \in \{1, \ldots l\}$
s	a firing function $s(t_i)$, associating every transition t_i from T with a Boolean condition over the set of input variables E and with an optional waiting condition ($\tau \geq$ time value)
p	a priority function $p(t_i)$, assigning an optional priority value from the non-negative integers to each transition where smaller numbers indicate higher priority (0 = highest priority)

Syntax: Constraints

Transitions always connect two different states; loops are not allowed in safety automata. From every state of the safety automaton, the initial state has to be reachable directly via one single transition. Therefore, the set of transitions is composed of two disjoint sub-sets T_{activ} and T_{init} as follows:

$T = T_{activ} \cup T_{init}$
T consists of two sub-sets T_{activ} und T_{init}

$T_{init} = Z \backslash \{z_1\} \times \{z_1\}$
T_{init} contains a transition to the initial z_1 state from every other state

$T_{activ} \subseteq Z \times Z \backslash \{z_1\}$
T_{activ} contains all other transitions

$(zi, zi) \notin T\ \forall\ i \in \{1, \ldots n\}$
There are no loops

The priorities of transitions originating from the same state need to be different:

$j \neq k$ implies $p(z_i, z_j) \neq p(z_i, z_k)\ \forall\ i,j,k \in \{0, \ldots n\}$

When the activation input is set to False, the safety automaton switches to the initial state regardless of the current state and all other input signals. This transition is assigned the highest priority and no further conditions:
$s(t) = NOT\ e1\ \forall\ t \in T_{init}$
$p(t_i) = 0\ \forall\ t \in T_{init}$

The activity output should show in the external interface whether the safety automaton is currently in its initial state.

$o(z_1, a_1) = True$
$o(z_i, a_1) = False\ \forall\ i \in \{2, \ldots l\}$

Semantics: Execution Rules

The basis of the introduced model is a non-autonomous I/O automaton. This means that the execution of the model is seen as synchronized with its environment. In automation systems this is normally realized by a time-driven cyclic call of the modeled function. The following rules apply for the execution:

1. When the automaton is triggered, it is checked whether in the active state, under the current setting of input variables, and – if applicable – the current value of the state timer, a transition can fire (i.e., the transition originates from the current state and its transition condition evaluates to true)
2. If there is one such transition, this transition is fired. If there are several transitions, the one with the highest priority is fired. In both cases, the automaton is set to the new destination state and the state timer is reset and restarted. Furthermore, the output variables are set to the values indicated by the new state.
3. If there is no transition that can be fired, the automaton remains in the currently active state.
4. The automaton always fires only one single transition. There are models such as Grafcet and Interpreted Petri Nets where a maximum step containing the firing of several transitions is executed until a stable state (state where no transition can fire under the current input signal setting) is reached [11]. This behavior leads to transient states and is not transparent for the user. Hence, it should not be used in safety critical systems where simplicity and diagnosability are of prime importance.

2.4 Example

To demonstrate the safety automaton, the safety function block SF_Equivalent defined by the PLCopen is modeled in the following and compared to the original model. Figure shows the interface description of the block as provided by the PLCopen. The function of this block is to check whether the two inputs *S_ChannelA* and *S_ChannelB* are both *TRUE*. In this case, the output *S_Equivalent=TRUE* is set. If one of the inputs is *FALSE* or the block is not activated, *S_Equivalent=FALSE* is set. In addition, several error condition are checked by the block

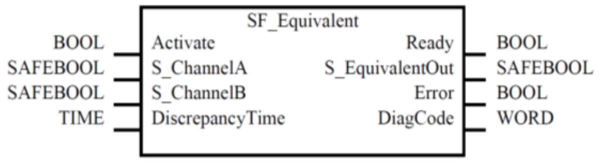

Figure : SF_Equivalent function block from PLCopen [1]

Figure shows the automaton defining the behavior of the SF_Equivalent function block as given by the PLCopen. From the initial state *Idle*, the automaton changes to the state *Init* when the input signal *Activate* is set to true. From now on the output signal *Ready=TRUE* is set and the safety function works. In the presented model, several output assignments and transitions are described only implicitly or by elaborated shorthand. For example: The transition in the top left without source state and priority 0 means that this transition can be taken from every state with the highest priority. The implicit assignment of output variables is described by the dashed lines. For all states above the uppermost dashed line, *Ready=FALSE* holds and for those below *Ready =TRUE* holds. This graphical representation needs too much interpretation for the safety domain.

Figure shows the SF_Equivalent function block modeled as a safety automaton. All transitions and all output signal settings are defined explicitly.

2.5 Comparison to PLCopen automata

As stated above, in the safety automaton, all transitions are modeled explicitly. Furthermore for each state, the values of all output signals are given explicitly. The PLCopen automaton uses implicit and /or informal additional information. This could be a possible source of errors or misinterpretations. To this end, the safety automaton is more formal but also less error-prone.

In the example, the safety automaton needs one more state than the PLCopen automaton. The reason for this is that the PLCopen automaton does not strictly follow the Moore model. In the state *Error*, the PLCopen automaton sets two different ErrorCodes depending on the previous state, i.e., like in a Mealy model depending on the state from which the automaton reached the *Error* state. To model this in a safety automaton, we need two states *Error1* and *Error2*.

Due to additional states like this, the safety automaton may result in a less compact description. However, the Moore property, meaning that output settings can be derived directly from the state information, is essential if the state of the system should be analyzed. Especially, since in most engineering environments the state information will be directly visible in the visualization but not necessarily its history. Looking at this state information should be sufficient for personal to determine which outputs should be active at the moment to detect possible malfunctions directly.

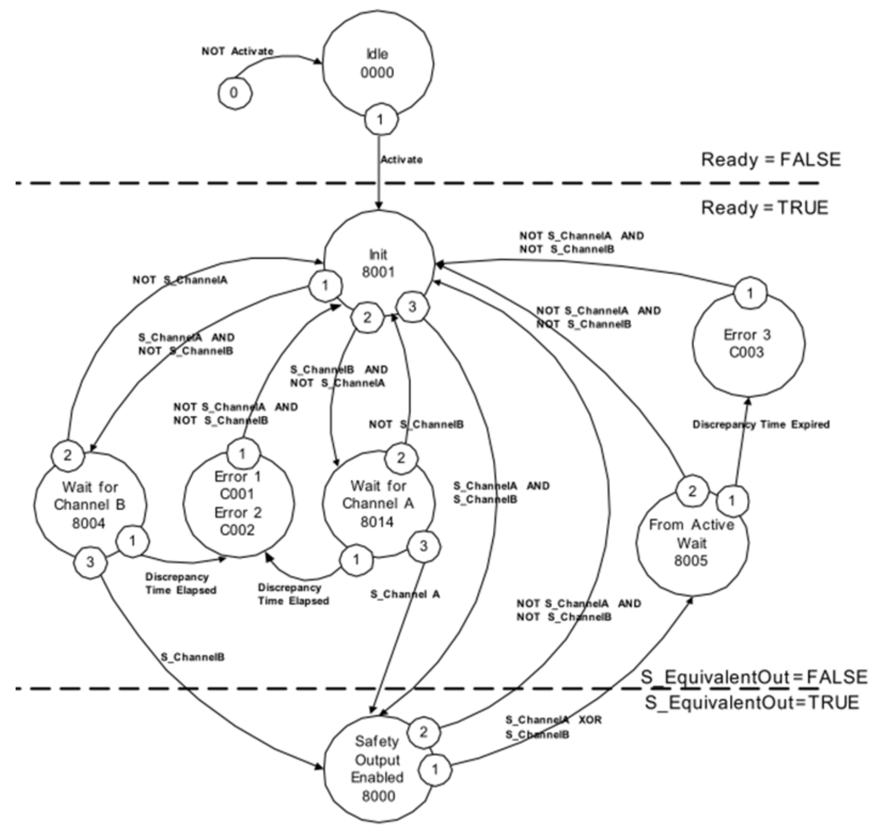

Figure : SF_Equivalent modeled by an automaton according to PLCopen [1]

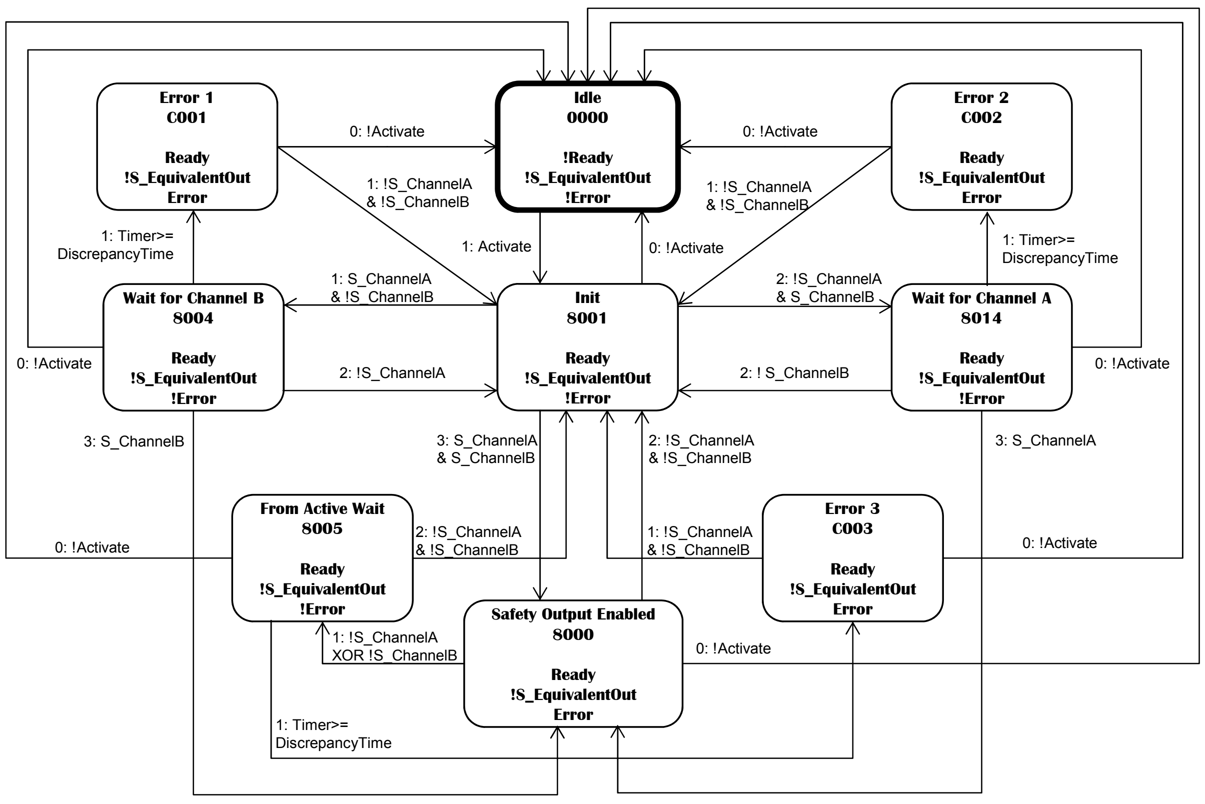

Figure : SF_Equivalent modeled as safety automaton

3 Implementation of Safety Automata

3.1 Transformation rules to SFC

The transformation rules from a safety automaton into SFC are simple. Thanks to considering this step already in the definition of safety automata, there is a one-to-one transformation of elements according to Table .

Element of safety automaton	Corresponding SFC element
state	step
output variable assignment	action
arc	transition
firing condition	transition condition

Table : Transformation rules from safety automaton to SFC

3.2 Example

This section presents the prototypical implementation of the safety function block SF_Equivalent using the PLC programming environment CoDeSys. Figure shows an instance of the function block in CoDeSys. The result is equivalent to the interface definition defined by PLCopen shown in Figure .

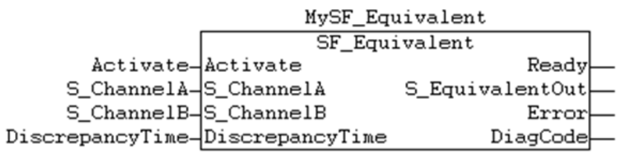

Figure : The safety function block SF_Equivalent described in CoDeSys according to IEC61131

Figure shows the corresponding executable SFC. Each step in this SFC corresponds to one state in the safety automaton. Furthermore, each step assigns values to all output variables of the safety function block. The SFC contains exactly the same transitions as the corresponding safety automaton.

Priorities are modeled in CoDeSys – according to IEC 61131 – by the geometrical order of transitions from left to right where the leftmost transition has the highest priority.

When implementing a safety automaton with SFC – as generally when using SFC – the number of jumps should be kept minimal. In this example, the number could be kept small by grouping the jumps back to the Init state. The names of the steps should ideally correspond to the state names in the safety automaton. The DiagCode in our case is realized by entry actions in the SFC steps.

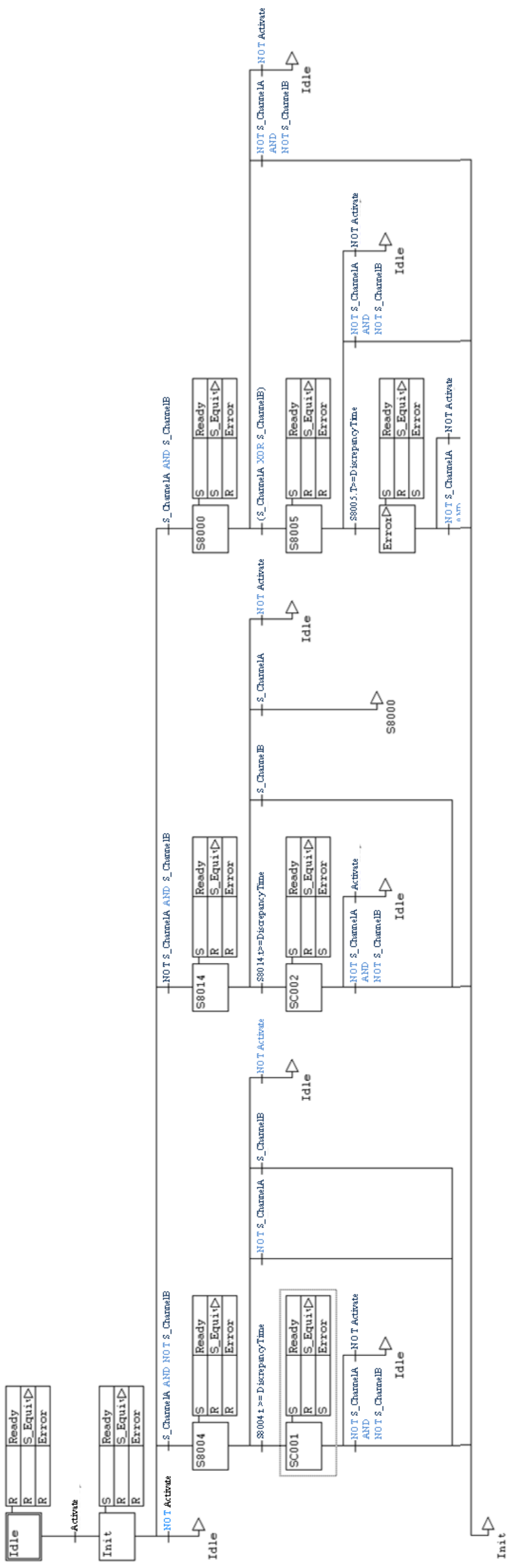

Figure : SF_Equivalent in CoDeSys SFC

4 Testing

4.1 Introduction

This section describes how test cases can be generated automatically from safety automata. The generated test cases can be used for different purposes: they can be used to (i) analyze the automata themself, and (ii) to test an implementation running on either a Soft-SPS or a real SPS in the context of real hardware, firmware, and run time environment. The generation of test cases can be done completely automatically and depending on the desired test depth according to different coverage criteria. The automated test case generation has several advantages: (i) high effort reduction compared to the manual creation of test cases and (ii) high and reproducible quality of test case execution and evaluation. The main reason for this is the direct link between generated test cases and safety automata. Depending on the desired test depth, different coverage criteria can be selected and test cases can be generated w.r.t. these criteria.

We recommend state coverage for safety functions of lower criticality. This recommendation is for all safety functions with safety integrity level 2, following the international safety standard IEC 61508. For safety functions with safety integrity level 3, we recommend complete transition coverage, i.e., all transitions of the safety automata will be covered by test cases at least once. Safety functions with safety integrity level 4 should be covered with MC/DC, the modified condition and decision coverage. Beside these (structural) coverage criteria, there exist also other criteria that can be used for test case generation. For example, it is possible to attach probabilities to transitions and generate test cases according to these probabilities. In this way, the safety automaton describes a Markov chain. The advantage is that well-established and rich mathematical theory of Markov chains and sampling becomes available for precise software quality estimations. More detailed information can be found in [9].

Level	Criterion	Coverage
SIL 1	-	-
SIL 3	transition coverage	100 %
SIL 4	modified condition / decision coverage	100 %

Table : Required test coverage referring to safety-standard IEC 61508

4.2 Test case generation

In our example, function block SF_Equivalent, modeling and test case generation will be realized with graphwalker [5] und yEd [6]. In a first step, the automaton was modelled with yEd based on the safety automaton depicted in Figure .

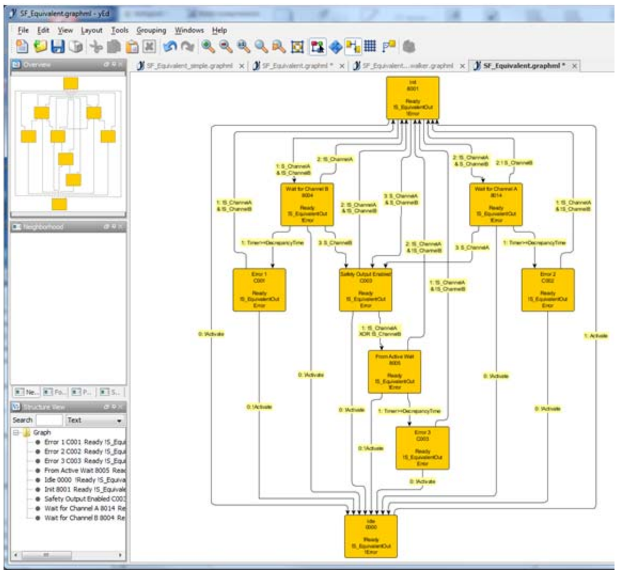

Figure : Safety automaton in the tool yED for SF_Equivalent

In a second step, the yEd model was saved in a special standardized XML format called graphml [8]. The model-based testing tool graphwalker can take models realized in this format as input and allows selecting start state, coverage criterion, and search algorithm for test case generation. Subsequently, the test cases will be generated and saved as text files. In our example, state *Idle* was selected as start state, A*-algorithm as search algorithm, and transition coverage as coverage criterion. Graphwalker produces a single test cases consisting of 50 test steps. Alternatively, the weaker criterion state coverage can be selected. In this case, a single test case is generated consisting of 29 test steps. This test case does not cover all transitions.

Criterion	Covered states	Covered transitions	Length of test case
State coverage	100% (10/10)	57% (15/26)	29
Transition coverage	100% (10/10)	100% (26/26)	50

Table : state and transition coverage statistics

It is possible to introduce special exit states that can be used to reduce the test case length and to increase the number of test cases. In this case, each path from the initial state to one of these exist states is considered to be a single test case. Furthermore, it is possible to focus test case generation on specific parts of the safety automaton. Given the fact that safety automata are usually rather small, we recommend to use complete transition coverage. A test case length of 50 test steps is still acceptable.

The tool graphwalker requires that an automaton is strongly connected, i.e., each state is connected (via a path) with all other states. This condition holds for all reasonable safety automata, especially for the safety automaton representing safety function SF_Equivalent. In a reasonable safety automaton, the initial state should be reachable from all states and each state should be reachable from the initial state.

4.1 Discussion

The generated test cases cover all transitions and hence all states of the safety automaton. Missing states, missing transitions, as well as erroneous transition conditions will be detected.

However, the SPS code could have some additional hidden states or additional transitions that could not be covered by the generated test cases since these additional states and transitions are not part of the safety automaton. In order to overcome this problem, we recommend performing a systematic review with the goal to detect additional states and additional transitions. This does not cause major problems since there exists a one-to-one correspondence between safety automaton and its implementation as SFC.

Execution and documentation of tests has not been treated in this section, but we recommend transforming test cases into test scripts that will be executed on the SPS. In this case, test cases will be exercised on the same target platform as the PLC code. Refer to [2] for more information on this method.

5 Conclusions and Outlook

This contribution defines for the first time the automata class „safety automata“, which closes an important gap in the method tool box of the automation engineer.

Based on safety automata, this contribution illustrates the major steps in the safety development chain by means of the safety function block SF_Equivalent - first the specification by means of the safety automaton, second the transformation of the automaton into a fully functional PLC code (SFC), and finally approaches for the generation of test cases that allow checking the automaton and the PLC code.

The similarity between safety automata and SFC is intended and opens up significant optimization potential in the safety development process. First, PLC code can be automatically generated via a minimal set of rules. One could even do without the automaton if a limited feature set of the SFC itself is used instead of the automaton. Furthermore, test cases can be automatically generated from the automaton. This is an important base for the test of the SFC.

The generation of test cases and the PLC code from the same automaton makes sense, contrary to popular opinion (e.g., [2]) since the automatic PLC code generation, compilation, download, hardware, or runtime environment may be flawed.

This work is part of committee work of the GMA 1.50, which currently develops a guideline to the development of safety-related control functions (VDI / VDE 3541). One focus of this work is the selection of appropriate descriptive tools and methods for developing safety function blocks for PLCs according to IEC 61131.

References

[1] PLCopen TC5: Safety Software Technical Specification, Version 1.0, Part 1: Concepts and Function Blocks. PLCopen, 2006.

[2] Drath R., Hoernicke M.: „Konzept für einen effizienten Entwicklungsprozess zur Erstellung von sicherheitsgerichteten SPS Bibliotheken“. Kongress Automation, Baden Baden, 2010

[3] Frey, G.; Drath, R.; Schlich, B.: Leitfaden zur Entwicklung von Safety-Applikationen auf Anwenderebene. Kongress Automation, Baden-Baden, 2011.

[4] IEC 61131-3 PLC programming languages.

[5] graphwalker, http://graphwalker.org/

[6] yEd, http://www.yworks.com/en/products_yed_about.html

[7] graphml, http://graphml.graphdrawing.org/

[8] Model-Based testing Tool JTorX https://fmt.ewi.utwente.nl/redmine/projects/jtorx

[9] Jesse H. Poore, Lan Lin, Robert Eschbach, and Thomas Bauer, "Automated Statistical Testing of Embedded Systems", Chapter in "Model-Based Testing for Embedded Systems", Series on "Computational Analysis, Synthesis, and Design of Dynamic Systems.", 2011.

[10] Lunze, J.: Relations between networks of standard automata and networks of I/O automata. Proceedings of the 9th International Workshop on Discrete Event Systems (WODES'08), Göteborg, pp. 425-430.

[11] Frey, G.: Automatic Implementation of Petri net based Control Algorithms on PLC. Proc. American Control Conference ACC 2000, Chicago (IL), pp. 2819-2823, June 2000.