SICA – A SOFTWARE COMPLEXITY ANALYSIS METHOD FOR THE FAILURE PROBABILITY ESTIMATION

Tero Tyrväinen[1], Ola Bäckström[2], Jan-Erik Holmberg[3], Markus Porthin[1]

[1] VTT Technical Research Centre of Finland Ltd.: P.O. Box 1000, Espoo, Finland, 02044, and tero.tyrvainen@vtt.fi and markus.porthin@vtt.fi
[2] Lloyds Register: P.O Box 1288, Sundbyberg, Sweden, 17225, and ola.backstrom@lr.org
[3] Risk Pilot Ab: Metallimiehenkuja 10, Espoo, Finland, 02150, and jan-erik.holmberg@riskpilot.fi

This paper presents a method, called SICA (SImple Complexity Analysis), for the complexity analysis of application software in computer based reactor protection systems of nuclear power plants. The complexity measures are utilised in the estimation of software failure probabilities. Complexity of software can be defined in several ways. The challenge is to find a practical and justifiable metric, which can be assumed to correlate with the reliability. The goal has been to develop a simple complexity analysis method, because reactor protection systems contain typically very many software modules, and their analysis can be time-consuming. The complexity analysis is performed based on functional diagrams used for requirements specification. Software modules are divided into three complexity categories: low, medium and high. In SICA, categorisation of modules is performed based on the number of feedback loops, the number of connected complex function blocks, the number of connected function blocks, and the number of inputs and outputs. The complexity analysis is demonstrated with application software module examples. The decision rules of the SICA method are simple to apply and the complexity category of a software module can be determined by a visual assessment.

I. INTRODUCTION

To assess the risk of nuclear power plant (NPP) operation and to determine the risk impact of digital systems, there is a need to quantitatively assess the reliability of software used in safety automation in a justifiable manner. This is challenging because software failures are in general mainly caused by systematic (i.e. design specification or modification) faults, and not by random errors, software based systems cannot easily be decomposed into components, and the interdependence of the components cannot easily be identified and modelled.

The logic of I&C functions in safety automation such as the reactor protection system is implemented in application software modules. Since application software is plant-specific and implementation specific, there is not enough operational data available currently for proper reliability analysis (Ref. 1). Therefore, other methods have to be used in the assessment. Complexity is considered a logical and suitable metric for this purpose. It can be assumed that more complex software has higher failure probability than simpler software if verification and validation (V&V) procedures are same.

This paper presents a software reliability analysis method based on these ideas and a novel software complexity analysis method, called SICA (SImple Complexity Analysis), designed for this purpose (Refs. 1 and 2). In the proposed software reliability method, application software failure probabilities are estimated based on the complexity of software and V&V level. This covers non-fatal failures in which the software continues operating and generates output despite the failure. The failures can either be failures on demand or cause spurious actuations. We have also addressed fatal failures and failures of system software (Ref. 1), but they are out of the scope of this paper.

The software complexity analysis is challenging because there are several possible definitions of complexity. Different aspects that increase complexity are quite well known, but their significances and combined effects are open to debate. It is easy to identify very simple and very complex software, but the cases between them can receive various opinions. Experts of different fields interpret complexity differently, e.g. I&C experts emphasise use of memories and process engineers focus on complex input-output relations. The goal of the research presented in this paper was to develop a complexity analysis method whose output metric can be assumed to correlate with the reliability. The method is kept simple, because reactor protection systems contain typically very many software modules, and their analysis can be time-consuming. Also, the method is

designed to make it possible for an analyst to estimate the characteristics of the application software, without having access to the source code itself.

II. SOFTWARE RELIABILITY ANALYSIS METHOD

The failure modes for application software can be divided into fatal and non-fatal failures. The fatal failures in application software will stop all on-going processes of application software running on the processor. The non-fatal failures will only affect the output of the current application software. Fatal and non-fatal failures may have the same impact on the signal generated; this is simply a matter of configuration and definition of end-states.

In this paper, only non-fatal failures are considered. Due to the nature of I&C functions performed by reactor protection system, we can assume that a non-fatal failure can be either failure to actuate or spurious actuation of the implemented function. Our method estimates the probability of failure per demand (pfd) for non-fatal failure of an application software module. This probability can then be divided into failures to actuate and spurious actuations as presented in Ref. 1.

A Bayesian belief network (BBN) model presented in Fig. 1 has been developed for the estimation of the pfd. The model is presented in a simplified form as follows. We suggest that a point estimate for the probability of non-fatal application software module failure is

$$E[P_{NSA} \mid F] = 1E\text{-}6 \times F, \qquad (1)$$

where F is a shaping factor that depends on complexity and V&V level. The point estimate may be interpreted as the mean value of the prior distribution of pfd, using suitable distribution assumptions (see Ref. 1). Potential values of the shaping factor are presented in TABLE I. It is assumed that an increase in complexity category increases the probability by a decade, and an increase in V&V level decreases the probability by a decade.

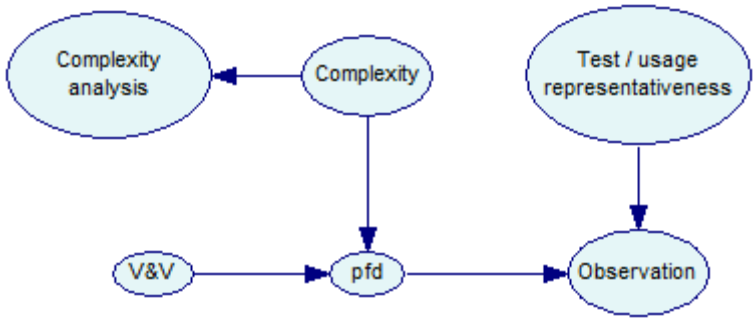


Fig. 1. A BBN for assessing software reliability using V&V class, software complexity and usage and test observations as evidence.

TABLE I. Shaping factor (F) values for different software categories

		Complexity		
		High	Medium	Low
V&V	0	10000	1000	100
	1	1000	100	10
	2	100	10	1
	3	10	1	0.1
	4	1	0.1	0.01

Only three complexity categories and five V&V levels are used. The approach is definitely rough, but it is considered sufficient because the uncertainties are high and the significance of an individual software module with regard to probabilistic risk assessment (PRA) results is supposedly quite small.

V&V levels are assumed to correspond to the safety classes in NPPs (Ref. 3) and the general IEC classification safety integrity levels (SIL) for electrical or electronic systems (Ref. 4) as presented in TABLE II. In NPP context, safety category

A can correspond to either SIL 3 or 4. Reactor protection systems belong to safety category A, and limitation functions can belong to category B.

TABLE II. V&V levels

V&V	Safety integrity level (SIL)	Safety category
0		Non-nuclear safety
1	1	C
2	2	B
3	3	A
4	4	

Operational data can be used to update the failure probability estimates as presented in Ref. 1. Currently available data is however so scarce that it does not change the estimates when reasonable prior distribution assumptions are used.

The numerical values presented are fully dependent on how the modules, complexity and V&V levels are defined. These numbers should not be taken as final accurate values but only as trial numbers. Ref. 1 claims that the probabilities generated by the method are reasonable compared to operating experience and failure probabilities used in PRA currently. However, the validation of the model is an important issue for further research. This paper focuses on the assessment of the complexity factor, and the justification of numbers is beyond the scope of this paper. Therefore, the reader should only consider numbers as examples, not as well justified data.

III. APPLICATION SOFTWARE MODULES

We apply the method presented in the previous chapter to application software modules. An application software module (AS module) is a piece of software that is representing a specific functionality. The application software is executed and controlled by the system software (run time environment) during an operating cycle. Each AS module usually corresponds to one individual function diagram group dedicated to a specific task. Depending on the specific case the application software can be represented by one or more AS modules.

The definition of an AS module should be adapted so that possible dependences between I&C functions will be properly covered (analogous to the definition of components of a system). It should be noted that the definition of module also affects the baseline probability 1E-6.

In Fig. 2, a signal acquisition and processing AS is presented. The software transforms two analog input signals into binary values by comparing them to threshold values. The software takes two similar signals from three other redundancies and performs 2-out-of-4 votings for both input signal types. Finally, the software sets the output to 1, if at least one voting has result 1, and to 0 otherwise. The AS module could be defined according to the red solid line, or three separate modules could be defined according to the red dotted line. As the output from this software is through the same interface, there is no reason to split the AS module into more than one AS module, so a proper definition would be according to the red solid line in this example.

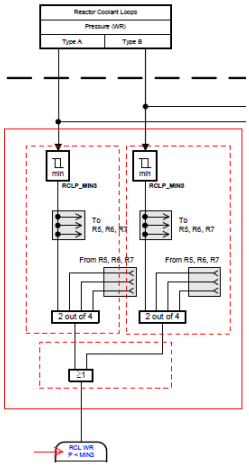


Fig. 2. Illustration of the definition of application software modules.

An estimate of the number of application software modules that form an average application software function is in the range of 5−20.

IV. COMPLEXITY ANALYSIS

For a complexity analysis method, the following properties are desirable:

- transparency
- possibility to quickly estimate the complexity of a software module
- complexity can be estimated without access to the source code, i.e., only logic diagrams are needed
- estimates stay consistent when AS modules are defined differently.

In Ref. 2, two complexity analysis methods were considered, ISTec's method (Ref. 5) and the SICA method (Ref. 1). In these two methods, the analysis is based on software logic diagrams that represent the software. In ISTec's method, a software complexity metric is calculated based on nine indicators using a BBN model. All the details of ISTec's method are not publicly known and it is complicated to calculate by hand. It has however been successfully automated. In the SICA method, complexity analysis is performed by a visual assessment using simple decision rules. ISTec's method and SICA are the only complexity analysis methods known to be applied in nuclear safety automation domain. In addition, the TOPAAS method for software reliability analysis accounts software complexity (Ref. 6), but its application area has been different. The TOPAAS method needs the source code of the analysed software for the complexity analysis.

IV.A. Factors that Affect Complexity

Factors that affect the complexity in application software modules were identified in Ref. 2. Discussions from Ref. 2 are presented in shortened form in the following.

IV.A.1. Function Blocks

Evidently, the number and the complexity of function blocks (or library functions) affect the complexity of a logic diagram. Complex function blocks typically use internal memories, perform complex computation or include many inputs, outputs and parameters.

IV.A.2. Interconnections between Function Blocks

The complexity of interconnections between function blocks affects the overall complexity significantly, but it is difficult to measure. A diagram that includes only simple function blocks, e.g. AND and OR, can also be moderately complex, if the number of function blocks is large and interconnections are complex.

Generally, the more connections there are between function blocks, especially complex function blocks, the more complex the software is. McCabe index (Ref. 7) is one option to measure the complexity of interconnections and software complexity in general, but the source code is needed for the analysis. McCabe index measures the number of linearly independent paths through the software's source code. ISTec (Ref. 5) has a similar approach to measure the complexity of interconnections. The counting of the number of interconnections in a diagram is impractical unless it can be automated.

IV.A.3. Feedback Loops

The use of feedback loops increases the software complexity significantly, and therefore, they do not appear often in digital safety automation in NPP, but some examples may exist.

IV.A.4. Inputs and Outputs

Having many inputs and outputs, especially of different types, complicates a software module in a sense that it is difficult to use in combination with other modules. If inputs have different types, they all need specific handling, which complicates the programming.

IV.A.5. Upstream and Downstream Logic Diagrams

If there are many logic diagrams that provide input to the analysed logic diagram and many logic diagrams where the outputs of the analysed logic diagram lead to, the analysed diagram is a part of a complex entity, and hence, using it is not easy. This factor correlates significantly with the number of inputs and outputs.

IV.B. SICA

In the development of SICA, the goal has been to develop a simple complexity analysis method, because reactor protection systems contain typically hundreds of application software modules, and their analysis can be time-consuming. The aim is that an expert can perform the analysis by a short visual assessment of a logic diagram. SICA accounts the complexity of function blocks, the interconnections between function blocks and inputs and outputs in the determination of the complexity category of a logic diagram.

Categorisation of modules is based on a few basic ideas:

1. The number of items (blocks, connections, inputs, outputs) correlates with complexity.
2. Some function blocks are by nature complexity increasing (more difficult to design and error-prone). For example, time-related blocks, flip-flops and modified function blocks that implement non-standard functionality increase complexity according to experience gathered from a formal model checking study (Ref. 8). Therefore, complex function blocks are defined in the method.
3. A feedback loop is a special structure, which is rare but possible. Therefore, it requires a special rule.
4. The division to three complexity levels seems appropriate. Two levels are probably not enough and having more than three levels would not provide much additional value.

In the SICA method, all the function blocks that use internal memory are categorised as *complex function blocks*. Also, those function blocks for which the sum of inputs, outputs and parameters is more than 10 are categorised as *complex function blocks*.

Categorisation of modules is performed using the following metrics:

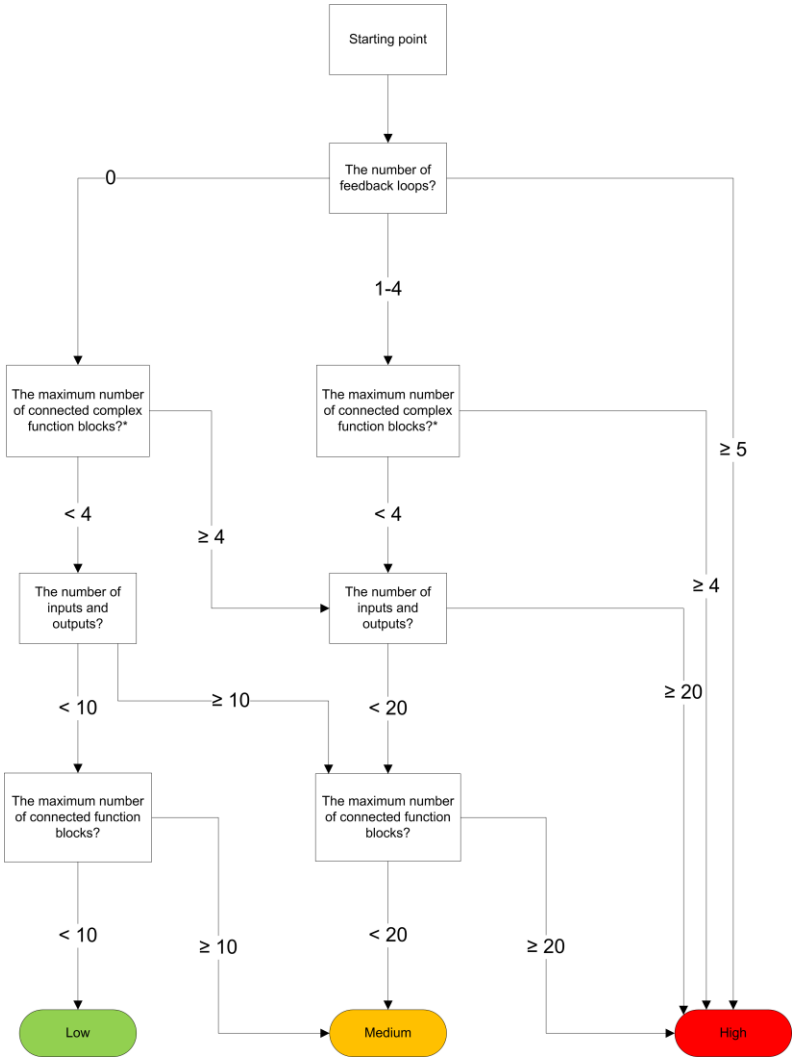
- the number of feedback loops (not including feedback loops inside function blocks)
- the number of connected complex function blocks
- the number of connected function blocks
- the total number of inputs and outputs.

Function blocks are defined to be "connected" if they affect the same output signal, i.e., function blocks are located in signal paths that are involved in the processing of the same signal. Note that it is required that the function blocks affect the same output because if they do not affect the same output, the software consist of many not so complex parts and is not that complex (except maybe due to complex input/output relations). The number of connected function blocks can be calculated for each output signal involved in the software module and the maximum value is used in complexity analysis.

The decision rules to categorise software modules into complexity classes, *low*, *medium* and *high*, are presented in Fig. 3.

In the following, few concepts related to the SICA analysis are explained more in detail:

- **Feedback loop** is a path in a logic diagram that starts and ends at the same point (any point in the feedback loop can be considered as a start/end point). The path can continue through internal outputs and inputs of the diagram to form a feedback loop. A feedback loop is formed by a single path. If a part of a feedback loop has a "redundant" part (another path with the same start point and end point), there are two feedback loops. In other words, multiple feedback loops can have a common part.
- **Input** of a logic diagram is a single signal coming to the diagram from outside of the diagram (e.g. from another diagram or measurement sensor). A logic diagram can also include internal inputs that come from the same diagram, but they are not counted as inputs of the diagram in the SICA method.
- **Output** of a logic diagram is a single signal going out of the diagram (e.g. to another diagram or device). A logic diagram can also include internal outputs that lead to internal inputs in the same diagram, but they are not counted as outputs of the diagram in the SICA method.
- A logic diagram can include **hidden redundancies**, i.e. only one of identical parts in the diagram is explicitly shown in the documentation. Inputs, outputs, function blocks and feedback loops must be calculated from all redundant parts of a diagram in the SICA method even if they are not explicitly shown in the documentation.



* all function blocks that use internal memory or those for which the sum of inputs, outputs and parameters is more than 10.

Fig. 3. Rules to identify application software complexity from Ref. 1.

V. EXAMPLES

Fig. 4 presents functional diagrams of five generic AS modules. TABLE III presents the results of SICA analysis. The complex function blocks of the examples are presented in Fig. 5. These function blocks are complex because they use internal memory.

VI. DISCUSSION

The SICA method takes most of the factors affecting complexity into account. The complexity of interconnections between function blocks is measured by counting connected function blocks. It is a quite simplified metric for that but a more detailed solution would make the analysis complicated. Inputs and outputs are treated together for simplicity and because there have not been claims that they should have different weights. Upstream and downstream diagrams are neglected by SICA to keep the analysis simple enough. It is not considered a big defect because the number of upstream and downstream diagrams correlates with the number of input and outputs.

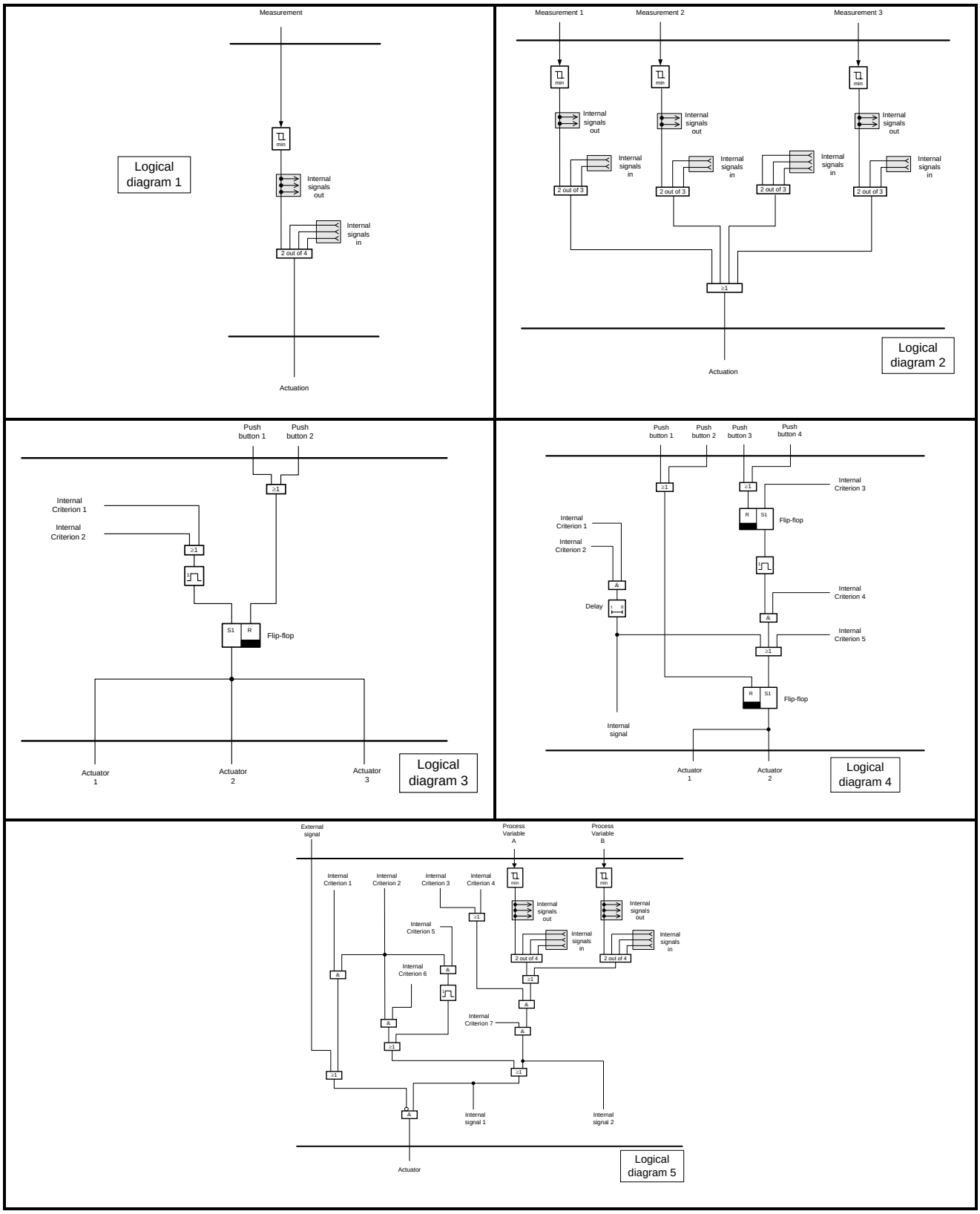


Fig. 4. Examples of application software module diagrams (Ref. 2).

TABLE III. The results of SICA analysis

Logical diagram	The number of feedback loops	The maximum number of connected complex function blocks	The number of inputs and outputs	The maximum number of connected function blocks	Complexity category
1	0	1	2	2	Low
2	0	3	4	8	Low
3	0	2	5	4	Low
4	0	4	6	9	Medium
5	0	3	4	16	Medium

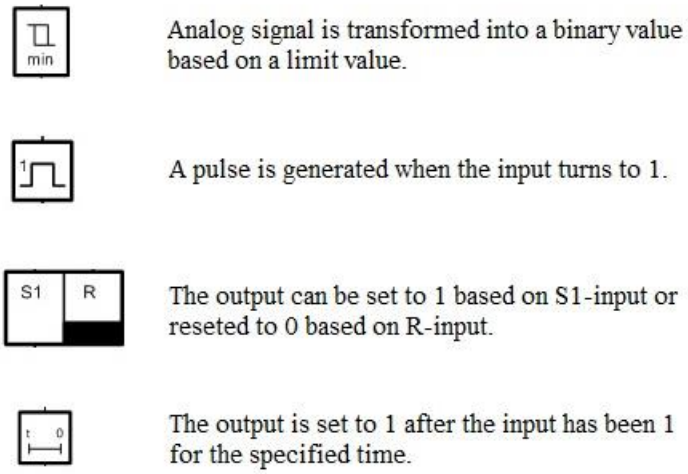


Fig. 5. Complex function blocks.

A user of the SICA method should notice that the parameter values and decision rules are only suggestions of the authors. These rules and parameters have been seen suitable based on the analysis of a moderate number of software diagrams. Still, they are just expert judgements, and there are no unambiguously correct rules to define complexity categories. A user of SICA is allowed to choose different parameters and modify the decision rules to fit his/her purposes.

The classification rules need to be studied more. Ideally, we should have a lot data or use many experts to justify the criteria. We have a good idea of which kind of diagrams are simple. The main challenge is to define rules when a diagram becomes so much more complex that the error probability could be assumed to increase by a certain factor, such as 10.

It would also be beneficial to benchmark SICA against ISTec's method. Some comparison was performed in Ref. 2, but it was very limited because the details of ISTec's method are not known. Based on the comparison, SICA was developed further as discussed in Ref. 1, but more comprehensive benchmarking would be required so that it could be said whether the methods give similar results and what are the differences. The benchmark study could increase the confidence on the method.

SICA analysis can be considered simple. Learning of the method can take some time because there are several decision rules and parameters to remember. Complex function blocks need also to be identified, and it might be good practise to do that before the analysis of diagrams. But after learning the method and knowing which function blocks are complex, the categorisation of most of the software modules is just a matter of visual assessment of few seconds.

So far, the SICA method has been applied to reactor protection systems which belong to safety category A, but it is also fully applicable at least to safety category B I&C software. In lower safety categories, the degree of complexity is usually higher (complexity class high).

VII. CONCLUSIONS

This paper presented a software reliability analysis method that estimates failure probabilities based on software complexity and V&V level. For the complexity analysis part, the SICA method was proposed. Software modules are divided into three complexity categories to keep the analysis simple enough. Despite its simplicity, SICA takes into account all the

main aspects of complexity: complexity of function blocks, interconnections between function blocks and input-output relations.

ACKNOWLEDGMENTS

The work has been financed by NKS (Nordic nuclear safety research) and SAFIR2018 (The Finnish Research Programme on Nuclear Power Plant Safety 2015–2018). NKS conveys its gratitude to all organizations and persons who by means of financial support or contributions in kind have made the work presented in this report possible. Software module diagram examples were provided by Mariana Jockenhövel-Barttfeld and Andre Taurines from AREVA GmbH in Ref. 2.

REFERENCES

1. S. AUTHEN, O. BÄCKSTRÖM, J.-E. HOLMBERG, M. PORTHIN, and T. TYRVÄINEN, "Modelling of Digital I&C, MODIG – Interim Report 2015, NKS-361," Nordic Nuclear Safety Research (NKS), Roskilde, Denmark (2016).
2. O. BÄCKSTRÖM, J.-E. HOLMBERG, M. JOCKENHÖVEL-BARTTFELD, M. PORTHIN, A. TAURINES and T. TYRVÄINEN, "Software Reliability Analysis for PSA – Final Report, NKS-341," Nordic Nuclear Safety Research (NKS), Roskilde, Denmark (2015).
3. INTERNATIONAL ELECTRONIC COMMISSION, "Nuclear Power Plants – Instrumentation and Control Important to Safety – Classification of Instrumentation and Control Functions, IEC 61226, Ed. 3.0," Geneva, Switzerland (2009).
4. INTERNATIONAL ELECTRONIC COMMISSION, "Functional safety of electrical/electronic/programmable electronic safety-related systems (E/E/PE, or E/E/PES), IEC 61508, ed. 2.0," Geneva, Switzerland (2010).
5. J. MÄRTZ, H. MIEDL, A. LINDNER and C. GERST, "Komplexitätsmessung der Software Digitaler Leittechniksysteme, ISTec-A-1569," Institut für Sicherheitstechnologie (ISTec) GmbH, Garching, Germany (2010).
6. E. BRANDT, A.A. BUCCHIANICO, J. EKRIS, J.F. GROOTE, W. GEURTS, G. HESLINGA and G. KOLK, "TOPAAS: Een Structurele Aanpak Voor Faalkansanalyse van Software Intensieve Systemen," Rijkswaterstaat Ministerie van Verkeer en Waterstaat, Netherlands (2011).
7. T. MCCABE, "A Complexity Measure," *IEEE Transactions on Software Engineering*, **SE-2**, *4*, 308 (1976).
8. J. LAHTINEN, J. VALKONEN, K. BJÖRKMAN, J. FRITS, I. NIEMELÄ and K. HELJANKO, "Model Checking of Safety Critical Software in the Nuclear Engineering Domain," *Reliability Engineering and System Safety*, **105**, 104 (2012).