MODEL CHECKING REVEALS HIDDEN ERRORS IN SAFETY-CRITICAL I&C SOFTWARE

Antti Pakonen, Teemu Mätäsniemi and Janne Valkonen
VTT Technical Research Centre of Finland
P.O. Box 1000, FI-02044 VTT, Finland
antti.pakonen@vtt.fi; teemu.matasniemi@vtt.fi; janne.valkonen@vtt.fi

ABSTRACT

Function blocks are a common programming language for safety-classified instrumentation and control (I&C) systems. While designs of elementary function blocks can be proven error-free through rigorous component and unit testing procedures, verification and validation of complex function block diagrams might only be based on inspections and reviews. Arguments for error-free software are often supported by claiming high quality in software development practices and referring to operational experience.

Model checking is an efficient formal method for the verification of (hardware or software) system designs. A model checker is a software tool that checks that a model of a system satisfies given requirements in all possible situations. The difference to more traditional V&V methods, such as testing and simulation, is that the analysis is exhaustive – covering all possible executions.

We have been studying the use of model checking for verifying nuclear power plant software design in several research projects. The method has also been adopted into practical use; as an example, we have been consulting the Finnish Radiation and Nuclear Safety Authority (STUK) on evaluating NPP I&C system designs using model checking since 2008.

In this paper, we present a method for model checking of function block based systems. Since model checking also requires the formal specification of system requirements, we also present a method for requirement specification based on requirement templates.

Key Words: Model checking, Verification and Validation, I&C software

1 INTRODUCTION

The use of digital I&C systems in nuclear power plants is on the rise. In addition to new plants, old analogue control systems are being replaced in renewal projects. When compared with the analogue solutions, digital systems are – despite their many advantages – inherently more complex, which justifiably raises concerns regarding the correctness of both hardware and software design. Different verification and validation (V&V) methods aim at ensuring that the designs are error-free.

Model checking is a formal computer-aided method for verifying that a model of a system behaves according to given system requirements in all possible behaviors or execution paths of the model. Long used in microprocessor manufacturing, model checking has also been successfully applied in the evaluation of I&C software. The difference to more traditional methods such as simulation and testing is that the analysis is exhaustive. Since all model behaviors are taken into account, hidden errors can be found from systems that have successfully passed other tests.

At VTT, we have been studying the use of model checking in the verification of programmable I&C system designs. Our experience has shown that the method is highly useful, especially for verifying function block based binary logic circuits that are common in safety-critical systems built on programmable logic controllers (PLCs).

Due to the lack of model checking tools dedicated to digital I&C systems, a lot of expertise and manual work is needed in using the method. Since the analysis is based on the system requirements presented in a formal language, the issue is also closely related to the art of requirements engineering. To ease the application of model checking, we have been working on tools and practices to support the user, and even to automate some of tasks involved. Our eventual objective is to integrate model checking with the tools used for engineering I&C software, thus facilitating the discovery of design errors as early as possible.

2 VERIFICATION OF I&C SOFTWARE

Function blocks are one of the most common programming languages used to implement safety functions in I&C systems based on PLC hardware. The function blocks are executed in cycles. At each processing cycle, the blocks are executed in the order specified by the way they are wired together in diagrams. Each block reads its inputs, updates its internal variables, and sets the outputs according to its internal logic. The IEC 61131-3 standard defines five programming languages for PLCs, one of which is function block diagram (FBD), but many major PLC manufacturers tend to favor non-standard vendor-specific languages.

In safety-critical applications, it is important to make sure that I&C systems always act according to their specified requirements, which is the aim of verification activities. Exhaustive verification of function block based systems is difficult because of the combinatorial explosion of the possible system states or executions. Each elementary block typically only has a few possible states, and the design can easily be proven error-free by component testing. When function blocks are combined into diagrams, things get tricky. (The term “logic block” is often used in the context of FPGAs, where the execution is not cyclic, but the verification challenges due to combinatorial explosion are similar.)

If we consider a simple circuit with n binary inputs, we can see that there are 2^n different input combinations. A circuit with 30 inputs will have 10^9 input combinations, for a system with 100 inputs the number of combinations is 10^{30}. However, most circuits will also have blocks with internal memory, analogue inputs, or feedback loops which further increase the amount of possible execution paths. Due to the combinatorial explosion, it becomes practically impossible to cover all the possible system behaviors by testing or simulating through all the input sequences.

As there is no way to cover all execution paths, many organizations rely only on manual design inspections and reviews. Arguments for error-free software are often supported simply by claiming high quality in software development practices and referring to operational experience. Design guidelines and testing practices help in detecting errors, but cannot guarantee the absence of software errors with possibly disastrous effects [1]. Formal methods like theorem proving have been proposed, but they typically call for detailed expertise, and can only prove certain aspects of the program structure or behavior.

3 MODEL CHECKING

Model checking [2] is a computer-aided formal method for verifying the correct behavior of a (hardware or software) system design model against the specified requirements. In addition to modeling the system being analyzed (as in, e.g., simulation), also the system requirements are formalized to a machine-processable language. A software tool called a model checker is then used to examine if all possible model behaviors fulfill the specified requirements. If the model checker finds a model execution that is contrary to a requirement, the execution is returned as a counterexample that demonstrates the unwanted model behavior. By analyzing the counterexample, the modeler can find solid evidence of a design error. If the counterexample is caused by an error in the model, rather than an error in the system being analyzed, it will demonstrate an unrealistic model execution (the model behavior is contrary to the system specification). The modeling error can then be corrected.

The key difference to more traditional V&V methods – such as simulation and testing – is that model checking covers all possible behaviors of the modeled system. In testing and simulation, only a certain set of test sequences can be covered (see Figure 1) in reasonable time. With model checking, it is therefore possible to find hidden errors in systems that have already undergone conventional V&V.

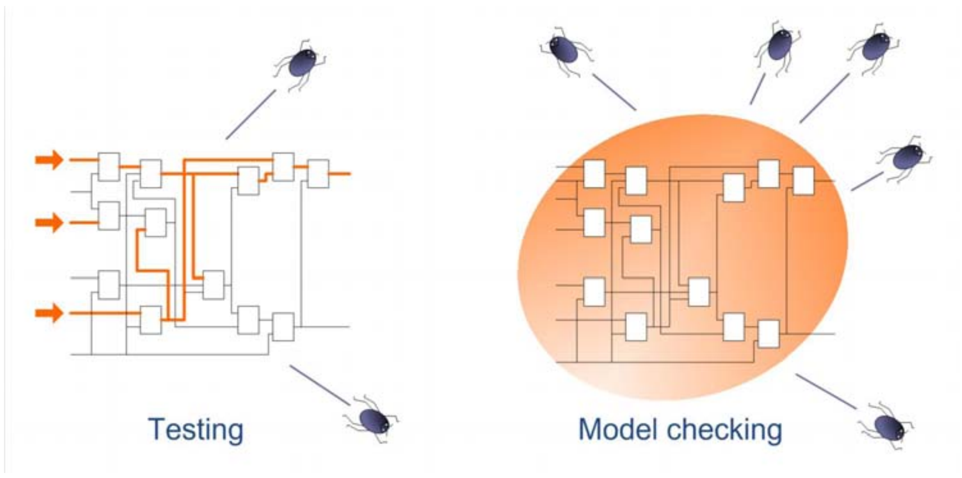


Figure 1. Only a limited number of test cases can be analyzed by simulation and testing. Model checking covers all possible behaviors of the system model, revealing hidden errors. (Modified from [3])

It is worth noting that model checking is not a brute force method. All the possible system executions and states are not analyzed one by one. Instead, the model structure is comparable to a finite state machine, and checking the structure for a certain property is basically a graph search.

Model checking is not a new invention. On domains such as microprocessor manufacturing, the method has been utilized since the 1990's. The later development of model checking techniques and tools, and the exponential growth of processing power has enabled the use of model checking in ever more complex and diverse applications [4]. In addition to the safety-critical domains like aviation and space, more down-to-earth use is also commonplace. For example, Microsoft uses model checking for the verification of device driver source code [5].

3.1 The Scope of Model Checking

Exhaustive verification is quite a promise, and indeed, the reliability of the analysis requires that both the system model and the requirements are correctly formalized. As mentioned above, errors in the model or the requirements usually result in a counterexample that will reveal the error when analyzed.

The challenge is to make sure that the model is expressive enough, and the necessary simplifications do not crucially limit model behavior. As the model is usually expressed as a type of a finite state machine, the method can only be effectively applied to the analysis of rather straightforward binary logic. Arithmetically complex logic cannot be verified, while analogue signals and complex timing logic can only be modeled with limited accuracy.

Furthermore, it is difficult to show that all the relevant aspects of all the necessary requirements have been taken into account. The requirement specification serving as the starting point can be incomplete or vague, and it is up to the modeller to consider all necessary aspects. For example, the so-called negative requirements ("the safety function must *not* be initiated if there is *no* justification for it") are seldom explicitly stated in the original requirement documents.

3.2 The Overall Model Checking Process

For many application areas (such as digital I&C software design) there are no ready dedicated tools for model checking. The overall process of model checking will then contain many tasks that require manual work and specific expertise (see Figure 2), making the process time-consuming and prone to errors.

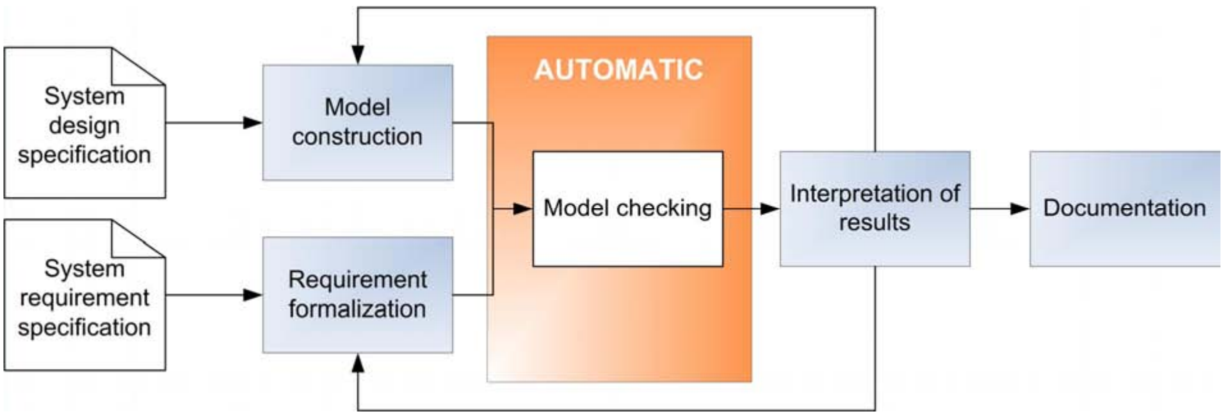

System design specification
System requirement specification
Model construction
Requirement formalization
AUTOMATIC
Model checking
Interpretation of results
Documentation

Figure 2. The overall process of performing model checking contains many steps, some of which may require manual work. (Modified from [6])

First, a model is constructed on the basis of the system design specification. As the modeling languages used in model checkers are typically more constricted than the specification languages used in the original system design, the modeler will typically have to resort to simplifications and abstractions. Expertise is needed to avoid omitting relevant aspects of system behavior.

Further specialized expertise is needed in requirement formalization. For model checkers, the requirements are typically specified using temporal logic, allowing for the definition of proper model behavior over time (see Section 5.1 for an example). Temporal logic requires adherence to a strict logic, but the original requirements are often given as natural language statements. The conversion can be difficult, especially if the original requirements are vague, ambiguous, or incomplete.

A model behavior that is inconsistent with the requirements is returned as a counterexample. A modeler will often spend a lot of time interpreting the counterexamples, since inconsistency can also be caused by errors or miscalculations in model construction or requirement formalization. Some iteration is usually needed.

4 MODEL CHECKING OF I&C SOFTWARE

4.1 Practical Experiences

At VTT, we have been studying the use of model checking for verifying NPP I&C software design under the Finnish Research Program on Nuclear Power Plant Safety (SAFIR) since the year 2007 [6][7][8][9][10]. After early successes in industrial pilot cases, the method has been quickly adopted into practical use. As an example, we have been consulting the Finnish Radiation and Nuclear Safety Authority (STUK) on evaluating NPP I&C system designs using model checking since 2008. We have also consulted the Finnish power company Fortum in verification of nuclear I&C. We have also successfully applied model checking in domains other than nuclear, for example for the verification of machine and factory automation systems.

We have been mostly working with the open source symbolic model checker NuSMV [11], which is based on a state machine representation and a discrete model of time. Other model checkers have also been studied, for example UPPAAL [12] for working with timed automata. NuSMV has proven to be computationally powerful for the evaluation of complex binary circuits. We are able to analyze models with state spaces as large as 10^{40} with computation times of minutes or less.

As the analysis is exhaustive, we have been able to find hidden errors in systems that have already undergone V&V using other methods. Since conventional methods can easily discover the more obvious problems, the errors found with model checking tend to deal with more unlikely, if not downright strange, scenarios. The errors may include elements such as:

- an operator or a maintenance person performing improper or ill-timed actions,
- unusual, unlikely signal pulses of exactly the right length at exactly the right time,
- faulty, conflicting or missing input data, or
- several of the above-mentioned combined.

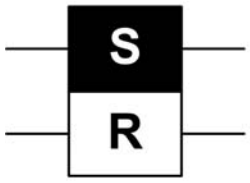
Scenarios such as listed above are challenging to take into account in test planning. The benefit of model checking is that such scenarios do not have to be thought of, as *all* possible scenarios are automatically taken into account.

It should be emphasized that model checking is not applicable to all kinds of I&C systems. Complex, algorithm-rich control systems cannot be modeled with any reasonable accuracy using modeling languages used by tools such as NuSMV. However, straightforward binary logic, which is (or at least should be) favored in safety-critical applications, is very well suited for model checking. Still, as there exist no model checking tools dedicated to the I&C domain, a lot of expertise is required.

4.2 A Modular Approach to Model Construction

For I&C software specified with function block based languages, we have been able to ease the modeling task by adopting a modular approach. If we first construct a library of code module elements that correspond to the set of elementary vendor-specific function block types, the task of constructing the model for model checking is then reduced to copying the application structure from the original design diagrams.

As an example, Figure 3 presents a basic type of a common function block, a flip-flop memory switch, and a code example for modeling the flip-flop as a NuSMV code module. Having now specified the block class, we can invoke an instance of the block and define the wiring of input signals from other blocks with a single line of code.



```
MODULE SR(set_input, reset_input)

    VAR
        memory : boolean;

    DEFINE
        set_output :=
        case
            set_input : TRUE;
            reset_input & !set_input : FALSE;
            TRUE : memory;
        esac;
        reset_output := !set_output;

    ASSIGN
        init(memory) := FALSE;
        next(memory) := set_output;
```

Figure 3. A common symbol for a bistable flip-flop memory switch, and the corresponding model code for the NuSMV model checker.

Most of the actual coding effort has been done once the modeler has constructed the library of elementary function block modules. The modeling of any function block diagram built of the blocks is now rather straightforward. However, there are still some aspects of the modeling task that require special care and interpretation:

- NuSMV uses a discrete time cycle, and proper handling of timing requires that the modeler finds the right level of accuracy without limiting model behavior or causing state space explosion.

- Analogue signals have to be discretized, again finding the right accuracy.
- Asynchronous systems are hard to model realistically.
- Models that are simply too complex have to be dealt with in parts [10].

For related approaches to model checking of function block diagrams, see [13] and [14].

4.3 A Graphical Toolset for Modeling

By adopting a modular approach to the modeling of function block based systems, we can simplify the application of model checking. Still, if the block connections have to be written down manually, the task is arduous. Less modeling expertise is needed, but writing down hundreds (if not thousands) lines of simple, repetitive code is still frustrating and error-prone work.

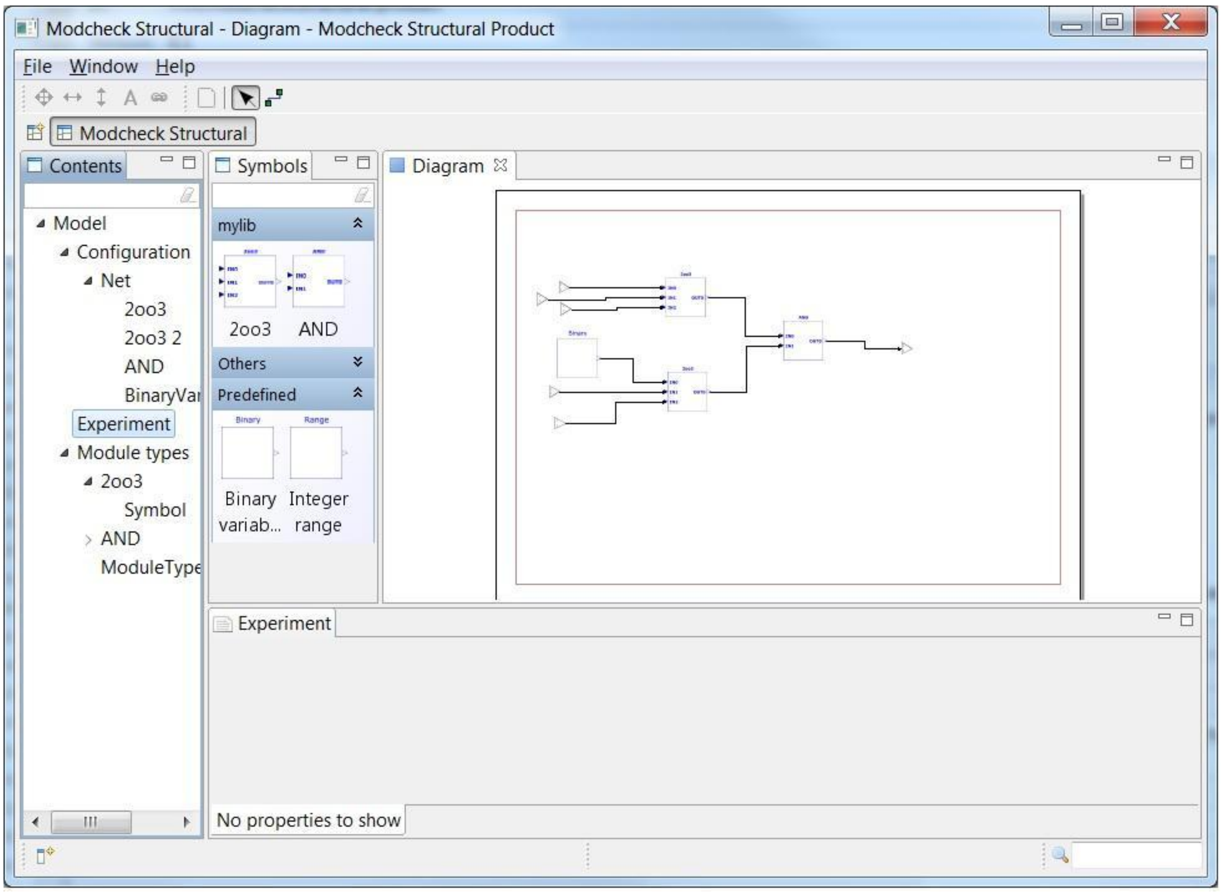


Figure 4. In the Simantics platform application, the modeler is able to specify the model for model checking by wiring together elementary function blocks using a graphical user interface.

Since the diagram modeling phase is a straightforward process, we have been developing a graphical user environment for model construction. Our solution is based on Simantics, an open source platform for modeling and simulation [15][16][17]. Simantics is based on a client-server architecture and a semantic modeling kernel, and the purpose is to connect and co-use a wide range of simulation and engineering tools. For example, tools such as OpenModelica, BALAS, Apros, OpenFoam, Comos and SmartPlant have been integrated into Simantics.

For our purposes, Simantics provides a ready graphical modeling framework. We have been developing a model checking plugin for Simantics based on NuSMV, and are now able to construct the NuSMV model by wiring blocks together in a 2D graphical view (see Figure 4). Applying model checking is now much more efficient, and syntactic errors are diminished. Issues such as timing and complex algorithms still call for some coding effort, but the overall work can significantly be reduced.

The ultimate benefit we expect to get from using Simantics is the flexible integration of model checking with the tools used for designing and engineering I&C software. If the I&C software engineering tool can be integrated with Simantics, we should be able to directly generate the model for model checking based on the original design sheets. Instead of model construction, our objective would be automatic model generation [18]. If model checking could be applied at the push of a button already in the system design phase, hidden errors could be discovered early in the system life-cycle, saving money and time. Still, practical challenges other than the model construction will also have to be solved (see next Chapter).

5 RESOLVING PRACTICAL CHALLENGES IN MODEL CHECKING

5.1 Formalization of the System Requirements

In model checking, the correct functioning of the system model is verified against a given set of requirements, which have to be formalized to a language understood by model checker software. Model checking is therefore firmly connected to the art of requirements engineering.

The system requirements are seldom available in formal representations. Commercial requirement management tools are used quite rarely, and there are no established standards for requirement specification languages. Textual descriptions, engineering drawings and different lists are favored, since they are readily understood by all stakeholders. However, using freeform text can make the requirement specification inconsistent, ambiguous and incomplete. Formalization and verification of such requirements is challenging, especially for an independent third-party reviewer.

For model checkers such as NuSMV, the requirements are formalized using temporal logic, which allows for the representation of propositions in terms of time. Two kinds of operators are used: modal operators such as “Globally” (in all states) or “Future” (eventually), and common logical operators. Even for fairly simple requirements, the temporal logic formulas can easily become convoluted and impossible to understand (see Figure 5). Typically, the modeler will also have to write down several temporal logic formulas to capture all necessary aspects of one requirement (for example the so-called negative requirements like “the system must *not* initiate the safety function if the alarm limit is *not* exceeded”).

```
(G ! Alarm ) | ( ! Alarm U ( Alarm & F Shutdown -> ( ! Shutdown U (
(Temperature < 55 ) & ! Shutdown & X ( ! Shutdown U Feedback_pump = OFF ) ) ) )
```

Figure 5. Temporal logic specifications can easily become convoluted and difficult to read. This NuSMV example translates to: "after the alarm signal, the temperature must first be below 55 and the feedback pump must then be off before the shutdown is activated".

Research on controlled (or constricted) natural language attempts to bridge the gap between representations that are generally acceptable but not machine-processable, and languages that require special expertise but enable automatic processing. Controlled natural language is designed to serve as a specification and knowledge representation language that avoids the ambiguity and vagueness of full natural language, and enables efficient processing. Engineers used to natural language should be able to adopt a controlled language tool, and the controlled language could be specified in such a way that it will enable transformation to, e.g., temporal logic.

We are currently studying the use of templates, or fill-in-the-blanks boilerplates, to facilitate the transformation of requirement clauses from controlled natural language to temporal logic. The idea is to capture logical structures that are typical for functional requirements in I&C, and then define the mapping from those structures to the specifications used in temporal logic (see Figure 6). See also [19] and [20].

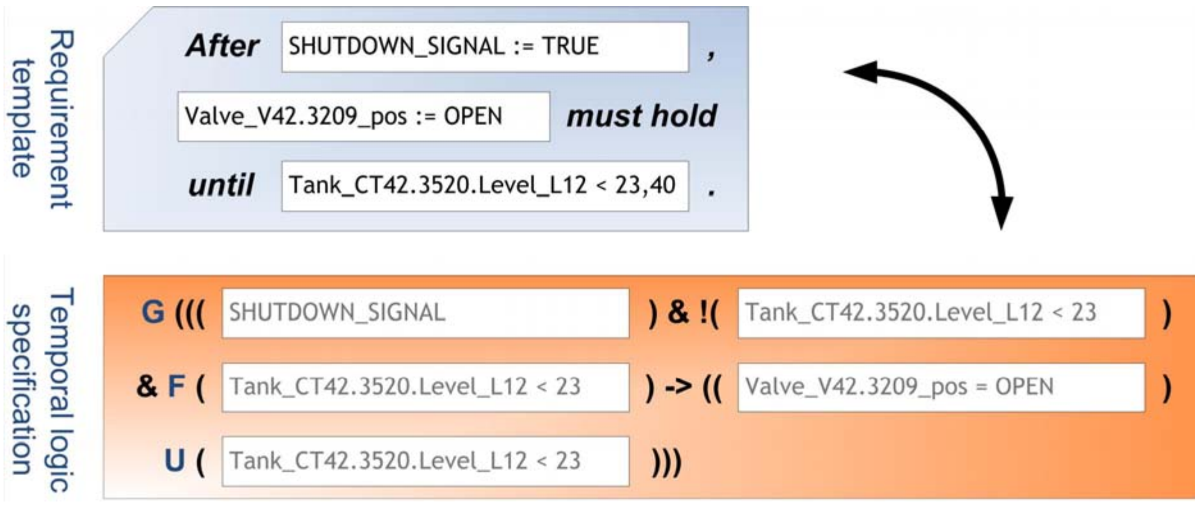

Requirement template
After
SHUTDOWN_SIGNAL := TRUE
,
Valve_V42.3209_pos := OPEN
must hold
until
Tank_CT42.3520.Level_L12 < 23,40
.
Temporal logic specification
G (((
SHUTDOWN_SIGNAL
) & !(
Tank_CT42.3520.Level_L12 < 23
)
& F (
Tank_CT42.3520.Level_L12 < 23
) -> ((
Valve_V42.3209_pos = OPEN
)
U (
Tank_CT42.3520.Level_L12 < 23
)))

Figure 6. A fill-in-the-blanks template to facilitate the transformation of requirements from controlled natural language to temporal logic.

Other ideas for tools to support the modeler in understanding complex temporal logic formulas can be found in [18].

5.2 Interpretation of the Counterexamples

When a model checker discovers a model behavior that is contrary to the system requirements, the execution path is returned as a counterexample scenario. By analyzing the counterexample, the modeler is able to trace the origin of the error. NuSMV, for example, outputs the counterexample as hundreds of lines of text, showing how the model variables change with time. Browsing through the textual representation is arduous work.

As the counterexample represents a sequence of model states, a much more illustrative representation would be a graphical display based on the manner the original system design is presented. For function block diagrams, this would mean an animation showing how signal values change over time by, e.g., changing the colors of the block connections, or displaying numerical monitors or trends for analogue values. By playing the animation back and forth in a step-by-step fashion, the modeler could then easily identify the origin of the error.

The visualization of the counterexample with 2D graphics in the Simantics workspace is a feature we are currently working on. Other methods for assisting the user in processing counterexamples are presented in [21] and [22].

6 CONCLUSIONS

Model checking is a powerful method. At VTT, we have successfully applied model checking for the verification of I&C software designs on different industrial domains, and have been able to find hidden design errors in systems that have already undergone conventional V&V. Others have reported similar results in fields such as aviation and aerospace industry. The Finnish nuclear sector has also shown interest in model checking.

As with formal methods in general, a lot of experience is needed in the application of model checking. The computational power of the method is based on the comparatively simple modeling languages – typically a type of finite state machine language is used for representing the system being analyzed. Accordingly, it takes know-how to understand the kind of simplifications and abstractions one can make without sacrificing the accuracy needed for exhaustive verification. It quickly becomes clear that the method does not apply to mathematically complex, algorithm-rich control systems. However, in safety-critical applications, straightforward binary logic is favored, and model checking becomes an invaluable tool.

Since there exist no model checking tools dedicated to the verification of I&C software, we have been working with model checkers such as NuSMV. As a result, a lot of effort is needed for the construction of the model, the formalization of the system requirements, and interpretation of the analysis results. Due to the amount of manual work, we have mostly been using model checking for independent third-party review of software that has already gone through the developer’s V&V cycle. Finding errors this late is costly, since all changes have to be propagated through many steps in the overall system development cycle.

For software specified with function block diagram languages, we have been able to ease the model checking process by adopting a modular approach – if we first construct a library of model checker code components that corresponds to the set of elementary function block types used, the modeling task of a function block diagram consists of basically just duplicating the block wiring logic. For the formalization of the system requirements, we are investigating the use of controlled natural language patterns in the shape of boilerplates.

Our technical solutions are largely based on Simantics, an integration platform for modeling and simulation. Our ultimate goal is to utilize the interoperability provided by Simantics, and integrate model checkers with the tools used for the design and specification of safety-critical I&C software. Once it becomes easy to use model checking already in the system design phase, most hidden errors will be found early, the reliability of safety-critical software will improve, and a lot of time and money will be saved.

7 REFERENCES

1. N. Völker, B. J. Krämer, "Automated Verification of Function Block-Based Industrial Control Systems", *Science of Computer Programming*, **Volume 32**, pp. 101-113 (2002).
2. E. M. Clarke, Jr., O. Grumberg, D. A. Peled, *Model Checking*, The MIT Press (1999).
3. D. Cofer, M. Whalen, S. Miller, "Model-Checking of Safety-Critical Software for Avionics", *ERCIM News 75*, October 2008, pp. 15-16 (2008).
4. T. Ball, B. Cook, V. Levin, S. Rajamani, "SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft", *Proceedings of the 4th International Conference on Integrated Formal Methods (IFM 2004)*, Lecture Notes in Computer Science, Volume 2999, pp. 1-20 (2004).
5. J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, "Symbolic Model Checking: 10^20 States and Beyond", *Information and Computation*, **Volume 98**, pp. 142-170 (1992).
6. J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, "Model checking methodology for supporting safety critical software development and verification", *ESREL 2010 Annual Conference*, Rhodes, Greece, September 5-9 (2010).
7. J. Valkonen, K. Björkman, J. Frits, I. Niemelä, "Model checking methodology for verification of safety logics", *The 6th International Conference on Safety of Industrial Automated Systems* (SIAS 2010), Tampere, Finland, June 14-15 (2010).
8. K. Björkman, J. Valkonen, J. Ranta, "Verification of Automated Changeover Switching Unit by Model Checking", *Seventh American Nuclear Society Topical Meeting on Nuclear Plant Instrumentation, Control and Human-Machine Interface Technologies (NPIC&HMIT 2010)*, Las Vegas, Nevada, November 7-11 (2010).
9. K. Björkman, J. Frits, J. Valkonen, J. Lahtinen, K. Heljanko, I. Niemelä, J.J. Hämäläinen, "Verification of Safety Logic Designs by Model Checking", *Sixth American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control, and Human-Machine Interface Technologies (NPIC&HMIT 2009)*, Knoxville, Tennessee, April 5-9 (2009).
10. J. Lahtinen, T. Launiainen, K. Heljanko, J. Ropponen, *Model Checking Methodology for Large Systems, Faults and Asynchronous Behaviour*, SARANA 2011 work report, VTT Technology 12, Espoo, Finland (2012).
11. "NuSMV: a new symbolic model checker," http://nusmv.irst.itc.it/ (2012).
12. "UPPAAL V.4 integrated tool environment," http://www.uppaal.com/ (2012).

13. J. Yoo, S. Cha. E. Jee, "A Verification Framework for FBD Based Software in Nuclear Power Plants", *15th Asia-Pacific Software Engineering Conference (APSEC '08)*, Beijing, China, December 3-5 (2008).

14. D. Soliman, G. Frey, "Verification and Validation of Safety Applications based on PLCopen Safety Function Blocks using Timed Automata in Uppaal", *2nd IFAC Workshop on Dependable Control of Discrete Systems (DCDS)*, Bari, Italy, June 10-12 (2009).

15. "Simantics – a software platform for modeling and simulation," https://www.simantics.org/ (2012).

16. T. Karhela, A. Villberg, H. Niemistö, "Open Ontology Based Integration Platform for Modeling and Simulation in Engineering", to be published in: *International Journal of Modeling, Simulation and Scientific Computing*, World Scientific Publishing (2012).

17. A. Villberg, T. Lehtonen, T. Karhela, K. Kondelin, "Applying Semantic Modelling Techniques in Large Scale Process Simulation", *Proceedings of the 1st IFAC Workshop on Applications of Large Scale Industrial Systems (ALSIS '06)*, Suomen Automaatioseura (2006).

18. A. Pakonen, J. Lahtinen, V.-P. Kuutti and T. Karhela, "Integrating Model Checking with Safety-Critical I&C Software Design", *Seventh American Nuclear Society International Topical Meeting on Nuclear Plant Instrumentation, Control and Human-Machine Interface Technologies (NPIC&HMIT 2010)*, Las Vegas, Nevada, November 7-11 (2010).

19. M. B. Dwyer, G. S. Avrunin, J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification", *Proceedings of the 21st International Conference on Software engineering (ICSE '99)*, ACM, New York (1999)

20. K. Loer, M. Harrison, "Towards Usable and Relevant Model Checking Techniques for the Analysis of Dependable Interactive Systems", *Proceedings of the 17th IEEE international conference on Automated software engineering (ASE '02)*, IEEE Computer Society, Washington, DC (2002)

21. H. Goldsby, B. H. C. Cheng, S. Konrad, S. Kamdoum, "A Visualization Framework for the Modeling and Formal Analysis of high Assurance Systems", *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006)*, Genova, Italy, October (2006).

22. I. Beer, B.-D. Shoham, H. Chockler, A. Omi, R. Trefler, "Explaining Counterexamples Using Causality", *Lecture Notes in Computer Science*, **Volume 5643**, pp. 94-108 (2009).