Title	Integrating model checking with safety-critical I&C software design
Author(s)	Pakonen, Antti; Lahtinen, Jussi; Kuutti, Veli-Pekka; Karhela, Tommi
Citation	7th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human-Machine Interface Technologies (NPIC&HMIT 2010). Las Vegas, Nevada, USA, 7 - 11 Nov. 2010
Date	2010
Rights	Copyright © (2010) American Nuclear Society. Reprinted from 7th International Topical Meeting on Nuclear Plant Instrumentation, Control and Human-Machine Interface Technologies (NPIC&HMIT 2010). ISBN 978-0-89448-084-3. This article may be downloaded for personal use only

INTEGRATING MODEL CHECKING WITH SAFETY-CRITICAL I&C SOFTWARE DESIGN

Antti Pakonen, Jussi Lahtinen, Veli-Pekka Kuutti and Tommi Karhela
VTT Technical Research Centre of Finland
P.O. Box 1000, FI-02044 VTT, Finland
antti.pakonen@vtt.fi; jussi.lahtinen@vtt.fi; veli-pekka.kuutti@vtt.fi; tommi.karhela@vtt.fi

ABSTRACT

Model checking is a formal method that can be used to verify hardware or software system designs. A model of the system is constructed, and that model is then checked against the system requirements. The difference to more conventional verification and validation (V&V) techniques is that the analysis is exhaustive – it covers all possible behaviors of the system model.

We have been working on model checking in the context of safety-critical I&C systems in several research projects, resulting in successful pilot applications. As an example, we have been consulting the Finnish Radiation and Nuclear Safety Authority (STUK) on verifying NPP I&C system designs. While the tools for performing the actual model checking are mature, there is still a lot of manual work needed for constructing the model on the basis of the design information, formalizing the system requirements, and interpreting the model checker results. The actual processing of the analysis is swift, but most of the work is spent on repetitive, dull tasks that could relatively easily be automated.

In order to ease the application of model checking already at the early phases of the system design process, we have been developing a set of tools to automate some of the tasks in the model checking process and guide the modeler in those tasks that still need human interpretation. Our solutions are largely based on the Simantics platform – an open-source development project aimed at the efficient integration of different modeling and simulation tools.

Key Words: model checking, instrumentation and control systems, software development

1 INTRODUCTION

The shift from analogue I&C to digital systems has raised concerns regarding the correctness of software design in safety-critical applications. Traditionally, verification and validation (V&V) of programmable control systems has been based on methods such as testing, simulation, and deductive verification. All of these methods have their shortcomings. Testing, while being a crucial element, cannot conclusively prove the reliability of complex systems due to the sheer number of required test cases. Similarly, all of the possible behaviors of a complex system cannot be covered by running simulations. Deductive verification can be used to prove aspects of the correctness of a system, but the techniques are time-consuming, require high expertise, and are subject to interpretation errors. None of the traditional approaches can exhaustively prove the absolute reliability of the examined system with any reasonable amount of time and effort.

Model checking [1] is a computer-aided formal method for verifying the correct functioning of a system design model against the specified requirements of the system. The analysis is relatively fast to perform and *exhaustive* – covering all the possible behaviors of the system

model. The model is typically based on a state machine representation, while the specifications are written in a language called temporal logic, the constructs of which allow the formal specification of system behavior in time. If the model can exhibit a behavior that is contrary to requirements, the model checker will output an execution path demonstrating this behavior as a counter-example. By examining the counter-example, the origin of the system fault can be located and corrected.

Various model checking tools have been developed. NuSMV [2], mainly used in our work, is a state-of-the-art symbolic model checking tool.

Our experience has shown that model checking can effectively be applied to V&V of digital I&C systems. It is however noteworthy that model checking does not apply to all kinds of control systems – continuous, complex, algorithm-rich control systems can not be modeled to reasonable detail with modeling languages used by tools such as NuSMV. However, the kind of straightforward binary logic that is (or at least should be) used in safety-critical I&C corresponds to a state-based representation. A state explosion problem can occur if the system under observation is e.g. very large or contains many feedback loops, but generally even quite complex binary logic can be exhaustively analyzed within seconds or minutes.

It is obvious that the capability to automatically discover hidden faults early in the design process would be a huge benefit for any system designer. However, while the analysis done by the model checker tool is quite quick, there is a troublesome amount of manual work needed for constructing the model, formalizing the system requirements, and analyzing the results. Accordingly, we have been working on developing software tools for automating the entire process of a model checking task.

2 MODEL CHECKING OF I&C SYSTEMS

2.1 Applicability of Model Checking for V&V of Digital I&C Systems

Model checking has been used since the early 1980's in e.g. verifying microprocessor design. Advances in computing power and the scalability [3] have enabled the use of model checking in new application areas and in ever more complex applications. Other than microprocessor verification, model checking techniques have been successfully used at least in verification of data communications protocols, verification of real-time controllers, and verification of source code in device drivers [4]. Examples of companies that have long utilized model checking include NASA and Microsoft. Application of model checking in evaluating digital I&C design is a fairly new topic, but research under the Finnish Research Programme on Nuclear Power Plant Safety (SAFIR) has quickly resulted in successful pilot cases ([5][6][7][8][9]), and eventually in the employment of model checking in large commercial projects. For example, we have been consulting the Finnish Radiation and Nuclear Safety Authority (STUK) on verifying NPP I&C system designs, with model checking used as a key method. We have also carried out several successful pilots in industry areas other than nuclear.

Model checking is not intended to replace existing V&V methods, but to complement them. Nevertheless, our work so far has shown that the method is a valuable addition to the set of methods used in I&C system evaluation. The tools are mature enough for real-world application, and in several of the pilot cases we have been able to discover design faults in systems that had

already undergone conventional V&V processes. Model checking is not, however, applicable to all kinds of systems, as the modeling language assumes certain simplifications. Nevertheless, safety I&C systems are (or at least, should be) based on rather straightforward binary logic, and such applications are inherently suitable for model checking.

2.2 The Overall Model Checking Process

The model checking process is illustrated in Figure 1. Based on a system description document, the relevant parts of the system to be modeled are selected, and the boundaries, constraints, and the abstraction level of the model are defined. Once this foundational work is done, the main phases follow. The different phases of the overall task of performing model checking are:

1. Construction of the model based on the system design specification
2. Formalization of the system requirements
3. Running of the model checker
4. Interpretation of the results
5. Documentation of the overall results

Currently, at least for applications such as digital I&C systems, the only phase that is automated is the actual analysis performed by the model checking tools. Other phases are still done manually, and are thus prone to human errors, and potentially either repetitive or confusing.

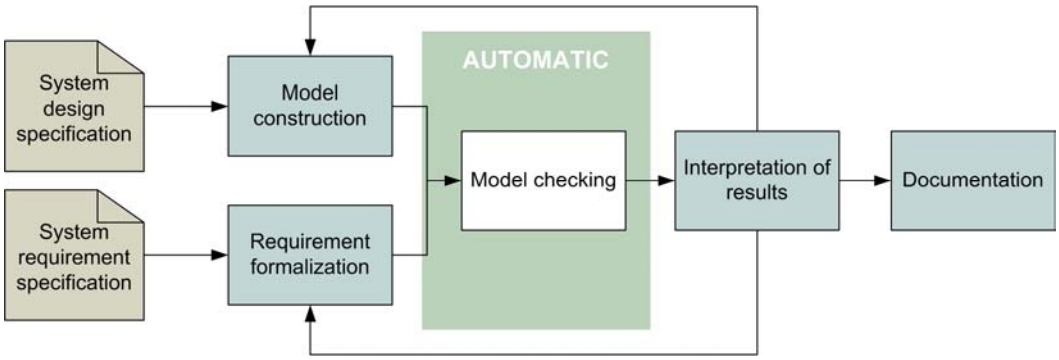

Figure 1. The Model checking process (Modified from [9])

First, using the system design specification as an input, a model has to be created. The model is a formal representation of the system that should include all the relevant behaviors, while leaving out insignificant details. The model has to be such that the requirements can be formalized as statements on the interfaces of the model.

The second step is the formalization of the system requirements to a format understood by the model checker. The specification language depends on the model checking tool, typically temporal logic is used. As the requirements are often given as natural language statements, converting them into a formal specification can be difficult, especially if the original requirements are ambiguous, vague, or incomplete.

After modeling and requirement formalization the model checking tool can be executed. The tool will automatically examine whether the model corresponds to the specified requirements. If the specification is fulfilled, the correctness of the model is declared. If the model can behave in a manner inconsistent with the requirements, an execution path demonstrating such behavior is

given as a counter-example. This counter-example allows the modeler to find faults in the original system design. However, a counter-example can also be a symptom of an error in the construction of the model or the specification of the requirements. Once the origin of the error is found, the model or the requirements can be refined (possibly introducing further errors). Since errors in modeling typically result in counter-examples, model checking can be seen as a self-fixing, iterative process that guides the modeler**.**

Interpretation of the results consists of examination of the counter-examples and deducing of the incorrect system behavior based on them. The interpretation can be straight-forward if the behavior of the system can be easily understood from the model behavior. Usually, however, the counter-examples are presented in such a format that manual interpretation can be complicated and time-consuming for larger models.

3 PRACTICAL CHALLENGES IN MODEL CHECKING

While tools for performing model checking are mature and available, there are still many challenges in its adaptation. In terms of **theoretical** problems, the greatest challenge has been the computational effort required due to the state explosion problem [10]. State explosion means that the number of states grows exponentially with respect to the size of the model. Even though the resulting state space is still finite, the model checking task might be too complex for existing methods and computers. The state explosion problem is not fully solved. However, advances in techniques such as symbolic methods [10], abstraction [10][11], partial order reduction [10][11][12], and bounded model checking [13][14] have made the model checking of increasingly complex systems possible.

Another problem is the reliability of model checking results. When a specification is satisfied, the model checking tool does not provide any evidence of this claim. It is quite difficult to confirm the correctness of a positive model checker result. Documented previous operational usage, certification, and cross-checking with another model checker are ways to build confidence in the tool. Combining model checking with theorem proving could alleviate the problem, but the techniques [15][16] need further research. If model checking is used only to locate errors in the system, and not to prove the correctness of the system, ensuring the reliability of the results is not the biggest concern.

Still, the main obstacles for efficiently applying model checking are found from the more **practical** challenges involved. Because of the amount of manual work needed, there are many phases in the overall model checking process that are slow, tedious and prone to errors:

1. **Construction of the model based on the system design specification**
2. **Formalization of the system requirements**
3. **Interpretation of the generated counter-examples**

Construction of the model requires expertise in both the formalism in which the system design is expressed, and the modeling language used by the model checker. If the formalisms are by their nature different, a lot of deduction and interpretation is needed. For NuSMV, for example, the model has to conform to a state machine representation. When possible, the modeling task can be made more efficient by adopting a modular approach. As digital I&C software is usually represented by function block diagrams, we have found it a practical

approach to first model the collection of elementary function blocks, reducing the overall model specification to assembling the overall application from reusable components. The latter task requires less modeling expertise, but is still quite tedious and prone to errors.

As the system requirements are usually stated in natural language, **formalization of the system requirements** to the safety properties processed by the model checker is a challenging task. Statements that can at the worst be ambiguous and vague need to be converted to precise statements in the formal specification language (such as temporal logic in the case of NuSMV). For any given one requirement, an expert using a model checker will also often have to derive *several* temporal logic clauses in order to cover all the aspects of the requirement. Furthermore, complex temporal logic formulas can easily become convoluted. It may later be difficult for the original modeler to later completely understand what the written formula originally was meant to state.

When a model checker discovers an execution path of the system model that is contrary to the system requirements, the execution path is presented as a counter-example. NuSMV, for example, outputs textual lists of data, showing how the signal variables of the model change through time. The format is all but illustrative, and for large models, can consist of thousands of lines of text. **Interpretation of the generated counter-examples** takes a large amount of effort in the overall modeling task, as the counter-examples do not specifically highlight where the problem is, but simply present all the data. Furthermore, a counter-example can be a symptom of an error in the modeling task as well as an actual fault in the system design, and it is impossible for a computer program to decide which one is to blame. Since errors in modeling usually result in unnecessary counter-examples, the modeler will have to process through many model checker outputs, even when the correctly constructed system model will eventually fulfill all the requirements.

4 INTEGRATING V&V TOOLS WITH THE SIMANTICS PLATFORM

Simantics [17][18] is a platform for modeling and simulation originally developed at VTT Technical Research Centre of Finland, but currently released and maintained as an open source tool by THTH (Association of Decentralized Information Management for Industry, www.ththry.org). Simantics has a client-server architecture with a semantic modeling kernel and an Eclipse framework based client software with plug-in interfaces. The idea is that a semantic approach to modeling and simulation (motivated by the ideas behind the Semantic Web) enables users of the platform to connect and co-use a wide range of different simulation and engineering tools. Several commercial and non-commercial simulation and engineering tools such as OpenModelica, BALAS, Apros, OpenFoam, Comos and SmartPlant are integrated to the environment. After the data model of the system is modeled as an ontology to Simantics and the interfaces for transferring system data to the platform are implemented, ontology mappings can be defined between different systems. These mapping rules will keep the different models consistent and thus provide automated model integration.

Starting point for Simantics platform development has been the following needs:

- Advanced operating environment for existing modeling and simulation tools
- Simulation and design system integration
- Co-use of different modeling and simulation tools

- Simulation and control system co-use
- Team work and information management for modeling and simulation

There exist many simulation solvers both in academia and industry that have sophisticated simulation algorithms but lack a good operating environment. The operating environment should provide certain pre- and post-processing capabilities, as well as connections to external applications like design tools or control systems. Pre-processing capabilities include features like 2D-flowsheeting support, discretization support (meshing), as well as support for model validation, model structure browsing and editing, model component reuse, model documentation and searches, experiment configuration, model version control and team features. Post-processing capabilities include features like 2D chart and 3D visualization of the results, animations of the results both in 2D and 3D, experiment control visualization etc. However, there should be no need for all different parties to maintain their own operating environments. Instead, one framework should be implemented which can then be further specialized to these different purposes.

Traditionally design systems (CAD) and computational systems have been separate entities in many areas of engineering. Naturally, there are exceptions, like electronic circuit design or discrete manufacturing processes, where the design has been done in a simulation-aided manner for a long time. Obviously, the more deterministic the target process or product is, the easier it has been to utilize computational models. However, in many engineering sectors like process industry, machine and vehicle industry, and in construction industry the tradition has been different. In addition to 2D and 3D CAD there is a legion of separate computational tools that can be utilized in different phases of the engineering process. V&V tools especially benefit from integration to automation design systems like e.g. Siemens Comos Logical.

When the products and the production processes that are modeled are complex, there is a definite need for the co-use of different computational, multi-level models. As an example, the Advanced Process Simulator – Apros [19] is used (among other things) for simulation-based automation design, mainly in power, pulp and paper industries. The modeling approach is dynamic and mechanistic. The control system components are modeled in the same simulation environment. As many companies use Apros as their control system design tool, there would be major benefits in being able to use different V&V tools (such as different model checkers) together with their simulation solution. In our case, this would mean mapping of the Apros control system ontology to the model checking ontology. This way the V&V model could be automatically generated from the simulation model. Combining model checking with simulation assisted testing would form a toolset for high quality automation in safety critical processes.

Modern engineering projects are networked. This also, in turn, sets increasing demands on the design and computational environment. Until now, computational environments have almost always been standalone applications. There has been very little team work support for sharing experiment results with other users. Another problem is version control – both of the model configuration and the computational results. There is a clear need for the future integration platform to support team work and version control. Naturally, any V&V tool would also significantly benefit from such features.

5 AUTOMATING THE MODEL CHECKING PROCESS

5.1 An Approach Based on Reusable Function Blocks

The overall model checking process described in Chapter 2 certainly does not represent how the model checking task should be done. Rather, it is the way that model checking unfortunately must be done because of the lack of suitable tools. However, for certain kind of I&C systems – namely, those that are designed using function block based programming languages – we have been able to find ways of creating reusable components of (NuSMV modeling language) code. Creating a software library of the elementary function blocks allows us to reduce the modelling task of any system based on those blocks to simply copying the application structure from the design diagrams. And, given that the system design is available in a suitable electronic form, the Simantics platform enables us to automatically convert the design specification to code used by the model checker.

Given the assumption that we are dealing with a standardized function block based application, we can now re-imagine the overall model checking process (Figure 2). Almost the entire process of creating the model and analyzing the results can now be automated, or at least computer-supported to some degree. Some of the steps we have already been able to automate, some of them we are currently working on.

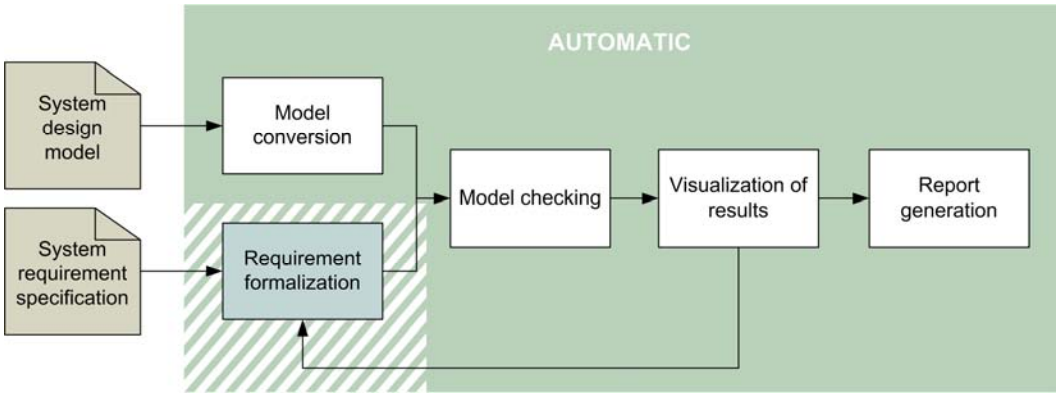

Figure 2. The Model checking process with increased automation and user support

Obviously, not all safety-critical I&C application are designed using function block diagrams. For such systems, we lose the capability of easily being able to convert the system model to the Simantics platform, and the modelling task remains as work that must be done manually. One should also note that even for function block based systems, there might still be a need for the modeller to occasionally modify the model code, for example to investigate timing issues or fault tolerance.

5.2 Model Conversion

In Chapter 2, one step of the overall process was called “model construction”. But now, our objective is automatic model *conversion*. A function block diagram consists of reusable elementary function blocks that can be wired together to form composite function blocks, which can then again be used to form larger compositions. This modular structure corresponds to the modular structure of the modelling language of the NuSMV model checker. Accordingly, once we have constructed a “component library” of the elementary function blocks, we can take any design based on those blocks and create the NuSMV model by figuratively wiring the blocks together. This phase used to involve writing the wirings in terms of simple code, but using Simantics, we can either 1) allow the user to concretely wire the blocks together using a 2D graphical editor, or 2) automatically convert the model if the function block diagrams are available in electronic form. As a result, the task is much quicker, and syntactic errors in code are diminished.

Figure 3 is a screenshot from the Simantics application. In this case, the model is being specified by wiring together elementary function blocks from the block library. We are currently able to generate the NuSMV code and perform model checking for an application composed of the elementary blocks, so that only the code for the block library and the requirement formalization are done manually.

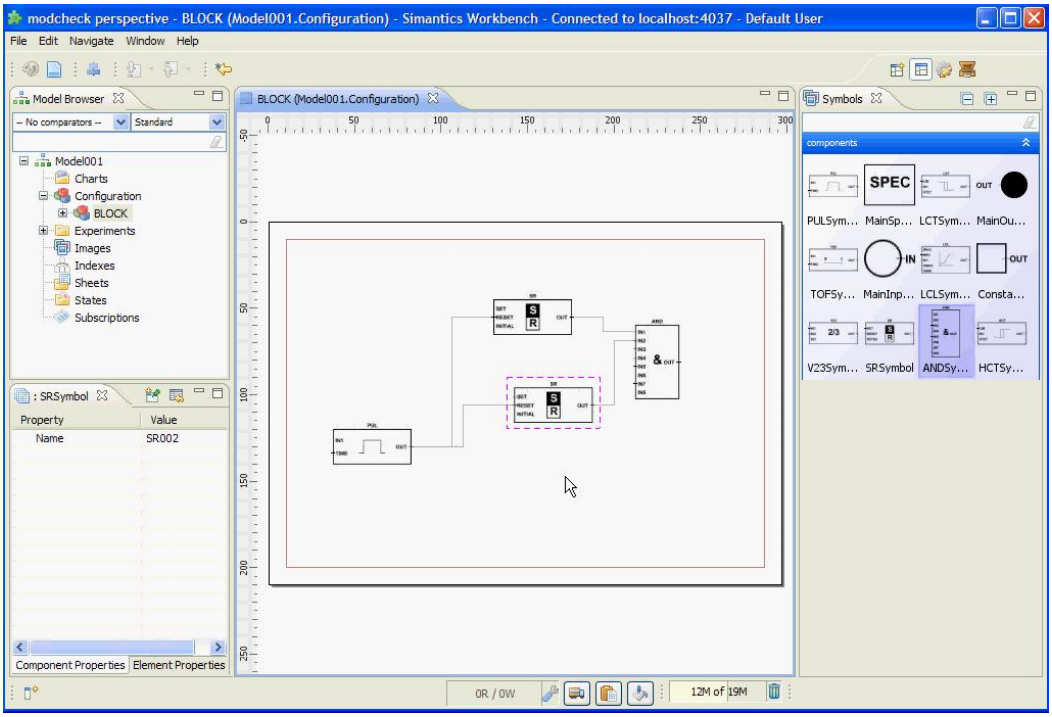

Figure 3. The Simantics platform workspace – the modeler is specifying the system model for model checking by wiring together elementary function blocks selected from the block library on the right.

It is now apparent that the actual modeling effort can be reduced to the maintenance of the library of elementary function blocks, the implementation of which is often specific to the application. Although there is a standard for PLC programming languages (the IEC 61131-3) that

describes a function block diagram language, many vendors prefer their own languages. For our way of performing model checking, the validity of the system model hinges on the correctness of the elementary function block library, although generally, the model checking process itself will reveal incorrectly specified modules, as the problems caused will pop up in the counter-examples. It is also noteworthy, that with vendor systems, one must often deal with black-box elements – the designer of the application has no knowledge of the implementation of the elementary function blocks, only their functional specification. Through e.g. type testing, vendors have shown the validity of the blocks, and model checking can only reveal errors in the ways of how those blocks are used.

The integration of NuSMV to Simantics is currently in "first draft" phase. Applications consisting of composite function blocks can already be converted and run through NuSMV, but further work on the tool is still needed.

5.3 Requirement Formalization

Converting the system requirements to the format needed by the model checker – formulas in temporal logic – is a task that cannot be completely automated. System requirements are rarely expressed in a strictly formal way, and even if they were, generation of temporal logic formulas is not a straight-forward task. Also, the so called "negative requirements" that describe how the system should *never* behave are typically not explicitly mentioned, partially because their evaluation using traditional V&V methods is quite difficult. As a result, in order to find hidden faults, an expert using a model checker may often derive *several* temporal logic clauses based on any one given requirement. If, for example, some safety I&C system is required to start a safety function when a tank level measurement is too low, it may not be explicitly stated that the safety function must *not* be invoked without proper cause. However, such a negative requirement is easy to verify with a model checker, and it is requirements such as these that often reveal hidden faults.

So, while complete automation may be out of reach, tools that would support the modeler in formalizing the requirements can still be outlined. The modeler needs assistance in understanding complex temporal logic formulas that can easily become convoluted. Concepts for supportive tools we have been working on include:

- Visualizing the temporal logic formulas as graphs (state machine diagrams) for easier interpretation.
- Visualizing exemplar execution paths that either conform to or are contrary to a temporal logic formula e.g. as graphs. See also [20].
- Translating the temporal logic formulas to natural language (e.g. "Starting from system state A, the system should never set the output B, until we get an input C.") to expose e.g. syntactic incorrectness. The modeler could also be using such a tool to describe the formulas, rather than writing temporal logic. See also [21].
- A set of templates for oft-used temporal logic constructs, i.e. "design patterns" for formulas.
- A "requirement management" tool for mapping each temporal logic clause to its origin in the system requirements, supporting the back-tracking of fault sources.

5.4 Visualization of Results

When the model checker discovers a fault, i.e. an execution path of the system model that goes against the system requirements, that execution path is given as a counter-example scenario. NuSMV, for example, typically outputs hundreds of lines of text.

For the system designer, it would naturally be easiest to see the counter-example presented in the familiar, original formalism used in the designer's tool. For function block diagrams, that would mean visualizing through 2D animation how the signal values change through time, either by coloring the wires for binary signals, or displaying the values of analog signals, timers etc. next to the wires or block ports. The designer could then play the execution back and forth to trace the source of the fault. Furthermore, the counter-examples could also be exported as scenarios for a process simulator. In this manner, the effects of faults in the I&C system under inspection could be e.g. examined concurrently with a simulation model of the controlled plant.

The visualization of the counter-examples in 2D graphics in the Simantics platform is a feature we will be working on in the future. For other approaches for helping the user understand counter-examples, see e.g. [22], [23].

6 CONCLUSIONS

Model checking has proved to be such an efficient tool for verifying and validating safety-critical I&C system software that it could soon be extensively used in both design and evaluation of nuclear power plant control systems. We have not only focused on gaining the necessary modeling experience, but also on developing tools for efficient application of model checking already in the system design phase. Discovering faults in control system design as early as possible is a key factor in reducing costs, as all changes in safety-critical software design propagate through numerous steps in the overall system development life-cycle.

While some model checkers already contain graphical tools for specifying at least some aspects of the system under inspection, the fact is that few (if any) model checkers are designed with any specific application area in mind. The NuSMV tool alone can be used for a wide range of application types, and taking the modular approach and reusing the elementary modules is surely not the only way to do the modeling, let alone a specifically supported one. However, taking the modular approach as a starting point, and employing the interoperability provided by the Simantics platform, we have been able to partially automate the model construction task. Further development is underway for automating more of the tasks involved in the overall process of performing model checking of I&C system design.

Our starting point has been that the system under inspection is expressed with function block diagrams. Naturally, this is not the case for many safety-critical I&C applications. Neither is it necessitated by any model checking tool. If the design specification does not use function blocks, model checking can still be – and certainly has been – utilized. However, the developments presented in this paper depend on the modular structure of function block diagrams, at least for now. For other kind of applications there still is a need for more manual work and interpretation for the modeler.

The Finnish nuclear sector has shown interest in model checking, as well as the interoperability provided by the Simantics platform – the ability to e.g. convert control system

software models from tools used by designers to the format needed by process simulators such as Apros. The ability to include model checking in the mix is a natural next step, and the expected benefits are mutual: easier implementation, more powerful tools.

7 REFERENCES

1. E. M. Clarke, Jr., O. Grumberg, D. A. Peled, *Model Checking*, The MIT Press (1999).
2. R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, A. Tchaltsev, "NuSMV 2.5 User Manual", ITC-IRST, http://nusmv.irst.itc.it/.
3. 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).
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. 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).
6. J. Valkonen, M. Koskimies, V. Pettersson, K. Heljanko, J.-E. Holmberg, I. Niemelä, J. J. Hämäläinen, "Formal verification of safety I&C system designs: Two nuclear power plant related applications", *Enlarged Halden Programme Group Meeting - Proceedings of the Man-Technology-Organisation Sessions, C4.2.*, Institutt for Energiteknikk, Halden, Norway (2008).
7. J. Lahtinen, "Model checking timed safety instrumented systems", Research Report TKK-ICS-R3, Helsinki University of Technology, Department of Information and Computer Science, Espoo, Finland (2008).
8. J. Valkonen, V. Pettersson, K. Björkman, J. Holmberg, M. Koskimies, K. Heljanko, I. Niemelä, "Model-based analysis of an arc protection and an emergency cooling system", VTT Working Papers 93, VTT Technical Research Centre of Finland (2008).
9. 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)
10. E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, "Progress on the state explosion problem in model checking", *Lecture Notes in Computer Science*, **Volume 2000**, pp.176–194 (2001).
11. A. Valmari, "The state explosion problem", *Lecture Notes in Computer Science*, **Volume 1291**, pp. 429–528 (1996).
12. P. Godefroid, "Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem", *Lecture Notes in Computer Science*, **Volume 1032** (1996).

13. A. Biere, A. Cimatti, E.M. Clarke, Y. Zhu, “Symbolic model checking without BDDs”, *Lecture Notes in Computer Science*, **Volume 1579**, pp. 193–207 (1999).
14. A. Biere, K. Heljanko, T. Junttila, T. Latvala, V. Schuppan, “Linear encodings of bounded LTL model checking”, *Logical Methods in Computer Science*, 2(5:5) (2006).
15. D. Peled, L. Zuck, “From model checking to a Temporal Proof”, *Proceedings of the 8th international SPIN workshop on Model checking of software (SPIN '01)*, Springer-Verlag, London (2001).
16. K. S. Namjoshi, “Certifying model checkers”, *Lecture Notes in Computer Science*, *Volume 2102* (2001).
17. Simantics – a software platform for modeling and simulation, https://www.simantics.org/.
18. 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)
19. Apros Process Simulator Software, http://www.apros.fi/en/.
20. E. Clarke, H. Veith, “Counterexamples Revisited: Principles, Algorithms, Applications”, *Lecture Notes in Computer Science*, **Volume 2772**, pp. 41-43 (2004).
21. K. Loer, M. Harrison, “Integrating Model Checking with the Industrial Design of Interactive Systems”, *26th International Conference of Software Engineering* (2004).
22. SY Shen, Y. Qin, SK Li, “A Faster Counterexample Minimization Algorithm Based on Refutation Analysis”, *Proceedings of the conference on Design, Automation and Test in Europe (DATE '05)*, pp. 672—677. IEEE Computer Society. Washington, DC, USA (2005)
23. S. Chaki, A. Groce, O. Strichman, “Explaining abstract counterexamples”, *ACM SIGSOFT Software Engineering Notes*, **Volume 29**, pp. 73—82, (2004)