Title	A Toolset for model checking of PLC software
Author(s)	Pakonen, Antti; Mätäsniemi, Teemu; Lahtinen, Jussi; Karhela, Tommi
Citation	18th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA2013, 10-13 September 2013, Cagliari, Italy Proceedings.
Date	2013
Rights	© 2013 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

A Toolset for Model Checking of PLC Software

Antti Pakonen, Teemu Mätäsniemi, Jussi Lahtinen, Tommi Karhela
VTT Technical Research Centre of Finland
P.O. Box 1000, 02044 VTT
Espoo, Finland
antti.pakonen@vtt.fi, teemu.matasniemi@vtt.fi, jussi.lahtinen@vtt.fi, tommi.karhela@vtt.fi

Abstract

Model checking is a powerful formal verification method that can also be used to evaluate PLC software. A lot of manual work and some expertise are still needed. Proposed methods for automating the process rely on standardised specification languages, but PLC software is often vendor-specific, and the source code for function blocks may not even be available.

We propose a toolset for model checking of function block based software. After manually modelling the elementary function block library, the model of any block diagram can be specified with easy-to-use graphical tools. The counterexamples output by the model checker can also be visualised using a "living" function block diagram. Our toolset is based on integrating the popular model checker NuSMV with the open source modelling platform Simantics.

1. Introduction

Model checking has been proved to be an effective method for the verification and validation (V&V) of programmable logic controller (PLC) software. Due to the exhaustive analysis, design errors can be found in control systems that have already been evaluated using more traditional methods such as testing or simulation. Verification of the functionality of C or Java code may not be fully feasible yet, but function block diagrams – a very common programming language for PLCs – can more easily be translated to formalisms used in model checking.

Still, model checking is not widely used in control engineering. One reason is the lack of dedicated tools, which means that a lot of manual work is needed to carry out the process.

In this paper, we propose a toolset for model checking of PLC software expressed with function block diagrams. After a function block model code library has been constructed and verified, the modelling of any application can be done using graphical user interfaces, or the block diagram structure can be directly imported from design data. The counterexamples produced by model checkers can also be presented in a convenient format.

In the following, we specifically avoid using the abbreviation FBD when discussing function block diagrams, since FBD is closely associated with the language defined by IEC 61131-3. Our approach is as well suited to any vendor-specific function block specification as it is to the IEC 61131-3 FBD.

2. Model checking of PLC software

2.1. Model checking

Model checking [1] is a powerful formal method for the verification of a system design model. A software tool called a model checker is used to automatically determine if a specified property is true, taking into account all possible states and executions of the model. Because of the exhaustive (but still quite fast) analysis, model checking has obvious advantages over traditional approaches such as simulation or testing.

The main challenge in model checking is to deal with the state explosion problem. The number of model states grows exponentially with the number of model inputs and interacting components [1]. A key technique in avoiding the problem is to employ symbolic verification, which is based on the manipulation of Boolean formulas. Binary decision diagrams (BDD) are used in several tools to allow the verification of systems whose extremely large state space would render explicit enumeration methods useless [2]. The model is usually based on a finite state machine (FSM) representation, and a variant of temporal logic is typically used to formulate the properties.

If a model checker finds a model behaviour that is contrary to a specified property, the behaviour is returned to the user as a counterexample (an error trace). Analysis of the counterexample can reveal a design error, but also an error in the way the model or the property was specified. The method can therefore be said to be self-repairing to some degree.

NuSMV is a BDD-based symbolic model checker allowing for the representation of both synchronous and asynchronous finite state systems, with a discrete

representation of time. Properties can be specified using Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) [1]. The open source model checker has been quite widely adopted, especially in the field of research, and it has also been used in our work.

2.2. PLC software verification

Since 2007, we have been applying model checking in practical customer work, by performing independent verification of I&C software for both the Finnish Radiation and Nuclear Safety Authority (STUK), and recently also the Finnish nuclear power company Fortum. Our experience has shown that model checking is a very useful addition to the set of existing V&V tools [3].

The problem is that a lot of work and expertise is needed in applying model checking to a particular domain such as control software. Since no dedicated tools are available, manual work and ad hoc solutions are needed to construct the model. Attempts at automating the process are based on standard languages that are hardly universally adopted (see next Chapter).

Specifying the properties to be verified is another problematic issue, linked to the general challenges in requirement engineering. Natural language requirement specifications are often vague and ambiguous, whereas model checking depends on exact formal representation.

A very significant practical problem is the effort spent on the interpretation of counter-examples [4]. Typically, the counterexample is returned as hundreds of lines of text listing signal values in different states. Manually browsing through such data – visualised, at best, with hundreds of trend graphs – makes it difficult to pinpoint the underlying problem.

3. Related research

Model checking of PLC software is a topic addressed by several authors using a range of different methods [5][6]. The attempts at facilitating the use of model checking often aim at automating aspects of the overall process, with emphasis usually on transforming the model automatically based on the original PLC program. Several approaches are proposed based on the standard programming languages of IEC 61131-3: Function Block Diagram (FBD) [7][8], Instruction List (IL) [5], Structured Text (ST) [6], Sequential Function Chart (SFC) [9], and Ladder Diagram (LD) [10], just to list a few.

The issue with these approaches is that they are (quite understandably) based on IEC 61131-3, which – although quite well known – is not universally adopted. Also, it is assumed that the implementation or the source code of the elementary function blocks is known and available, which may not be the case. Furthermore, the methods often have limitations related to, e.g., processing of timing or analog signals: the models may require manual tweaking to properly handle delay or counter blocks [8], or such variables can be omitted altogether [6][9].

In order to make the counterexamples returned by model checkers easier to understand, different approaches have been proposed. From a theoretical perspective, minimisation algorithms that eliminate irrelevant variables from the counterexample are a common topic [11]. A more practical viewpoint is to focus on the visualisation method.

An intuitive visualisation is often achieved through the use of timing graphs, or trends [4], a common way to display numerical data over time. Still, when dealing with hundreds of model variables, it might be difficult to understand what is relevant.

In general, the preferable visualisation method depends on the users' background and application domain, with "model animation" sometimes being a favoured view [12]. The idea is to animate the counterexample in the context and presentation of the original system model, which in our case would be the function block diagram.

4. Modular approach to model checking of PLC software

4.1. The function block model code library

Our approach for model checking function block based PLC software is based on the manual specification of model checker code for each elementary function block. The justification for resorting to manual block specification is as follows:

1. Instead of using standard languages as specified by IEC 61131-3, many major PLC system vendors use vendor-specific function blocks.
2. Often, vendors are unwilling to disclose the actual implementation algorithms for the elementary function blocks (black box), addressing them as non-propriety intellectual property. Only the functional description is handed to customers.
3. Even if the internal logic of the blocks were revealed (white box), it is likely that the implementation is based on languages such as Java or C, which does not usually enable direct representation in tools such as NuSMV.

The starting point for modelling is therefore most often the *functional description* of the elementary function blocks. The corresponding model checker code is then written for each block. Model checking can be used to ensure that the block model code is correct, by verifying properties derived from the functional description.

As a small example, Figure 1 below shows a fragment of a (non-standard) function block diagram with two

connected blocks: a greater-than comparison block, and a bistable set-reset memory block.

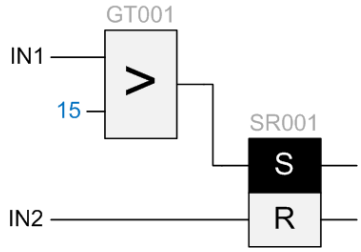

Figure 1. An exemplar fragment of a function block diagram with two connected blocks.

The corresponding code in NuSMV for the two function blocks would then read as follows:

```
MODULE GT(in, limit)
DEFINE
   out := (in > limit) ? TRUE : FALSE;

MODULE SR(set, reset)
VAR
    memory : boolean;
DEFINE
    out1 := case
       set : TRUE;
       !set & reset : FALSE;
       TRUE : memory;
   esac;
   out2 := !out1;
ASSIGN
   init(memory) := FALSE;
   next(memory) := out1;
```

The code for invoking and connecting the two block objects on the diagram would then read:

```
VAR
   IN1 : 4..20;
   IN2 : boolean;
   GT001 : GT(IN1, 15);
   SR001 : SR(GT001.out, IN2);
```

NuSMV contains only quite basic expressions and simple types (boolean, integer, enumeration, bit word, array), meaning that algorithmically rich function blocks (e.g., PID) cannot be modelled with reasonable precision. Complex control logic is therefore either abstracted away, or completely omitted from verification.

4.2. Function block diagram modelling

Once the library of model code modules (that corresponds to the vendor-specific set of elementary function blocks) has been constructed, the task of modelling any function block diagram is reduced to making the same block connections in the NuSMV model. This is achieved by either a) manually copying the diagram structure, or b) importing the diagram structure directly from another tool. For these purposes, we use the Simantics tool introduced in the following chapter.

4.3. Challenges in diagram modelling

A key obstacle for the applicability of model checking is the state space of the FSM growing too rapidly. In our work, we have rarely found this to be an issue. Especially when verifying binary logic, the method scales very well, with analysis times for FSMs with 10^{40} states still in the range of minutes, if not seconds. Even with analog (integer) values and simple math, NuSMV is very efficient. Problems typically arise from excessive amount of feedback loops, and, e.g., function blocks storing integer data into memory.

There are, however, aspects of modelling function block diagrams that require specific attention from the modeller: timing, asynchrony (in distributed systems), and the discretisation of analog variables. Also, since NuSMV can only handle integer numbers, some scaling back and forth is sometimes necessary to properly model simple arithmetic.

5. A graphical toolset for model checking

5.1. Simantics

Simantics is an open, ontology-based integration platform for different modelling and simulation tools [13]. Originally developed at VTT, it is now released and maintained as an open source tool by THTH Association of Decentralized Information Management for Industry (http://www.ththry.org).

Simantics has a client-server architecture based on the Eclipse framework. The plugin user interface and the semantic modelling kernel enable easy integration of range of (commercial and non-commercial) simulation and engineering tools. Examples of industrial tools already integrated into the environment include OpenModelica, BALAS, Apros, SULCA, OpenFoam, Comos and SmartPlant.

For our purposes, Simantics provides a graphical modelling framework with model structure browsing and editing, model component reuse, support for model validation, version control, team features and documentation, and most of all easy integration with an existing model checker tool. Eventually the design (CAD) system connections provided by Simantics should enable automatic model import based on data available from software development tools [13].

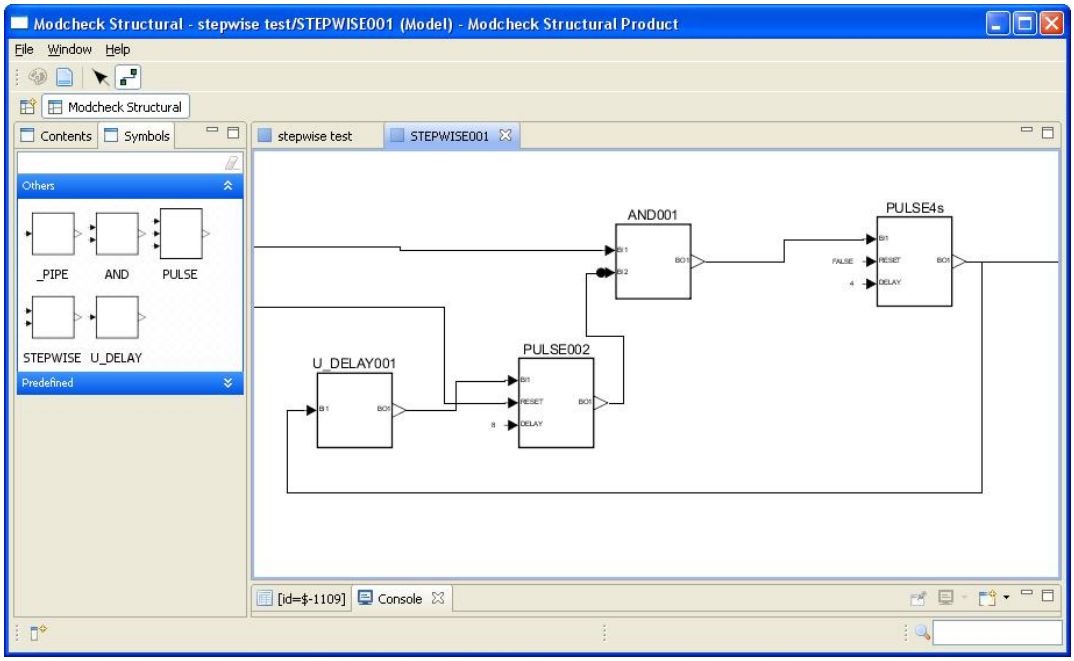

Figure 2. The Simantics modelling tool allows the user to specify the function block diagram in a graphical view, and generates the corresponding input file for the NuSMV model checker.

5.2. NuSMV integration

We have developed a model checking tool for Simantics, currently enabling us to construct or reproduce the function block diagram using a graphical user interface, and to transform the diagram structure to an input file for NuSMV.

The model checking tool consists of four main plugins. There are two ontology plugins, a plugin for NuSMV integration, and a plugin for user interface and model update features. The first ontology plugin defines foundation concepts for modelling, while the other plugin utilizes them to establish different predefined concepts, such as the variable types that are available. The NuSMV integration plugin enables configuration management for different model checker versions, and provides an interface for communication with NuSMV. The user interface plugin binds the elements together, and provides features for model management.

5.3. Modelling with Simantics

For the purpose of specifying the function block model code library, the user is provided with a text-based code editor. For each elementary function block type, the user will specify the block interface (input and output ports), write the internal code, and define the graphical element. Once constructed, the function block library can be exported and shared between different users.

After the function block model code library is complete, the user can construct the system models in a 2D graphical view by adding function block objects in a drag-and-drop fashion, and wiring them together (Figure 2). Default input values or binary port negations can also be assigned directly to block input ports.

Composite function block types can also be defined. The internal logic is first composed with a set of function blocks. The logic is then encapsulated within a composite block, and a suitable graphical element is again defined. After this, the composite block objects can be added to the diagrams in a drag-and-drop fashion. The (potentially multi-level) model hierarchy can be directly browsed, as double-clicking any composite block on the diagram will reveal the underlying internal logic.

The user can then generate an input file for NuSMV containing all the necessary model elements (module code for the used elementary blocks, as well as upper level modules recreating the block connections). The system properties are currently input using a plain text editor (See chapter 5.5).

5.4. Counterexample visualisation

A counterexample output by a model checker is an exemplar sequence of model states that shows a behaviour that is contrary to a specified property. Closer analysis of the counterexample can reveal a design error (or an error in the model or the property specification).

The tool displays the counterexample as an animated “living” function block diagram, by using monitors and different colours and line styles to show changing model variables (see Figure 3).

Binary signal values are shown by changing the colour and thickness of the block connection wire. The

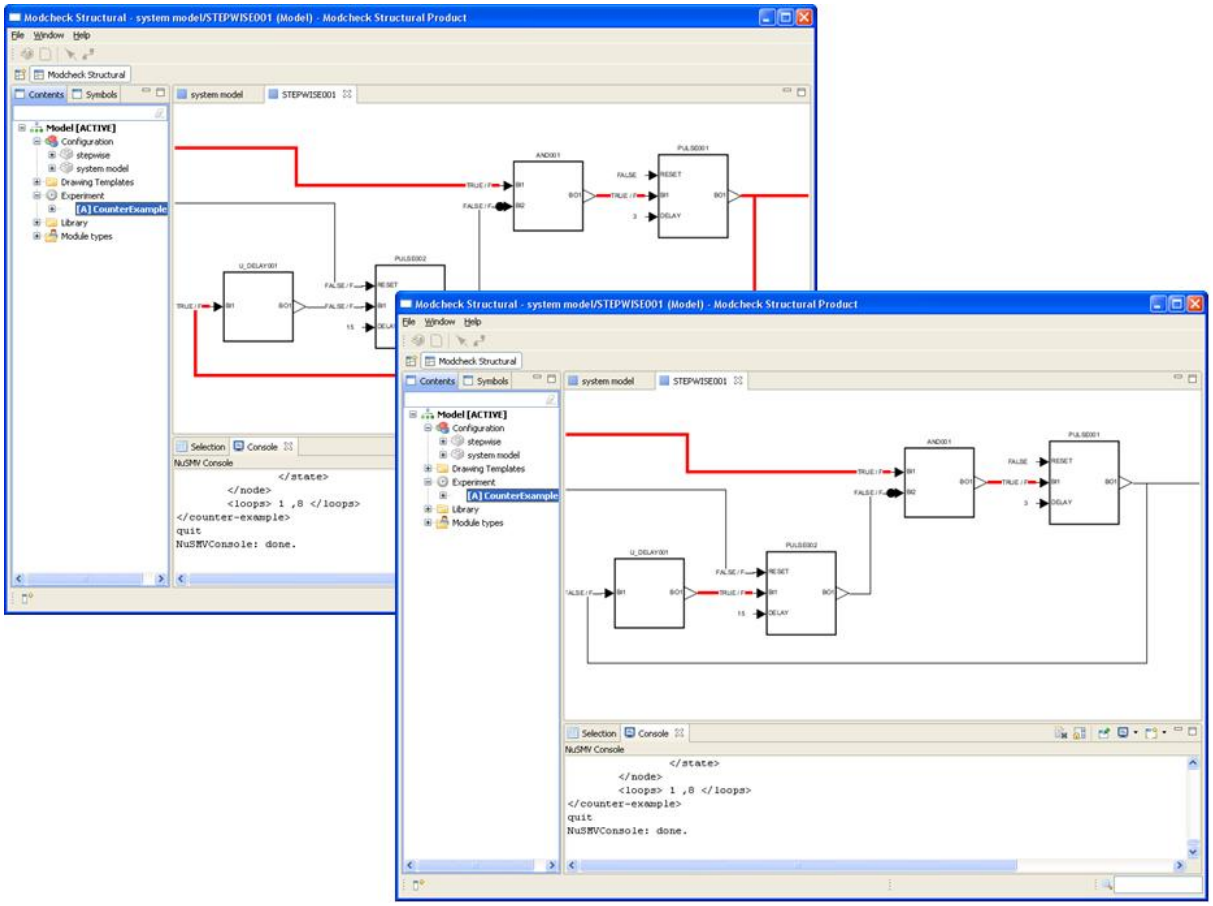

Figure 3. The counter-example output by the model checker is presented to the user as an animation illustrating how signal values change – a "living" function block diagram.

default setting is a thick red line for one (true), and a thin black line for zero (false). The changing thickness will add contrast, and using only one colour will help avoid contradictory interpretations and issues for users with impaired colour vision.

Analog values (number or text data) are shown with monitors attached to the input gates of each function block.

If supported by the function block specification, data indicating the status or validity of the signal can be visualised using dashed connection wires.

Since the counterexample is a sequence of states, the user is able to play it back and forth as an animation. The animation can be paused and browsed step-by-step. As the animation is being browsed, the user can also move between different levels of the model structure, i.e., to look inside composite function blocks. An engineer or analyst familiar with the function block diagram notation can quite quickly pinpoint the exact cause of the unwanted behaviour and the potential design issue.

The user is also provided with a trend view of model signals.

5.5. Future work

In addition to the system model, the properties specifying proper system behaviour are also needed as an input for the model checker. Typically expressed using temporal logic, the formulations can become convoluted and difficult to understand. To ease the task, we are currently working on a template-based approach for property formalisation, motivated by the seminal work on property specification patterns by Dwyer et al. [14].

Since Simantics is essentially an integration platform, our next objective is to implement automatic import of the function block diagram structure from, e.g., an existing (legacy) software development tool.

We are also taking part in more theoretical research with the Aalto University on topics such as 1) compositional analysis of very large models using iterative abstraction refinement, 2) developing tools and methods for analysing asynchronous phenomena in distributed systems, and 3) modelling of failure modes of the underlying PLC hardware.

6. Conclusions

Careful verification and validation of PLC software is an important issue for not only systems that are safety-critical, but also for applications whose downtime leads to substantial financial loss. Practise has shown that model checking is a valuable V&V method, since hidden errors can be found from systems already evaluated using more traditional means.

Although the use of relatively simple modelling languages means that systems with arithmetically complex elements (PID blocks, for example) cannot be effectively analysed, model checking is very powerful in the verification of more straightforward binary logic. Function block diagrams specify a clear input-output mapping and have a well-defined modular structure. Both these factors make model checkers like NuSMV a viable tool for the evaluation of PLC software.

Due to lack of domain-specific tools, however, application of model checking is all but mainstream in the control engineering domain. A lot of time and expertise are needed, as is common with formal methods. Other proposed methods of automating the work process rely on the use of standard programming languages (most often IEC 61131-3) and/or access to function block source code, both of which are far from guaranteed when dealing with many system vendors.

Having to specify the function block model code library manually may seem rudimentary, but in many cases, it might be the only option, and it only has to be done once for each vendor-specific function block specification. Certainly the approach is universal, and our claim that it is also very effective is supported by the several years of experience we have in model checking real-world systems in customer projects.

Using the open source modelling platform Simantics, we have been developing tools to support model checking of function block based software. The advantages have been obvious, since (notwithstanding the initial work phase of defining block model code library) specifying the model now requires very little understanding of the modelling languages involved.

By far the most useful feature, however, is the visualisation of counterexamples as a "living" function block diagram, a step-by-step animation that can be freely manipulated. This intuitive presentation is clearly superior to trend displays in helping the modeller or analyst find the root of the cause. This sophisticated feature would also be difficult to implement if the model specification used in model checking did not follow the modular structure of the original function block diagram.

References

[1] E. Clarke, Jr., O. Grumberg, D. Peled, *Model Checking*, The MIT Press, 1999.

[2] J. Burch, E. Clarke, K. McMillan, D. Dill, L. Hwang, "Symbolic Model Checking: 10^{20} States and Beyond", *Information and Computation*, Vol. 98, pp. 142-170, 1992.

[3] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain", *Reliability Engineering and System Safety*, Vol. 105, pp. 104-113, 2012.

[4] E. Jee, S. Jeon, S. Cha, K. Koh, J. Yoo, G. Park, P. Seong, "FBDVerifier: Interactive and Visual Analysis of Counter-Example in Formal Verification of Function Block Diagram", Journal of Research and Practice in Information Technology, Vol. 42, pp. 171-188, 2010.

[5] B. Schlich, J. Brauer, J. Wernerus, S. Kowalewski, "Direct Model Checking of PLC Programs in IL", *2nd IFAC Workshop on Dependable Control of Discrete Systems*, Bari, Italy, June 10-12, 2009.

[6] V. Gourcuff, O. De Smet, J. Faure, "Efficient representation for formal verification of PLC programs", *8th International Workshop on Discrete Event Systems*, Ann Arbor, MI, USA, July 10-12, 2006.

[7] D. Soliman, K. Thramboulidis, G. Frey, "Transformation of Function Block Diagrams to UPPAAL timed automata for the verification of safety applications", *Annual Reviews in Control*, Vol. 36, pp. 338-345, 2012.

[8] J. Yoo, S. Cha, E. Jee, "Verification of PLC Programs Written in FBD with VIS", *Nuclear Engineering and Technology*, Vol. 41, pp. 79-90, 2009.

[9] R. Huuck, *Software Verification for Programmable Logic Controllers*, Dissertation, University of Kiel, 2003.

[10] O. Rossi, P. Schnoebelen, "Formal Modeling of Timed Function Blocks for the Automatic Verification of Ladder Diagram Programs", *The 4th International Conference on Automation of Mixed Processes (ADPM 2000)*, Dortmund, Germany, Sep 18-19, 2000.

[11] K. Ravi, F. Somenzi, "Minimal Assignments for Bounded Model Checking", K. Jensen, & A. Podelski (Eds.): *Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science*, Vol. 2988, pp. 31-45, 2004.

[12] K. Loer, M. Harrison, "Integrating Model Checking with the Industrial Design of Interactive Systems", *26th International Conference on Software Engineering*, Edinburgh, Scotland, UK, May 23-28, 2004

[13] T. Karhela, A. Villberg, H. Niemistö, "Open ontology-based integration platform for modeling and simulation in engineering", *International Journal of Modeling, Simulation, and Scientific Computing*, Vol. 3, 2012.

[14] M. B. Dwyer, G. S. Avrunin, J. C. Corbett, "Patterns in Property Specification for Finite-State Verification", *Proceedings of the 21st International Conference on Software Engineering*, ACM, New York, 2012.