MOS: An Integrated Model-Based and Search-Based Testing Tool for Function Block Diagrams

Eduard Paul Enoiu*, Kivanc Doganay*†, Markus Bohlin†, Daniel Sundmark*, and Paul Pettersson*
*Mälardalen University, Västerås, Sweden
eduard.paul.enoiu@mdh.se, daniel.sundmark@mdh.se, paul.pettersson@mdh.se
†Swedish Institute of Computer Science, Västerås, Sweden
kivanc.doganay@sics.se, markus.bohlin@sics.se

***Abstract*—In this paper we present a new testing tool for safety critical applications described in Function Block Diagram (FBD) language aimed to support both a model and a search-based approach. Many benefits emerge from this tool, including the ability to automatically generate test suites from an FBD program in order to comply to quality requirements such as component testing and specific coverage measurements. Search-based testing methods are used to generate test data based on executable code rather than the FBD program, alleviating any problems that may arise from the ambiguities that occur while creating FBD programs. Test cases generated by both approaches are executed and used as a way of cross validation. In the current work, we describe the architecture of the tool, its workflow process, and a case study in which the tool has been applied in a real industrial setting to test a train control management system.**

***Index Terms*—model-based software testing, search-based software testing, timed automata, programmable logic controllers.**

I. Introduction

Industrial safety-critical systems implemented in *Programmable Logic Controllers* (PLCs) are widely used in avionics and the railway domain. One of the programming languages defined by the *International Electrotechnical Commission* (IEC) [1] for PLCs is the *Function Block Diagram* (FBD), a standard widely used to implement safety-critical software [2]. Programs developed in FBD are transformed into program code, which is compiled into machine code automatically by using specific engineering tools provided by PLC vendors. Due to the fact that the program transformation is implemented differently depending on the tool vendor, the resulting code contains less, the same, or more instructions and operations than the FBD program [3], [4]. Structural coverage is used in safety-critical systems not only to satisfy the criteria but also to identify missing functionality both in the code and the FBD program. Hence, it is important to provide an approach for testing FBDs from program level to code level and to build knowledge on how this can influence the testing activity.

Testing tools for FBDs have been under scientific study for some time, and relies mostly on functional testing or simulation methods [5], [6]. There has been no research on applying model-based and search-based testing approaches for FBD programs in an industrial setting.

In this paper search-based test generation focuses on satisfying particular coverage criteria at the FBD code level. Even though it is code that is executed while testing, the semantics of the FBD program under test is not preserved at code level. Therefore, we use a model-based method of generating test data for FBD programs before its transformation to executable code that can ensure compliance to quality requirements including unit testing and model coverage requirements.

MOS is a tool for model-based and search-based testing of safety-critical systems implemented using the FBD language, developed at *Mälardalen University* since 2012. As shown in Fig. 1, it allows its users to automatically generate test suites from FBD programs. The model-based testing approach is based on behavior models and uses a model-checker to automatically generate test suites. The output of the MOS tool is used as test programs that can be used on the system under test. The main features include:

- A framework for producing test suites for FBD programs using the model checker's ability to generate diagnostic trace witnessing a submitted test property or coverage criteria. This is achieved by using the UPPAAL [7] model checker to perform symbolic reachability analysis of FBD programs modeled as a network of timed automata.
- A set of coverage criteria, including *decision coverage* and *condition coverage* are used to specify for which requirement a test suite should be generated.
- Support for search-based test generation to maximize *modified condition/decision coverage* (MC/DC) for assignments with logical expressions in the FBD program code.
- Compatibility with the file format used for generating FBD programs [8].

II. Preliminaries

PLCs are used in control software systems from nuclear power plants to traffic control systems. A PLC is an industrial real-time system, containing a processor, a memory, connected together by a bus. Software on a PLC runs in a loop, in

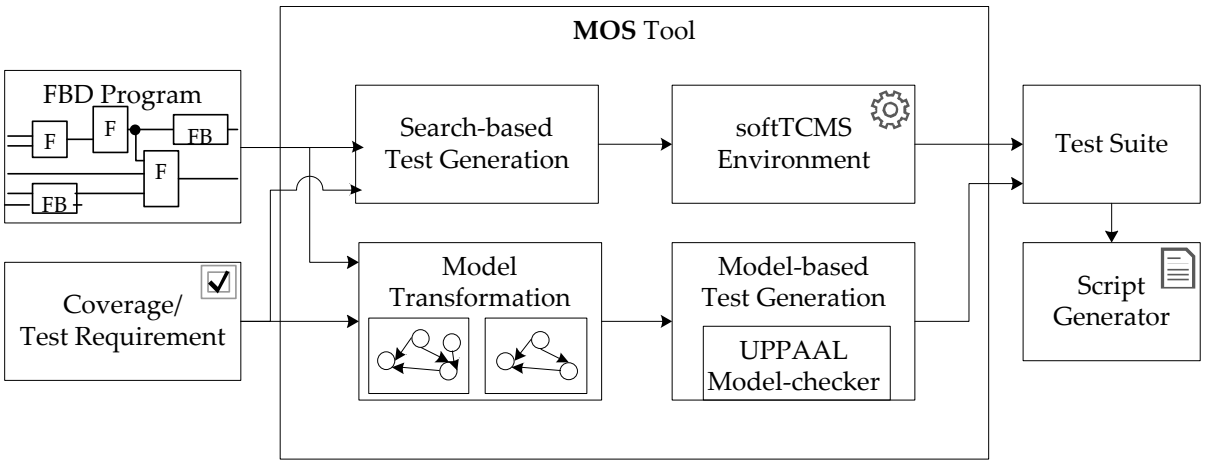

Fig. 1. Combined Testing Tool Architecture and Environment.

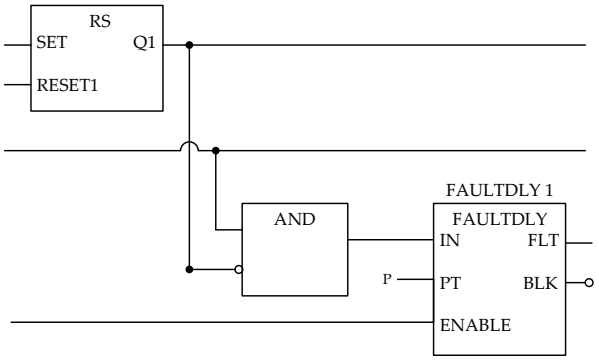

Fig. 2. An FBD program showing the graphical nature of the language.

which the iteration follows the *"read-execute-write"* semantics. When a PLC reads all inputs, it executes the computation, and then writes to its outputs without interruption. FBD, a PLC programming language standardized by IEC 61131-3 [1], is used in practice because of its graphical representation and its data flow model of the computation. An example of an FBD program depicting a `Loadshed Contactor Control` is shown in Fig. 2. An FBD program is composed of *Functional Elements* (*FE*) defined as *Function Blocks* (*FB*) and *Functions* (*FUNC*). Independent of the PLC choice language for IEC 61131-3, *FUNC*s and *FB*s are the base for a structured and hierarchical FBD program. They are supplied by the manufacturer, defined by the developer, or predefined in a library.

Basically, the model elements are equivalent to predicates and instrumentation points shown in a circuit diagram fashion. For instance in Fig. 2 the system consists of some basic logic, timer, state functions such as AND, and function blocks as FAULTDLY and RS (Reset-Set Latch). A *FUNC* does not have any internal state and its output is determined only by the current inputs. In Fig. 2, AND is a *FUNC*. Differently, FAULTDLY and RS are *FB*s because they both maintain an internal state and are producing outputs based on this state and inputs. The IEC 61131-3 standard proposes a hierarchical software architecture for structuring and running any FBD program. This architecture specifies the syntax and semantics of a unified control software based on a PLC configuration, resource allocation, task control, program definition, function and function block repository, and program code [2], [9], [10]. The program code is generated from the FBD program used on a specific PLC by utilizing a model-to-code transformation.

III. Tool Overview

In this section, we describe the MOS tool architecture focusing on the search-based and model-based capabilities. The workflow of the tool and the architecture is shown in Fig. 1. The user creates an *FBD Program* and a *Test Requirement*. The *FBD Program* is handled by both *Search-Based Test Generation* and the *Model-Based Test Generation*. To specify the coverage information we use a *Test Requirement* query file to be used in the MOS tool. The tool currently supports: (i) search-based test case generation, (ii) model-based test generation, and (iii) simulation of the test platform using a Train Control and Management System Simulation (SoftTCMS Environment)[1]. The tool is developed with the purpose to automate and improve the FBD program testing process. Faults found automatically in this process can be fixed as the developer is getting feedback from the tests immediately. FBD program testing has a direct impact on the final implementation, with respect to functional, performance, and other quality attributes. Therefore, guaranteeing that FBD programs meet the specified requirements is beneficial for detecting faults early in the testing process.

A. Model-Based Test Generation for FBDs

The modeling language used by MOS is based on the timed automata (TA) model. In MOS, FBD programs are built from interconnected components with well-defined interfaces and transformed to TA models. MOS is based on functional and timing behavior models and uses a verifier to parse the model language. The model-based test generation is tailored for FBD

[1]SoftTCMS Environment is a software simulation of the entire train.

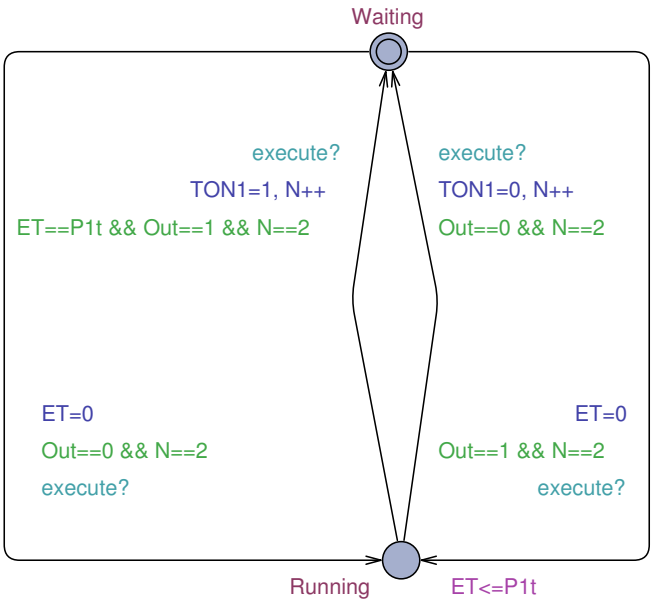

Fig. 3. Timed Automata Model for a TON Function Block.

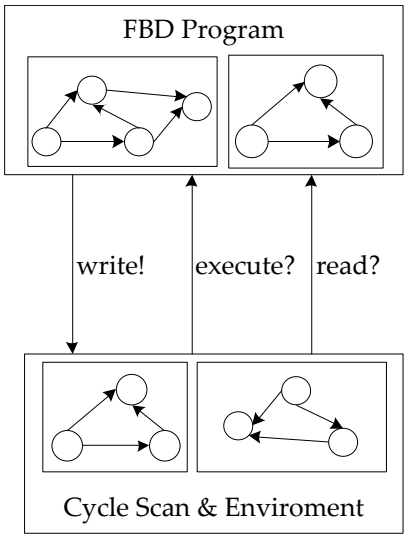

Fig. 4. Timed Automata Network used by the Model-based Test Generation.

programs with both safe and timed behavior. We provide a generator for producing test suites for FBD programs using a model checker's ability to generate a diagnostic trace witnessing a submitted test property or coverage criteria from the *Test Requirement* file. This is achieved by using the UPPAAL [7] model checker to perform symbolic reachability analysis of FBD programs modeled as a network of TA. Basically, the reachability algorithm explores the symbolic state space of a TA model and generates time-optimal traces based on a variation of the A*-algorithm [11]. UPPAAL is a model-checker using TA as a modeling language [2]. It supports *real-valued clocks* and different data types like *bounded integers* and *arrays*. The verification language supports properties such as *safety, liveness, and reachability*.

1) Timed Automata: The TA model was introduced by Alur and Dill [12] and has gained a lot of attention as a modeling language for timed systems. We give here a short description for readers unfamiliar with this model.

Let X be a finite real-valued variables called clocks and $B(X)$ the set of guards, which are finite conjunctions of constraints of the form $x \bowtie n$, where $x \in X$, $n \in \mathbb{N}$, and $\bowtie \in \{<, \leq, =, \geq, >\}$. A *timed automaton* (TA) over clocks X and actions Act is a tuple $\langle L, l_0, E, I \rangle$ where L is a finite set of locations, l_0 is the initial location, $E \subseteq L \times B(X) \times Act \times L$ is the set of edges and $I : L \to B(X)$ assigns invariants to locations. In the case of and edge $\langle l, g, a, r, l' \rangle \in E$, we write $l \xrightarrow{g,a,r} l'$ where the label g is a guard of the edge, r is the data- or clock reset assignments of the edge, and a is the action of the edge.

A network of TA $T_1 \parallel \ldots \parallel T_n$ is a parallel composition of n TA over X and Act, synchronized actions (i.e., $a!$ is complementary with $a?$) and shared variables. We refer the reader to [13] for more information on the theory of TA.

2) Transforming FBDs to Timed Automata: To support model-checking, MOS enables the transformation of FBD programs into TA models, being one step away from test suite generation with the UPPAAL tool. The transformation maps all the interface elements alongside the existing timing annotations within the FBD program into a TA model. MOS performs an automatic transformation of the FBD program to a TA model that obeys the *read-execute-write* semantics. As a result of the transformation local automata are composed in parallel into a TA network. The purpose of the transformation is to construct a target model by associating the TA with a corresponding behavior. For example, a rather straightforward model of the `TON` function block is shown as a TA model in Fig. 3. The FBD program interacts with other TA via `execute?` action. `TON` is modeled by a standard time on timer that sets the output $TON1$ to true if *Out* input variable is true at least as long as the time $P1T$.

In the context of testing an FBD program we assume that the test specification is given as a closed network of TA as shown in Fig. 4. This model can be seen as two sub-networks, one modeling the *FBD Program* and the other one modeling its *PLC Cycle Scan*. This TA model is executed in a cycle loop, in which the the iteration of the FBD program follows the run-to completion semantics. The program operates in a specific environment, which obviously can be considered completely unconstrained containing all possible interactions between the TA network elements.

3) Test Generation for FBD Programs: Generation of test suites for an FBD program starts by manually formulating a set of informal test properties and continues with formalization of these such that the model is used for each test property. In this context, a test requirement is a specific test property that the tester would like to formulate. Properties of TA can be expressed as logical formulae in the Timed Computational

[2]The UPPAAL tool is available at www.uppaal.org.

Tree Logic (TCTL) [14]. In this paper we focus on properties of the form $\exists \Diamond p$, called reachability properties, where $\exists$ is the existential quantifier, and $\Diamond$ is the temporal operator. A reachability property states that there is a path in which the p location in the TA is reached.

For using the test generation capability of the UPPAAL model-checker, the test property must be formulated using the TCTL logics and checked by the TA model. Often we are interested in a test suite that ensures that the FBD program is covered in several ways. This ensures that a certain level of thoroughness has been achieved during the test generation process. Hessel et al. [15] already proposed a way to apply coverage criteria to TA models. In addition, we propose the usage of coverage analysis directly on the FBD program. For example, for decision coverage we analyze every decision points in the FBD program. Full decision coverage indicates that each decision in the FBD program has taken every outcome at least once. We implement a mechanism to facilitate this by specifying a set of decision parameters. We annotate the TA model with an auxiliary boolean variable v_i for each decision d_i to be covered. For every edge with destination $d_i : l \xrightarrow{g,a,r} d_i$, v_i is added to r assignment. The reachability property for full decision coverage will require that all v_i to be true.

B. Search-Based Software Testing for FBDs

Unlike the model-based approach, search-based test generation needs to execute the software under test. However, the FBD implementations are not directly executable on the target hardware. Instead, it is converted to C code using a proprietary FBD-to-C compiler, and then compiled with a C compiler (such as gcc) to be run on the target system. The generated code is not a special dialect of C, unlike what is common in the embedded domain (e.g., Embedded C and DSP-C), but is C99 compliant. Therefore, it is possible to compile it to run on a regular (non-embedded) computer. However, the C code uses a set of special libraries related to the actual hardware devices on a train. The SoftTCMS platform, developed by CrossControl, implements mock libraries and simulates most of the important peripheral devices on the train, such as the I/O Bus. Simply switching to SoftTCMS libraries allows us to run the same unit or component level test cases on the simulated environment, without the need to change the code under test.

Even though the C code generated from the FBD implementation is C99 compliant, it has some special properties that are not typically seen in software in other domains, due to the underlying FBD semantics. Each FBD program becomes a C function that do not use any static variable to access any global variable. So syntactically, functions appear to be stateless, which is actually not the case. The scheduler that periodically calls each FBD program is responsible for passing the same global data as an argument, which can be used by the FBD to preserve state. As an example, consider the following C code that implements a simple function block `R_TRIG`, which is one of the standard blocks in the FBD language:

```
void R_TRIG(struct_R_TRIG *data)
{
  data->Q = data->CLK & !data->M;
  data->M = data->CLK;
}
```

`R_TRIG` implements the rising edge detector, which outputs 1 only if the input at the previous execution cycle was 0, and it is 1 in the current execution cycle. The caller is responsible for preserving the integrity of the `data` variable instance of appropriate structure type, and call the `R_TRIG` function with the same input variable at each execution cycle. The input data element `CLK` is marked as an input variable, and `Q` as an output variable, while `M` is unmarked. This implies that a valid test case should be manipulating only the value of `CLK`, but not the non-input variables. However, this information is not visible at the C level, but only in the original FBD implementation.

Another property of FBDs that is lost at C code level, is the variables with fixed values, or the so called *parameters*. An FBD can have input values that are actually constant values through the system execution. An initialization procedure at the boot time of the system sets these parameters to their respective values, after which they are not altered. These values are passed to FBD function as part of the input data structure without any distinction that indicates the difference from other non-input variables.

Even though we create test input by executing the C code, it also needs to conform to the FBD semantics. Therefore, we parse the above mentioned semantic information from the FBD implementation, and create test inputs accordingly. The exact form of the C code is dependent on the particular FBD-to-C compiler that is used. However, there is a mismatch between what is visible in the C code and the actual FBD semantics, regardless of the compiler, making it necessary to parse the relevant information from the underlying FBD implementation.

1) Search Algorithm: The current implementation uses a slightly modified hill climbing algorithm to generate test data that satisfies MC/DC coverage for logical predicate assignments. As an example, consider an AND gate with three inputs. This FBD element gets compiled into a single C assignment in the form of $P = x \ \& \ y \ \& \ z$; where x, y and z are boolean variables, possibly computed by other functional elements. In order to have full MC/DC coverage, three distinct values are required for the (x,y,z) tuple. The goal set G_P consisting of the required tuples is:

$$G_P = \{(1,1,1),(0,1,1),(1,0,1),(1,1,0)\}$$

The hill climbing algorithm uses the minimum of the Euclidean distances from the current observed values to the elements of G_P, as an heuristic function, which can be formalized as below.

$$fitness = Minimum(|t_{obs} - t_i|,\ t_i \in G_P)$$

Where t_{obs} is the observed values for the (x,y,z) tuple during the execution of the current search node in the input space. Once a target tuple is observed (i.e., $fitness = 0$) it is removed

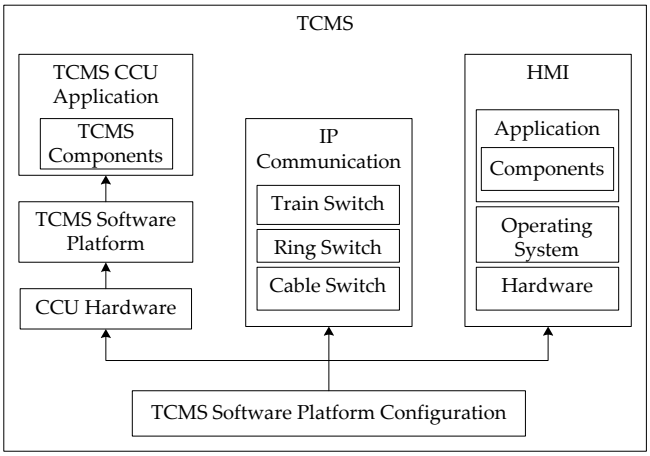

Fig. 5. A Simplified View of the Train Control and Management System.

from the goal set G_P, so that the search can continue in order to find other tuples. This heuristic drives the search towards the nearest element of the goal set, which is a slight improvement over the approach where the elements are separately targeted in a fixed order.

2) Search Space: The obvious choice of search space for the algorithm is the input space, representing possible inputs to the FBD program under test. However, if the FBD program has elements that are stateful, such as the *function blocks* explained in section II, it may not be possible to set an internal variable to the needed value in a single function call. Therefore, we represent a node in the search space as a sequence of input vectors. Subsequently, the fitness of an input sequence is the minimum among the fitness values of the function calls that are part of the sequence.

IV. Case Study

MOS has so far been applied on a real world case study in cooperation with *Bombardier Transportation AB*. We present here how the tool is applied to test a MITRAC *Train Control and Management System* (TCMS) provided within the ATAC research project [16]. TCMS is a distributed system, built on open standard *IP-technology* that allows easy integration of control and communication functions for high speed trains. The TCMS system consists of many different kinds of hardware and software units as shown in Fig. 5. The *TCMS Software Platform Configuration* is a major part of TCMS. Just like the *TCMS Application* refers to the distributed train control system, the *TCMS Software Platform Configuration* refers to the configuration of all configurable software items of TCMS. The *Central Computing Units* (CCUs) contain all FBD programs controlling the train. The PLC development tools used for developing these programs are based on the *MULTIPROG* software. The FBD code is generated using PLC development tools used specifically for TCMS.

The MOS tool aims to support testers and developers when testing FBDs at both program and code level. TCMS contains the software-based components of the train control system and is in charge of the operation-critical safety-related functionality of the train. Testing must therefore be focused on the most important functionality as exhaustive tests are practically unfeasible. The focus of testing must be considered in the context of the integrated train control system which controls all train operations, including propulsion, line voltage and passenger comfort systems. The goals of using MOS on TCMS component testing are to:

- *Systematically find faults in TCMS components.*
- *Aid the TCMS tester and developer in his ongoing work.*
- *Automatically improve test coverage for each TCMS component.*

A. Results

In the case study, we used MOS to produce test suites for ensuring that an FBD program is covered in several ways and that a certain level of thoroughness is achieved in the test generation process before the actual FBD program-to-code transformation. The FBD programs were transformed and modeled using MOS tool. Based on the TA model of the FBD program we used coverage analysis directly on the FBD programs based on the following criteria: (i) *decision coverage*, (ii) *condition coverage*, and (iii) *decision-condition coverage*. To perform the actual testing process, a complete test interface was built that supports automated generation of tests. MOS takes as input the FBD program together with a test goal, and generates ready-to-use test suites. The generation time for goals expressed as reachability properties is between 0.15 and 0.53 seconds [3] with 5 MB memory usage. Obviously, the environment model can pose restrictions to our results potentially obtaining more expensive test suites with regard to consumed resources. Still, for FBD programs in TCMS, MOS model-based test generation scales up for different models and coverage criteria. We refer the reader to [17] for more results on model-based test suite generation for FBD programs.

From our ongoing experiments and experiences with the MOS tool, the TCMS component test process is based on the actual FBD programs rather than requirements making it a perfect candidate for using both model-based and search-based testing. In this case, MOS assists the tester to identify automatically test cases, which can be scripted and executed by the developers. Testing TCMS components with MOS is done in isolation from the rest of the TCMS software in a controlled simulated environment, making the component tests a good target for coverage-based test generation and therefore a potential benefit to automatic evaluation of test cases.

Furthermore, we used MOS to generate test cases at the TCMS code level using the search-based testing approach. To ensure compliance to code coverage requirements, we generated test data that maximizes the MC/DC coverage of the C code generated from the FBD programs, which is the actual artifact to be eventually executed during operation. We also compared the performance of the current search algorithm

[3]Experiments were executed on a machine with 2.4 Ghz Intel Core i5 and 8 GB 1333 Mhz DDR2.

(a slightly modified version of hill climbing) against random testing on around 300 different FBD programs. Results show that most FBD programs have many easy to cover structures that allow random testing to perform surprisingly well against the more informed search algorithm. However, while trying to find test data for MC/DC coverage for more complex structures (e.g., predicates consisting of many clauses), hill climbing outperforms random testing. We discuss the details of this study (search-based vs. random testing) in [18].

Both model-based and search-based approaches are suitable for incremental testing. As new functionality is added to TCMS, new test cases can be added automatically to the component test specification by using both approaches, which is suitable for testing efforts early in the development process.

B. Implications

In this section we describe implications of our combined model-based and search-based test generation approach for research and practice.

Using the search-based test generation capability of the MOS tool it is possible to define the goal set differently in order to generate input data that achieves different types of testing goals. This is trivial for logical coverage (e.g., condition coverage), but if the testing goal is complex and structured differently it may not be possible. For example, it is not obvious how to translate an arbitrary TCTL formula into a set of concrete values, as in the MC/DC example. Such goals can be fulfilled by using MOS to formally verify temporal properties of the TA model and generate test cases.

An important fundamental limitation of the search-based approach is that, if the algorithm can not find an input data that satisfies a particular testing goal, it is not possible to deduce that the goal is infeasible. In this case the model-based test generation capability can be used to formally reason if the goal is reachable in the model. On the other hand, custom built functional elements that may be difficult to reason about (e.g. due to complicated or ambiguous specifications) do not affect the applicability of search algorithms.

V. Conclusions

The MOS tool has been applied in a real world case study in cooperation with Bombardier Transportation AB, where a train control and management system has been tested. The software developed using the FBD language was modeled in detail using TA framework. In the case study, we used MOS to produce test suites based on model coverage criteria. We also implemented search-based testing algorithms for FBD code in order to produce meaningful tests. As a proof-of-concept, the MOS tool was used to generate test input for train control software developed by Bombardier Transportation AB. Results from this ongoing research clearly indicate that MOS can be used for automating and systematically test complex safety-critical systems.

VI. Acknowledgments

This work was supported by VINNOVA, the Swedish Governmental Agency for Innovation Systems, under grant agreement number 2011-01379, within the ATAC project.

References

[1] K. John and M. Tiegelkamp, "IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems," *Decision-Making Aids*, 2010.

[2] W. A. Halang, "Languages and Tools for the Graphical and Textual System Independent Programming of Programmable Logic Controllers," *Microprocessing and Microprogramming Journal*, vol. 27, no. 1, pp. 583–590, 1989.

[3] L. Baresi, M. Mauri, A. Monti, and M. Pezze, "PLCTools: Design, Formal Validation, and Code Generation for Programmable Controllers," in *International Conference on Systems, Man, and Cybernetics*, vol. 4. IEEE, 2000, pp. 2437–2442.

[4] D. Thapa, C. M. Park, S. C. Park, and G.-N. Wang, "Auto-Generation of IEC standard PLC code using t-MPSG," *International Journal of Control, Automation and Systems*, vol. 7, no. 2, pp. 165–174, 2009.

[5] M. Younis and G. Frey, "Formalization of existing PLC programs: A survey," *Proceedings of Computing Engineering in Systems Applications*, pp. 0234–0239, 2003.

[6] G. Hassapis, "Soft-testing of Industrial Control Systems Programmed in IEC 1131-3 languages," *ISA Transactions: The Journal of Automation*, vol. 39, no. 3, pp. 345–355, 2000.

[7] K. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 1, no. 1, pp. 134–152, 1997.

[8] E. Estévez, M. Marcos, D. Orive, E. Irisarri, and F. Lopez, "Xml based visualization of the iec 61131-3 graphical languages," in *International Conference on Industrial Informatics*, vol. 1. IEEE, 2007, pp. 279–284.

[9] M. Öhman, S. Johansson, and K. Årzén, "Implementation Aspects of the PLC standard IEC 1131-3," *Control Engineering Practice*, vol. 6, no. 4, pp. 547–555, 1998.

[10] J. Thieme and H. Hanisch, "Model-based Generation of Modular PLC Code using IEC61131 Function Blocks," in *Proceedings of the International Symposium on Industrial Electronics*, vol. 1. IEEE, 2002, pp. 199–204.

[11] G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, and J. Romijn, "Efficient Guiding towards Cost-Optimality in UPPAAL," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 174–188, 2001.

[12] R. Alur and D. Dill, "Automata for Modeling Real-time Systems," *Automata, languages and programming*, pp. 322–335, 1990.

[13] R. Alur, "Timed Automata," in *Computer Aided Verification*. Springer, 1999, pp. 688–688.

[14] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in Dense Real-time," *Information and computation*, vol. 104, no. 1, pp. 2–34, 1993.

[15] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou, "Time-Optimal Real-Time Test Case Generation Using UPPAAL," *Lecture Notes in Computer Science, Formal Approaches to Software Testing*, pp. 114–130, 2004.

[16] ATAC- ITEA2 Consortium. (2012). [Online]. Available: http://atac.testautomation.fi/

[17] E. P. Enoiu, D. Sundmark, and P. Pettersson, "Model-based Test Suite Generation for Function Block Diagrams using the UPPAAL Model Checker," in *Proceedings of Sixth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2013, to appear.

[18] K. Doganay, M. Bohlin, and O. Sellin, "Search based testing of embedded systems implemented in IEC 61131-3: An industrial case study," in *Proceedings of Sixth IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2013, to appear.