Review of Model-Based Testing Approaches in Production Automation and Adjacent Domains – Current Challenges and Research Gaps

***Abstract*—As systems have been and are becoming more and more complex, the task of quality assurance is increasingly challenging. Model-based testing is a research field addressing this challenge and many approaches have been suggested for different applications. The goal of this paper is to review these approaches regarding their suitability for the domain of production automation in order to identify current trends and research gaps. Adjacent domains where approaches are introduced in order to identify promising techniques which may be interesting for the field of production automation. The different approaches are classified and clustered according to their main focus which is either testing and test case generation of/from formal models, test case generation from semi-formal models, test case generation from fault models or test case selection and regression testing.**

***Index Terms*—Model-based testing, automated production systems, conformance testing, regression testing, fault injection, survey**

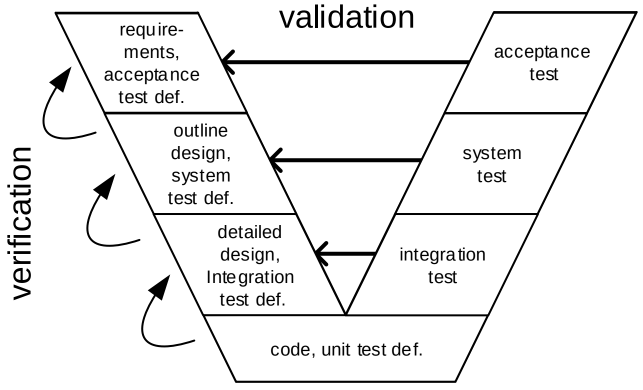
validation
verification
require-ments, acceptance test def.
outline design, system test def.
detailed design, Integration test def.
code, unit test def.
acceptance test
system test
integration test

Fig. 1. Test-driven development process within the V-Model

I. Introduction

TESTING is the "activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" [1]. As systems in industrial automation are becoming more complex [2], also due to trends such as increasing reconfigurability [3], flexibility [4] or autonomous and intelligent behavior [5], the challenge of validation has gained significance. The high standards regarding non-functional requirements such as quality, timing and safety aspects [6] of automated production systems or conformance test of critical controllers as advocated by certification bodies and standards [7], [8] further increase the challenge. In [2], it is shown that in currently established tools for the development of industrial control software, testing activities are rarely automated and have to be conducted manually. Furthermore, "the process of deriving tests tends to be unstructured, not reproducible, not documented, lacking detailed rationales for the test design, and dependent on the ingenuity of single engineers" [9]. Consequently, many works have been conducted in research on the key challenges of improving and automating the testing process in the domain of production automation. Methods aiming towards this goal can be part of model-based and model-driven development processes, which are increasingly established [10]. The key research questions include 1) the definition of user-friendly models enabling the abstraction of automation systems' structure and behavior in order to handle their complexity, 2) the automatic generation of test cases from these models, decreasing the error-prone tasks of manually deriving test cases from informal requirements, 3) the inclusion of hardware effects and especially hardware failures in the models and fault models, 4) the minimization of the effort of regression testing through analyzing suitable specification or code models, and finally, 5) the automatic execution of test cases on the system under test. In test-driven development processes, it is aimed at specifying the test cases along with the system specification which is further refined in each step (see Fig. 1). Test-driven processes are commonly applied in software engineering and have also been proposed for automated production systems [11]. However, they are mostly still not systematically applied in the field of production automation.

The complementary techniques to testing, which are based on the validation that the implementation behaves equivalently to its specification, are formal verification, model-checking and theorem-proving, which prove "that the internal semantics of a model is consistent, independently from the modeled system" [12] (see Fig. 1). These techniques prove that the semantics and the behavior of a model are consistent. However, they mainly aim at verifying non-functional properties such as reachability, liveness and absence of deadlock, independently from the expected functional behavior. A comprehensive overview on formalization of models for verification purposes may be found in [13] and [14].

The goal of this paper is to review current model-based testing approaches in production automation and close adjacent domains, such as embedded systems where model-based testing is already applied more often, in order to identify promising approaches which might be adopted, put current trends into context and define current challenges and research gaps. This paper focuses on validation of functional and non-functional requirements concerning the Programmable Logic

Controller (PLC) application software rather than structural analysis, stress testing or performance analysis [15]. Also, the systems' software behavior is regarded as deterministic, excluding stochastic testing.

The remainder of this paper is structured as follows. In the following section, classification criteria are established in order to distinguish and structure current research approaches. In the subsequent sections, the different research fields are discussed, structured based on the classification criteria concerning the testing goal and the formality of the models used as a basis for test case generation. In section III, testing of formal models and test generation from formal models, targeting the research questions 2 and 5, are described. Then, test case generation from semi-formal models, including approaches which target research questions 1, 2 and 5, is explained in section IV. Test generation based on fault models (research questions 1-3 and 5) is explained in section V. Change impact analysis of specification or code models for regression testing (section VI) (research question 4) completes the discussion of the different research fields. Subsequently, research gaps in model-based testing in production automation are identified and a conclusion is given.

II. Methodology, Classification and Definitions

Recently model-based testing and related terms and definitions such as test case, test selection criteria and test case specification have been updated and defined in [9]. The classification criteria for model-based testing approaches proposed in this paper use these and other definitions from [16] and [1] related to testing , and put them into context regarding the key research questions in the field of production automation. An overview of the classification criteria (*C1-C5)* may be found in Fig. 2.

1) Model/specification – Classification criterion C1: To validate automation systems, different measures are taken regarding the *testing goal* which differs according to the manner and information included in the *specification* also called *model paradigm (C1a).*

If only the expected *behavior (C1a-i)* is specified, the *functional compliance* may serve as the testing goal. If fault models do exist, the reliability of the system, which is determined by the reaction to faults, can be tested and research question 3 is targeted. This can be done for example by using fault injection [17] (see section V). Another research field is concerned with defining and analyzing change models and therefore focuses on regression testing tackling research question 4 (see section VI).

Furthermore, the works of different research groups can be distinguished by the level of *formality (C1b)* which is presumed of the models. Many approaches require formal specifications to generate test cases (see section III) while others focus on providing more user friendly modeling languages for system engineers in order to specify the system (research question 1, see section IV). The latter often include a process to formalize the modeling languages further to be able to generate test cases.

2) Test Selection Criteria – C2: The test cases are generated based on the specification, i.e. models, and on *test selection criteria* which "define[...] the facilities that are used to control the generation of tests" [9] (research question 2). As an example, when a state machine is used as model/specification (C1a-i, C1b-i), several test cases may be created using a *coverage criterion (C2a)* such as transition coverage. For black-box testing, a typical criterion would be data coverage. Further test selection criteria may be found in [9].

3) Test Cases – C3: Besides test case generation based on a predefined model and static test selection criteria, sometimes test cases are generated "on-the-fly" by comparing the outputs emitted by the *system under test* (SUT) during a test run to the expected ones according to the specification and adapting the test cases depending on the local *test verdict* (pass of fail). This type of test generation and execution is called *online testing (C3a-ii).* On the opposite side, *offline testing (C3a-i)* refers to the separation of the offline test generation and later execution. If the test case also manipulates the input from other test components, such as the injection of faults, this is called *feedback manipulation (C3b).*

4) Test Bed – C4: The execution of test cases against the SUT is made possible by the *test bed* which is the "environment containing the hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test" [1] (research questions 3 and 5). One determining factor of the test bed is the kind of environment feedback it includes. Common test cases usually simply consist of predefined inputs and the SUT's outputs which are compared to the expected ones. The input and expected output signals are static, i.e. they are not influenced by the SUT's behavior, and are specified using implicit knowledge or assumptions about the SUT's environment's behavior. In contrast to this, the SUT's output signals can be dynamically fed back into its inputs through a function representing the SUT's environment's behavior, e.g. a plant's behavior represented by a *model (C4-i)*, i.e. simulation. On the one hand, test sequences for including complex environment behavior are simplified, shortened and can be based on physical correlations, rather than implicit knowledge. On the other hand, extensive effort has to be invested into the simulation's definition and verification.

Other test beds include a setup with a connection to the *real (C4-ii)* plant (hardware), or some parts of it, in order to include the actual feedback in the test execution.

5) System Under Test (SUT) – C5: The setup of the SUT with the test bed determines which *type (C5a)* of test is conducted. Some approaches do only test models of the implementation (*Model in the Loop - MiL (C5a-i)*) while other tests are done using the implemented software (*Software in the Loop - SiL (C5a-ii)*) using static sequences or a simulation.

In [18], two major forms of simulation in industrial automation are identified: System simulation and *Hardware in the Loop - HiL (C5a-iii)* simulation. In system simulation, both control software, i.e. the SUT, and simulation are running on the same system - usually a standard computer. In this case, the control software is running on a soft PLC, which can either implement the simulation or can be connected to an external simulation (SiL). In [19], this type of simulation

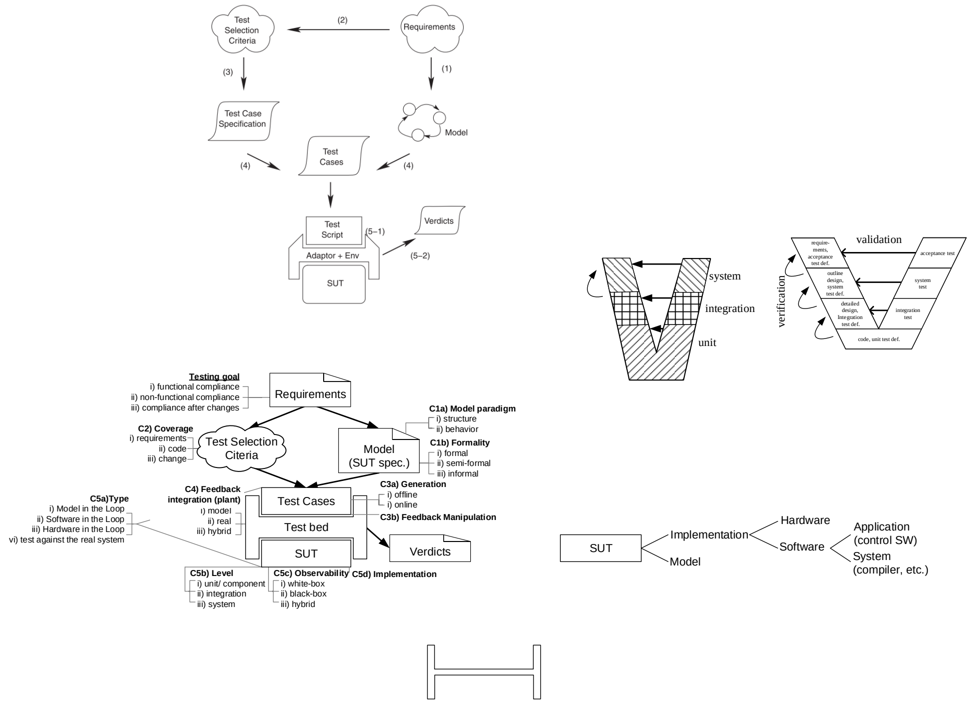

Fig. 2. Overview model-based testing and classification based on [9].

and its verification are investigated. The second major form is HiL simulation/testing. A "typical HiL setup would include a controller with loaded control code connected to a testing environment" [20]. For automated production systems this means that the control software is executed on the target hardware, i.e. a real PLC, and connected to the simulation system via a field bus. The field of HiL has become broader as in "past years [...] HiL has expanded to encompass component testing as well" [20].

While MiL approaches do help to root out faults in the early design phases and SiL approaches root out faults during the implementation phase [17], only HiL approaches enable the validation of the integrated system and the inclusion of hardware related effects (research question 3), as every model is some form of abstraction and only the system itself is completely accurate [21].

The SUT also determines the testing *level (C5b)*, which ranges between *unit/module/component test (C5b-i)*, *integration testing (C5b-ii)* and *system testing (C5b-iii)*. Testing approaches are needed for all stages of the development process.

Furthermore, the observability of the implementation must be considered (C5c): White-Box testing is "a type of testing in which you examine the internal structure of a program" [22] and therefore the implemented code is used as a basis for test case generation. Black-Box testing, which is an approach where the "internal structure is ignored. Test data are derived solely from the application's specification" [22].

In this paper, approaches are furthermore distinguished by their applicability in the field of production automation. Some approaches are developed for testing *implementations (C5d)* of languages which are designed for use in production automation systems such as the IEC 61131-3, which is established within industry right now [23], or the IEC 61499 which has been advocated by many researchers [24]. Other approaches have been developed in different domains and therefore for different languages but may be applicable in the field of production automation.

The different validation techniques aim at detecting faults in an SUT before the system is commissioned for operation. *Fault, error,* and *failure* are defined in [25]. A *failure* refers to "an event that occurs when the delivered service deviates from correct service", which was originally specified or expected. The *failure* is caused by a deviation from the expected system state. This "deviation is called an error" and "the adjudged or hypothesized cause of an error is called a fault".

6) Test Verdict: After the execution of a test sequence (once a final state has been reached), the test verdict is reported and documented which can either be:

- *pass*: the equivalence relation between the specification and the implementation model is fulfilled: the implementation conforms to its specification
- *fail*: the equivalence relation is not fulfilled and counter-examples (or traces) can be given: the implementation does not conform to its specification
- *inconclusive*: the execution on the test has not permitted to assess the equivalence of the two models neither to give counter-examples. This case corresponds to test execution where the observed behavior of the implementation does

not lead to a *fail* state (no counter example can be found), but where a part of the behavior to be tested could not be observed. For more details about this verdict in the context of on-the-fly test cases generation see [26].

In order to avoid false positive and false negative verdicts, a conformance test must be:

- *valid* (or *exhaustive*): "the test suite comprises all combinations of input values and preconditions" [16], therefore every implementation which does not conform to its specification must be detected and rejected;
- *non-biased* (or *sound*): no implementation which conforms to its specification should be rejected.

III. Testing of formal models and test sequence generation from formal models

A. Testing of formal models

A promising solution to develop testing techniques is to benefit from the results of the researches of the Computer Science and Discrete Event Systems communities in the domain of conformance test of formal models. In these works, the specification is given in the form of a formal model such as a finite state machine [27], a transition system [28]–[30], a timed automaton [31] or a Petri net [32]. The implementation is supposed to behave according to the same formalism, e.g. if the specification is defined by a Moore machine, the implementation should also behave like a Moore machine and not like, for instance, a Mealy machine. The interested readers are referred to the above-mentioned paper to get more details about these formal languages. The goal of these testing techniques is then to validate the inclusion or equivalence relations between the two models. The equivalence between a specification model and an implementation model are usually tested according to their *observable behavior* and their *trace equivalence*. Below are some examples of conformance relations between an implementation i and its specification s, for formal definitions and details about these relations the interested readers are referred to [28], [29]:

- $i \leqslant_{tr} s$: i conforms to s wrt. the relation $\leqslant_{tr}$ if and only if for all test sequences composed of the input alphabet of the models, the traces of observable actions of i are included in the traces of observable actions of s. This relation does not consider inputs and outputs but observable actions.
- i **ioconf** s: i conforms to s wrt. the relation **ioconf** if and only if for all test sequences *generated from the traces of* s, the set of observable outputs of i is included in the set of observable outputs of s. This relation allows partial specifications because the test sequences are generated only from the traces of the (partial) specifications.
- i **ioco** s: i conforms to s wrt. the relation **ioco** if and only if for all test sequences generated from the suspension traces of s (traces that also represent the absence of emitted output: specified quiescence, or missing output actions), the sequence of emitted outputs of i is included in the sequence of emitted outputs of s.

In the context of model-based testing of automated production systems, the use of conformance relations defined over the observable input/output relations is more appropriate because the internal behavior of the controller cannot always be observed (black-box testing). Recently, [33] proposed a new conformance relation for model-based testing of PLCs.

As mentioned earlier a test sequence can either be generated offline or online. In the first case, the test sequence is a straight sequence of input/output couples. In the second case, the continuation of the test execution depends on the observed outputs from the previous test step. Both cases can be modeled as state-machine or labeled transition systems where each final state defines the test verdict.

Since the verdict of a test is based on the observation of the behaviors of two models (specification and implementation), an important issue is to be able to ensure the state synchronization and the state identification of those models. A review of the usual state identification and synchronization techniques is presented in [27], [34]–[36]. Even though the basic techniques are well established, several research activities are still conducted on the improvement of those techniques [37], [38] and their application to others formal models [39]

Conformance of an implementation to its specification requires that a *test objective* or *test selection criteria* be first defined. A classical test objective, when critical systems are considered, is to cross at least once each edge of the directed graph that represents the structure of the formal model; this permits to check every state change from each state of the formal model. Then, the test sequence can be constructed from this model. However, depending on the scale of the system, the security level and the assumptions made on the implementation, different test objectives can be defined. The table I lists different techniques that can be applied depending on the assumptions made on the implementation. A more complete overview of the main testing challenges and the different testing techniques developed to improve the test coverage and the reliability of the test results is given in [40]. It is also of importance to note that most of these works have been developed using event-based formalism (vs. signal-based formalism).

As testing is based on the validation of the equivalence (or inclusion relation) of two behaviors (an implementation and its specification), testing can be seen as an exploration process, which permits to explore the behavior of an implementation and compare it to its specification, and requires the execution of the implementation. In contrast to testing, model-checking is based on the verification of properties that should hold for a behavior. Model checking can be seen as a confirmation process, it is used to confirm that a property holds or not for the whole behavior. Symbolic approaches are used to handle the scalability of verification techniques. Even though symbolic approaches cannot be applied during the execution phase of testing (during the execution, for each test step, the implementation is solicited with a set of fixed values, not with a set of value ranges), model-checking techniques can be used during the first phase to generate test sequences [49]–[53]

B. Model transformation

In order to apply the fruitful theoretical results on specification and implementation used in the automation industry

TABLE I
ASSUMPTIONS ON FORMAL MODELS

Assumptions	Proposed solutions	References
The implementation can have extra states	Bounded number of extra states	[41]
Time	Test boundaries (maximum duration of a timer)	[42]
Variables within a range	Domain testing and/or boundary testing	[43]
Variables within a range	Symbolic approaches	[44]–[46]
Partial specifications	Verifying partial inclusion: Impl. $\subseteq$ Spec1 $\wedge$ Impl. $\subseteq$ Spec2 $\wedge \ldots$	[47]
Partial specifications	Inferring partial models	[48]

there is a need to transform industrial models into more formal models. Several works have considered the issue of model transformation from industrial (or standardized) models into (semi-) formal models in order to apply existing formal techniques and tools. The table II gives an overview of existing works on such model transformations.

Many of the transformations presented in the table II have been developed in the context of verification. Some of the existing methods only consider the source code of the input language while some others also consider a model of the PLC operation. When only the source code is considered (i.e. without a model of the controller execution), the input model can be considered as an infinitely reactive model (i.e. the response time delay equals zero). Thus, the transformation to another formal model can be facilitated by the use of their meta-model. For instance, model-to-model transformations can be obtained using the model transformation tool ATL [77].

The transformation into formal models also permits the combination of the model of the software, with a model of the execution of the controller – if not already done – and also with a model of the plant that is to be controlled. The more information the composed model contains, the more reliable the simulation and the results of the verification and validation methods is [78].

IV. Test sequence generation from semi-formal models

Formal models are up to now rarely used in industry. Mostly partial models or specifications, and informal requirements specifications are used. To bridge this gap and to support system and test engineers in creating models for testing, semi-formal modeling languages and notations are further developed and formalized to receive a basis for test case generation (see table III). Furthermore, model-based approaches are increasingly applied and developed in production automation. Using similar models for test case generation is the logical next step to further support and improve the development process.

As the Unified Modeling Language (UML) is one of the most widely used notations for modeling the structure and behavior of the software up to now, it is no surprise that many approaches do focus on deriving test cases from this language. In [79] and [80], useful diagrams for modeling and deriving test cases from the UML are identified for the field of automation software development and especially for IEC 61499 implementations. Structure diagrams such as component diagrams are used to model the context and the interfaces of the SUT. Interaction diagrams are recommended for the extraction of test sequences. In [80], the extraction of test sequences from state charts using round-trip path coverage is shown. A first application of the recommended test case generation process using state chart diagrams especially for IEC 61499 applications is shown in [81].

In [82], an approach to automatically generate test cases from the UML state charts by first transforming them into a formal model (extended safe place/transition nets) is introduced. In order to make the transformation possible, some restrictions on the model elements used are done. Given the formal model, the test case generation is easily made possible using methods such as unfolding the nets.

Making UML models, and in this work especially sequence diagrams, executable is another focus of using UML diagrams in the testing process. In [83], the semantics of sequence diagrams are adapted in order to make direct IEC 61131-3 code generation possible. In this way the modeled test scenarios can be executed directly.

As UML models are already a wide-spread notation also for testing, organizations have started to standardize the language in the context of testing using the profiling mechanism of the UML. The UML Testing Profile (U2TP) has standardized the way to specify the SUT, its context and the specific test cases. The test case scenarios are modeled using the UML sequence diagrams. To make these test cases executable, a transformation from the U2TP to the Testing and Test Control Notation (TTCN-3) has been proposed by [84], which has been established especially in the field of communication. However, up to now, no approaches could be found that have evaluated the applicability of the U2TP in the field of production automation. In [85], UML test case generation approaches from state charts are combined with the aim of making them executable by mapping them to the TTCN-3. The evaluation of the approach is done using a simple communication protocol but the extension of the approach in order to test PLC control software applications is planned as well.

In recent years the Systems Modeling Language (SysML) is increasingly established for supporting the development process of real-time systems [86]. However, investigations on the possibilities to derive test cases from these models or adapting these models are still missing. Another interesting development that the testing community could benefit from is the improvement of the communication between tools. In [87], an approach to automatically consolidate different domain models from the field of production automation to receive a correct model using AutomationML and MathML is presented. The generation of test cases from such models is still an open topic though.

TABLE II
MODEL TRANSFORMATION FROM INDUSTRIAL OR STANDARDIZED LANGUAGES INTO FORMAL MODEL

Input language	Output formal model	References	Remarks
IL (IEC 61131-3), ST (IEC 61131-3)	Timed Net Condition/Event System, Petri Net, Timed automaton, SIGNAL equations, Model-checker language (SMV)	[54]–[59]	Most of the transformations are performed for verification purposes. The selected references consider the cyclic behavior of industrial controllers such as PLC for the execution of IL programs.
LD (IEC 61131-3)	Time Petri Net, Model-checker language (UPPAAL automata)	[60]–[62]	Most of the transformations are performed for verification purposes. The selected references consider the cyclic behavior of industrial controllers such as PLC. [62] even considers multitask systems.
FBD (IEC 61131-3), CFC, IEC 61499	SIGNAL equations, Esterel, Interface automata, Model-checker language (UPPAAL automata, SMV)	[51], [58], [63]–[67]	Most of the transformations are performed for verification purposes. The selected references consider the cyclic behavior of industrial controllers such as PLC. [67] presents a modeling of each block by an interface automaton, the focus is placed on the IO relations: this approach could be adapted to testing of sub-components where only the IOs can be observed (black-bock testing).
SFC (IEC 61131-3)	Timed automata, Model-checker language (SMV),	[68]–[71]	SFC (61131-3) is a graphical language with hierarchical relations used to represent mainly sequential behaviors. The semantics defined in the standard contains ambiguity. An improved semantics is proposed in [71].
GRAFCET (IEC 60848)	Monolithic automaton, Mealy machine, Petri Net	[72]–[76]	Grafcet (60848) and SFC (61131-3) share similarities: SFC has been defined from Grafcet. The main difficulty with Grafcet is the stability research. The method presented in [72]–[74] is dedicated to black-box testing and stresses on logic input/output relations.

TABLE III
TEST GENERATION FROM SEMI-FORMAL MODELS

Source	Domain	C1) Specification and Formality	C2) Test Selection Criteria	C3) Execution and Feedback Manipulation	C4) Test type	C5) Test type, Implementation, Observability and Level
[79]–[81]	production automation	UML (structure: system and context; interaction, mainly state charts: generation of specific test cases)	depending on diagram, extraction of sequences from interaction diagrams (transition parameter variation, path coverage)	offline	as spec.	IEC 61499
[82]	production automation	UML state charts	model transformation, path unfolding, path coverage	offline	as spec.	n.a.
[83]	production automation	UML sequence diagram	as spec.	offline	both SiL and HiL possible	IEC 61131-3, unit

V. TEST GENERATION FROM FAULT MODELS - TESTING OF UNINTENDED BEHAVIOR

An important topic that must be addressed when testing and validating automated production systems besides the intended behavior is the reaction to faults that may occur within or without the system as this determines the reliability. The faults that must be regarded are not only software faults, but also other possible causes of failures of automation systems such as the failure of hardware or influences of the environment. These faults must be handled by the software of automation systems by error handling routines. To prove the validity and correctness of systems, fault injection (FI) is a method that has been established in order to measure the dependability. Fault injection is used as a means to evaluate error handling mechanisms concerning fault detection and error handling. FI approaches can be divided into hardware-implemented FI (HWIFI), where faults are for example injected by forcing pins, software-implemented FI (SWIFI), where faults of the system are emulated by the software, and model-implemented FI (MIFI) also called simulation-based FI [88]. While HWIFI and SWIFI are mostly used on prototypes or for system testing, MIFI is rather used in earlier conceptual and design phases to give early feedback to engineers [17]. It is possible that the approach implements one kind of fault injection (e.g. MIFI) but still is another kind of test (e.g. SiL), because the fault might be injected by the test case through the test bed (e.g. fault is injected in the simulation model but the SUT is the implemented software not running on the final system). FI is determined by the *faults (F)* that are injected, a set of *activations (A)*, the *readouts (R)*, i.e. the logging of the system reaction or the outputs, and the actions or measures (M) that are derived from the analysis of F, A and R, as defined in [89].

Testing approaches may also be divided into approaches which aim at finding some classes of faults and approaches which aim at testing the reaction of a system to these classes. The latter explicitly define a fault model, i.e. the possible faults, which are described as mutants or saboteurs.

A *fault model* defines the types of possible faults of a system in respect to several different criteria such as the phase of creation (design, implementation, etc.), the dimension (hardware faults, software faults), the system boundary (introduced from within or without the system, etc.) or the persistence (transient or permanent faults) [25].

The type of faults F commonly injected by FI, in some papers called mutants, are mostly hardware faults in contrast to software faults, human made faults, etc. (for classification of faults see [25]). Another aspect considered by many approaches is the timing behavior of the fault which is either permanent, which means that it occurs and is permanent from this point on, or if it occurs only at certain time intervals or at random. The activation *A* is differentiated whether the fault is injected before runtime or during operation. By injecting faults during operation the opportunity to activate the fault by time-triggered or event-triggered conditions is created, making it possible to realize more complex fault scenarios.

The three different approaches - HWIFI, MIFI and SWIFI are further explained in the following subsections. An overview is shown in Table IV.

1) HardWare-Implemented Fault Injection (HWIFI): Many tools and methods for testing the reaction to faults from integrated circuits and especially microprocessors have been established. The methods vary between FI with contact such as the injection of faults on pin level and FI without contact such as heavy-ion-radiation or electromagnetic interference [91]. A third means of introducing faults is the use of built-in logic especially designed for the SUT [90]. The specification is based on faults *F* which typically occur within (micro-) processors such as bridging faults, stuck-at faults, bit flips or power surges [17]. In the field of electrical engineering the fault models are still being updated and further developed [99]. For methods that are controllable and reproducible such as pin-level fault injection the activation *A* is possible to be time-triggered or event-triggered. An overview and more detailed descriptions of the different tools and methods may be found in [17] and [91].

Hardware-in-the-loop test benches have been developed for more complex systems such as PLCs, where the PLC is for example connected with a simulation environment [100]. However, no special attention has been given on FI techniques in this field and studies illustrating which faults and fault models would be useful to inject in such systems and the benefits that might be gained from such testing techniques are not available.

2) Model-Implemented Fault Injection (MIFI): In respect to MIFI a distinction can be made whether faults are injected into hardware models or into software models. Hardware models in the domain of electrical engineering are usually modeled using the Very High Speed Integrated Circuit Hardware Description Language (VHDL) and therefore targeted at integrated circuits. The faults that are to be injected are also based on faults which may occur within microprocessors as mentioned in the previous section. These faults are specified as mutants or saboteurs and integrated into the model or at the interfaces of the model to simulate faults [92]. In the automotive domain several MIFI approaches have been suggested [88]. These approaches make use of the fact, that MATLAB/Simulink models are commonly used modeling automotive systems. In [88] it is not only aimed at testing only on simulation level. The result of the test runs against simulation are used for test case generation for the real systems. In the field of production automation an executable UML state chart simulation model is used for FI in [96]. As the approach focuses on testing the application, the program is sliced to extract all possible execution paths leading to a defective component in order to reach full path coverage. The MIFI approaches mostly only have an offline activation *A*, as the faults *F* are predefined within the model.

3) software-implemented fault injection (SWIFI): As for MIFI and HWIFI, tools for injecting faults in integrated circuits have been evaluated and are available for use [101]. As interfaces and additional functions are standardized for testing and have been introduced in this field, it is additionally made easier to access different locations to inject faults [102]. In [95], a FI approach for embedded system's is introduced in order to validate specified safety functions. The approach targets specific functions which must hold during all circumstances. Specific fault scenarios or the definition of user-friendly notations are not the main focus of the approach. Next to FI during co-simulation, [93] also proposes to use model-based approaches in the automotive domain to introduce faults into the code during code-generation out of Matlab/Simulink models. The components that will be tested as failing are selected in the model, then code is generated where this fault will occur during execution. In [94], a similar approach is suggested using SCADE models. In [98], a method to inject faults during runtime is presented suggesting kernel-based FI on different architectural levels of embedded systems. It is analyzed how and where different faults may be injected in order to test the integration between application, operating system and hardware. The integration of application and operating system is tested by manipulating the communication protocols in between them. The integration testing between operating system and hardware is done using three further mutation operands. If possible the global variables are manipulated such as communications device errors. If the addresses of the hardware are read-only variables different mutants are proposed such as disconnecting the hardware or changing the voltage supply for I/O device errors and power-supply. The method seems a viable way to test the integration of the components within embedded systems such as a PLC as proposed in this paper. The test of the control software in respect to hardware faults is not focused on in the paper. [97] suggests a SWIFI approach, where test cases are generated from timing sequence diagrams for the field of production automation. In a preceding survey [97] evaluates timing sequence diagrams as a notation which is commonly used in the domain of production automation. It is assumed that the diagram depicts the expected behavior and any kind of deviation should be handled by the software. Accordingly, when generating the test cases, test cases with a deviation from the timing sequence diagram (fault operator) are generated. The test execution is done using a setup with the real automation system while injecting the fault directly in the software. The approach is shown to work for processes with discrete behavior. The approach focuses on deviations from the process behavior (*F*: process faults) as sensor values are used for analyzing the behavior. The activation *A* is done during the execution of the system. Communication faults or human made faults are not especially in the focus of this work.

TABLE IV
FAULT INJECTION IN AUTOMATED PRODUCTION SYSTEMS AND EMBEDDED SYSTEMS

Source	Domain	C1) Specification and Formality	C2) Test Selection Criteria	C3) Execution and Feedback Manipulation	C4) Feedback Integration	C5) Type, Implementation, Observability and Level
[17], [90], [91]	integrated circuits	stuck-at, open, complex logical faults, ...	as spec. contact or contactless, time-triggered	offline	HWIFI (pin-level or insertion, heavy-ion radiation, electromagnetic)	real system, microprocessors/ integrated circuits, system
[92]	integrated circuits	fault model (saboteurs, mutants), some mutants generated automatically	as spec.	offline	MIFI	VHDHL model, integration/ system
[88]	automotive	fault models, failure mode function (mutants and saboteurs), requirements (Simulink assertion blocks), time window	as spec. minimal cut sets	offline	MIFI	Matlab/Simulink, unit
[93], [94]	automotive	selection of component/ fault nodes in simulation model	as spec.	offline	MIFI/ SWIFI (injection in generated code)	C (generated out of SCADE), unit
[95]	embedded systems	mathematical description of functions and safety properties	mutation operators (insertion of an additional request, re-ordering of a pair of requests)	offline	SWIFI	MiL
[96]	production automation	fault operators	path coverage, possible inputs, code splicing, possible paths leading to component	offline, simulation: UML SC for PLC	MIFI	SiL, IEC 61131-3, unit
[97]	production automation	timing sequence diagram	fault operators (missing signal)	offline	SWIFI	real system, IEC 61131-3, system
[98]	production automation	target variable chosen manually, mutant created accordingly, every path checked with every mutant	5 mutation operators according to target	offline	SWIFI	C, FBD, integration

4) short summary of testing embedded system's reaction to faults: The use of FI to test hardware faults is widely spread to validate the dependability of integrated circuits. In this field all HWIFI, MIFI and SWIFI approaches have been tested and evaluated. The automotive and aerospace domains make use of the models available during the development process and introduce faults on model level. The execution is done using simulation or code-generation adopting the MIFI or SWIFI approach. In domains where models are scarcely available such as machine and plant automation, FI techniques on application level have not been exploited very much so far even though it is a field where reliability and dependability are of huge interest.

VI. TEST SELECTION FROM CHANGE MODELS - REGRESSION TESTING

During development and operation of automated production systems, the system's software has to be changed and adapted regularly. The changes are categorized in [103] as adaptive, corrective and perfective. Adaptive changes are due to changing environments or requirements, such as changes in hardware or new needed functions. Corrective changes are introduced whenever faults within the software are discovered and fixed. Perfective changes are made during optimization processes, e.g. to shorten production cycle times. Whenever the software is changed, an investigation whether faults are introduced into the software and its compliance to the specification has to be conducted. This process is known as *regression testing*.

The main challenge in regression testing in production automation is to test a changed software system thoroughly, while minimizing the effort to do so. However, if done manually, regression testing is tedious and prone to be incomplete, as dependencies within programs can be intricate. Scenarios leading to errors might be missed and testing efforts are high and have to be repeated with every change. Nevertheless, this type of testing is still dominant in this engineering domain. Model-based testing methods, as described in the previous sections, can help in this regard, by offering ways to automate the test execution and generation. Based on a set of available test cases, regression testing can be conducted by selecting suitable test cases for retesting [104] depending on identified changes, which is done within a *Software Change Impact Analysis* (CIA). The goal of this analysis is to select the test cases that are most likely to find new errors introduced by the changes, but keeping the time of retesting lower than a simple ”retest-all” approach, where all tests are re-executed [105]. Therefore, the time needed for the analysis plus the execution of the selected test cases is supposed to be lower than executing all test cases. This selection is done under the assumption that program execution is deterministic and nothing but the code or the specification changes.

CIA can be based on dependencies between software entities, such as functions, classes or statements or on traceable dependencies between the software and other software related artifacts, such as function specifications or interlocking definitions. The former is known *dependency based change impact analysis*, while the latter is called *traceability based change* *impact analysis* [106].

1) Traceability based CIA: This type of CIA is based on the program's specification rather than the changed code itself. The term traceability refers to the ability to trace changes from the specification to its corresponding code through appropriate definitions within the specification. There are different approaches for identifying changes in available specifications and deriving possible influences on test cases for regression testing. As structural models are common in computer science as an artifact for program design, many works take these models to gather information for regression testing regarding integration tests. For selection of integration tests, information about object interdependencies and changes are gathered, which is then used to identify the parts of the program affected by the changes. These entities are then scheduled for regression tests.

A suitable model for this process is the UML Class Diagram, which includes information about the program structure, interfaces and interdependencies between objects. Several works use this model for selecting regression integration tests [107]–[109]. Other works do not rely on this notation, but create their own formalized model from it, such as the Component Dependency Model [110], or directly specify needed dependencies within a formal description language, such as the Specification Description Language (SDL) [111]. These models focus on relevant dependencies and allow automatic analysis, e.g. graph and dependency analyses, to find relevant changes and assess their impact on testing. In most of the mentioned approaches, certain change classes are defined, which are linked to an influence on test cases. This information is then used to select the test cases.

For regression testing of single units, other specification models are needed, as no information about behavior is stored within class diagrams or other structural models. Several works identifyy suitable models for defining behavior, such as the UML Sequence Diagram [110], [107], Extended Finite State Machines (EFSM) [112] or the SDL [111]. Again, identification of influences of changes is conducted using change classes, which define whether a test case is influenced or not. Changed entities are identified by comparing the sets of sub-elements of the respective models.

2) Dependency based CIA: This type of CIA can be used for regression testing without relying on models describing structure or behavior of the software. The models needed for analyzing affected entities are directly generated from the code. This can be either done statically, meaning before test execution or dynamically, based on information about previous test execution.[1] Common static dependency analysis methods include building call graphs [113], analyzing which entities call other entities, or program dependence graphs [114], adding information about data dependencies. The resulting graphs include all possible object interconnections, which can be problematic as changes can lead to assumptions about their influence including a lot of false positives, i.e. parts of the code which seem to be influenced, but are not.

For dynamic methods, execution traces are recorded during test case execution, giving a clearer image on what is actually affected by the test case. Dynamic call graphs [115], dynamic program slices [116] or control flow graphs [117] and data flow graphs [118] can be extracted from this information. While false positives are reduced, the found information is only valid for the executed scenarios. Also, in many cases code instrumentation is needed for recording the test execution to gather all needed information, which can alter the system's behavior as additional code is executed.

Traceability based CIA is especially interesting, if certain parts of the program code are not accessible ("black box"), e.g. in compiled libraries, as all approaches in the dependency based CIA create a model directly from the code. In contrast, the latter is interesting because no additional manual modeling is needed.

3) Short summary on the applicability of the investigated approaches: As summarized in Table V, most of the analyzed methods were developed within the domain of computer science.

Regarding traceability based CIA, the uncommon use of formalized models for software design in production automation hinders an application in this domain. While advantages in later phases can be seen, this domain is still hesitant introducing modeling on a level that can be used for the presented approaches (close enough to the code).

In dependency based CIA there are many similarities between programming in the domains of computer science and production automation, but most of the presented concepts are not directly applicable to automated production systems' program code. Only recently, first advances towards applying similar approaches in this domain were made [119]. Problems regarding applicability are mostly rooted in the dominant programming standard IEC 61131-3 used in the domain of production automation, which consist of different graphical as well as textual programming languages. Even though most production automation integrated development environments convert all of the possible languages to one textual language that uses a similar syntax as those used in computer science, there are differences regarding structure and behavior that make adaptation complicated.

Even though advances towards object-orientation are being made, the structure of IEC 61131-3 programs is neither completely object-oriented (e.g. many global accesses), nor procedural (e.g. class like function blocks). All of the analyzed approaches are directly aimed at one of these paradigms.

Regarding behavior, cyclic execution of the code significantly influences the programming paradigm, and thus hinders a direct application of the approaches which usually assume a single execution of the program per test case. The test cases used in the publications are designed accordingly: single input vector test cases that do not allow adequate testing of state machines, which are common in automated production systems.

VII. Discussion and Research Gaps

Reflecting the presented work in this paper, many promising approaches in the field of model-based testing in production automation have emerged.

[1]The expressions static and dynamic model generation are not to be confused with static and dynamic feedback inclusion in test cases.

TABLE V
METHODS FOR SOFTWARE CHANGE IMPACT ANALYSIS

Source	Domain	C1) Model Paradigm and Formality	C2) Test Selection Criteria	C3) Generation	C5) Test type, Implementation, Observability
[108], [109]	computer science	structure, semi-formal (UML class diagram)	change of specification	none	integration tests, object-oriented, allows black box
[110]	automotive, embedded systems	structure and behavior, semi-formal (component dependency model, UML sequence diagram) and formal (directed graph)	change of specification	none	integration tests, allows black box
[111]	computer science	structure and behavior, formal (specification description language)	change of specification	none	integration tests, object-oriented, allows black box
[107]	computer science	structure and behavior, formal (UML sequence diagram, class diagram)	change of specification	none	integration tests, allows black box
[112]	computer science	behavior, formal (extended finite state machine)	change of specification	none	integration tests
[113]	computer science	behavior, formal (call graph)	change of code	none	integration test, system test, object-oriented, white box
[114]	computer science	behavior, formal (program dependency graph)	change of code	none	integration test, procedural, white box
[115]	computer science	behavior, formal (dynamic call graph)	change of code	previous execution	integration test, system test, object-oriented, white box
[116]	computer science	behavior, formal (dynamic program slice)	change of code	previous execution	integration test, system test, object-oriented, white box
[117]	computer science	behavior, formal (dynamic control flow graph)	change of code	previous execution	unit test, system test, procedural, white box
[118]	computer science	behavior, formal (dynamic data flow graph)	change of code	previous execution	unit test, integration test, system test, procedural or object-oriented, white box

Research question 1: The definition of user-friendly modeling languages – especially based on the UML – as a basis for test case generation and methods to derive test cases from these models have been investigated. However, comprehensive surveys and case studies on the acceptance and usability of these models or test case generation from SysML models in production automation still remain an open challenge.

Research question 2: There are a number of approaches for transformation of semi-formal modeling languages into formal models, as well as algorithms and test selection criteria. To find industrial relevance of the approaches, a thorough industrial evaluation could help assessing the relevance for the domain of production automation. A special focus should be given to approaches that include not only the source code but also at least the execution model of the implementation of the plant model.

Research question 3: Defining appropriate fault models and generating test cases to test the reaction to faults is still an emerging field in the field of production automation, where mainly other domains have been conducting research until now.

Research question 4: The same is true for change impact analysis and regression testing approaches which have mainly been researched in the field of computer science so far. Therefore, the investigation on how to apply these methods in the field of production automation remains an open issue.

Research question 5: Last but not least, the automatic execution of test cases is also a field where a lot of work remains to be done. As mentioned in [2], full support of current tools for PLC platforms is still missing. Approaches that integrate the automatic execution in available tools or new tools targeted at PLC platforms have scarcely been found.

VIII. CONCLUSION

In this paper, model-based testing approaches have been reviewed in context of the current challenges within the field of production automation. These challenges have been identified as the definition of user-friendly models as a basis for test case generation, the automatic test case generation from these models, the inclusion of hardware effects, the minimization of testing efforts during regression testing and the automatic execution of the generated test cases. Furthermore, the classification criteria in order to classify the different analyzed approaches are introduced, ordered by requirements and models as a basis for testing, test selection criteria, executable test cases, test bed, system under test and test verdict. In conclusion, it has been found that especially regarding the definition of user-friendly notations as a basis for test case generation, testing of system's reaction to faults and regression testing more work remains to be done. For the latter two, promising approaches have been introduced in electrical engineering and computer science which may be interesting for the field of production automation when adapted to the domain-specific requirements.

REFERENCES

[1] *Systems and software engineering – Vocabulary*, ISO/IEC/IEEE 24765:2010 Std., 2010.

[2] A. Dubey, "Evaluating Software Engineering Methods in the Context of Automation Applications," in *2011 9th IEEE International Conference on Industrial Informatics (INDIN)*, 2011, pp. 585 – 590.

[3] D. Schütz, A. Wannagat, C. Legat, and B. Vogel-heuser, “Development of PLC-Based Software for Increasing the Dependability of Production Automation Systems,” *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2397–2406, 2013.

[4] T. Cucinotta, A. Mancina, G. F. Anastasi, G. Lipari, L. Mangeruca, R. Checcozzo, and F. Rusinà, “A Real-Time Service-Oriented Architecture for Industrial Automation,” *IEEE Trans. Ind. Informat.*, vol. 5, no. 3, pp. 267–277, 2009.

[5] P. Leitão, V. Mařík, and P. Vrba, “Past, Present, and Future of Industrial Agent Applications,” *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2360–2372, 2013.

[6] M. A. Wehrmeister, C. E. Pereira, and F. J. Rammig, “Aspect-Oriented Model-Driven Engineering for Embedded Systems Applied to Automation Systems,” *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2373–2386, 2013.

[7] IEC 60880, *Nucl. power plants - Instrumentation and control systems important to safety - Software aspects for computer-based systems performing category A functions*, 2nd ed. International Electrotechnical Commission, 2006.

[8] IEC 61850-10, *Communications Networks and Systems in Substations - Part 10: Conformance testing*, 2nd ed. International Electrotechnical Commission, 2005.

[9] M. Utting, A. Pretschner, and B. Legeard, “A taxonomy of model-based testing approaches,” *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297–312, 2012.

[10] V. Vyatkin, “Software Engineering in Industrial Automation: State-of-the-Art Review,” *IEEE Trans. Ind. Informat.*, vol. 9, no. 3, pp. 1234–1249, 2013.

[11] R. Hametner, I. Hegny, and A. Zoitl, “A Unit-Test Framework for Event-Driven Control Components Modeled in IEC 61499,” in *Emerging Technology and Factory Automation (ETFA)*, 2014, pp. 1–8.

[12] J.-M. Roussel and J.-J. Lesage, “Validation and Verification of grafcets using finite state machine,” in *Proceedings of IMACS-IEEE’CESA’96’*, 1996, pp. 1–6.

[13] G. Frey and L. Litz, “Formal methods in PLC programming,” in *Systems, Man, and Cybernetics, 2000 IEEE International Conference on*, vol. 4. IEEE, 2000, pp. 2431–2436.

[14] M. B. Younis and G. Frey, “Formalization of existing plc programs: A survey,” in *Proceedings of CESA*, 2003, pp. 0234–0239.

[15] A. Girbea, C. Suciu, S. Nechifor, and F. Sisak, “Design and Implementation of a Service-Oriented Architecture for the Optimization of Industrial Applications,” *IEEE Trans. Ind. Informat.*, vol. 10, no. 1, pp. 185–196, Feb. 2014.

[16] E. Van Veenendaal, “Standard glossary of terms used in software testing,” *International Software Testing Qualifications Board*, no. 2.3, pp. 1–53, 2014.

[17] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, “Fault Injection Techniques and Tools,” *Computer*, vol. 30, no. 4, pp. 75–82, 1997.

[18] M. Barth and A. Fay, “Automated generation of simulation models for control code tests,” *Control Engineering Practice*, vol. 21, no. 2, pp. 218–230, Feb. 2013.

[19] H. Carlsson, B. Svensson, F. Danielsson, and B. Lennartson, “Methods for Reliable Simulation-Based PLC Code Verification,” *IEEE Transactions on Industrial Informatics*, vol. 8, no. 2, pp. 267–278, May 2012.

[20] F. Gu, W. S. Harrison, D. M. Tilbury, and C. Yuan, “Hardware-in-the-loop for manufacturing automation control: Current status and identified needs,” in *Automation Science and Engineering, 2007. CASE 2007. IEEE International Conference on*. IEEE, 2007, pp. 1105–1110.

[21] ——, “Hardware-In-The-Loop for Manufacturing Automation Control: Current Status and Identified Needs,” in *2007 IEEE International Conference on Automation Science and Engineering*. IEEE, Sep. 2007, pp. 1105–1110.

[22] G. J. Myers, C. Sandler, T. Badgett, and T. M. Thomas, *The art of software testing, Second Edition*. John Wiley & Sons, 2004.

[23] K. Thramboulidis, “IEC 61499: Back to the well proven practice of IEC 61131?” in *IEEE 17th Conference on Emerging Technologies Factory Automation (ETFA)*, Sept 2012, pp. 1–8.

[24] A. Zoitl and H. Prähofer, “Guidelines and Patterns for Building Hierarchical Automation Solutions in the IEC 61499 Modeling Language,” *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2387–2396, 2013.

[25] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

[26] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho, “Using on-the-fly verification techniques for the generation of test suites,” in *Computer Aided Verification*. Springer Berlin Heidelberg, 1996, pp. 348–359.

[27] D. Lee and M. Yannakakis, “Principles and methods of testing finite state machines - a survey,” in *Proc. IEEE*, vol. 84, no. 8, 1996, pp. 1090–1123.

[28] J. Tretmans, “Test generation with inputs, outputs and repetitive quiescence,” *Software — Concepts and Tools*, vol. 17, no. 3, pp. 103–120, 1996.

[29] ——, “Model based testing with labelled transition systems,” in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, R. M. Hierons, J. P. Bowen, and M. Harman, Eds. Springer, 2008, vol. 4949, pp. 1–38.

[30] S. Pickin, C. Jard, T. Jéron, J.-M. Jézéquel, and Y. Le Traon, “Test synthesis from UML models of distributed software,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 4, pp. 252–269, 2007.

[31] M. Krichen and S. Tripakis, “Conformance testing for real-time systems,” *Formal Methods in System Design*, vol. 34, no. 3, pp. 238–304, 2009.

[32] M. Pocci, I. Demongodin, N. Giambiasi, and A. Giua, “Testing Experiments on Synchronized Petri Nets,” *IEEE Trans. Autom. Sci. Eng.*, vol. 11, no. 1, pp. 125–138, Jan 2014.

[33] A. Guignard and J.-M. Faure, “A conformance relation for model-based testing of PLC,” in *Proc. of the 12th Int. Workshop on Discrete Event Syst.*, 2014, pp. 412–419.

[34] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, “Test selection based on finite state models,” in *IEEE Trans. Softw. Eng.*, vol. 17, no. 6, 1991, pp. 591–603.

[35] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko, “FSM-based conformance testing methods: A survey annotated with experimental evaluation,” *Inform. and Softw. Technol.*, vol. 52, no. 12, pp. 1286 – 1297, 2010.

[36] A. T. Endo and A. Simao, “Evaluating test suite characteristics, cost, and effectiveness of FSM-based testing methods,” *Inform. and Softw. Technol.*, vol. 55, no. 6, pp. 1045–1062, 2013.

[37] A. Petrenko, A. Simao, and N. Yevtushenko, “Generating checking sequences for nondeterministic finite state machines,” in *IEEE 5th Int. Conference on Softw. Testing, Verification and Validation (ICST)*. IEEE, 2012, pp. 310–319.

[38] R. M. Hierons and U. C. Türker, “Distinguishing sequences for partially specified FSMs,” in *NASA Formal Methods*. Springer, 2014, pp. 62–76.

[39] M. Pocci, I. Demongodin, N. Giambiasi, and A. Giua, “A new algorithm to compute synchronizing sequences for synchronized petri nets,” in *TENCON 2013-2013 IEEE Region 10 Conference (31194)*. IEEE, 2013, pp. 1–6.

[40] C. Kaner, J. Bach, and B. Pettichord, *Lessons learned in software testing*. John Wiley & Sons, 2008.

[41] A. Simão, A. Petrenko, and N. Yevtushenko, “Generating reduced tests for FSMs with extra states,” in *Testing of Software and Communication Systems*. Springer, 2009, pp. 129–145.

[42] K. El-Fakih, N. Yevtushenko, and H. Fouchal, “Testing timed finite state machines with guaranteed fault coverage,” in *Testing of Software and Communication Systems*. Springer, 2009, pp. 66–80.

[43] L. J. White and E. I. Cohen, “A domain strategy for computer program testing,” *IEEE Trans. Softw. Eng.*, no. 3, pp. 247–257, 1980.

[44] W. d. L. Andrade, P. D. Machado, T. Jéron, and H. Marchand, “Abstracting time and data for conformance testing of real-time systems,” in *IEEE 4th Int. Conf. on Softw. Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2011, pp. 9–17.

[45] W. d. L. Andrade and P. D. Machado, “Generating test cases for real-time systems based on symbolic models,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 9, pp. 1216–1229, 2013.

[46] C. Cadar and K. Sen, “Symbolic execution for software testing: three decades later,” *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[47] A. Petrenko and N. Yevtushenko, “Testing from partial deterministic FSM specifications,” *IEEE Trans. Comput.*, vol. 54, no. 9, pp. 1154–1165, 2005.

[48] M. Shahbaz and R. Groz, “Analysis and testing of black-box component-based systems by inferring partial models,” *Software Testing, Verification and Reliability*, vol. 24, no. 4, pp. 253–288, 2014.

[49] C. Constant, T. Jéron, H. Marchand, and V. Rusu, “Integrating formal verification and conformance testing for reactive systems,” *IEEE Trans. Softw. Eng.*, vol. 33, no. 8, pp. 558–574, 2007.

[50] A. Armando, G. Pellegrino, R. Carbone, A. Merlo, and D. Balzarotti, “From model-checking to automated testing of security protocols: Bridging the gap,” in *Tests and Proofs*. Springer, 2012, pp. 3–18.

[51] E. P. Enoiu, D. Sundmark, and P. Pettersson, "Model-based test suite generation for function block diagrams using the UPPAAL model checker," in *IEEE 6th Int. Conf. on Softw. Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2013, pp. 158–167.

[52] M.-C. Gaudel, "Checking models, proving programs, and testing systems," in *Tests and Proofs*. Springer, 2011, pp. 1–13.

[53] M.-C. Gaudel, R. Lassaigne, F. Magniez, and M. de Rougemont, "Some approximations in model checking and testing," *arXiv preprint arXiv:1304.5199*, 2013.

[54] H.-M. Hanisch, J. Thieme, A. Luder, and O. Wienhold, "Modeling of PLC behavior by means of timed net condition/event systems," in *6th Int. Conf. on Emerging Technologies and Factory Automation Proceedings, 1997. ETFA'97*. IEEE, 1997, pp. 391–396.

[55] M. Heiner and T. Menzel, "A petri net semantics for the PLC language instruction list," in *Workshop on Discrete Event Systems (WODES '98)*, 1998, pp. 161–166.

[56] A. Mader and H. Wupper, "Timed automaton models for simple programmable logic controllers," in *Proc. of the 11th Euromicro Conf. on Real-Time Systems*. IEEE, 1999, pp. 106–113.

[57] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen, "Towards the automatic verification of PLC programs written in Instruction List," in *IEEE Int. Conf. on Syst., Man, and Cybernetics*, vol. 4. IEEE, 2000, pp. 2449–2454.

[58] F. Jiménez-Fraustro and É. Rutten, "A synchronous model of IEC 61131 PLC languages in SIGNAL," in *13th Euromicro Conf. on Real-Time Syst.* IEEE, 2001, pp. 135–142.

[59] V. Gourcuff, O. De Smet, and J. Faure, "Efficient representation for formal verification of PLC programs," in *8th Int. Workshop on Discrete Event Syst.* IEEE, 2006, pp. 182–187.

[60] B. Zoubek, J.-M. Roussel, and M. Kwiatkowska, "Towards automatic verification of ladder logic programs," in *Proceedings of IMACS-IEEE CESA'03: Computational Engineering in Systems Applications*, 2003.

[61] D. F. Bender, B. Combemale, X. Crégut, J. M. Farines, B. Berthomieu, and F. Vernadat, "Ladder metamodeling and PLC program validation through time petri nets," in *Model Driven Architecture–Foundations and Applications*. Springer, 2008, pp. 121–136.

[62] H. Bel Mokadem, B. Berard, V. Gourcuff, O. De Smet, and J.-M. Roussel, "Verification of a timed multitask system with UPPAAL," *IEEE Trans. Autom. Sci. and Eng.*, vol. 7, no. 4, pp. 921–932, 2010.

[63] O. Pavlovic and H.-D. Ehrich, "Model checking PLC software written in function block diagram," in *3rd Int. Conf. on Softw. Testing, Verification and Validation (ICST)*. IEEE, 2010, pp. 439–448.

[64] D. Soliman, K. Thramboulidis, and G. Frey, "Transformation of function block diagrams to UPPAAL timed automata for the verification of safety applications," *Annual Reviews in Control*, 2012.

[65] A. Wardana, J. Folmer, and B. Vogel-Heuser, "Automatic program verification of continuous function chart based on model checking," in *35th Annual Conference of Industrial Electronics*. IEEE, 2009, pp. 2422–2427.

[66] L. H. Yoong, P. S. Roop, V. Vyatkin, and Z. Salcic, "A synchronous approach for IEC 61499 function block implementation," *IEEE Trans. Comput.*, vol. 58, no. 12, pp. 1599–1614, 2009.

[67] H. Prähofer and A. Zoitl, "Verification of hierarchical IEC 61499 component systems with behavioral event contracts," in *11th IEEE Int.l Conf. on Ind. Informatics (INDIN)*. IEEE, 2013, pp. 578–585.

[68] D. L'Her, P. Le Parc, and L. Marcé, "Proving sequential function chart programs using automata," in *Automata Implementation*. Springer, 1999, pp. 149–163.

[69] M. Remelhe, S. Lohmann, O. Stursberg, S. Engell, and N. Bauer, "Algorithmic verification of logic controllers given as sequential function charts," in *IEEE Int. Symposium on Comput. Aided Control Syst. Design*. IEEE, 2004, pp. 53–58.

[70] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg, "Verification of PLC programs given as sequential function charts," in *Integration of Software Specification Techniques for Applications in Engineering*, ser. Lecture Notes in Computer Science, H. Ehrig, W. Damm, J. Desel, M. Große-Rhode, W. Reif, E. Schnieder, and E. Westkämper, Eds. Springer, 2004, vol. 3147, ch. Part V: Verification, pp. 517–540.

[71] N. Bauer, R. Huuck, B. Lukoschus, and S. Engell, "A unifying semantics for sequential function charts," in *Integration of Software Specification Techniques for Applications in Engineering*. Springer, 2004, pp. 400–418.

[72] J. Provost, J.-M. Roussel, and J.-M. Faure, "Translating Grafcet specifications into Mealy machines for conformance test purposes," *Control Engineering Practice*, vol. 19, no. 9, pp. 947–957, 2011.

[73] ——, "A formal semantics for Grafcet specifications," in *Proceedings of the IEEE 7th International Conference on Automation Science and Engineering (CASE 2011)*, 2011.

[74] ——, "Generation of single input change test sequences for conformance test of programmable logic controllers," *IEEE Trans. Ind. Informat.*, vol. 10, no. 3, pp. 1696–1704, Aug. 2014.

[75] F. Schumacher, S. Schröck, and A. Fay, "Transforming hierarchical concepts of GRAFCET into a suitable petri net formalism," in *Manufacturing Modelling, Management, and Control*, vol. 7, no. 1, 2013, pp. 295–300.

[76] F. Schumacher and A. Fay, "Transforming time constraints of a GRAFCET graph into a suitable petri net formalism," in *IEEE Int. Conf. on Ind. Technology (ICIT)*. IEEE, 2013, pp. 210–218.

[77] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Science of computer programming*, vol. 72, no. 1, pp. 31–39, 2008.

[78] N. Schetinin, N. Moriz, B. Kumar, A. Maier, S. Faltinski, and O. Niggemann, "Why do verification approaches in automation rarely use HIL-test?" in *IEEE Int. Conf. on Ind. Technology (ICIT)*. IEEE, 2013, pp. 1428–1433.

[79] R. Hametner, D. Winkler, T. Östreicher, S. Biffl, and A. Zoitl, "The Adaptation of Test-Driven Software Processes to Industrial Automation Engineering," in *IEEE International Conference on Industrial Informatics (INDIN)*, 2010, pp. 921–927.

[80] T. Hussain and G. Frey, "UML-based Development Process for IEC 61499 with Automatic Test-case Generation," in *IEEE Conf. on Emerging Technologies and Factory Automation (ETFA'06)*, 2006, pp. 1277–1284.

[81] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, and A. Zoitl, "Test case generation approach for industrial automation systems," in *The 5th International Conference on Automation, Robotics and Applications*. IEEE, Dec. 2011, pp. 57–62.

[82] J. Krause, A. Herrmann, and C. Diedrich, "Test case generation from formal system specifications based on UML State Machine," in *atp–International*, 2008, pp. 47–54.

[83] B. Kormann, D. Tikhonov, and B. Vogel-Heuser, "Automated PLC Software Testing using adapted UML Sequence Diagrams," in *14th IFAC Symposium of Information Control Problems in Manufacturing*, Bucharest, Romania, 2012, pp. 1615–1621.

[84] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din, "From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing," in *Testing of Communicating Systems*, R. Khendek, Ferhat and Dssouli, Ed. Springer Berlin Heidelberg, 2005, pp. 289–303.

[85] B. Kumar, B. Czybik, and J. Jasperneite, "Model based TTCN-3 testing of industrial automation systems — First results," in *IEEE Conference on Emerging Technologies and Factory Automation*. IEEE, 2011, pp. 1–4.

[86] G. DeTommasi, R. Vitelli, L. Boncagni, and A. C. Neto, "Modeling of MARTe-Based Real-Time Applications With SysML," *IEEE Trans. Ind. Informat.*, vol. 9, no. 4, pp. 2407–2415, 2013.

[87] E. Estévez and M. Marcos, "Model-Based Validation of Industrial Control Systems," *IEEE Trans. Ind. Informat.*, vol. 8, no. 2, pp. 302–310, 2012.

[88] R. Svenningsson, J. Vinter, H. Eriksson, and M. Törngren, "MODIFI: A MODel-Implemented Fault Injection Tool," in *Computer Safety, Reliability, and Security, Lecture Notes in Computer Science Volume 6351*, 2010, pp. 210–222.

[89] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *IEEE Trans. Softw. Eng.*, vol. 16, no. 2, pp. 166–182, 1990.

[90] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson, "GOOFI: generic object-oriented fault injection tool," *Proc. Int. Conf. on Dependable Syst. and Networks*, pp. 83–88, 2001.

[91] H. Ziade, R. Ayoubi, and R. Velazco, "A Survey on Fault Injection Techniques," *The Int. Arab J. of Inform. Technology*, vol. 1, no. 2, pp. 171–186, 2004.

[92] J. Baraza, J. Gracia, D. Gil, and P. Gil, "Improvement of fault injection techniques based on VHDL code modification," in *10th IEEE Int. High-Level Design Validation and Test Workshop*. IEEE, 2005, pp. 19–26.

[93] H. Schlingloff and S. Vulinovic, "Model based dependability evaluation for automotive control functions," in *Modeling and simulation for public safety*, 2005.

[94] J. Vinterl, L. Bromander, P. Raistrick, and H. Edlerl, "FISCADE - A Fault Injection Tool for SCADE Models," in *3rd Inst. of Eng. and Technology Conf. on IET*, 2007, pp. 1–9.

[95] D. Powell, J. Arlat, H. N. Chu, F. Ingrand, and M. Killijian, "Testing the Input Timing Robustness of Real-Time Control Software for Autonomous Systems," in *2012 9th European Dependable Computing Conf.* IEEE, May 2012, pp. 73–83.

[96] B. Kormann and B. Vogel-Heuser, "Automated Test Case Generation Approach for PLC Control Software Exception Handling using Fault Injection," in *IECON 2011 - 37th Annual Conf. of the IEEE Ind. Electronics Soc.*, 2011, pp. 365 – 372.

[97] S. Rösch, D. Tikhonov, D. Schütz, and B. Vogel-Heuser, "Model-based testing of PLC software: test of plants' reliability by using fault injection on component level," in *IFAC World Conference, accepted paper*, 2014.

[98] A. Sung, B. Choi, W. E. Wong, and V. Debroy, "Mutant generation for embedded systems using kernel-based software and hardware fault simulation," *Inform. and Software Technology*, vol. 53, no. 10, pp. 1153–1164, Oct. 2011.

[99] J. Arlat and Y. Crouzet, "Physical Fault Models and Fault Tolerance," in *Models in Hardware Testing*, ser. Frontiers in Electronic Testing, H.-J. Wunderlich, Ed. Dordrecht: Springer Netherlands, 2010, vol. 43, pp. 217–255.

[100] H. Schludermann, T. Kirchmair, and M. Vorderwinkler, "Soft-commissioning: Hardware-in-the-loop-based verification of controller software," in *Proc. of the 2000 Winter Simulation Conf.*, no. Microsoft 1995, 2000, pp. 893–899.

[101] J. Carreira, H. Madeira, and J. G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," in *Dependable Computing and Fault Tolerant Systems 10*, 1998, pp. 245–266.

[102] P. Yuste, J. C. Ruiz, L. Lemus, and P. Gil, "Non-intrusive Software-Implemented Fault Injection," in *Dependable Computing*. Springer Berlin Heidelberg, 2003, pp. 23–38.

[103] IEEE Std 1219-1998, *IEEE Standard for Software Maintenance*. The Institute of Electrical and Electronics Engineers, Inc., 1998.

[104] G. Rothermel and M. Harrold, "Analyzing regression test selection techniques," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 529–551, 1996.

[105] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, Mar. 2010.

[106] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. Los Alamos, CA: The Institute of Electrical and Electronic Engineers, Inc., 1996.

[107] Q. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and A. Nadeem, "An approach for selective state machine based regression testing," in *Proc. of the 3rd Int. workshop on Advances in model-based testing (A-MOST '07)*. New York, New York, USA: ACM Press, 2007, pp. 44–52.

[108] L. Briand, Y. Labiche, and G. Soccar, "Automating impact analysis and regression test selection based on UML designs," in *Proc. Int. Conf. on Softw. Maintenance*. IEEE Comput. Soc, 2002, pp. 252–261.

[109] Y. Le Traon, T. Jeron, J.-M. Jezequel, and P. Morel, "Efficient object-oriented integration and regression testing," *IEEE Trans. Reliab.*, vol. 49, no. 1, pp. 12–25, Mar. 2000.

[110] P. Caliebe, T. Herpel, and R. German, "Dependency-Based Test Case Selection and Prioritization in Embedded Systems," in *2012 IEEE 5th Int. Conf. on Softw. Testing, Verification and Validation*. IEEE, Apr. 2012, pp. 731–735.

[111] Y. Chen, R. L. Probert, and H. Ural, "Regression test suite reduction based on SDL models of system requirements," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 21, no. 6, pp. 379–405, Nov. 2009.

[112] B. Korel, L. Tahat, and B. Vaysburg, "Model based regression test reduction using dependence analysis," in *Proc. Int. Conf. on Softw. Maintenance*. IEEE Comput. Soc, 2002, pp. 214–223.

[113] B. Ryder and F. Tip, "Change impact analysis for object-oriented programs," *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pp. 46–53, 2001.

[114] H. Leung and L. White, "A study of integration testing and software regression at the integration level," in *Proc. Conf. on Softw. Maintenance*. IEEE Comput. Soc. Press, 1990, pp. 290–301.

[115] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A Tool for Change Impact Analysis of Java Programs Categories and Subject Descriptors," *ACM Sigplan Notices*, vol. 39, no. 10, 2004.

[116] A. Orso, T. Apiwattanapong, and M. J. Harrold, "Leveraging field data for impact analysis and regression testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, p. 128, Sep. 2003.

[117] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Trans. Softw. Eng. and Methodology*, vol. 6, no. 2, pp. 173–210, Apr. 1997.

[118] Y. Chen, D. Rosenblum, and K. Vo, "TestTube: A system for selective regression testing," in *Proc. of the 16th Int. Conf. on Softw. Eng. (ICSE '94)*, 1994, pp. 211–220.

[119] S. Ulewicz, D. Schütz, and B. Vogel-Heuser, "Software changes in factory automation - towards automatic change based regression testing," in *40th Annual Conf. of the IEEE Ind. Electron. Soc.*, 2014.