Introducing Plant Features to Model-Based Testing of Programmable Controllers in Automation Systems

Canlong Ma*, Julien Provost

Technical University of Munich, Safe Embedded Systems, 85748 Garching, Germany

Abstract

This paper proposes a model-based test generation approach for programmable controllers that aims at reducing the length of a test sequence by applying plant features. The proposed approach does not require detailed or full knowledge of the plant behavior of a system under test, but it can achieve a remarkable reduction with simple plant features. As a result, the obtained test sequence can be significantly shorter than ones generated by complete testing methods; and meanwhile, it still reaches full coverage of the nominal behavior of the system under test. This makes it feasible to test large-scale systems, or to serve as an early test in the validation of safety critical systems. The proposed approach has been illustrated on a large-scale case study.

Keywords: model-based testing, plant feature, programmable controller, discrete event system, verification and validation

1. Introduction

Dependability of automation systems is measured and ensured by verification and validation methods such as model checking and testing. Model checking evaluates on the model level if a system meets some certain properties such as temporal logic by exhaustively exploring the whole state space. It has been widely accepted in the verification of software applications. For example, model checking has been applied to programmable controller programs by constructing formal models from semi-formal specifications [1], or by transforming implementations written in standard languages into an intermediate model [2].

An automation system is comprised of hardware and software components. Such a system often interacts with its physical environment through its sensors and actuators, while its components also interact with each other internally. Therefore, formal verification alone is not sufficient to guarantee the correctness of such systems, because the final system behavior which is performed by the hardware and influenced by the environment cannot be evaluated on the model level. In complement to formal verification, testing is strongly recommended and even compulsorily required as a validation method by many industrial standards such as IEC 61508, IEC 61511 and ISO 26262.

Test generation, as an important test activity, has been studied since long. Manual selection of test cases is the most intuitive and also most commonly used method. It is usually straightforward, expert-based and useful in much practice [3], [4], etc. Nevertheless, its disadvantages such as requiring individual customization, time-consuming, and error-prone, have become big obstacles for many modern industry applications where numerous new ideas, innovative technologies, and various user requirements are emerging.

The new solution is automatic test generation, for example with model-based techniques. Complete testing (CT) is naturally the first idea: to generate the test cases directly from the specification models. It by nature takes all possible combinations of input signals from all states into consideration. Similar to what model checking does, complete testing covers the whole behavior of a system under test (SUT), and it is therefore highly advantageous for safety critical systems. However, it is often not feasible for most practical applications, because the number of test cases and subsequently the length of a test sequence grow exponentially with the number of inputs, which as a result become soon incredibly large when the system size gets larger.

It is obvious that a testing with 100% coverage of all system behavior is hardly achievable for large-scale applications. Some researchers then proposed random testing as a compromise, aiming at reaching a high coverage with a relatively small set of test cases, and hoping they can find as many errors as possible. For example, [5] generated test cases based on the element identifier and function block-tree traversal; [6] used coverage metrics to implement a symbolic execution engine; [7] proposed an assessment approach to support increasing system test coverage through effectively identifying untested code and untested behav-

*Corresponding author
Email addresses: canlong.ma@tum.de (Canlong Ma), julien.provost@tum.de (Julien Provost)

ior of an SUT. However, as research results indicated, testings with coverage criteria satisfaction alone are not always powerful; they can be poor at effectively finding faults in some applications [8]. Since no system behavior has been considered in the generation of these random testings, critical faults may remain untested, and testers can never guarantee that a system would always work correctly, not only when faults occur, but also during the normal execution.

To provide a solution to this dilemma, this paper proposes a model-based test generation approach that guarantees full coverage of the nominal behavior of an SUT with a shortened test sequence. In this paper, an SUT can refer to an overall system, a subsystem, or a component which contains a controller. Here, "nominal behavior" means all the expected functional behavior that a controller should perform, which is documented in the specification[1]. Furthermore, when necessary, faulty behavior can also be easily included into the set of test cases, which is presented in Sec. 5.3 of this paper.

The core idea is to involve not only specification models but also plant features in the test generation. In an automation system, sensors and actuators are usually considered as *plants* while controllers can be understood as implemented *specifications*. In this paper, plant features are signal relations modeled in a simplified way which requires only limited design effort. In test generation, they serve as filters to the system behavior: only part of the behavior that conforms to the plant features will remain to be tested, the rest behavior that violates them will be filtered out. Consequently, the number of generated test cases and the length of a test sequence could be significantly reduced. Therefore, it becomes feasible to test large-scale systems with full coverage of their nominal behavior.

This paper provides a finalized version of this plant feature approach, while early results during the development have been published in [9] and [10]. The contributions of this paper are: modified mathematical formalism, improved algorithms that apply plant features in a very early phase of test generation, accomplished software tool chain that combines automatic test sequence generation mechanism, and a new application case study. Compared to previous results, the main improvements are: the state space of the computation is shrunk from very early phases, the number of test cases and subsequently the length of a final test sequence are further reduced. Besides, now the test generation process is fully automated, which means, after the specification and plant are modeled and reviewed (these need to be done manually), users just trigger the program, and then they can obtain an executable test sequence without other manual work.

The paper is structured as follows: In section 2 and 3, the related work and the mathematical formalism used to model specification and plant features are introduced. Section 4 and 5 present an overview of the testing objective and test generation process on programmable controllers, and the detailed techniques and algorithms used in the test generation, respectively. A large-scale case study is illustrated in section 6. Finally, section 7 concludes this work.

[1] In this paper, only non-timed implementations are considered, time dependencies are not considered.

2. Related work

2.1. Plant models in verification and validation

The importance of using plant models in verification and validation of programmable controllers has been generally acknowledged for a long time [11] [12] [13].

Most research work uses plant models for verification purpose such as model checking [14] [15], simulation [16] [17], and simulation-based test (in the scope of verification) [18] [19]. However, as pointed out in [20], simulation-based verification methods may always encounter two issues: real-world errors are not discovered in a simulated world, and errors are discovered that do not exist in the real world. The concern of the first issue is also valid for model checking, since a formal model is an abstraction of a real system with assumptions and constraints, as introduced in Sec. 1.

To cope with the first issue, one popular research direction is to build a better simulation interface and environment that are more close to the real world [17] [20] [19], another direction is to develop better plant modeling methods, i.e., automatic methods, thereby maximally avoiding human errors in the construction of models, improving modeling efficiency, and enhancing the overall applicability of plant models in verification and validation [21] [18] [22].

On the other hand, this shortage of verification can also be overcome by validation through testing, as explained in Sec. 1. The idea of having plant models in testing has also been considered and investigated recently. [23] created an automated test case generation approach for industrial automation applications where specification and plant models are specified by Unified Modeling Language (UML) state chart diagrams. However, the generation criterion is still about reaching high coverage rather than analytically considering system behavior, which is the goal this paper aiming to reach.

The concern of the first issue does not apply to the approach presented in this paper, because the test objective is to validate if the behavior of a controller conforms to its specification models (which will be explained in details in Sec. 4), not the behavior of the environment. The use of plant feature models is to limit the scope of behavior to be tested. As long as the plant features are correctly modeled, no extra error will be introduced to the testing activities.

The second issue can also be relieved by the approach presented in this paper. By applying plant features, test cases that are not/less meaningful in the real system are filtered out from the generated test sequence. More specifically, the proposed approach guarantees full coverage of

the nominal (plus optional faulty) system behavior, and maximally removes test cases that are not relevant.

2.2. Virtual commissioning

In automation engineering, virtual commissioning is a popular technology that tests a controller through simulation virtually with plant models before the real system is implemented. It requires the virtual plant environment to be fully described at the level of sensors and actuators with regard to aspects such as logic, geometry, and kinematics, which requires significant amount time and efforts for the modeling work [24].

Compare to virtual commissioning, the approach presented in this paper does not need a complete and concrete plant model that can run a simulation. The modeling effort is therefore much less. Furthermore, the two techniques should actually complement each other rather than compete since their purposes are different. With virtual commissioning, users can try things out very early even when the control code is not implemented, which is helpful but is not considered as serious functional testing, since real commissioning with real system is still necessary afterwards. With the approach in this paper, if a controller passes the test, it is formally proved to conform to its specification under nominal situation.

2.3. Open-loop / closed-loop testing

When testing a controller in a real system, there are two different architectures: open-loop and closed-loop testing ([25], page 91-95).

In open-loop testing, test stimuli are input signals for a controller, which are generated in advance and directly sent to the controller during test execution. For example, in [5], the applicable test cases are created according to the functional requirements and data flow of Function Block Diagram (FBD). The generated test cases are applied on the open-loop testing of several PLC devices in a smart home system.

As for the latter, a controller is embedded with real or simulated system plant so that the controller and the plant form a closed-loop; test stimuli are sent to the system plant and indirectly influence the controller. For example, [21] presented an automated procedure for constructing plant models for closed-loop simulation and testing of programmable controllers.

In this paper, plant features (models) are involved in the test generation, but not in the test execution. Therefore, the testing technique presented in this paper should be categorized into the group of open-loop testing. More details of the testing workflow are given in Sec. 4.

3. Mathematical background

In this paper, the specification and plant features of a system are modeled as Moore finite state machines. Boolean signals are used as inputs and outputs.

Higher-level modeling languages such as Grafcet, Statechart, and Petri net are not considered in this paper; however, the approach presented in this paper could be extended to most of them. Specifically, it has no issues to work with Grafcet thanks to researches done by [26]; with Petri net, the approach works with its reachable graph; with Statechart, it can work if the execution semantics has been modified accordingly.

As a general criterion, the approach can be applied on signal-based models with stability search, where all possible paths are considered independently of the execution/evaluation order of each transition [26]. Considering a set of models that can run in parallel, a situation is *stable* if no transition in any of the models can be fired unless the values of input signals are changed; otherwise, it is *transient*. The stability search semantics implies that the firing of transitions continues until a stable situation is reached. This semantics is used in the composition of models in this section.

3.1. Communicating Moore machine extended with Boolean signals

Specification is modeled as a set of Moore finite state machines which can communicate with each other.

Formally, a communicating Moore machine extended with Boolean signals is defined by an 8-tuple $(L, l_{init}, I, C, O, G_\delta, \delta, \lambda)$[2], where:

- L is a finite set of locations. A location represents a logic state of a single model for a subsystem/component[3].
- l_{init} is the initial location, $l_{init} \in L$.
- I is a finite set of Boolean input signals.
- C is a finite set of internal Boolean communicating variables that are related to locations; a communicating variable is denoted as '$X(location)$', e.g., '$X(l_1)$'.
- O is a finite set of Boolean output signals.
- $G_\delta := expr(I, C)$ is a finite set of transition guards, which are Boolean expressions[4] built up by inputs and internal variables.
- $\delta : L \times G_\delta \to L$ is the transition function that maps the current location and transition guard to the next location; a transition is fired when its source location is active and its guard is evaluated as '1' (i.e., *True*); 'Δ' is used to denote a set of 'δ'.

[2] The subscript 'S' will be used to stand for *Specification*, the subscript 'P' for *Plant*: e.g. L_S and L_P mean the set of locations for specification and plant feature models.

[3] The term *state* is explicitly used to represent a state in the composed model (introduced in the next part of this section), where the internal Boolean communicating variables are not used anymore; while a location is used for a subsystem/component. For a subsystem, its state is defined by $L \times C$.

[4] Boolean operators used in this paper: $\wedge$: AND; $\vee$: OR; $\neg$: Negation.

- $\lambda : L \rightarrow 2^O$ is the output function that maps the locations to their corresponding output signals; 'Λ' is used to denote a set of 'λ'.

Moore machines are also represented in a graphical form in this paper. A simple specification example is given in Fig. 1. A location l is drawn as a rounded rectangle. It can either have an externally observable output[5], e.g., o_3 in l_3, or no observable output, e.g., $\emptyset$ in l_1.

A transition δ is represented by an oriented arc with its guard $g(\delta)$, e.g., $\neg i_1 \wedge i_2$ for the transition from l_1 to l_2. The use of an internal communicating variable in transition guards is not complicated. For example, when the location l_6 is activated, $X(l_6)$ is assigned the value '1'. If l_2 is active at the same time, then the transition from l_2 to l_3 can be fired.

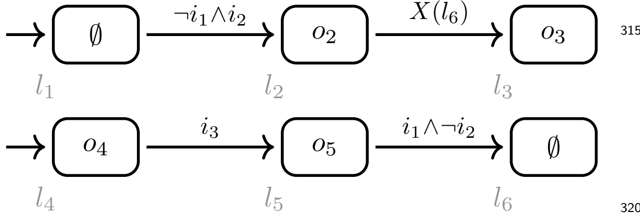


Figure 1: A simple Moore machine specification model with Boolean signals

3.2. Stabilized composed automaton

To generate test cases, in this paper, individual Moore machine models are first composed in parallel to build a monolithic model. An introduction of the parallel composition operation is not provided in this paper, readers can refer to [27] (page 79) for more information. Still, it is worth mentioning that the signal interpreted semantics with stability search, which has been introduced in the beginning of this section, is used in the composition. Therefore, the obtained model is named a *Stabilized Composed Automaton* (SCA) in this paper. A software program proposed in [26], Teloco, is used in this process.

Similar to an individual Moore machine, an SCA is defined by a 7-tuple $(S, s_{init}, I, O, G_e, e, \lambda_s)$, where:

- S is a finite set of states. A state represents a combination of locations from the individual models.
- s_{init} is the initial state, $s_{init} \in S$.
- I is a finite set of Boolean input signals (same as used in the individual models).
- O is a finite set of Boolean output signals (same as used in the individual models).
- $G_e := expr(I)$ is a finite set of evolution guards, which are Boolean expressions built up by inputs.
- $e : S \times G_e \rightarrow S$ is the evolution function that maps the current state and evolution guard to the next state; a *transition* between states is named an *evolution*.
- $\lambda_s : S \rightarrow 2^O$ is the output function that maps the states to their corresponding output signals.

3.3. Plant feature modeled as Moore machine extended with Boolean signals

As introduced in a previous publication [9], plant features are signal relations that can be described with three different languages: natural language, finite state machine, and temporal logic, which can be converted into each other. With the current implemented approach, the plant feature descriptions in the format of natural language or temporal logic need first to be converted into Moore machine models. For simplicity reason, this paper uses Moore machines with Boolean signals to model plant features.

The formalism of a plant feature model is similar to a specification model with two differences:

- $\lambda_P : L_P \rightarrow 2^I$: *inputs* of specification models are used as *outputs* in plant feature models.
- $G_{P,\delta_P} := expr(I, O)$: both *inputs* and *outputs* of specification models can be used in the transition guards in plant feature models.

A simple plant feature model example is given in Fig. 2, which can interact with the specification models in Fig. 1 since they have common signals.

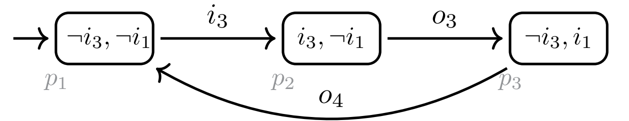


Figure 2: A simple Moore machine plant feature model with Boolean signals

This model can be understood as follows: initially, the signals i_1 and i_3 are both *False*, then i_1 remains $False$ when i_3 is activated; as soon as o_3 takes place, i_1 is activated and i_3 is deactivated; after o_4 occurs, the values of i_1 and i_3 turn $False$ again as described in the initial location.

4. Testing objective and test generation process

The objective of conformance testing is to check whether the behavior of an implemented programmable controller conforms to the behavior of its specification models [26]. Therefore, test cases and test sequences are generated from specification models. It is worth noting that the correctness of specification models is not part of the testing objective and therefore not in the scope of this paper. However,

[5] For readability reasons, only active outputs are presented, i.e., in l_3, o_3 implicitly means $o_3 \wedge \neg o_2 \wedge \neg o_4 \wedge \neg o_5$.

it is always suggested to assure the specification model correctness through requirement engineering measures such as review and inspection. For example, the specification models in the case study in this paper have been reviewed carefully.

As presented in Fig. 3, a model-based testing process for a programmable controller consists of four steps:

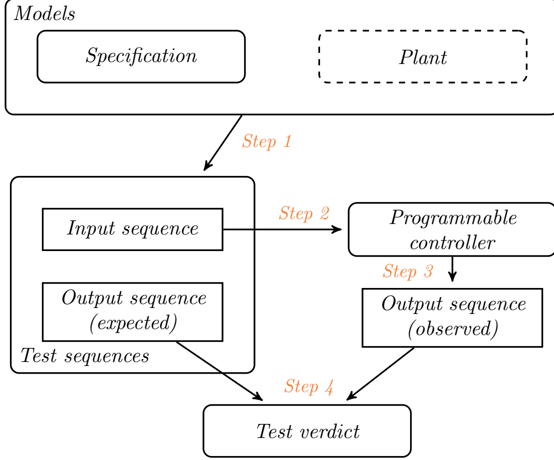


Figure 3: Workflow of testing a programmable controller

- **Step 1:** generate a test sequence (with input and output) from models (specification models with/without plant feature models)
- **Step 2:** feed the input sequence to the programmable controller
- **Step 3:** execute the implemented program on the controller
- **Step 4:** compare the observed output sequence to expected one, and record if the controller passes the test

The focus of this paper lies in the first step: construction of a *test sequence*, which is presented with more details in Fig. 4. The yellow blocks in Fig. 4 correspond to a classic process of test generation, i.e., complete testing. With complete testing, all possible behavior of the SUT should be covered, and no piece of system behavior should be filtered out, so only specification models are considered, no plant feature is involved. This test generation method has been presented in details in [26]. Firstly, all individual specification models are composed to obtain an SCA; then, an equivalent Mealy machine model is built from the SCA by explicitly representing all Boolean conditions of an evolution over the Boolean input set; the last task is to construct a test sequence which passes through different states and evolutions. A test case, as a single unit of the test sequence, is built up with a pair of one input and one output from the Mealy machine.

The length of a test sequence, as its core matter, is determined by two factors: the number of test cases, and

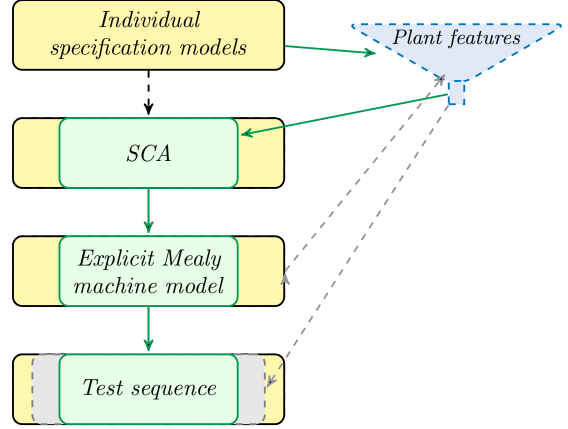


Figure 4: Framework of involving plant features in the test generation. Yellow blocks: generation of complete testing without plant features; Gray block and arrows: earlier version of test generation with plant features ([9], [10]); Green blocks and arrows: current version of test generation with plant features.

the ordering and repetition of test cases. The second factor comes into being because in practice, a state can have several outgoing evolutions, and some states have more evolutions to be tested than others. Therefore, in a test sequence, some evolution arcs need to be traversed several times. This is an instance of the Chinese Postman problem [28], and can be formulated as '*Find a minimum length closed path that visits each edge in the graph at least once*'. This paper uses the solution presented in [26].

The first factor is what this paper deals with. For a large-scale system, when the number of inputs of an SUT grows linearly, the sizes of SCA and Mealy machine model grow exponentially, and subsequently the number of test cases also grows exponentially, which results in the well-known *state space explosion* issue.

That was the motivation of involving plant features in the test generation. As mentioned in the introduction in this paper, plant features serve as filters to the system behavior to be tested. Hence, the state space explosion issue could be relieved to practically acceptable standard.

In [9] and [10], plant features are used after the Mealy machine model has been generated (see Fig. 4). As a result, the number of test cases can be remarkably reduced, and consequently the length of the generated test sequence is also remarkably shortened, when a set of simple plant features are involved.

In this paper, plant features are applied early in the generation of SCA (see Fig. 4, and the algorithms are presented in the next section). The new and additional advantages with regard to [9] and [10] are:

(1) Lower memory load for the test generation computer: Not only the length of test sequence is shortened, but also the sizes of SCA and Mealy machine model are reduced.

(2) Further shortening of the test sequence: In the generation of SCA, some states appearing in the complete

testing might not be reachable due to the interaction among plant features and specification models. Therefore, the SCA and Mealy machine model would contain fewer states to be tested.

It is worth reminding that, this method does not require very detailed or full plant feature models, but only fragments of knowledge from plant feature models. Modeling of such plant features is actually not complex, even simple and straightforward from a logical point of view. For example, for two signal i_1 and i_2, if they should not be true at the same time, they constitute a *mutual exclusion* relation; otherwise if i_2 can only be true when i_1 is true, then they constitute a *premise* relation. Such basic signal relations can be used as patterns to build other (more complex) signal relations. Details of the *mutual exclusion* and *premise* patterns of plant features can be found in [10].

Of course, the more plant features can be modeled, the smaller part of system behavior considered relevant for testing, and the greater reduction to the length of the test sequence can be achieved. A practical case study is provided in this paper to help readers obtain some intuitive feeling.

Additionally, this method can be combined with the idea of *fault injection*, a class of testing techniques which involves faulty behavior supplementary to the nominal behavior testing.

5. Test case generation with utilization of plant feature models

In an automation system, the controller is implemented according to *specification*, while the rest elements such as sensors and actuators are considered as *plant*. As presented in Fig. 5, specification and plant constitute a closed-loop. More specifically, plant is controlled by specification, directly as for actuators and indirectly as for other components; while the reachable space of specification is influenced by plant on the other hand.

In this paper, plant features are sorted into two levels: level 1 - signal relations among sensors, level 2 - signal relations among sensors and actuators. Following are two intuitive examples. As for level 1, in a water tank with two level sensors (high and low), and in a nominal situation, when the high level sensor gives the value *True*, the low level sensor should also give the value *True*. As for level 2, on a conveyor belt, only when the belt is running, sensors at input and output can change their values. In other words, in a nominal situation, a workpiece cannot move from the input to the output unless the belt runs.

5.1. Level 1: Signal relations among sensors

Alg. 1 presents the algorithm to consolidate plant features from plant feature models of level 1. The plant feature model given in Fig. 6 is used as a simple example to help illustrate the algorithm.

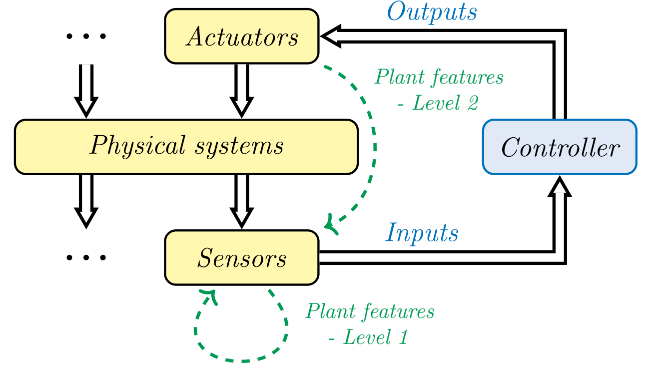


Figure 5: Specification and plant in an automation system

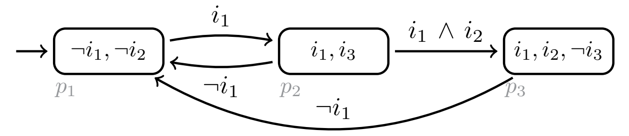


Figure 6: Example: plant feature model of level 1

L_P and Λ_P are the inputs of the algorithm, and represent the set of locations and outputs in a plant feature model, respectively. It is worth noting that the outputs and transition guards[6] of plant feature models are constituted by inputs from specification models. PF is the output of the algorithm, and represents the set of consolidated plant features, which will be used in the generation of SCA. A consolidated plant feature is defined with two attributes: *scope* and *cond*. The former indicates under which condition will this plant feature be used during the generation of SCA. The latter stores the formulated signal conditions that the evolution guards in the SCA should fulfill.

Firstly, the outputs of one location build up a basic element of a signal condition (line 3 to 6). For example, in Fig. 6, for location p_1, i_1 and i_2 should be both *False*. This model contains another input i_3, the value of which can be either *True* or *False* for location p_1, since it is not explicitly specified.

Every location in a plant feature model represents a part of the plant behavior. The final signal condition consists of the signal conditions of all the locations (line 7 to 8). In Fig. 6, it applies that in a nominal behavior, at least one of the following three conditions should be fulfilled: i_1 and i_2 be both *False*; i_1 and i_3 be both *True*; i_1 be *True*, i_2 be *True*, and i_3 be *False*.

Since sensor values are not modified by controllers, signal relations on this level are valid for all states. Therefore, the *scope* of a plant feature with level 1 is assigned $GLOBAL$ (line 9).

[6] For the current version, transition guards in plant feature models level 1 are not used, because timing features are not considered yet.

Algorithm 1: Consolidating plant features of level 1, i.e., signal relations among sensors

Input: L_P, Λ_P
Output: PF

1 **begin**
2 $\quad \lambda_{PF} := False$; /* initialization */
3 $\quad$ **foreach** $l_P \in L_P$ **do**
4 $\quad\quad \lambda_{PF,l_P} := True$; /* initialization */
5 $\quad\quad$ **foreach** $\lambda_P \in \Lambda_P(l_P)$ **do**
6 $\quad\quad\quad \lambda_{PF,l_P} := \lambda_{PF,l_P} \wedge \lambda_P$;
$\quad\quad\quad$ /* all the signal constraints in one location of a plant feature model need to be fulfilled at the same time, so merge them with 'AND' */
7 $\quad\quad \lambda_{PF} := \lambda_{PF} \vee \lambda_{PF,l_P}$;
$\quad\quad$ /* it is accepted as nominal if the signal constraints in any location of a plant feature model are fulfilled, so merge them with 'OR' */
8 $\quad pf.cond := \lambda_{PF}$;
9 $\quad pf.scope := GLOBAL$;
$\quad$ /* the plant features of level 1 are valid for all the states in the SCA */
10 $\quad PF := PF \cup \{pf\}$;

5.2. Level 2: Signal relations among sensors and actuators

The algorithm for consolidating plant features from plant feature models of level 2 is presented in Alg. 2. The plant feature model given in Fig. 7 is used as a simple example to help illustrate the algorithm.

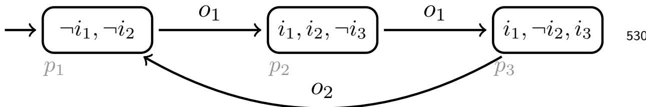


Figure 7: Example: plant feature model of level 2

L_P, Λ_P, Δ_P and G_{P,δ_P} are the inputs of the algorithm, and represent the set of locations, the set of outputs, the set of transitions, and the set of transition guards in a plant feature model, respectively. The outputs of plant feature models are also constituted by inputs from specification models, same as for level 1. The difference is that, transition guards in plant feature models level 2 are built up with outputs from specification models. PF is the output of the algorithm. It is defined with the attributes *scope* and *cond* in the same way as for level 1.

Similar to level 1, firstly, the outputs of one location build up a basic element of a signal condition (line 3 to 6). In the example of Fig. 7, for location p_1, i_1 and i_2 should be both *False* while i_3 can be either *True* or *False*.

A pair of location and transition in a plant feature model build up a candidate of plant feature. The condition is the consolidated outputs in the location (line 7). The scope is the transition guard, which is indeed Boolean expressions of outputs from specification models (line 8). In the generation of SCA later on, only the states whose outputs fulfill the Boolean expression (valued as *True*) will apply this plant feature. In Fig. 7, the first plant feature candidate, i.e., for location p_1, has the condition $\neg i_1 \wedge \neg i_2$ and the scope as o_2.

The following part of Alg. 2 deals with the issue that one scope of plant feature might lead to different conditions. Every condition represents a part of plant behavior for a scope. The final signal condition for a scope consists of all possible signals conditions (line 10 to 14). For example, in Fig. 7, two plant feature candidates have the same scope o_1, and different conditions, $i_1 \wedge i_2 \wedge \neg i_3$ and $i_1 \wedge \neg i_2 \wedge i_3$. The two candidates are merged into a final plant feature that has the scope o_1 and the condition $i_1 \wedge (i_2 \wedge \neg i_3 \vee \neg i_2 \wedge i_3)$.

5.3. Modifying plant features for fault injection

Apart from nominal behavior, when users also want to test how their system will behave if faults occur, they can use a type of testing techniques called fault injection. Usually, users select the faults to inject into the set of test cases based on their practical experience and domain knowledge. For instance, if they find some components are more error-prone than others in their applications, they would like to explicitly inject these errors when they generate the tests.

More knowledge and techniques about fault injection can be found in [29]. In this paper, fault injection can be realized conveniently by modifying the plant feature models. By definition, plant feature models represent the nominal situation that all sensors and actuators work properly. Therefore, in order to introduce a fault into the set of test cases, users just need to remove the corresponding plant feature models, or modify them to be less restrictive. As a result, the set of test cases becomes larger and the test sequence becomes longer, since some test cases that were previously excluded by the nominal behavior are now also considered. In extreme cases, when all the plant feature

Algorithm 2: Consolidating plant features of level 2, i.e., signal relations among sensors and actuators

```
Input: L_P, λ_P, Δ_P, G_{P,δ_P}
Output: PF
1  begin
2      PF_temp := ∅; PF_rm := ∅; /* initialization */
3      foreach δ_P ∈ Δ_P | l_{P,src} × g_{P,δ_P} → l_{P,des} do
4          λ_{PF,l_P} := True; /* initialization */
5          foreach λ_P ∈ Λ_P(l_{P,des}) do
6              λ_{PF,l_P} := λ_{PF,l_P} ∧ λ_P;
               /* the signal constraints in one location need to be fulfilled at the same time */
7          pf.cond := λ_{PF,l_P};
8          pf.scope := g_{P,δ_P};
           /* the plant features of level 2 are valid only for the states of SCA which hold the relevant
              actions */
9          PF_temp := PF_temp ∪ {pf};
10     foreach pf_ref ∈ PF_temp do
11         foreach pf_cpr ∈ PF_temp\PF_rm do
12             if pf_ref.scope = pf_cpr.scope and pf_ref.cond ≠ pf_cpr.cond then
13                 pf_ref.cond := pf_ref.cond ∨ pf_cpr.cond;
                   /* if an action can lead to different pf conditions, merge them with 'OR' (ref: reference;
                      cpr: compare) */
14             PF_rm := PF_rm ∪ {pf_cpr};
               /* save the used and redundant plant features in the set PF_rm (rm: remove) */
15     foreach pf ∈ PF_temp\PF_rm do
16         PF := PF ∪ {pf};
```

models are removed, users perform then actually complete testing, with which all possible faults are considered.

5.4. Applying plant features in the generation of SCA

The generation of SCA is done by synchronous composition of individual specification models with stability search.

With the new method proposed in this paper, plant features are applied in the generation of SCA as introduced in Sec. 4. The main point is that, when an evolution guard from one state is created, it will be combined with consolidated plant features. This step is presented in Alg. 3.

Given an evolution guard, for plant features level 1, i.e., the plant feature scope is $GLOBAL$, the evolution guard is modified by simply adding the plant feature condition into it (line 3 to 5).

For plant features level 2, firstly the evolution will be checked, if outputs of its source state of this fulfill the plant feature scope. If yes, then this plant feature condition will also be added to the evolution guard (line 6 to 10). For example, a system has an output set $O_S := \{o_1, o_2, o_3\}$, if the scope of a plant feature is $o_1 \wedge \neg o_2$, and the source state of an evolution has the outputs $\{o_1, o_3\}$, the plant feature should be applied for this evolution; but it will not be applied to another evolution whose source state has the outputs $\{o_1, o_2\}$.

6. Case study: A flexible manufacturing system

A benchmark case study (Fig. 8) originally presented in [30] is used in this paper to illustrate the proposed test generation approach.

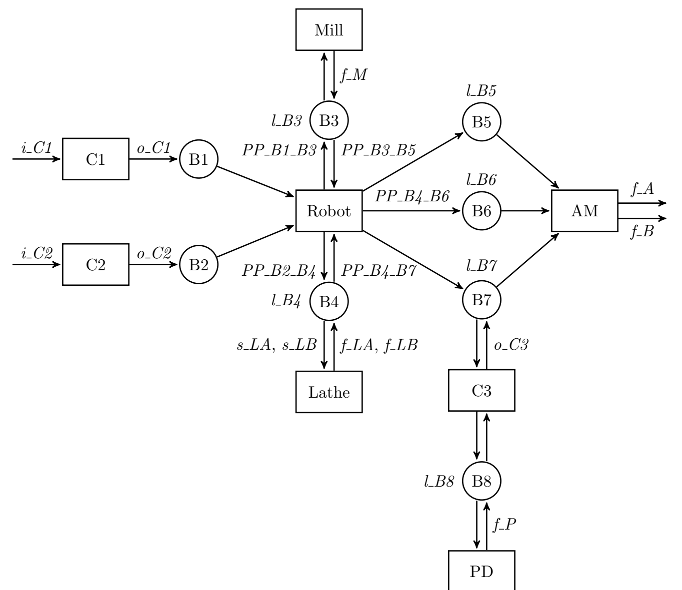


Figure 8: Case study: a flexible manufacturing system

Algorithm 3: Modification of evolution guards involving consolidated plant features in the generation of SCA

Input: $PF, s_S, g_{S,e}$
Output: $g_{S,e,wP}$
1 **begin**
2 $g_{S,e,wP} := g_{S,e}$; `/* initialization */`
3 **foreach** $pf \in PF$ **do**
4 **if** $pf.scope = GLOBAL$ **then**
5 $g_{S,e,wP} := g_{S,e,wP} \wedge pf.cond$;
`/* combine an original evolution condition with a plant condition, so that the final evolution condition in SCA conforms to this plant feature */`
6 **else**
7 $dict_O := getDict(state)$;
`/* return the output list of a state as a dictionary data */`
8 **if** $applyValue(pf.scope, dict_O) = True$
`/* output of this plant feature is valued as True with the output data of this state */`
9 **then**
10 $g_{S,e,wP} := g_{S,e,wP} \wedge pf.cond$;

6.1. Description of the system

As presented in Fig. 8, a flexible manufacturing system (FMS) consists of eight devices: three conveyors C1, C2 and C3, a mill, a lathe, a robot, a painting device (PD), and an assembly machine (AM). The devices are connected through buffers Bj, $j = 1, \ldots, 8$, each with capacity of one piece.

The FMS system is modeled with 19 input and 14 output signals, as listed in Tab. 1.

New products enter the system with C1 and C2. C1 supplies blocks and C2 supplies pegs. The blocks go through the mill, and the pegs go through the lathe to be shaped conical (type A) or cylindrical (type B). Cylindrical pegs are additionally painted through the painting device. The end products are blocks with attached conical pegs (type A) and blocks with cylindrical painted pegs (type B). The flow of products in the system is mainly directed by the robot and the buffer specifications.

The specifications are modeled with seven Moore machines, which contain 2, 2, 2, 3, 3, 5, and 6 locations, respectively. For the sake of brevity, the two models for the lathe and B4, and the robot are selected as illustrative examples and presented in Fig. 9.

6.2. Complete test generation

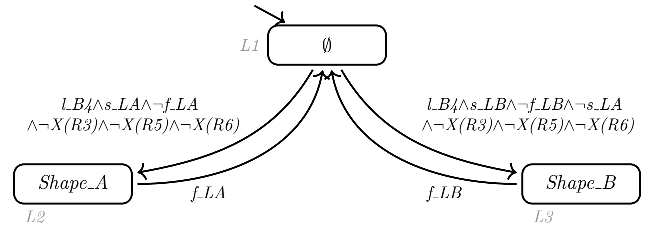
Applying Teloco [26], the SCA of the seven specification models contains 1170 states and 368,626 evolutions. Since the system has 19 inputs, the Mealy machine of the SCA contains $1170 * 2^{19} = 613{,}416{,}960$ test cases.

Based on that, an executable test sequence is obtained with 845,525,235 steps.

6.3. Test generation with plant features

Plant features are modeled by inspecting the physical structures and functional relations of the system. For the

Specification model - Lathe & Buffer4



Specification model - Robot

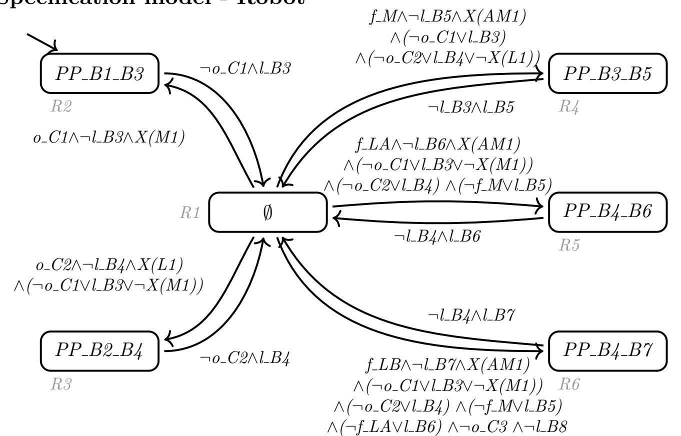


Figure 9: Specification models for two subsystems: *Lathe-Buffer4* and *Robot*

Table 1: Inputs & outputs of the flexible manufacturing system

Input	Description
i_C1 / i_C2	activated when a new product (block / peg) is detected at the input of C1 / C2
o_C1 / o_C2 / o_C3	activated when a product (block / peg / painted cylindrical peg) is detected at the output of C1 / C2 / C3
l_B3 / l_B4 / l_B5 / l_B6 / l_B7 / l_B8	activated when a workpiece is loaded in B3 / B4 / B5 / B6 / B7 / B8
f_M	activated when the mill finishes milling a block
s_LA / s_LB	activated when the lathe starts to shape a peg to be conical (type A) / cylindrical (type B)
f_LA / f_LB	activated when the lathe finishes shaping a peg to be conical (type A) / cylindrical (type B)
f_P	activated when the painting device finishes painting a cylindrical peg
f_A / f_B	activated when the assembly machine finishes assembling a final product (a block with a conical peg (type A) / a block with a cylindrical painted peg (type B))

Output	Description
Run_FW_C1 / Run_FW_C2 / Run_FW_C3	the conveyor belt C1 / C2 / C3 runs in a forward direction
Run_BW_C3	the conveyor belt C3 runs in a backward direction
PP_B1_B3 / PP_B2_B4 / PP_B3_B5 / PP_B4_B6 / PP_B4_B7	the robot picks and places a product from one buffer to another
Mill	the mill mills a block
Shape_A / Shape_B	the lathe shapes a peg to be conical (type A) / cylindrical (type B)
Paint	the painting device paints a cylindrical peg
Assemble	the assembly machine assembles a final product, i.e., a block with a peg

case study of FMS, 17 plant feature models have been built. As illustrative examples, the plant feature models for the lathe and B4, and the robot are presented in Fig. 10.

In *pl1*, i.e., the first plant feature model for the lathe and B4, *l_B4* is a premise of *f_LA, s_LA, f_LB* and *s_LB*, because the lathe can only operate when there is a workpiece available from B4. The models *pl2* and *pl3* describe a similar plant feature of premise relation between *f_LA vs. s_LA*, and *f_LB vs. s_LB*, respectively.

In *pl4*, *f_LA* and *f_LB* are mutually exclusive, since the lathe cannot do both types of shaping operations simultaneously.

In *pr1*, i.e., the first plant feature model for the robot, when the robot does the action *PP_B3_B5*, a workpiece is taken away from B3, and thus the sensor signal *l_B3* turns immediately to be *False*. *l_B3* will eventually turn *True* when another action *PP_B1_B3* is taken. Similar plant features exist among some other output and input signals, as presented in *pr2*, *pr3*, *pr4*, respectively.

Combining the 17 plant feature models with the 7 specification models in the test generation, the newly obtained SCA contains 970 states and 134,637 evolutions. The newly generated Mealy machine contains 12,514,080 test cases. The final executable test sequence is then generated with 18,363,192 steps.

6.4. Test generation with fault injection

In practice, faults can occur in sensors and actuators, in software and hardware, in the beginning and after a long time of operation, etc. For instance, a sensor can misdetect whether a workpiece is loaded, and an actuator can fail to pick and place a product. To make an example, let us suppose something goes run with the sensor signal *l_B3* and the actuator signal *PP_B3_B5*. That means the nominal signal relation between *l_B3* and *PP_B3_B5* does not hold anymore.

In order to test how the controller would behavior when the above-discussed situation occurs, corresponding faults can be easily injected into the expected set of test cases, by removing *pr1*, the first plant feature model in Fig. 10.

After updating the models and re-executing the test generation, a set of 16,523,040 test cases in the Mealy machine, and a test sequence with the length of 23,955,766 are obtained. The numbers have increased by 32.0% and 30.5% respectively compared to the tests for pure nominal behavior (since now more system behavior are tested), but they have still decreased by 97.3% and 97.2% respectively compared to CT.

6.5. Comparison of results

The test generation results of different methods are presented in Tab. 2, including also the results of the previous algorithms [10]. In summary, with the proposed method, a remarkably smaller set of test cases and also a significantly shorter test sequence are obtained compared to the ones generated with a complete testing.

In practice, the cycle time of a PLC now varies roughly from 1ms to 100ms. For example, if a controller has the cycle time of 10ms, after applying plant features in the test generation, the test execution time is reduced from 2348 hours (97.8 days) to 51 hours (2.1 days) for full coverage of nominal behavior, and to 67 hours (2.8 days) for full coverage of nominal behavior with extra consideration of certain faults, respectively. When the execution time for complete testing is possibly unacceptable to users, they are able to identify whether their controllers work properly in a normal/expected environment with much shorter testing time, which probably fulfills their most urgent demands at first.

[7] Executable sequence is not generated with previous algorithms.

Table 2: Results and comparison of different test generation methods on the case study

Test Generation method	Size of SCA		Number of test cases in the Mealy machine	Length of executable test sequence
	#state	#evolution		
Complete testing (CT)	1170	368,626	613,416,960	845,525,235
Plant features approach (previous algorithms; nominal behavior)	1170	368,626	13,739,040	N/A[7]
- Comparison to CT -	-0%	-0%	-97.8%	N/A
Plant features approach (new algorithms; nominal behavior)	970	134,637	12,514,080	18,363,192
- Comparison to CT -	-17.1%	-63.5%	-98.0%	-97.8%
Plant features approach (new algorithms; with fault injection)	1090	153,596	16,523,040	23,955,766
- Comparison to CT -	-6.8%	-58.3%	-97.3%	-97.2%

7. Conclusion and outlook on future work

The model-based test generation method proposed in this paper aims at reducing the number of test cases / length of test sequence in testing programmable controllers, by utilizing plant features extracted from a system under test. Meanwhile, the obtained shortened test sequence still achieves full coverage of the nominal behavior of the system under test. This can be a good remedy for large-scale systems where a complete testing is usually not realistic due to system complexity; or serve as a first validation step for safety critical systems, which enables to detect faults earlier.

Plant features are modeled as finite state machines in this paper. It is worth mentioning that this method does not require detailed or full plant feature models. Any fragment of plant knowledge can contribute to the reduction. Furthermore, users can insert a selected set of faults into the target behavior to be tested by modifying the plant features.

Currently, only Boolean signals are taken into account for the control logic. Actually, this approach can be extended to handle other types of signals as well, e.g., digital signals with integer values or analog signals. It is obvious that the state space of test would be even larger since the signals can have multiple values. To cope with the state-space explosion issue, equivalence class partition techniques [31] are of interest. The executed test cases will be then representatives selected from each equivalence partition. As a result, large and possibly infinite input data types and ranges can be reduced into a limited set of equivalence partitions.

In this paper, plant features have been classified into two levels: signal relations among sensors, and signal relations among sensors and actuators. The two levels all deal with current values of sensors and actuators. In addition to the current results, other kinds of plant features including timing features, and temporal features such as historical traces of sensor and actuator values can also affect their current values. For example, only after a machine has been running for a certain amount of time, it can send a signal that an operation is finished. Another example, a belt is off at the beginning and the end, but if it has been turned on for a while in-between, then the position of a product on the belt should have been changed. To achieve a better description of the nominal system behavior and consequently a more efficient test sequence, these extensions will be considered in the continued research. Some above mentioned ideas have already been realized and presented in [32].

8. Acknowledgments

This research was supported and funded by Technical University of Munich. The authors do not declare any conflict of interest.

References

[1] D. Soliman, G. Frey, Verification and validation of safety applications based on PLCopen safety function blocks, Control Engineering Practice 19 (9) (2011) 929–946. doi:10.1016/j.conengprac.2011.01.001.

[2] B. F. Adiego, D. Darvas, E. B. Viñuela, J. C. Tournier, S. Bliudze, J. O. Blech, V. M. G. Suárez, Applying model checking to industrial-sized PLC programs, IEEE Transactions on Industrial Informatics 11 (6) (2015) 1400–1410.

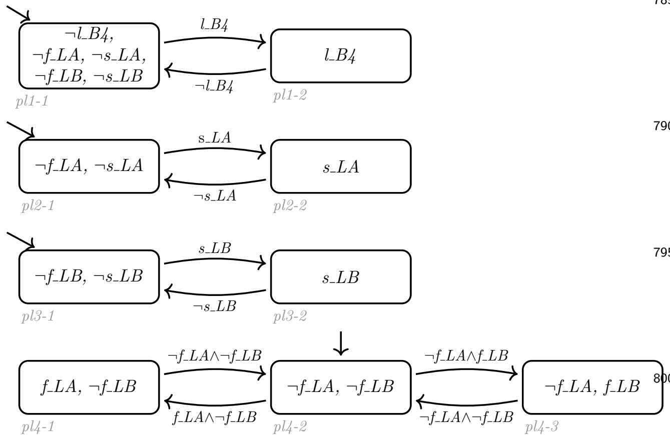
[3] J. Mcgregor, Testing a software product line, Tech. rep., Carnegie Mellon University (2001).

[4] E. Jee, D. Shin, S. Cha, J.-s. Lee, D.-h. Bae, Automated test case generation for FBD programs implementing reactor protection system software, Software Testing, Verification and Reliability 24 (8) (2014) 608–628. doi:10.1002/stvr.

[5] P. Mani, M. Prasanna, Automatic test case generation for programmable logic controller using function block diagram, in: International Conference on Information Communication and Embedded Systems (ICICES), 2016, pp. 1–4.

[6] D. Bohlender, H. Simon, N. Friedrich, S. Kowalewski, S. Hauck-Stattelmann, Concolic test generation for PLC programs using coverage metrics, in: Discrete Event Systems (WODES), 2016 13th International Workshop on. IEEE., 2016, pp. 432–437.

Plant models - Lathe & Buffer4 (Level 1)



Plant models - Robot (Level 2)

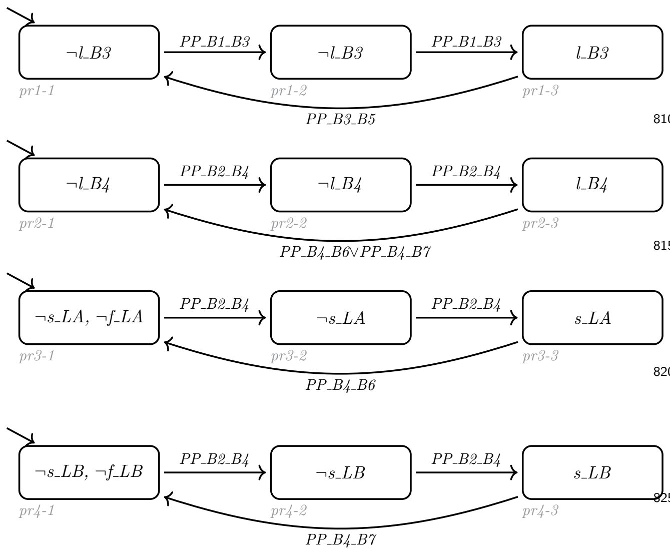


Figure 10: Plant feature models for two subsystems: *Lathe-Buffer4* and *Robot*

[7] S. Ulewicz, B. Vogel-Heuser, Increasing system test coverage in production automation systems, Control Engineering Practice 73 (2018) 171–185. doi:10.1016/j.conengprac.2018.01.010.

[8] G. Gay, M. Staats, M. Whalen, M. P. Heimdahl, The risks of coverage-directed test case generation, IEEE Transactions on Software Engineering 41 (8) (2015) 803–819. doi:10.1109/TSE.2015.2421011.

[9] C. Ma, J. Provost, Using plant model features in generation of test cases for programmable controllers, in: 20th World Congress The International Federation of Automatic Control, 2017, pp. 11655–11660.

[10] C. Ma, J. Provost, A model-based testing framework with reduced set of test cases for programmable controllers, in: 13th IEEE Conference on Automation Science and Engineering (CASE), 2017, pp. 944–949.

[11] M. Rausch, B. H. Krogh, Formal verification of PLC programs, in: Proceedings of the American Control Conference, Vol. 1, 1998, pp. 234–238. doi:10.1109/ACC.1998.694666.

[12] M. B. Younis, G. Frey, Formalization of existing PLC programs: A survey, in: Proceedings of CESA, Lille, France, 2003, pp. 0234–0239.

[13] J. M. Machado, B. Denis, J.-J. Lesage, J.-M. Faure, J. C. Ferreira Da Silva, Logic controllers dependability verification using a plant model, IFAC Proceedings 39 (17) (2006) 37–42. doi:10.3182/20060926-3-PL-4904.00007.

[14] J. M. Machado, B. Denis, J.-J. Lesage, J.-M. Faure, J. C. Ferreira Da Silva, Increasing the efficiency of PLC program verification using a plant model, in: 6th International Conference on Industrial Enginneering and Production Management (IEPM'03), 2003.

[15] P. Ovsiannikova, D. Chivilikhin, V. Ulyantsev, A. Shalyto, Closed-loop verification of a compensating group drive model using synthesized formal plant model, in: Emerging Technologies and Factory Automation (ETFA), 2017 22nd IEEE International Conference on. IEEE, 2017, pp. 1–4.

[16] I. Hegny, M. Wenger, A. Zoitl, IEC 61499 based simulation framework for model-driven production systems development, in: 2010 IEEE 15th Conference on Emerging Technologies & Factory Automation (ETFA 2010), no. October, 2010, pp. 1–8. doi:10.1109/ETFA.2010.5641364.

[17] C.-h. Yang, G. Zhabelova, C.-w. Yang, S. Member, Cosimulation environment for event-driven distributed controls of smart grid, IEEE Transactions on Industrial Informatics 9 (3) (2013) 1423–1435.

[18] M. Barth, A. Fay, Automated generation of simulation models for control code tests, Control Engineering Practice 21 (2) (2013) 218–230. doi:10.1016/j.conengprac.2012.09.022.

[19] S. Suess, S. Magnus, M. Thron, H. Zipper, U. Odefey, V. Faessler, A. Strahilov, A. Klodowski, T. Baer, C. Diedrich, Test methodology for virtual commissioning based on behaviour simulation of production systems, in: IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2016, pp. 1–9. doi:10.1109/ETFA.2016.7733624.

[20] H. Carlsson, B. Svensson, F. Danielsson, B. Lennartson, Methods for reliable simulation-based PLC code verification, IEEE Transactions on Industrial Informatics 8 (2) (2012) 267–278. doi:10.1109/TII.2011.2182653.

[21] H. T. Park, J. G. Kwak, G. N. Wang, S. C. Park, Plant model generation for PLC simulation, International Journal of Production Research 48 (5) (2010) 1517–1529. doi:10.1080/00207540802577961.

[22] I. Buzhinsky, V. Vyatkin, Automatic inference of finite-state plant models from traces and temporal properties, IEEE Transactions on Industrial Informatics 13 (4) (2017) 1521–1530.

[23] R. Hametner, B. Kormann, B. Vogel-Heuser, D. Winkler, A. Zoitl, Test case generation approach for industrial automation systems, in: The 5th International Conference on Automation, Robotics and Applications, 2011, pp. 57–62. doi:10.1109/ICARA.2011.6144856.

[24] C. G. Lee, S. C. Park, Survey on the virtual commissioning of manufacturing systems, Journal of Computational Design and Engineering 1 (3) (2014) 213–222. doi:10.7315/jcde.2014.021.

[25] J. Babić, Model-based approach to real-time embedded control systems development with legacy components integration, Doctoral dissertation, Sveučilište u Zagrebu (2014).

[26] J. Provost, J. M. Roussel, J. M. Faure, Translating Grafcet specifications into Mealy machines for conformance test purposes, Control Engineering Practice 19 (9) (2011) 947–957. doi:10.1016/j.conengprac.2010.10.001.

[27] C. G. Cassandras, S. Lafortune, Introduction to discrete event systems, 2nd Edition, Springer Science & Business Media, 2009. doi:10.1109/tac.2001.905709.

[28] M.-k. Kwan, Programming method using odd or even pints, Acta Mathematica Sinica 10 (1960) 263–266.

[29] S. Rösch, B. Vogel-Heuser, A light-weight fault injection approach to test automated production system PLC software in industrial practice, Control Engineering Practice 58 (2017) 12–23. doi:10.1016/j.conengprac.2016.09.012.

[30] M. H. De Queiroz, J. E. R. Cury, Modular multitasking supervisory control of composite discrete-event systems, in: IFAC Proceedings Volumes (IFAC-PapersOnline), Vol. 16, 2005, pp. 91–96. doi:10.3182/20050703-6-CZ-1902.00300.

[31] ISTQB, Standard glossary of terms used in Software Testing,

855 version 2. Edition, Vol. 3, 2014.

[32] C. V. Jordan, J. C. Herrero, J. Provost, Extension of the Plant Feature Approach Introducing Temporal Relations, in: Automation Science and Engineering (CASE), 2018 IEEE International Conference on, 2018, pp. 1164–1169.