TECHNISCHE UNIVERSITÄT MÜNCHEN

Fakultät für Maschinenwesen

Professur für Sichere Eingebettete Systeme

Advances in Model-Based Testing of Programmable Controllers: Automatic Test Generation using Design-to-Test and Plant Features

Canlong Ma, M.Sc.

Vollständiger Abdruck der von der Fakultät für Maschinenwesen der Technische Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr.-Ing. Boris Lohmann

Prüfer der Dissertation:

1. Prof. Dr. Julien Provost
2. Prof. Dr.-Ing. Georg Frey

Die Dissertation wurde am 06.12.2018 bei der Technische Universität München eingereicht und durch die Fakultät für Maschinenwesen am 05.07.2019 angenommen.

Anhang I

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich bei der **Fakultät für Maschinenwesen** der TUM zur Promotionsprüfung vorgelegte Arbeit mit dem Titel: **Advances in Model-Based Testing of Programmable Controllers: Automatic Test Generation using Design-to-Test and Plant Features**, in **Professur für Sichere Eingebettete Systeme** unter der Anleitung und Betreuung durch **Prof. Dr. Julien Provost** ohne sonstige Hilfe erstellt und bei der Abfassung nur die gemäß § 6 Ab. 6 und 7 Satz 2 angebotenen Hilfsmittel benutzt habe.

☐ Ich habe keine Organisation eingeschaltet, die gegen Entgelt Betreuerinnen und Betreuer für die Anfertigung von Dissertationen sucht, oder die mir obliegenden Pflichten hinsichtlich der Prüfungsleistungen für mich ganz oder teilweise erledigt.

☐ Ich habe die Dissertation in dieser oder ähnlicher Form in keinem anderen Prüfungsverfahren als Prüfungsleistung vorgelegt.

☐ Die vollständige Dissertation wurde in **mediaTUM** veröffenlicht. Die **Fakultät für Maschinenwesen** hat der Veröffentlichung zugestimmt.

☐ Ich habe den angestrebten Doktorgrad noch nicht erworben und bin nicht in einem früheren Promotionsverfahren für den angestrebten Doktorgrad endgültig gescheitert.

☐ Ich habe bereits am _____ bei der Fakultät für _____
_____ der Hochschule _____
unter Vorlage einer Dissertation mit dem Thema _____
_____
die Zulassung zur Promotion beantragt mit dem Ergebnis: _____
_____

Die öffentlich zugängliche Promotionsordnung der TUM ist mir bekannt, insbesondere habe ich die Bedeutung von § 28 (Nichtigkeit der Promotion) und § 29 (Entzug des Doktorgrades) zur Kenntnis genommen. Ich bin mir der Konsequenzen einer falschen Eidesstattlichen Erklärung bewusst.

Mit der Aufnahme meiner personenbezogenen Daten in die Alumni-Datei bei der TUM bin ich

☐ einverstanden

☐ nicht einverstanden

_____

Ort, Datum, Unterschrift

Acknowledgment

I would like to express my acknowledgments to many people throughout the whole research journey during my PhD.

First, I would like to thank my supervisor, Prof. Dr. Julien Provost, a great mentor not only for my research but also for my personal development. He always took the time to discuss with me, answer my questions, and give me advices. During these years, I have learned a lot from him, not only the knowledge and technologies, but also the passion, enthusiasm, and patience for research and work.

I would also like to thank my second reviewer, Prof. Dr.-Ing. Georg Frey, for giving me great feedback and suggestions, as well as to thank Prof. Dr.-Ing. Boris Lohmann for serving as my committee chair.

I thank my dear SES colleagues Claudius Jordan, Laurin Prenzel, and Nancy Elhady, for supporting me and providing me with great suggestions for any issue I ever had, in work and life. I wish you great success in your research and life. During my PhD, I also thank my friends Minjie Zou, Wei Zhu, Jiejia Hu, and all my other friends for always encouraging me and giving me valuable advices whenever I needed.

I want to deliver special thanks to my parents, sister and family for understanding, supporting and encouraging me in my whole life, even though we lived physically far away most of the time during my PhD. Finally, most thanks to Congying, my wife, you are the light of my life.

Canlong Ma
November 28th, 2018, Munich

Abstract

In this thesis, two novel approaches aiming at increasing the effectiveness and efficiency during the model-based testing of programmable controllers in automation systems are presented: **design-to-test (DTT)** and **plant features (PFs)**.

These two approaches deal with black-box conformance testing, where the specifications and implementations can be modeled as finite state machines (FSMs). Given an automation system, the testing objective is to validate whether the implemented controller conforms to expected input-output behavior with regard to their specification models. However, existing testing methods suffer from various issues and are therefore not well applicable for current industrial applications.

On the one hand, the DTT approach aims to improve the *effectiveness* of complete testing, which is indispensable for critical systems. The specification models are automatically checked and modified with limited design overhead in order to improve the testability of their physical implementation, namely its controllability, observability, and single-input-change testability. This approach also guarantees, by design, that the behavior of the implementation remains unchanged during its normal execution, i.e., when disconnected from a test bench.

On the other hand, the PF approach attempts to enhance the *efficiency* of testing for large scale systems where complete testing is hardly realistic. Plant features are manually modeled using simple templates (which also limits the modeling overhead), and then automatically fed into test generation. As a result, the input space of a system under test and the number of *meaningful* test cases can be significantly reduced, and consequently, the length of an executable test sequence can also be significantly shortened. It is worth mentioning that the obtained shortened test sequence guarantees full coverage of the whole *nominal* behavior of a system under test.

Based on case studies, these two approaches outperform the current methods and advance the model-based testing of programmable controllers.

Zusammenfassung

In dieser Arbeit werden zwei innovative Ansätze vorgestellt, die die Effektivität und Effizienz modellbasierten Testens für programmierbare Steuergeräte in Automatisierungssystemen erhöhen sollen: Design-to-Test (DTT, dt. Entwurf-für-Testen) und Plant Features (PFs, dt. Anlageneigenschaften).

Die beiden Ansätze befassen sich mit Black-Box-Konformitätstests für programmierbare Steuergeräte, wobei die Spezifikationen und Implementierungen als endliche Automaten modelliert werden können. Bei einem Automatisierungssystem besteht das Testziel darin, zu validieren, ob das implementierte Steuergerät dem erwarteten Eingabe-Ausgabe-Verhalten hinsichtlich seiner Spezifikationsmodelle entspricht. Bestehende Testmethoden leiden jedoch unter verschiedenen Problemen und sind daher für derzeitige industrielle Anwendungen nicht gut anwendbar.

Zum einen zielt der DTT-Ansatz darauf ab, die *Effektivität* vollständiger Tests zu verbessern, die für kritische Systeme unverzichtbar sind. Die Spezifikationsmodelle werden automatisch mit begrenztem zusätzlichen Entwurfsaufwand überprüft und modifiziert, um die Testbarkeit der physikalischen Implementierung zu verbessern, d.h. die Steuerbarkeit, die Beobachtbarkeit und die s.g. single-input-change testability (dt. Testbarkeit mittels einzelner Eingangsgrößenwechsel). Dieser Ansatz garantiert, dass das Verhalten der Implementierung während ihrer normalen Ausführung unverändert bleibt, d.h. wenn sie nicht mit einem Prüfstand verbunden ist.

Zum anderen zielt der PF-Ansatz darauf ab, die *Effizienz* von Tests für große Systeme zu verbessern, bei denen vollständige Tests kaum realistisch sind. Anlageneigenschaften werden manuell mit einfachen Vorlagen modelliert (was auch den Modellierungsaufwand begrenzt) und dann automatisch in die Testgenerierung einbezogen. Dadurch können der Eingangsraum eines zu testenden Systems und damit die Anzahl

der *aussagekräftigen* Testfälle deutlich reduziert und die Länge einer ausführbaren Testsequenz auch deutlich verkürzt werden. Es ist wichtig zu erwähnen, dass die erhaltene verkürzte Testsequenz eine vollständige Abdeckung des gesamten *Nennverhaltens* eines getesteten Systems garantiert.

Auf der Grundlage von Fallstudien übertreffen diese beiden Ansätze die derzeitigen Methoden und fördern das modellbasierte Testen für programmierbare Steuergeräte.

Contents

List of Figures

List of Tables

List of Acronyms

BIST	built-in-self-test
CCT	complete conformance testing
CTL	computational tree logic
DFT	design-for-test
DTT	design-to-test
DTT-MAT	design-to-test MATLAB tool box
FBD	function block diagram
FSM	finite state machine
HIL	hardware-in-the-loop
IC	integrated circuit
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
IL	instruction list
IM	intermediate model
IUT	implementation under test
LD	ladder diagram
LTL	linear temporal logic

MBT	model-based testing
MIC	multiple-input-change
MIL	software-in-the-loop
PF	plant feature
PIL	processor-in-the-loop
PLC	programmable logical controller
SCA	stabilized composed automaton
SFC	sequential function chart
SIC	single-input-change
SIL	model-in-the-loop
ST	structured text
SUT	system under test
SysML	systems modeling language
UML	unified modeling language
VLSI	very-large-scale integration

1 Introduction

1.1 Motivation

Nowadays, automation engineering is facing challenges in designing and testing. It involves knowledge and technology from multiple fields such as mechanical and electrical engineering as well as computer science. Besides, automation systems are often composed of multiple subsystems that are distributed and interact with each other.

Fig. 1 presents the classic V-model of system development. The left wing of the '*V*' represents the design phase. The first step is to collect requirements from users, which are usually informal, e.g., descriptions in the form of natural language. Then, engineers consolidate all the requirements and create formal specification models. During the design phase, the models are refined and detailed step by step, from the most abstract system level (on the top) throughout subsystem, architecture, component levels until the most detailed unit level (on the bottom).

Meanwhile, each time models are created or detailed, they are always checked against the models/requirements one level higher. This procedure is called verification, including formal and informal techniques. Formal verification is a type of popular verification techniques aiming at proving the correctness and consistency of formal models with respect to certain formal specifications or properties. Recent research and development of formal verfication techniques can be found in [3] [4] [5].

The right wing of the '*V*' represents the integration of implemented parts, i.e., from unit level (on the bottom) back to system level (on the top). Each time after the integration for one level is done, the obtained implementations will be validated whether they

conform to the initial specifications/requirements.

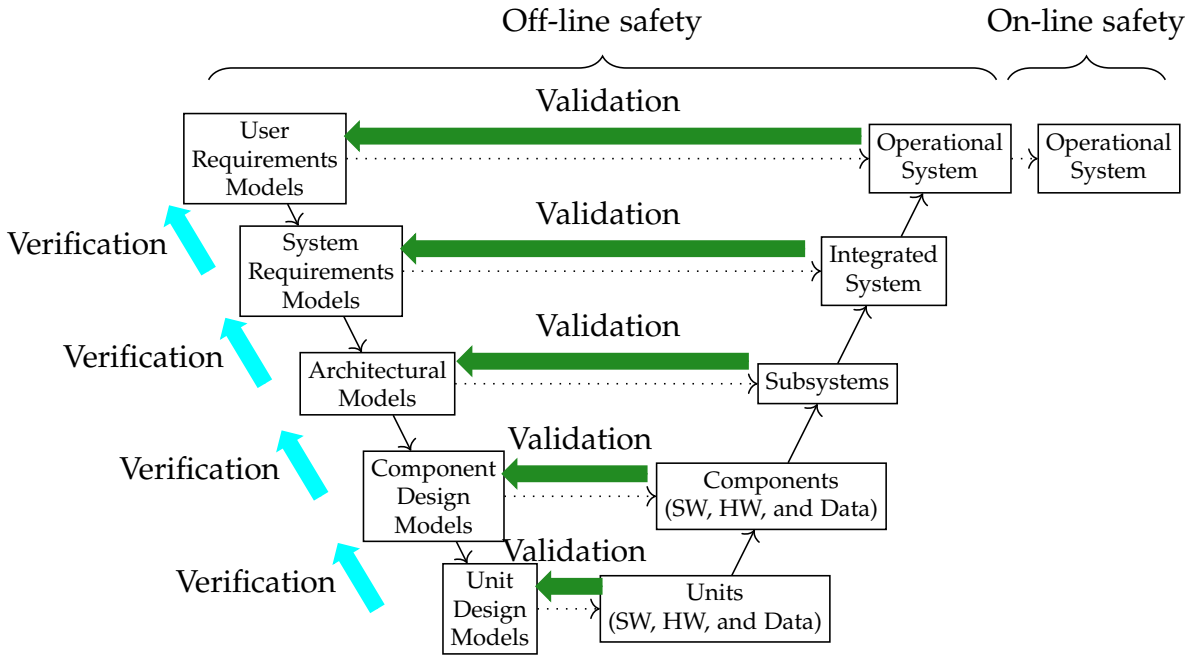


Figure 1: V-model in the system development: design, verification, validation and diagnosis

Verification and validation are both important off-line safety measures to ensure the correctness of a product or process [6]. They are not the same thing, although they are often confused. Succinctly, the difference can be understood as follows [7]:

- Verification: "Are we building the product right?"
- Validation: "Are we building the right product?"

After the steps of the both wings have been finished, the system will be put into use and enters the so-called 'on-line mode' (the most right block in Fig. 1). In this phase, some other measures such as diagnosis, reconfiguration, maintenance and repair are used to ensure the on-line safety of a system [8].

1.2 Problem statement of testing

Testing, as "an 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"([9], page 368), is considered as an important validation means.

Automation systems are comprised of hardware and software components, which interact with each other through various communication protocols, and often also interact with the physical environment with peripherals such as sensors and actuators.

On one hand, in many applications especially critical fields such as railway, power production and medical devices, the functionality of automation systems are becoming more and more complicated [10]. Verification techniques can verify the models, but they cannot guarantee the correctness of the final implemented system which is also strongly influenced by the hardware and environmental factors. For example, a single-input-change (SIC) issue that can lead to undetected behavioral difference between initial design and final implementation is resulted from hardware execution characteristics [11], which can not be discovered by verification techniques. To cope with these issues, testing is strongly recommended and even compulsorily required by many industrial standards such as IEC 61508 [12], IEC 61511 [13] and ISO 26262 [14] as a validation technique on top of verification methods.

On the other hand, in many fields such as manufacturing, the life cycles of industrial products/processes are also constantly shortening. As a consequence, the introduction of new processes and modification of existing designs are carried out more frequently, which causes frequent changes in system specifications [15]. This also requires efficient system engineering approaches including not only useful designing tools but also powerful testing techniques.

Referring to Fig. 1, usually, testing is not considered until the design and implementation steps are finished. Besides, research work has shown that in current practice, most testing activities are still conducted manually while design tasks have been supported by abundant tools to be done automatically [16] [17].

Manual testing methods are mostly straightforward, expert-based, and have been proven useful in practice decades-long [18], [19], etc. Nevertheless, their disadvantages are obvious: individually customized, time-consuming, and error-prone. The shortcomings have become big obstacles which are hardly bearable for the testing demands of modern automation systems, especially for safety-critical and large scale systems.

Among all applications, safety-critical automation systems have in particular following characteristics: intensive interaction with sensors and actuators; strict requirements on real-time performance; and high demands on dependability and safety [20]. Therefore, safety-critical systems, or safety-critical parts of a system, need to be tested completely. To be specific, the testings should cover all possible behavior of a system/part under all situations it could have.

As for large scale systems (which are usually not completely critical), the largest testing challenge is to overcome the so-called 'state space explosion' issue. Briefly, when the number of inputs of a system grows linearly, the total state space in test generation grows exponentially, and the obtained test sequence grows also exponentially. Consequently, complete testing is neither convenient to generate nor realistic to execute for large scale systems.

In this thesis, the tested targets are programmable controllers, which play a key role in many automation systems such as manufacturing and power plant. Compared to general computers, programmable controllers are dedicated to a limited set of specific tasks. They receive a variety of input signals from sensors, internal buses and external networks, make decisions according to the implemented specifications from users, and send commands to actuators. Owing to user demands of complex functions, high individualization, and frequent modifications, the development and application of programmable controllers are also becoming highly complex. This fact raises challenges not only to design tasks but also to verification and validation methods [21].

The motivation of this thesis is to present two innovative model-based approaches to cope with the above mentioned issues in testing programmable controllers of automation systems. The two approaches aim at following advantages in test generation: automatic, effective and efficient. Here, 'automatic' means demanding as less 'expert knowledge' and manual work as possible; 'effective' means that the test should always

give the correct verdict result; 'efficient' means that the test generation and execution are simple and fast, and should cost as less testing overhead as possible.

1.3 Contributions

This thesis is based on the research work during my PhD, most of which has been published in international conferences and journals (the list of publications is to be found in the appendix of this thesis).

Fig. 2 presents the frameworks of three test generation approaches[1]: complete conformance testing (CCT), design-to-test (DTT), and test generation with plant features (PFs). CCT is a classic and well-developed approach [22], and is used as a basis and reference of the two other approaches, DTT and PF, which are the main contributions of this thesis.

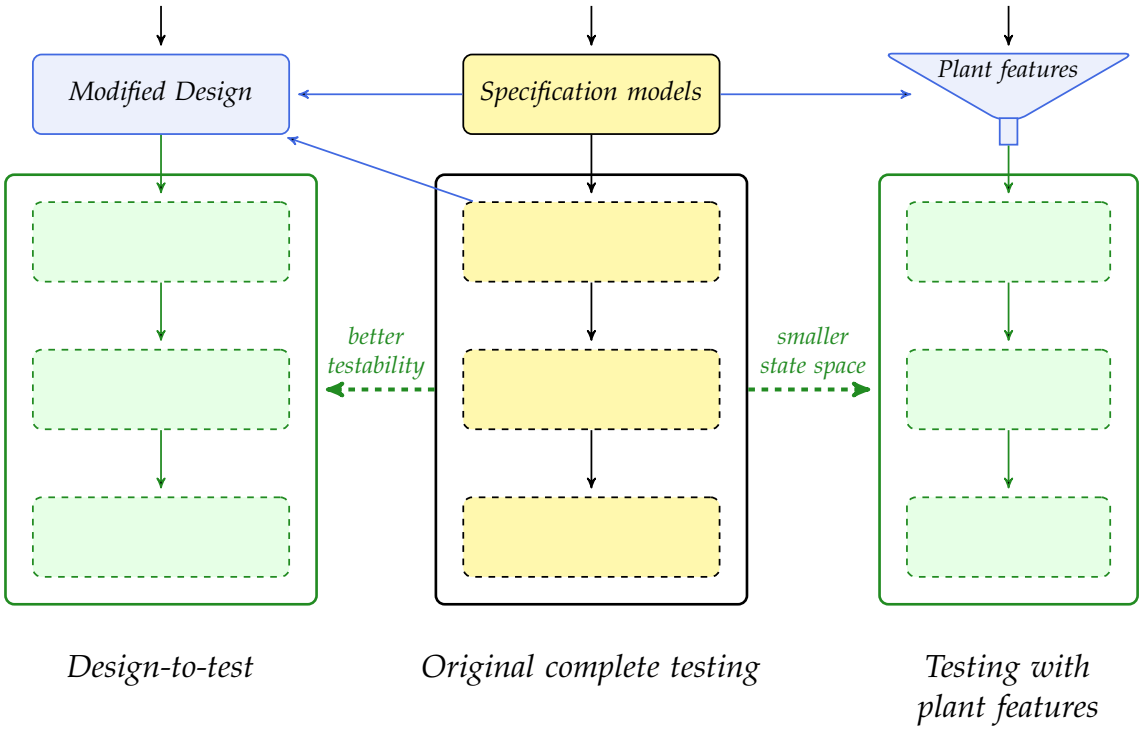


Figure 2: Simplified frameworks of complete testing, design-to-test, and test generation with plant features

[1]Details of the three approaches are given later in the thesis.

In brief, the DTT approach aims at improving the testability of a system under test, while the PF approach aims at reaching a reduction of state space in test generation and a shortened test sequence.

The essential contributions are briefly listed as follows:

- **DTT approach**
  - Identification of three testing issues of programmable controllers in practice: controllability, observability, single-input-change (SIC) testability
  - Proposal of DTT solutions to each testing issue: C-guard method, O-action method, T-guard method
  - Design of algorithms of the DTT approach
  - Implementation of a software toolbox
  - Application on industrial case studies
- **PF approach**
  - Informal and formal descriptions of plant features
  - Design of algorithms of the PF approach
  - Implementation of the PF approach and integration into existing test generation toolbox
  - Application on industrial case studies

1.4 Outline

The thesis is structured as follows:

Chapter 2 introduces the background knowledge in the field of programmable controllers, as well as some basic concepts in quality assurance measures.

Chapter 3 displays the state of the art, e.g., current development of verification and validation techniques for automations systems and programmable controllers .

The mathematical notations of Moore machines extended with Boolean signals, which is the formal modeling language used in this thesis, as well as the overall framework of testing objective, test execution unit, and test generation process used in this thesis, are provided in chapter 4.

Chapter 5 presents the DTT approach for black-box conformance testing of programmable controllers. Given an automation system, the testing objective is to check whether an implemented controller conforms to its expected behavior with regard to the specification models. The proposed design-to-test approach analyzes the specification models and automatically modifies them by inserting additional inputs and outputs with limited design and testing overhead, in order to improve the testability of their physical implementations. Two case studies are presented to illustrate and evaluate this approach.

Chapter 6 presents a model-based test generation approach for programmable controllers that aims at reducing the length of a test sequence by applying PFs. The proposed approach does not require detailed or full knowledge of the plant behavior of a system under test, but can achieve 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. Similar to chapter 5, this approach has been applied on two case studies.

Finally, chapter 7 gives the conclusions, discussions and perspectives of future work.

2 Background knowledge

2.1 Programmable controller

In this thesis, programmable controllers are the hardware where the executable programs are installed, executed and also tested. Programmable controllers are widely used in different kinds of applications and industries such as manufacturing, robotics, process control, power plant, and wastewater treatment.

A programmable controller, also known as programmable logical controller (PLC), is "a special form of microprocessor-based controller that uses a programmable memory to store instructions and to implement functions such as logic, sequencing, timing, counting and arithmetic in order to control machines and processes" [23].

Basically, a programmable controller is a specific computer which is designated for control tasks in industrial environment. It is required to run 24/7 and resist harsh physical and electrical factors such as vibration, temperature, humidity, noise and electromagnetic interference.

2.1.1 Programing languages

Usually, a programmable controller has been pre-programmed by manufacturers so that the control code can be programmed with rather simple and intuitive languages. IEC 61131 is an International Electrotechnical Commission (IEC) standard for programmable controller and covers aspects such as general information, equipment requirements and tests, user guidelines, functional safety, etc. In particular, IEC

61131-3 [24], the third part of this standard, defines the software architecture and programming languages of control program. Five programming languages have been officially approved: ladder diagram (LD), function block diagram (FBD), instruction list (IL), structured text (ST), and sequential function chart (SFC). LD and FBD are graphical languages, and IL and ST are textual language, while SFC can be either graphical or textual.

In this thesis, we focus on the generation of ST code. It is a high level language that syntactically resembles Pascal, and supports some complex statements such as conditional execution, iteration loops and functions [24]. This makes it powerful to handle large scale applications. A disadvantage might be that, many experienced engineers and technicians are more used to the graphical languages. Compared to LD and FBD, ST appears not very intuitive and straightforward for manual trouble-shooting.

Considering both the advantages and disadvantages, the main tendency in research and practice is to automatically generate the code with model-based methods, so that engineers do not need to write the complicated code manually. But this also requires reliable automatic testings to replace fully or partly manual validations.

2.1.2 Model-based development of applications

Model-based development methods are getting increasingly accepted for the design tasks of automation systems. They permit to achieve a high degree of automation and good re-usability.

Following the V-model introduced in chapter 1, firstly, formal specification models are created for the programmable controllers according to users' requirements which are usually informal.

To model the specifications, different formalisms/languages have been studied and applied. Following are some representative examples, [25] proposed PLC-automata, a new class of automata, which are tailored to deal with real-time properties of programmable logic controllers. The use of Petri nets was introduced in [26], while [27]

extended it to be signal-interpreted Petri nets. [28] presented an agile approach using unified modeling language (UML) to automatically generate IEC 61131-3 code. [29] extended the use of UML to systems modeling language (SysML) in order to achieve better compliance with the IEC 61131-3 items and rules. Recently, an object-oriented PLC programming approach adapted from UML and SysML was proposed in [30], and a PLC design approach based on Petri nets was proposed in [31].

In this thesis, we use Moore machines extended with Boolean signals, a kind of automaton which is and easy to understand and powerful to model a system. The formalism is presented in detail in chapter 4.

2.1.3 Cyclic execution

A significant feature of programmable controllers is the *cyclic execution mode*. When a programmable controller is turned on, it runs continuously its control program and updates its input and output signals. Each such loop is called a *cycle*, which includes mainly three steps:

- read values of input signals

- execute all instructions of the implemented programs

- update values of output signals.

It is worth noting that most programmable controllers also have a self-test or diagnostic step in a cycle, which is nevertheless not relevant to the topic of this thesis, and is therefore not considered.

The cycle time of a programmable controller varies from 1ms to 100ms in practice, in most of the cases around 10ms. The cyclic execution mode enables programmable controllers to fulfill the 'hard' real-time requirement, meaning that in the worst case it can respond to the input signals in the time of two cycles [23]. This is a crucial advantage of programmable controllers for industrial needs.

However, the cyclic execution mode can also bring issues in testing. For example, a

single-input-change (SIC) issue is presented in detail in chapter 5.

2.2 Error, fault, failure

In software engineering, an *error* is defined as a human mistake that results in an erroneous program. A *fault* is a manifestation of an error, also known as defect or bug. A *failure* is a deviation between the observed behavior and the required behavior of a software system [32].

An error, which is introduced by a person, always leads to a fault. A failure is caused by a fault or several faults in the software. A fault might lead to a failure, but not necessarily. In other words, a software system can contain faults but still never fail.

Different countermeasures are used in practice to cope with these issues. In particular, development engineers can make fewer errors through specific training and establishment of rules. Faults are mainly diminished through inspection, review and static analysis of the program, which are classified as static testing techniques in [33]. Failures are mostly detected by tests, which is usually referred to as dynamic testing [33].

2.3 Verification & validation

Verification and validation (V&V) are important quality assurance measures that take place after a programmable controller has been implemented with its executable code and before it can finally be put into use. By applying verification and validation, a system is determined whether and how good it meets the specifications and fulfills its intended purpose.

Formally, verification is defined as “the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase”([9], page 400).

Validation is defined as “the process of providing evidence that the software and its

associated products satisfy system requirements allocated to software at the end of each life cycle activity, solve the right problem, and satisfy intended use and user needs"([9], page 397).

As briefly mentioned in chapter 1, verification deals with the question "Are we building the product right", while validation handles the question "Are we building the right product". Apart from that, Tab. 1 provides some more comparisons between the two measures.

Table 1: Verification and validation

Verification	**Validation**
checks the system against specifications	checks the system against user requirements
mostly static mechanisms	dynamic mechanisms
usually does not execute the code	always executes the code
low level exercise	high level exercise
considers specifications and system designs as target	regards end-user product as target
always done by development team	usually carried out by extra testing team
usually done before validation	usually done after verification
methods: informal and formal verification	methods: white-box testing, gray-box testing, black-box testing

Several representative verification and validation techniques are presented in the next chapter.

3 State of the art

3.1 Introduction

This chapter presents recent research work in the field of this thesis, i.e., model-based testing of programmable controllers, and adjacent fields such as informal/formal verification techniques, and validation methods through testing.

In each of the following sections, a type of techniques or a family of related techniques are reviewed and discussed with regard to their characteristics in general and applicability for programmable controllers. In particular, some techniques have strong relations to the two approaches presented in this thesis. Therefore, their similarities and differences are highlighted in the discussions.

3.2 Informal methods

Informal methods of verification and validation are frequently used in modeling and simulation during the design of a system. These methods are more qualitative than quantitative, and rely more on experiences and opinions of experts than numerical results, that is why they are called *informal*.

Typical informal methods include inspection, review, code walk-through, etc. These methods are all manual examinations done by qualified personnel to check software products with regard to requirements, and to detect and identify software anomalies errors and deviations from standards and specifications [9]. For example, with inspection techniques, the behavior of an abstract conceptual model and a concrete runnable

model can be compared.

In the field of programmable controllers, informal methods such as checklist-based review and inspection are also widely used, as a complement to formal methods. Tools have been developed to support these techniques, i.e., to make them more automatic and systematic [34].

The largest advantage is that users can verify a model more quickly because they do not need a proper model, which may take some time to acquire and sometimes does not exist at all.

The shortcomings are also obvious, expert knowledge and experience might be working but can not be formally approved; and informal methods are hard to scale on different individual projects. For safety-critical systems, it is compulsorily required or strongly recommended to use systematic and formal methods such as formal verification and validation through testing to evaluate and ensure the safety levels.

3.3 Formal verification

When formal languages and mathematical techniques are used, a verification process is called formal verification. Some typical formal verification techniques are introduced in following parts.

3.3.1 Theorem proving

Theorem proving is one of the key approaches of formal verification. Firstly, a theorem prover is constructed based on a mathematical statement. A mathematical statement can only be verified as true when a proof can be deduced from the logic of the prover. In such a case, the statement is called a theorem.

Theorem proving has been studied for long. For example, a machine program for theorem-proving was proposed in [35] in 1962; the complexity of theorem-proving

procedures has been discussed in [36] in 1971; now, different theorem prover tools such as Coq [37] and SPASS [38] have been developed and provided to construct proofs interactively / automatically.

In the field of programmable controllers, theorem proving has also been applied. In [39], a formalization of TON-timers of programmable logical controller (PLC) programs has been proposed to apply the theorem prover Coq. [40] proposed a theorem-proving method for ladder diagram (LD) program by defining a formal framework with a specific algebra. Using the solver they developed, the verification can be done automatically.

A very practical reason for applying theorem proving, as introduced in [41], is that a substantial degree of control in the process of derivation of complex theorems can be achieved by most theorem provers. In some applications where a very expressive logic is desired to represent complex properties, theorem proving might be the only technology that can be resorted to. In other words, a great advantage of theorem proving is the avoidance of the state-space explosion issue [21].

As pointed out in [42] in 1999, the biggest issue that hindered the wide use of theorem proving was insufficient degree of automation compared to other techniques such as model checking and testing. Now, fortunately, various attempts of automatizing this approach has been done such as SPASS [38].

3.3.2 Model checking

Model checking is an automated approach that exhaustively explores the whole state space of a formal model of a system to prove it meets some certain properties, or otherwise to find a counter example.

The system is usually modeled as automata containing the initial states, possible other states, and the transitions between all the pairs of states. The model is an abstraction of the system by omitting irrelevant details with regard to the properties to be verified. Typical properties are temporal logic or other safety requirements such as the absence of deadlocks. The most common formal descriptions of temporal logic

are linear temporal logic (LTL) and computational tree logic (CTL). Complex temporal constraints can be formulated with LTL and CTL conveniently.

In practice, model checking has been well accepted and widely used in software applications where formal models are close to the final products such as verifying the implementation of controllers and evaluating communication protocols. Many tools have been developed in academia and industry, such as Spin [43], UPPAAL [44], NuSMV [45] and its extended version nuXmv [46], etc.

In the field of programmable controllers, model checking has also been widely applied by transforming the control programs into an adequate and behavioral equivalent formal model. For example, a systematic process of verifying function block diagram (FBD) control programs by NuSMV model checker has been proposed in [47]. It is realized by transforming the graphical FBD programs into formal textual forms called TextFBD and tFBD. In [5], a formal intermediate model (IM) has been built to transform control programs written in the form of structured text (ST), sequential function chart (SFC) to nuXmv model checker.

As pointed out in [48], nowadays model checking techniques and tools have become competitive for finding bugs in software products, and even outperformed the bug-findings capabilities of state-of-the-art testing tools when performing experiments on a benchmark set. However, for industrial automation systems, bug-finding is not the only task of quality assurance, other aspects such as the conformance relation between specification and implementation are often also of interest. Therefore, a combination of model checking with other techniques such as testing would be a good solution.

3.3.3 Static analysis

Static analysis encompasses a bunch of techniques that enable automatic analysis of software products, and detection of errors and error-prone conditions without executing the software programs [49].

Static analysis techniques work on the source code or intermediate representations of a program. Therefore, they are used intensively in compiler optimization [50]. However,

nowadays static analysis can also be employed independently from compilers. It has evolved to an important measure of software quality assessment and improvement technology, which can provide means to reveal bad code smells, violations of programming conventions and guidelines, and potential defects [51]. Also, static analysis tools can help reduce the effort of other verification processes such as code review by catching common mistakes prior to it [52]. The tools employed for static analysis are rich and manifold, such as SLAM [53] for C program verification in Microsoft, FindBugs [54] for finding bugs in Google, and Pixy for detecting web application vulnerabilities [55].

In the field of programmable controllers, the applications of static analysis are also popular [51], for example, the tool from 3S which became part of the CoDeSys Professional Developer Edition[1], the tool PLC checker from Itris[2], and the tool logi.LINT from Logicals[3]. Recently, a live static code analysis architecture has been proposed in [56], which aims at bridging the gap among different static analysis tools in different development processes by enabling the use of static analysis directly in a common development architecture. Instant feedback are given when a user is still editing the PLC software.

As concluded in [49], the main advantages of static analysis is that it can reduce the amount of testing workload in practice, and it can detect errors that cannot be found by testing such as improper resource management, illegal operations, dead or incomplete code, non-termination, uncaught exceptions, and race conditions.

However, static analysis only deals with static behavior of a system under test (SUT), and therefore cannot guarantee the absence of runtime errors. In practice, other techniques such as runtime verfication and testing are good supplement to static analysis.

[1]https://www.codesys.com/ (last visited on September 10th, 2018)
[2]http://www.itris-automation.com/ (last visited on September 10th, 2018)
[3]http://www.logicals.com/ (last visited on September 10th, 2018)

3.3.4 Runtime verification

Runtime verification refers to the techniques that allow checking whether a *run* of a system under scrutiny satisfies or violates a given correctness property [57].

A *run* of a system is understood as a possibly infinite sequence of the system's states, which are formed by current variable assignments, or as the sequence of (input/output) actions a system is emitting or performing. The checking is performed by using a monitor, which is a device that reads a finite trace and yields a certain verdict, which gives the value *True* or *False* [58].

Following are some tools supporting runtime verification and some applications in practice. JPaX is a Java PathExplorer runtime verification tool, which can monitor the execution of a Java program and check if it conforms with a set of temporal logic properties provided by users [59]. It has been essentially developed and used in the NASA Research Center, and it has been applied to many programs produced for rovers, spacecrafts and similar devices. Mop, short for 'Monitoring-Oriented Programming', is a formal framework for software development and analysis with runtime verification [60]. It automatically generates monitors from specified properties by users and then integrates them together with the user-defined code into the original system.

In the field of programmable controllers, runtime verification has recently also been applied. In [61], for advanced programmable controllers coupled with embedded hypervisors, online cyber-physical verification solutions have been directly integrated into the program scan cycle as well as online intrusion detection systems within the embedded hypervisor. With this approach, advanced security and verification solutions are allowed to be directly enforced within the programmable logic controller program scan cycle.

Compared to static verification techniques such as the above discussed techniques, the biggest difference, which can also be seen as an advantage, is that runtime verification does analysis and checking dynamically. Another distinguishing feature is that runtime verification technique does not require a model of the system, but only deals with observed executions of a system.

It is worth mentioning that runtime verification techniques deals only with detection of violation/satisfaction of properties from observations, but do not influence or change the program execution. For example, the reparation of the program with regard to a detected violation is based on the verification results, but the repairing itself is not part of the job of runtime verification.

Since runtime verification is also executed on running systems, it is therefore considered as a form of *passive testing*. The difference between the two techniques lies on the adjective 'passive', saying that runtime verification does not require creating test cases, but only needs to observe the behavior of a running system, while testing in general always actively needs test cases which is the run of a system. As a consequence, coverage metrics of code can be reached by testing but not by runtime verifications.

3.3.5 Brief discussions of formal verification techniques

The biggest shortcomings of informal verification techniques turn out to be the advantages of formal methods: they are highly convincing with formal proof, are capable to handle complicated systems systematically, and many of their processes can be automated.

Most verification techniques (one exceptional example is runtime verification) are executed on models, before a system or a component is really implemented. In one aspect, this brings an advantage that these techniques can be applied independently from influences of implementation, e.g., software environments and hardware platforms. In another aspect, these techniques can not verify the properties of an implemented system, e.g., real-time performance of a controller, the interaction between software and hardware. To overcome this, validation through testing is a good supplement, which are mostly executed on the implementation.

As a conclusion, for safety-critical systems, testing is strongly recommended or mandatorily required to be a supplement of verification techniques.

3.4 Validation through testing

Testing is defined as "an 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."([9], page 377).

For software systems, there are mainly two types of testing: functional and performance. The former includes black-box and white-box testing, while the latter concerns software availability, reliability, survivability, flexibility, durability, security, re-usability, and maintainability [20].

The approaches presented in this thesis, i.e., the design-to-test (DTT) approach and the plant feature (PF) approach, are categorized into the field of validation through testing. More specifically, the approaches focus on the aspect of functional testing.

3.4.1 Spontaneous, manual testing vs. systematic, automatic testing

Testing activities in a broad sense have taken place early in the development phase. For example, developers always execute their programs after they have coded a new piece or modified some parts. If a program does not run or the results are obviously not correct, developers will then do a debugging and search the faults. When the program executes and '*seems to produce reasonable*', the debugging is usually finished. In fact, what developers have done is already a test, which is spontaneous and manual. This kind of testing starts always when a failure is detected, is usually executed with random test cases, and ends if the developers think they have tested enough.

On the other hand, to ensure the safety and reliability of a complex system in critical applications, testing activities are more and more required to be systematic, and also recommended being automatic.

Compared to spontaneous tests, systematic tests are always well planned. Test cases are generated according to specifications, and consist of selected inputs and expected outputs. The test results are well documented. Tests will only be finished after previously defined test goals are reached, for example the SUT is fully tested or a

preset test coverage is reached.

The advantages are obvious:

- Test executions are traceable and reproducible, failed tests can easily be repeated.
- The efforts and benefits of a test is predictable
- It is economically efficient to have all the test cases, execution process and results well documented.
- Risk and liability can be reduced, since critical failures of a system can be systematically better avoided.

Spontaneous testings are more suitable for small and non-critical applications or early stage of development phase, since they are mostly done manually and based on expert knowledge and experience.

The main cost of systematic testing is the design overhead of the testing, while the test generation and execution can be done more or less automatically now. These characteristics make this type of testing highly scalable and therefore suitable for large scale and complex systems. For safety critical systems, systematic and automatic testing is strongly recommended and required, while spontaneous and manual testing plays a complementary roll.

The two approaches presented in this thesis focus more on systematic and automatic testing.

3.4.2 White-box, black-box, gray-box testing

White-box testing is defined as a type of testing that takes into account the internal mechanism of a system or component ([9], Page 349). It is good at revealing errors in hidden code, providing traceability of tests, setting up coverage metrics for designing of test cases, but require a high level of knowledge of the system, and might not detect unimplemented specifications or missing requirements [62].

Black-box testing is referred to as a type of testing that ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution condition ([9], Page 154). It has no requirement of programming knowledge and is efficient at handling large code segment. Besides, tester perception is very simple. The main disadvantages include low/limited coverage, and difficulty in designing the test cases [62].

The so-called 'gray-box testing' is a combination of white-box and black-box testing, in which the internal structure is partially known [63]. The advantages and disadvantages are also combinations and compromises between black-box and white-box testing.

White-box testing is suitable for early testing in development when a tester has full knowledge of source code. Black-box testing is widely applied in late phases of testing, where the internal structures are not easily accessible. Gray-box testing is well suited for intermediate phases or some special situations such as web applications, where the systems are often of distributed structures, the interface of which are clearly defined but the source code or binaries are absent [64].

In the testing of programmable controllers, black-box testing is in general more suitable since the internal structure of controllers are not easily accessible. It checks whether an implemented controller, seen as a black-box with inputs/outputs, behaves correctly with respect to its specification models.

However, with some modifications into the specification models, the benefits of gray-box testing have also been achieved with the DTT approach proposed in this thesis, which is presented in detail in chapter 5.

3.4.3 Open-loop / closed-loop testing

When testing a controller in a real system, there are two different architectures: open-loop and closed-loop testing ([65], 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 [66], the applicable test cases are created according to the functional requirements

and data flow of 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, [67] presented an automated procedure for constructing plant models for closed-loop simulation and testing of programmable controllers.

In this thesis, plant features (models) are involved in the test generation with the PF approach, but not in the test execution. Therefore, testing in this thesis should be categorized into the group of open-loop testing. More details are given in chapter 6.

3.5 Model-based testing

The term model-based testing (MBT) refers to a family of testing techniques that build test cases based on the behavior models extracted from an implementation under test (IUT) and its environment. Test cases for the IUT are constituted by traces of inputs and expected outputs generated from specification models [68].

Much research interest has been attracted to MBT. For example, [69] sorted many publications on model-based testing and provided a taxonomy that covers the key aspects of MBT approaches, so that different approaches can be classified and compared. [70] recommended the use of state-based languages (e.g., Moore machine used in this thesis) in modeling IUT behavior due to numerous theoretical results and suggestions by many programms. The survey revealed a large gap between theoretical research and practical applications. Large part of theoretical research has not been/cannot easily be transformed into practical tools.

The two approaches presented in this thesis also belong to the family of model-based approaches.

3.5.1 X-in-the-loop testing

Nowadays, automation systems are getting increasingly complex, and therefore the testing of embedded controllers is usually not a one-click action. Instead, tests are executed in a series of different stages. Four popular X-in-the-loop testings, i.e., model-in-the-loop (SIL), software-in-the-loop (MIL), processor-in-the-loop (PIL), and hardware-in-the-loop (HIL), are presented in the following part.

These X-in-the-loop testings belong to closed-loop testing where the software product is verified and validated in different target platforms. They have raised great interest in the testing practice of safety critical industrial software [71], applications on embedded controllers [72], and especially in the automotive industry [73]. Some key features of these X-in-the-loop testings are listed as follows.

Model-in-the-loop testing

In MIL testing, the controller and plant are both models without any physical hardware components. Testing is executed on simulations of abstract functional models. The computations are usually performed on a host computer, with floating-point arithmetic, and independent from final target platform [72].

It is suitable at early stages of the development cycle, from which developers can quickly get important feedback with regard to fulfillment of functional requirements.

Software-in-the-loop testing

In SIL testing, executable object code is tested instead of models. Nevertheless, the plant is still simulated models without hardware, and the execution of SIL testing is still done on a host computer [72].

The results of SIL testing is supposed to be comparable to the results obtained from MIL testing. However, they can differ. The main reason is that the executable code contains more information and restrictions such as mapping between different parts

and functions. Besides, in SIL testing the code computations are done with fixed-point arithmetic, which can also run on an embedded controller [71].

The SIL testing results reflect somewhat the actual code executions that would happen later on the final target platform.

Processor-in-the-loop testing

In PIL testing, executable object code is tested on not anymore on a host computer, but on an experimental hardware which contains the same processor as the final target system. A typical example of such experimental hardware is an evaluation board [71].

The aim is to verify the code behavior and measure code efficiency such as profiling and memory usage on the target processor.

Hardware-in-the-loop testing

In HIL testing, the software code runs on the final controller. The plant is still a simulation, but usually not running on a normal host computer but on a dedicated hardware designed for HIL testing, so that real-time communication between the controller and plant can be tested as in the real application. Additionally, in HIL testing, not only the control code but also the input output interfaces can be tested [73].

In this thesis, testing is executed on real programmable controllers and is therefore categorized into the field of HIL testing.

3.5.2 Conformance testing

Conformance testing is a kind of model-based testing that checks whether the behavior of an implementation conforms to the behavior of its specification models [22]. Model-based conformance testing methods have been investigated since long.

The term complete conformance testing (CCT) refers to a classic model-based technique that automatically generates test cases from specification models, which considers all possible combinations of input signals from all states. Therefore, CCT covers the whole behavior of a SUT and is highly advantageous for safety critical systems. The main limitation is that the number of test cases and subsequently the length of a test sequence grow exponentially with the number of inputs, which severely restricts its application to large-scale systems.

Thus, there has been an urgent demand to have advanced testing techniques that are capable to handle large scale systems.

Most recent research work aims at reaching high test coverage with a relatively small set of test cases. For example, [66] generated test cases based on the element identifier and function block-tree traversal; [74] used coverage metrics to implement a symbolic execution engine; [75] proposed an assessment approach to support increasing system test coverage through effectively identifying untested code and untested behavior of an SUT.

However, research results have indicated that testing with coverage criteria satisfaction alone are not always powerful; they can be poor at effectively finding faults in some applications, sometimes even worse that random testing [76].

On the other hand, system behavior of an SUT is seldom considered in these coverage-oriented methods. Thus, critical faults might be missing in a testing with high but not full coverage.

The two approaches proposed in this thesis are intended to support model-based conformance testing techniques, to be more exact, the DTT approach permits to providing better specification models for testing purpose, and the PF approach aims at shortening the length of test sequence by using plant features.

3.6 Test-driven development

Test-driven development (TDD) is a technique that considers testing at early phase in the development process.

In a typical TDD process [77], an automated functional test is implemented before any program code is written; then, quick designs or changes are made to the program in order to pass the test; once succeed, the new code will be re-factored and improved to fulfill non-functional requirements such as coding guidelines.

However, despite the promising prospects, TDD is far less applied in industry than expected [78]. Several research projects have been undertaken to investigate the reasons and obstacles that restrict the wide usage of TDD. In [78], a summary of published results from research projects and practical experiments concluded that although the TDD approach provides a better code coverage, it cannot be proven to be generally superior to other traditional approaches in terms of development time, change and maintenance cost. In [79], seven essential factors that limit the industrial adoption of TDD have been identified: *increased development time, insufficient TDD experience/knowledge, lack of upfront design, domain and tool specific issues, lack of developer skill in writing test cases, insufficient adherence to TDD protocol, and legacy code*.

Meanwhile, many attempts have been made to adapt TDD on specific industrial applications. For instance, in [80], TDD was adapted to C programming by applying a dual-targeting approach. In [81], a TDD process introduced adaption of unified modeling language (UML) models to enable effective test case derivation of automation systems.

Compared to TDD, the DTT approach presented in this thesis does not require expertise, and encounters no issue dealing with the code, since the DTT approach modifies specification on the model level. What's more, when the DTT approach is combined with automated testing tools such as Teloco [22], the final test code is generated fully automatically, so that developer skills in writing test cases are not needed.

3.7 Concept of design-to-test

Design-to-test (also called design-for-test (DFT) in some research work) was initially conceived for the testing of integrated circuits (ICs) [82]. It aimed at achieving a high fault coverage by inserting test points into circuits at the cost of increasing the circuit area and the number of pins of the board.

In this domain, DFT has been researched and applied widely. Most research work aimed at better performance in fault coverage and testing overhead. In [83], a low power built-in-self-test (BIST) test pattern generator that provides test vectors which can lower the energy consumption during test operation has been proposed. [84] proposed a test generator that saves test overhead to fulfill fault coverage requirements in very-large-scale integration (VLSI) testing. In [85], a BIST scheme that can lower the test time and silicon area cost in the testing of three-dimensional ICs has been presented. [86] proposed a test solution for monolithic three-dimensional ICs based on dedicated test layers, which are inserted between functional layers. The authors also showed that this technique is more cost-efficient than the Institute of Electrical and Electronics Engineers (IEEE) standard solution, i.e., it provides test schedules with minimum test time under power consumption and probe pad constraints.

Recent work has also extended the DFT techniques to other applications. [87] presented the problem of conventional DFT techniques in security-critical applications that these ICs can be hacked through the test mode. The authors also proposed their countermeasures which can be incorporated into their DFT framework and provide defense against those potential attacks. In [88], a new test strategy has been proposed to enhance the application of DFT on on-chip networks which require high reliability. With the proposed strategy, the impact of test procedures on the system performance can be minimized, and therefore the test frequency can be increased.

On the one hand, the benefits of these DFT techniques such as higher fault coverage, lower testing overhead and higher reliability are also of interest to the DTT approach presented in this thesis. On the other hand, the DTT approach supports model-based conformance testing between specification and implementation of a programmable controller; while the DFT methods used for ICs mainly support designing and manufacturing tests, which validate whether a hardware contains no designing or

manufacturing defects that affect its correct functioning.

3.8 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 [89] [90] [91].

Most research work uses plant models for verification purpose such as model checking [92] [93], and simulation [94] [95]. However, as pointed out in [96], simulation-based verification methods may always encounter two issues: real-world errors are not discovered in the 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 chapter 3.

To cope with the first issue, one popular research direction is to build better simulation interface and environment that are more close to the real world [95] [96], another direction is to develop better plant modeling methods, i.e., automatic methods, to maximally avoid human errors in the construction of models, improve modeling efficiency, and enhance the overall applicability of plant models in verification and validation [67] [4].

On the other hand, this shortage of verification can also be overcome by testing. The idea of having plant models in testing has also been considered and investigated recently. [97] created an automated test case generation approach for industrial automation applications where specification and plant models are specified by UML state chart diagrams. However, the generation criteria is still about reaching high coverage rather than analytically considering system behavior, which is the goal the proposed approach aiming to reach.

The concern of the second issue has also been considered in this thesis, i.e. with the PF approach presented in chapter 6. 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 PF approach guarantees full coverage of nominal (plus optional faulty)

system behavior, and maximally removes test cases that are not relevant.

4 Testing framework

4.1 Mathematical notation

4.1.1 Specification model

Communicating Moore Machine extended with Boolean signals

In this thesis, the specification of a system is modeled as a set of communicating Moore finite state machines (FSMs), which can communicate with each other. Boolean signals are used as inputs and outputs. It has been proven that many other modeling languages can be automatically or easily transformed into Moore machines, e.g., Mealy machine [98] and Petri net [99], IEC61131 [42], and GRAFCET [22].

Signals and events are two types of inputs that can be accepted by programmable controllers. They can be converted into each other easily. For example, a rising edge, as a typical event, is equivalent to a value change from '0' to '1' of a signal. Also, contrary to many event-based models, Moore machine extended with Boolean signals does not restrict only one change of signal values at once.

A communicating Moore machine extended with Boolean signals is defined by an 8-tuple
$(L, l_{init}, I, C, O, G_\delta, \delta, \lambda)$[1], where:

- L is a finite set of locations.

[1]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 models

- 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[2] built up by inputs and internal variables.
- $\delta : L \times G_\delta \rightarrow L$ 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'.
- $\lambda : L \rightarrow 2^O$ 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 their graphical form in this thesis. A simple example is given in Fig. 3. A location l is drawn as a circle or a rounded rectangle. It can either have an externally observable output[3], 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., $\neg a \wedge b$ 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.

[2]Boolean operators used in this paper: $\wedge$: AND; $\vee$: OR; $\neg$: Negation.
[3]For readability reasons, only active outputs are presented, i.e., in l_3, o_3 implicitly means $o_3 \wedge \neg o_4$.

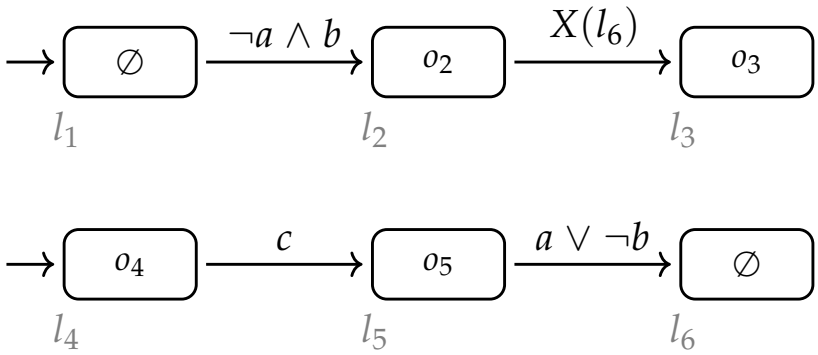


Figure 3: A simple Moore machine model example with Boolean signals

Stabilized Composed Automaton

The first step of the design-to-test (DTT) approach is to compose all individual Moore machine models in parallel with regard to signal interpreted semantics, i.e., with stability search. During the composition, a situation is *stable* if no transition in any of the Moore machines can be fired without changing the values of input signals; otherwise, it is *transient*. The stability search semantics implies that the firing of transitions continues until a stable situation is reached. For this purpose, Teloco proposed in [22] is used for composition in this thesis.

Similar to an individual Moore machine, an *stabilized composed automaton (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.
- $e : S \times G_e \to S$ 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.

4.1.2 Plant feature model

Moore machine with Boolean signals

In this thesis, plants can also be modeled as FSMs with Boolean signals to describe dependency relations between signals.

Similarly, a plant model is defined by an 7-tuple $(L_P, l_{P,init}, I_P, O_P, G_{P,\delta}, \delta_P, \lambda_P)$, where:

- L_P is a finite set of locations[4].
- $l_{P,init}$ is the initial location, $l_{P,init} \in L_P$.
- I_P is a finite set of Boolean input signals; $I_P := I \cup O$, the inputs and outputs from specification models can be used as input signals in a plant model.
- O_P is a finite set of Boolean output signals.
- $G_{P,\delta} := expr(I_P)$ is a finite set of transition guards, which are Boolean expressions built up by input signals.
- $\delta_P : L_P \times G_{P,\delta} \rightarrow L_P$ 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_P' is used to denote the set of 'δ_P'.
- $\lambda_P : L_P \rightarrow 2^{O_P}$ is the output function that maps the locations to their corresponding output signals; 'Λ_P' is used to denote the set of 'λ_P'.

[4]The subscript 'P' stands for 'Plant'.

A simple example of a plant model is given in Fig. 4, which interacts with the second specification model in Fig. 3.

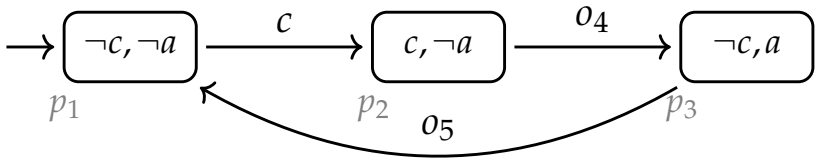


Figure 4: A simple Moore machine plant model with Boolean signals

This model can be understood as follows: initially, the signals a and c are *False*, a remains *False* when c is activated; as soon as o_4 takes place, a is activated and c is deactivated; after o_5 occurs, the values of a and c turn *False* again as described in the initial location.

4.2 Model-based black-box conformance testing of programmable controllers

4.2.1 Conformance testing of programmable controller: objective and 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 [22].

Test case & test sequence

According to [9] (page 368), a test case has been formally defined as 'a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement'.

In this thesis, for the conformance testing of a programmable controller, a test case

consists of a set of input signals, the expected output signals, the observed/real output signals, and the verdict of the result which is obtained from the comparison of expected and observed output signals.

Since the specification models are communicating Moore machines extended with Boolean signals in this thesis, the execution of several test cases can be automated by linking them together and building a test sequence.

Test process

As presented in Fig. 5, a complete model-based testing process for a programmable controller consists of four steps:

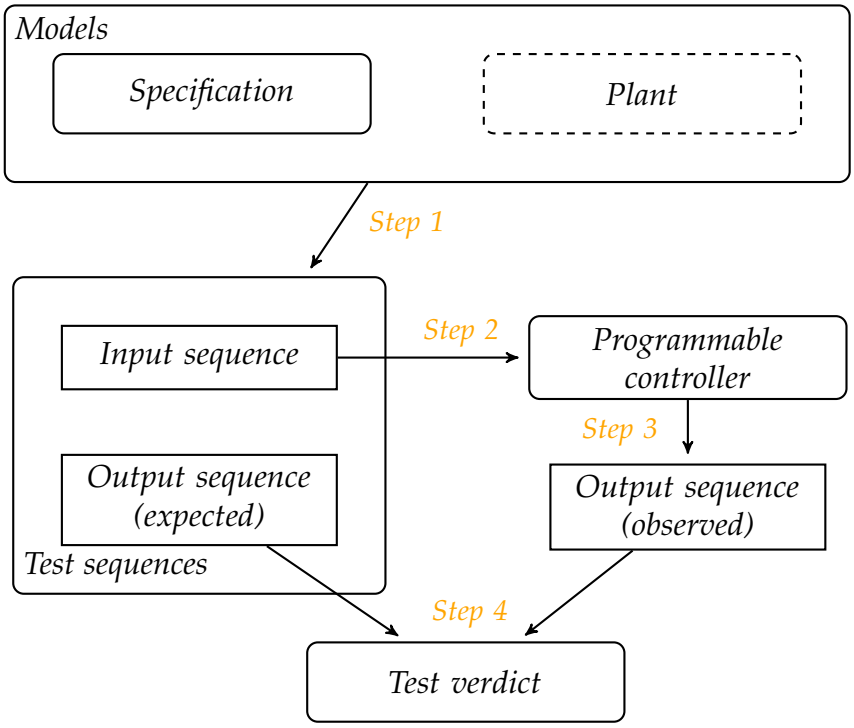


Figure 5: Workflow of testing a programmable controller

- **Step 1:** generate a test sequence (with input and output) from models (specification models with/without plant models)
- **Step 2:** feed the input sequence to the implemented 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

Test verdict

As presented in Fig. 6, if an implementation passes all sequences / a complete test sequence derived from its specifications, then the implementation is considered to completely conform the specifications. Otherwise, the sequence or step that detects the behavioral inconsistency between the implementation and specification, i.e., the difference between observed and expected sequence, is reported as a counter-example.

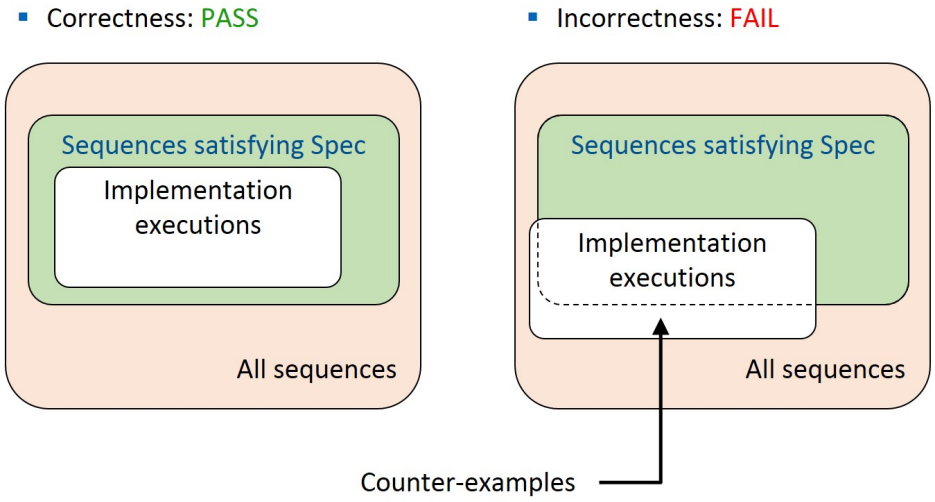


Figure 6: Test verdict of an implementation against its specifications

4.2.2 Black-box testing of a programmable controller: a testing unit

In this thesis, the implementation under test (IUT) are physical embedded programmable controllers that can store instructions and functions, and run in cyclic execution mode which fulfills hard real time requirement, while the specifications are Moore machine models extended with Boolean signals. In each cycle, a controller runs successively: reading the inputs, executing the programs and updating the outputs.

A test execution unit consists of three phases, each one containing a few steps [100]:

1. Before testing the transition of interest:

    - bring the specification model and the IUT to a certain state by inputting a signal sequence (synchronizing or homing sequences)

2. Testing the transition of interest:

    - apply the testing input signals to both the specification and the IUT

3. After testing:

    - if needed, apply a distinguishing sequence to both the specification and the IUT

    - observe the emitted output signals by the specification model and the IUT

    - compare the results and continue to the next unit of the test sequence

A simple example presented in Fig. 7 is used to illustrate a test unit of black-box testing for a Moore machine model specification implemented on a programmable controller. Here, the transition of interest is supposed to be the transition from l_4 to l_5, which corresponds to the second phase. From the initial location l_1, there are two paths to reach l_5, i.e., by applying three guards g_{12},g_{23},g_{34} or one guard g_{14}, which corresponds to the first phase. It is worth noting that applying the guard g_{14} is a more *economic* choice. After testing, the active location should be l_5, which however does not have an observable output. Considering the fact that there is no observable output in l_3 either, the location after testing cannot be directly identified. Therefore, the guard g_{56} is then applied, and if the output o_6 is observed, then it can be confirmed that the active location in one step before was l_5. These operations correspond to the third phase.

In summary, the second phase is considered as the real effective test, while the first and third phases constitute the testing overhead. Reducing the testing overhead and enhancing the effectiveness of testing is the aim of the DTT approach presented in this thesis, and it will be discussed in detail in chapter 5.

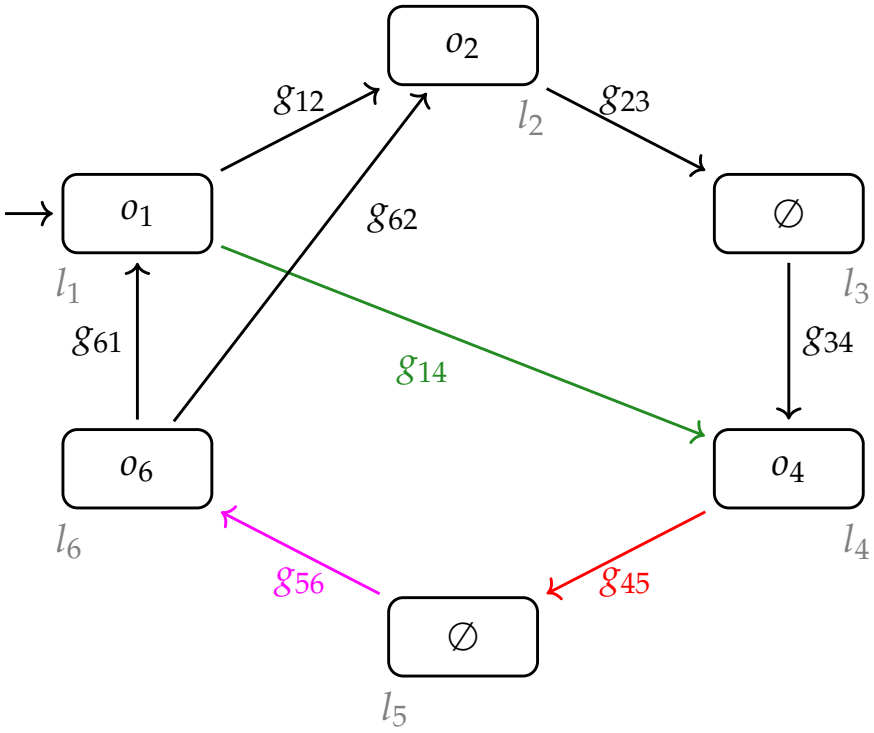


Figure 7: A simple specification example

4.2.3 Test generation of complete conformance testing

The focus of this thesis lies in the first step of testing process: construction of a *test sequence*. As introduced in chapter 3, complete conformance testing (CCT) is a type of black-box testing which tests the behavior of an IUT for all combinations of input signals from all the states with regard to its specification models. Fig. 8 presents the test generation process of complete conformance testing, which has been presented in [22].

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 evolutions by a set of minterms[5] 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 by a pair of input and output.

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

[5]A minterm is a basic element of an explicitly presented guard, e.g. if $g_{e(I,l)} = a \wedge \neg b$ and $I_S = \{a,b,c\}$, the corresponding minterms are $a \wedge \neg b \wedge c$ and $a \wedge \neg b \wedge \neg c$.

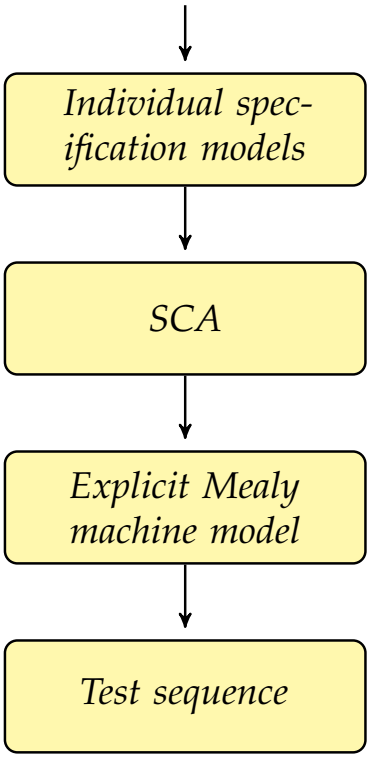


Figure 8: Process of complete conformance testing

number of test cases, and the ordering and repetition of test cases.

The first factor can be an issue in testing large scale systems. When the number of inputs of an system under test (SUT) grows linearly, the sizes of SCA and Mealy machine model grows exponentially, and therefore the number of test cases also grow exponentially, which results in the well-known *state space explosion* issue. This issue is what the plant feature (PF) approach presented in this thesis deals with, and will be discussed in detail in chapter 6.

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 transition arcs need to be traversed several times. This is an instance of the Chinese Postman problem [101], and can be formulated as '*Find a minimum length closed path that visits each edge in the graph at least once*'. This chapter uses the solution presented in [22].

5 Design-to-test approach

5.1 Introduction

This chapter presents a *design-to-test (DTT)* approach for black-box conformance testing of programmable controllers, where the specifications and implementations can be modeled as finite state machines (FSMs). The DTT approach analyzes the specification models and automatically modifies them with limited design and testing overhead, in order to improve the testability of their physical implementations. This approach also guarantees, by design, that the behavior of an implementation remains unchanged during its normal execution (i.e., when not connected to a test bench). Based on the proposed methods, a design-to-test MATLAB tool box (DTT-MAT) has been developed.

5.2 Core idea

Fig. 9 presents two V-models of system development. In a traditional engineering process (Fig. 9, left V-model), testing is executed mainly based on expert knowledge and usually not considered until the design phase is finished, or focuses on function unit test [102].

In contrast, the DTT approach proposed in this thesis takes testing performance of programmable controllers into consideration already during the design phase of specification models (Fig. 9, right V-model), and considers the whole system behavior rather than unit level. Before testing, the DTT approach automatically checks and modifies the initial specification models with limited overhead so that the final implementation

will be better testable, while keeping the nominal behavior unchanged during normal execution.

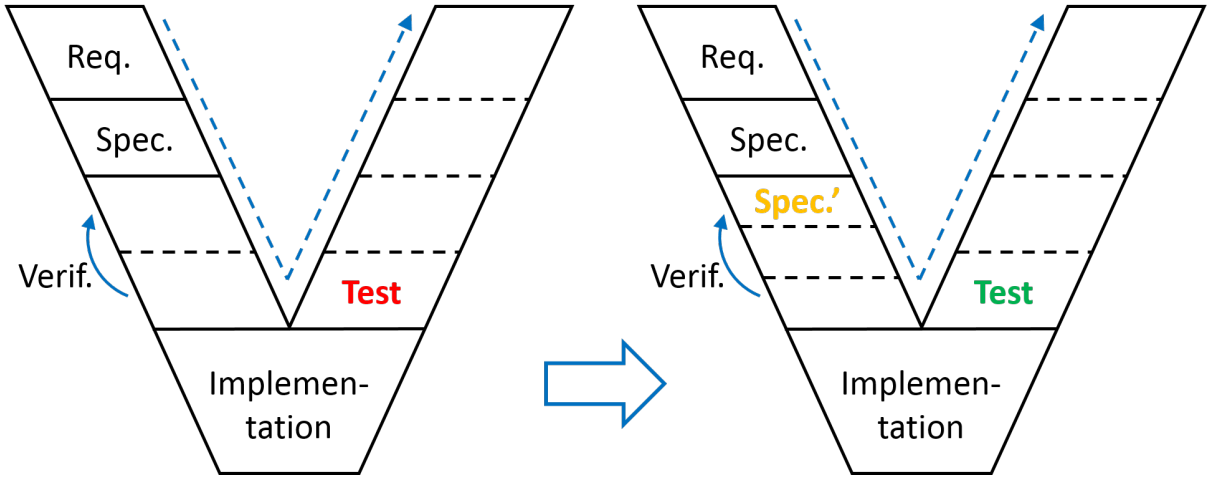


Figure 9: V-models of the system engineering process: classic and with the DTT approach

The modification of specification is inspired from code instrumentation techniques which are widely used in development and testing of software products [103], also introduced into embedded systems [104] [105]. However, no matter whether they use source code or byte code, whether in a static or dynamic way, these instrumentation methods all directly manipulate the specifications on the code level, while the proposed DTT approach modifies the specifications on the model level. Executable code is generated automatically from specification models afterwards. Thus, the DTT approach can be applied complementarily without issues to other methods required for certification, such as code inspection and model checking.

In brief, the DTT approach improves the testability, reduces the testing overhead with limited design overhead, while keeping the nominal behavior unchanged during normal execution. It is worth noting that during normal execution, the testing overhead code is deactivated, but it is not a dead code. Since this approach requires the specification (models), it is more suitable for internal testing rather than external testing.

The concept of the DTT approach was initially proposed in [106]. Then, a software toolbox based on this approach was developed and published in [107]. Due to the benefits of DTT, and especially the capability of achieving complete testing, this approach has been applied to critical systems in particular [108].

5.3 Testing issues & DTT methods

5.3.1 Black-box conformance testing on programmable controllers

As presented in chapter 4, in complete conformance testing, a test unit for a transition consists of three phases [100], which are briefly reminded as follows:

1. **Before testing**: activate a certain state in the implemented controller by inputting a signal sequence (synchronizing/homing sequences)

2. **During testing**: apply the set of input signals to the controller

3. **After testing**: compare the observed output signals to the expected outputs; if the output are not directly observable, apply a distinguishing sequence to the controller

During the three testing phases, several issues may occur, namely *observability*, *controllability* and *single-input-change (SIC)-testability* issues.

The main objective of the DTT approach is to slightly modify the specification models in order to automatically solve the three testing issues mentioned above. Fig. 10 presents the test generation process modified with DTT approach. Detailed explanations are given in following parts in this chapter.

Fig. 11 presents a schematic visualization of a specification model with testing issues, and the modified model by DTT approach. The initial model is depicted in black. Blue, purple and green drawings and texts correspond to the modifications, namely added T-guards, O-actions and C-guards.[1]

[1]This coloration will also be used in the model examples in the rest of this chapter.

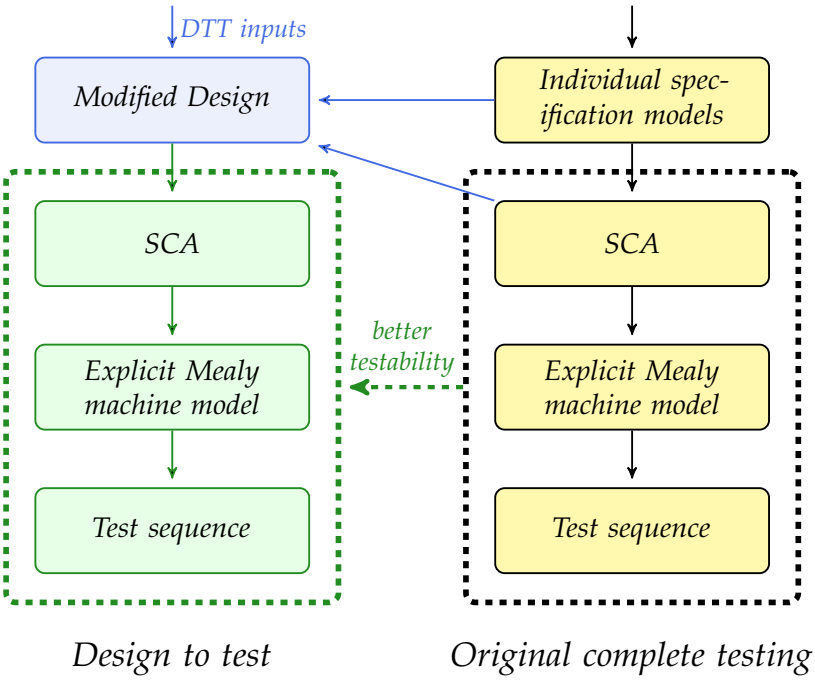


Figure 10: Process of complete conformance testing modified with the DTT approach

5.3.2 SIC-Testability & T-guard method

The concept of SIC test originates from testing of electronic circuits, because such tests are sensitive to address decoder faults, consume less power and can reach higher fault coverage than multiple-input-change (MIC) tests [109].

When testing a programmable controller, in the second testing phase, i.e., during a test step of a transition of interest, a set of input signals are read by the implementation under test (IUT). Because of cyclic input scanning, when several input signals change their values at the same time, the input values read by the IUT might deviate from the values supposed to be [11].

Fig. 12 presents the physical causes for this issue:

1. two events, i.e., changes of different physical signals, cannot happen exactly at the same time

2. according to the cyclic execution, a programmable controller reads the values of input signals only in the first phase during a cycle. If an input changes its value

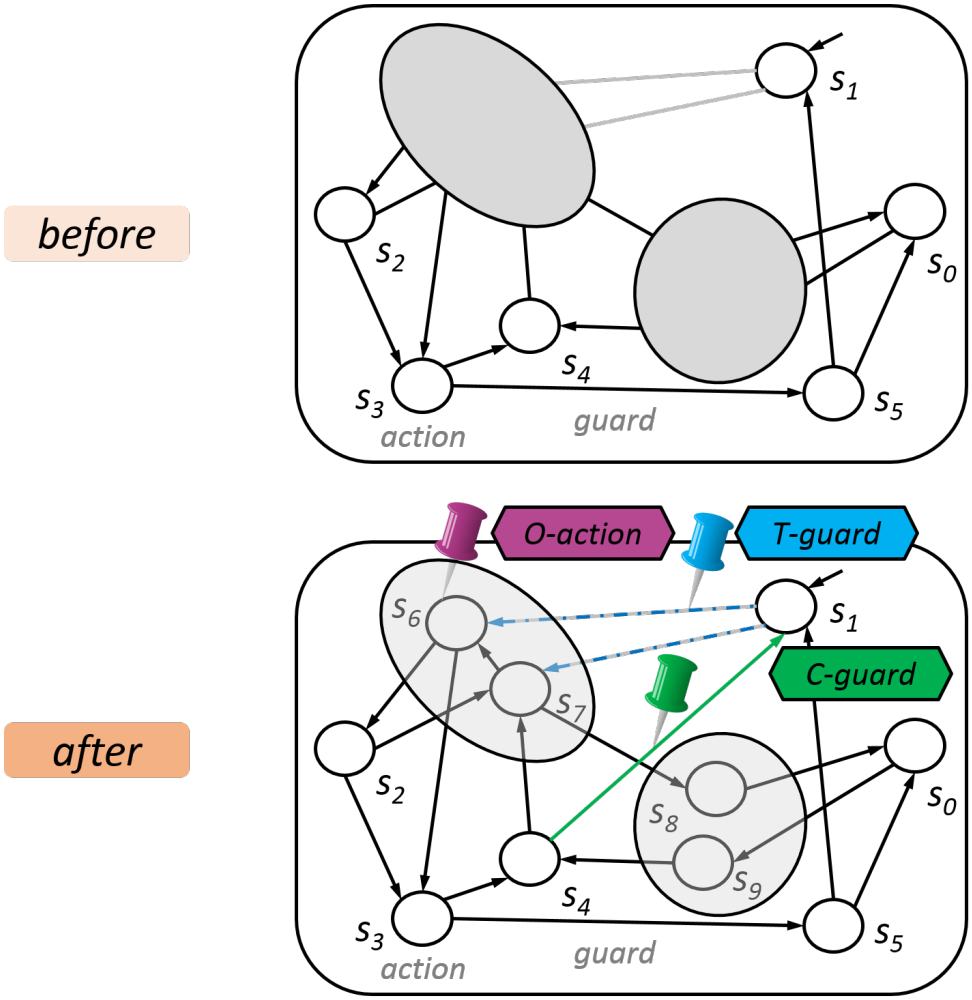


Figure 11: Basic idea of the DTT approach: adding T-guards, O-actions and C-guards to modify the initial specification model, so that the model will fulfill the SIC-testability, observability and controllability requirements

after this phase, the new value will be read in the next cycle

The example given in Fig. 13 will be used to help illustrate SIC-testability issue and the proposed T-guard method in DTT approach. Since the example contains only one individual model, S and E are equal to L and Δ.

In the initial model, in order to test the transition from location l_2 to l_3, two input signals, i.e., a and b, are supposed to change their values synchronously from '0' to '1'. This is therefore a MIC test step.

However, if a changes its value just before the input reading phase and b changes just after it (or b before a), the transition from l_2 to l_5 (or the transition from l_2 to l_4) will be

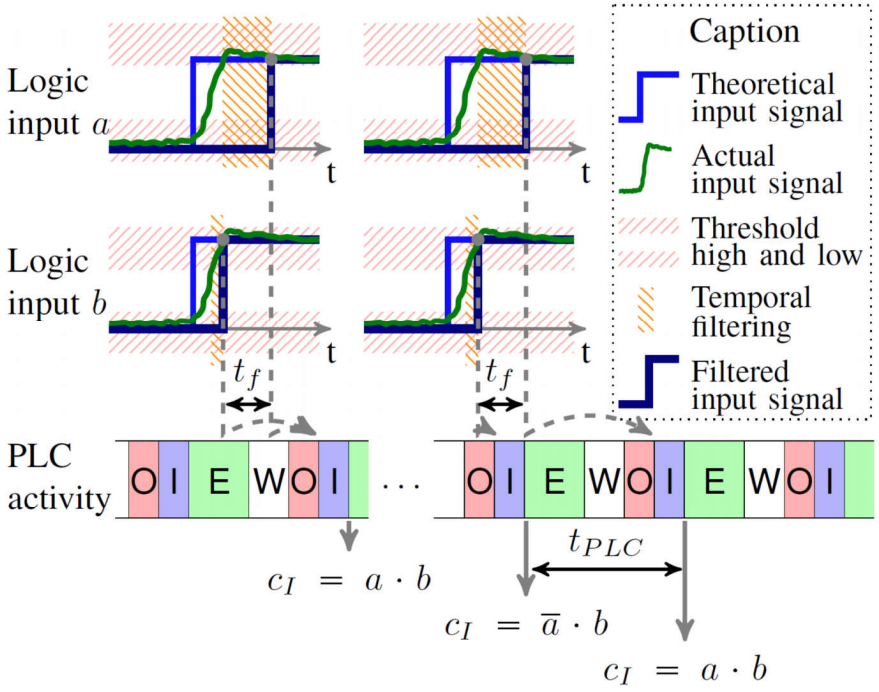


Figure 12: Physical causes of SIC-testability issue in programmable controllers

taken. Consequently, a transition that is executed in reality could be different from the one that should be tested.

Experiments have proven that the occurrence of this error can not be neglected, especially for large scale systems containing several I/O cards [110].

The proposed T-guard method solves this issue by transforming MICs into SICs with added T-guards (denoted as 'Tg' and depicted in blue in Fig. 13). The status changes of signals and locations after adding T-guards are presented in Fig. 14.

Before a MIC happens, the input signal Tg is set to the value '0', so all outgoing transitions from the current location, i.e., l_2, are frozen from being fired. After the MICs are stabilized, i.e., a and b have changed and stabilized their values, Tg is set to the value '1' again. In this way, only the correct outgoing transition will be fired. Now, the system can be completely tested without errors caused by asynchronism among input signals.

For a more complex system, Alg. 1 depicts the method used to achieve a full SIC-testability by adding a minimum set of T-guards to some transitions in the models.

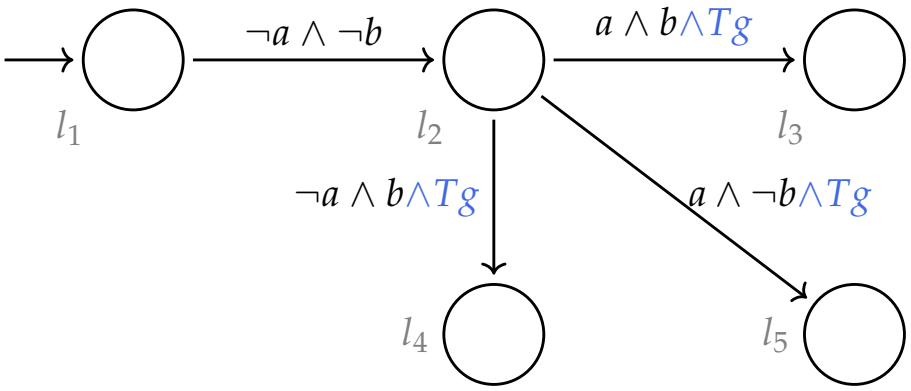


Figure 13: A simple Moore machine model updated with T-guards

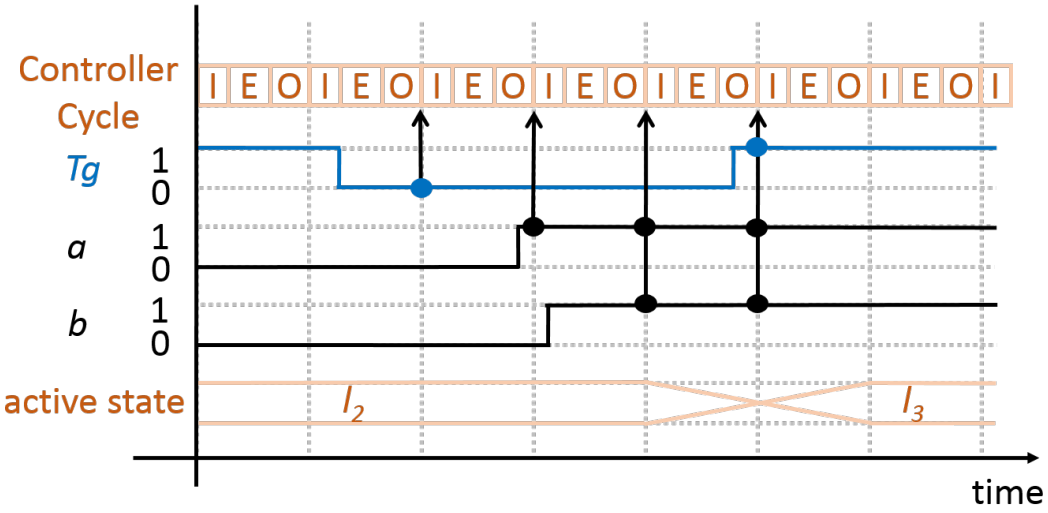


Figure 14: Status changes of signals and locations after adding T-guards

The example given in Fig. 13 will be used again to help illustrate the algorithm.

The inputs of the algorithm are L, Δ, G_δ and I, which are respectively the union of the sets of locations, the full set of the transition functions, the full set of the transition guards, and the full set of inputs of Moore machine models.

In Alg. 1, first G_{E-NSIC}, a subset of evolution guards in the composed model that are non-SIC-testable, is obtained by running the program Teloco [22]. Non-SIC-testable guards represent the input combinations that are not accessible by sole single input changes in existing evolutions. In the example in Fig. 13, G_{E-NSIC} contains one guard, i.e., $a \wedge b$ in the evolution from l_2 to l_3.

Then, all the inputs that are involved in the non-SIC-testable evolution guards are

Algorithm 1: Pseudo-code of the T-guard method

Input: L, Δ, G_δ, I
Result: G_{SIC}

Initialization: $I_{NSIC} := \varnothing$; $T_{target} := \varnothing$;
$L_T := \varnothing$; $G_{SIC} := G_\delta$;
begin
  $G_{E-NSIC} := Teloco(L, \Delta, I)$;
  foreach $g_{E-NSIC} \in G_{E-NSIC}$ ***and*** $i \in I$ **do**
    $I_{NSIC} \mathrel{+}= \{i \mid i \wedge g_{E-NSIC} = g_{E-NSIC}\}$;
    $I_{NSIC} \mathrel{+}= \{\neg i \mid \neg i \wedge g_{E-NSIC} = g_{E-NSIC}\}$;
    /* check if i or $\neg i$ is an element in g_{E-NSIC} */
  foreach $i \in I_{NSIC}$ **do**
    $i := False$;
    /* set their initial values to *False* */
  foreach $j \in I_{NSIC}$ **do**
    $j := True$;
    if $\bigvee_{G_{E-NSIC}} g_{E-NSIC} \neq False$ **then**
      $T_{target} \mathrel{+}= \{j\}$;
      $j := False$;
      /* this means j is an essential element in T_{target}, a minimum set of I_{NSIC}, so that all guards of G_{E-NSIC} will be protected by T-guards */
  foreach $g_\delta \in G_\delta$ ***and*** $l \times g_\delta \to l'$ **do**
    foreach $k \in T_{target}$ **do**
      if $k \wedge g_\delta = g_\delta$ **then**
        $L_T \mathrel{+}= \{l\}$;
        /* find all locations that have at least one outgoing transition that involves at least one input in T_{target} */
  foreach $l_T \in L_T$ **do**
    foreach $g_\delta \in G_\delta$ ***and*** $l \times g_\delta \to l'$ **do**
      if $l = l_T$ **then**
        $G_{SIC} \mathrel{-}= \{g_\delta\}$;
        $g_\delta := g_\delta \wedge Tg$;
        /* add a T-guard to all outgoing transitions from l_T */
        $G_{SIC} \mathrel{+}= \{g_\delta\}$;

identified (lines 5 to 7). The purpose of lines 8 to 14 is to obtain T_{target}, a minimum set of inputs, so that as long as these inputs are protected by T-guards, all the previous non-SIC-testable guards are SIC-testable. In the example, T_{target} could be $\{a\}$ or $\{b\}$.

If a location has any outgoing transition that contains a non-SIC-testable guard, all outgoing transition guards from this location should be protected by T-guards (lines 15 to 24). In the example, all the transitions outgoing from l_2 will be protected by T-guards.

The result of this algorithm is G_{SIC}, an updated set of the transition guards. All the previous non-SIC-testable parts of the states are now protected by T-guards, so any outgoing evolution that requires a multiple-input-change test step can be temporary frozen by the T-guards, i.e., by setting the value of the input signal Tg to the value '0'. Therefore, the system can be completely tested without errors caused by asynchronism among input signals.

5.3.3 Observability & O-action method

In order to realize the third testing phase, i.e., the identification of the current state of the IUT, two methods have been considered:

1. directly observing its output

2. applying a distinguishing sequence

The first method requires a strong hypothesis: Every state must have a unique observable output action. However, this is not always the case in real systems. The second method is more generally applicable. However, it is still not always possible to find such sequences, and if they exist, they might be of exponential length to the number of states [100]. This will obviously generate a huge testing overhead for large scale systems. This is what is referred to as the *observability issue*.

Fig. 15 presents an example of the observability issue as well as a solution by O-action method. Since the example contains only one individual model, S is equal to L.

Initially, four locations have the same output o_i, so that they suffer from observability issue. After adding two O-actions and setting their values to *True*/*False* correspondingly, the locations are now directly distinguishable from each other.

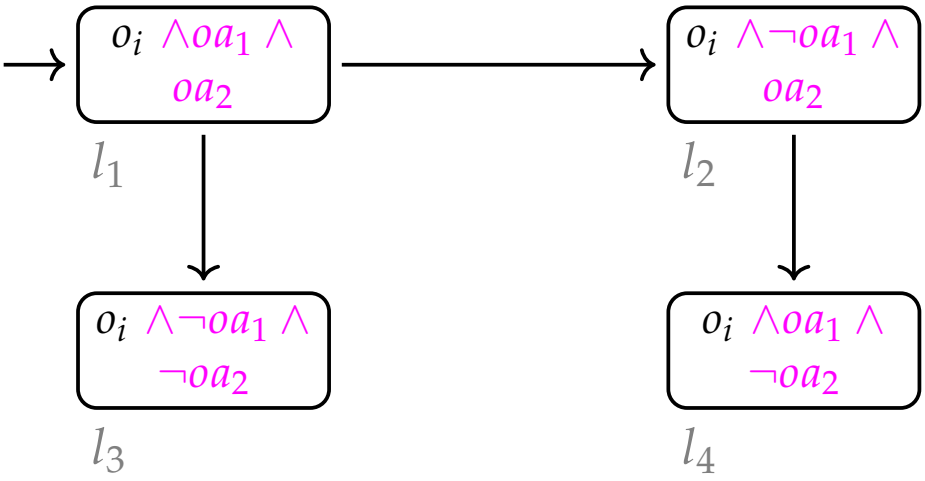


Figure 15: A simple Moore machine example updated with O-actions

For a more complicated system, Alg. 2 depicts the method to achieve full observability by adding a minimum number of O-actions into some locations.

In Alg. 2, S, L, Λ, Λ_s are defined in the same way as in Sec. 4 and Alg. 1, while $\#_{model}$ means the number of individual models. For example, in Fig. 15, $\#_{model}$ is equal to 1.

First of all, output actions of all the states are examined (lines 3 to 4). If at least one pair of states share the same actions, the individual locations inside the states will be further analyzed (lines 5 to 6). If two different locations in the same individual model have the same action, then they are identified as the cause of non-observability of those states. These locations will be collected in L_{NObs} (line 7). In Fig. 15, L_{NObs} contains all four locations since they all have the same action o_i.

Since each O-action can be set to the value '0' and '1', a list of n O-actions can be used to represent 2^n different outputs. Thus, a minimum set of O-actions can be obtained by calculating the logarithm of the size of L_{NObs} with the base of 2 (for Boolean signals) (line 8)[2]. In Fig. 15, two O-actions are the minimum required to distinguish four locations.

A unique O-action combination will be assigned to each location from L_{NObs} (lines 9

[2]The final result is an integer after using celling function.

Algorithm 2: Pseudo-code of the O-action method

Input: $S, L, \Lambda, \Lambda_s, \#_{model}$
Result: Λ_{Obs}
1 **Initialization**: $L_{NObs} := \varnothing$; $\Lambda_{Obs} := \Lambda$;
2 **begin**
3 **foreach** $(s_i, s_j) \in S^2, s_i \neq s_j$ **do**
4 **if** $\lambda_s(s_i) = \lambda_s(s_j)$ **then**
5 **for** $n = 1 : \#_{model}$ **do**
6 **if** $\lambda_n(l_i) = \lambda_n(l_j), l_i \in s_i, l_j \in s_j$ **then**
7 $L_{NObs} \mathrel{+}= \{l_i, l_j\}$;
8 $\#_{OA} := \lceil \log_2 (|L_{NObs}|) \rceil$ /* the number of necessary 0-actions */
9 $OA := \left[oa_1, oa_2, \cdots, oa_{\#_{OA}}\right] \in O^{\#_{OA}}$;
 /* oa is a single 0-action, 0A is a list of oa */
10 **foreach** $l \in L_{NObs}$ **do**
11 $\Lambda_{Obs} \mathrel{-}= \{\lambda(l)\}$;
12 $\lambda(l) := \lambda(l) \wedge minterm(OA)$;
 /* $minterm(OA)$ returns a unique combination of OA elements */
13 $\Lambda_{Obs} \mathrel{+}= \{\lambda(l)\}$;

to 13). As a result, the previous non-observable states will become fully observable in one step.

5.3.4 Controllability & C-guard method

Similar to the third phase, during the first testing phase the specification and the IUT should be brought to a specific state. The controllability issue concerns whether and how fast the IUT can be brought from an arbitrary state to another desired state. This issue can be solved by applying a homing or a synchronizing sequence. For complex systems, this process can also require long sequences and thereby generate high testing overheads. This is what is referred to as the *controllability issue*.

The goal of the proposed C-guard method is to limit/reduce the controllability issue, i.e., to shorten the distance between locations during testing. It is realized by adding a set of C-guard transitions to the models.

Also, in some models, some states may not be reachable from some other states. In the

example in Fig. 16 (since the example contains only one individual model, S is equal to L), once δ_{02} and δ_{24} have been taken, l_0 and l_1 cannot be reached anymore, so δ_{01} and δ_{13} cannot be tested without restarting the system. Thanks to the added C-guard to the initial location, all transitions can be completely tested with less manual intervention, and therefore, more automatically and conveniently.

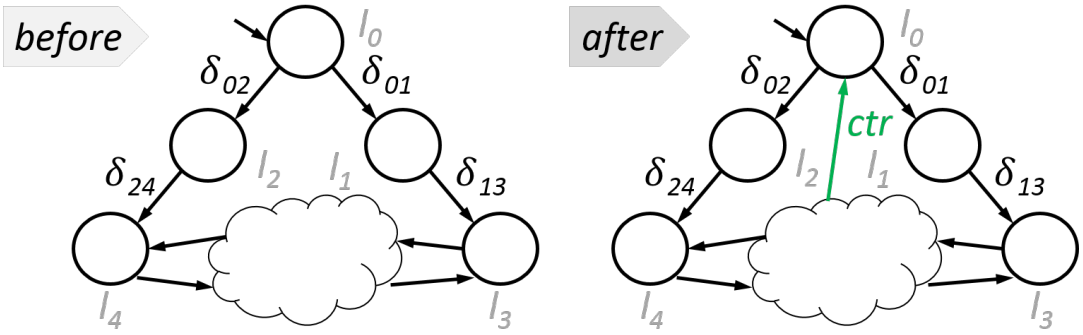


Figure 16: C-guard in testing transitions between unreachable locations

In the following part, Alg. 3 presents the algorithm of the proposed C-guard method, while one function, *EvoCalc*, is presented in Alg. 4.

The example given in Fig. 17 will be used to help illustrate the controllability issue and the C-guard method. Since the example contains only one individual model, S is equal to L.

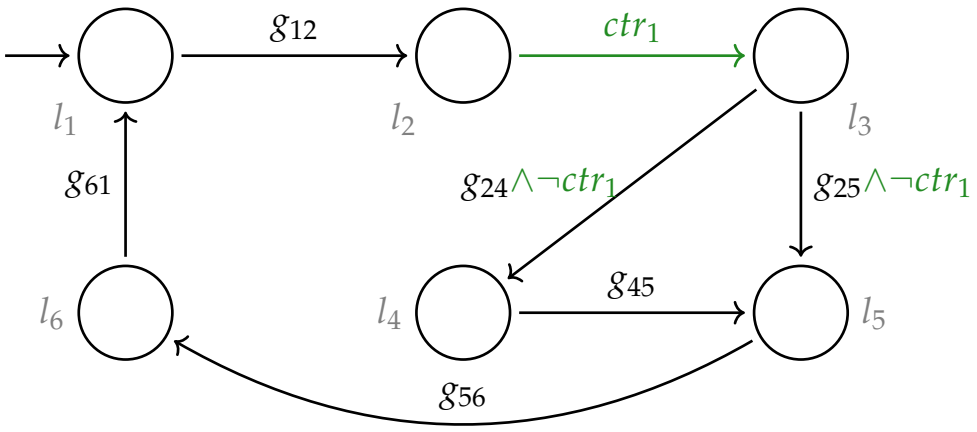


Figure 17: A simple Moore machine example updated with C-guards

In Alg. 3, $Path_S$ is defined as a path cost matrix for all pairs of states. During initialization (lines 3 to 9), if there is a direct evolution from one state to another, the path cost for the pair of states will be set to 1, if not, then to ∞. The Floyd-Warshall

algorithm is then applied to calculate indirect path costs between all pairs of states (line 10). For example, $Path_S$ for the system in Fig. 17 is displayed in Tab. 2. Values in black and green correspond to the path costs for the initial model and the model updated with C-guards, respectively.

Table 2: Path cost matrix for the system in Fig. 17

To \ From	l_1	l_2	l_3	l_4	l_5	l_6
l_1	0	$\infty/4$	3	3	2	1
l_2	1	0	4	4	3	2
l_3	$\infty/2$	$\infty/1$	0	$\infty/5$	$\infty/4$	$\infty/3$
l_4	$\infty/3$	$\infty/2$	1	0	$\infty/5$	$\infty/4$
l_5	$\infty/3$	$\infty/2$	1	1	0	$\infty/4$
l_6	$\infty/4$	$\infty/3$	2	2	1	0

After that, the maximum of $Path_S$ will be compared to $Limit_{Ctr}$, i.e., the desired path cost limit. If the maximum of path cost exceeds this limit, a set of new evolutions, E_{Ctr}, will be built (lines 11 to 13). Based on the result of E_{Ctr}, a minimum set of transitions with associated C-guards, Δ_{Ctr}, for individual models will be calculated (lines 14 to 19). In the example in Fig. 17, if the path cost from s_2 to s_3 exceeds $Limit_{Ctr}$, as a result of C-guard method, a new transition from s_2 to s_3 will be built with the guard ctr_1.

It should be noted that for stability reason the negation of the C-guards will be added to the guards of outgoing transitions from the destination state of the newly added C-guard (lines 20 to 27). In the example in Fig. 17, the guards on all the outgoing transitions from l_3 will be added with $\neg ctr_1$.

Alg. 4 presents the function *EvoCalc* (Alg. 3, line 12) in detail, which is called when the maximum value of $Path_S$ exceeds $Limit_{Ctr}$ (Alg. 3, line 11).

Elements of $Path_S$, i.e., $Path_S(s_i, s_j)$, are summed in rows and columns (lines 5 and 6). The maximum sums of path costs in rows and columns, i.e., max_{sum-r} and max_{sum-c}, are obtained through iterative comparison (lines 7 to 10).

If max_{sum-r} is larger than max_{sum-c}, it means that there is a *most critical destination state*, i.e., s_{max-i}, which takes the largest path cost to be reached from other states. A

Algorithm 3: Pseudo-code of the C-guard method

Input: $S, E, \Delta, G_\delta, Limit_{Ctr}, \#_{model}$
Result: Δ_{Ctr}, G_{Ctr}

1 **Initialization**: $E_{Ctr} := \varnothing$; $\Delta_{Ctr} := \Delta$; $G_{Ctr} := G_\delta$;
2 **begin**
3 **foreach** $(s_i, s_j) \in S^2$ **do**
4 **if** $s_i = s_j$ **then**
5 $Path_S(s_i, s_j) := 0$;
6 **else if** $\exists (s_i \times g_e \to s_j) \in \delta_L$ **then**
7 $Path_S(s_i, s_j) := 1$;
8 **else**
9 $Path_S(s_i, s_j) := \infty$;
10 $Path_S := Floyd\text{-}Warshall(Path_S)$;
11 **while** $\max(Path_S) > Limit_{Ctr}$ **do**
12 $(Path_S, e_{new}) := EvoCalc(Path_S)$;
 /* *EvoCalc* is presented in Alg. 4 */
13 $E_{Ctr} \mathrel{+}= \{e_{new}\}$;
14 **foreach** $(s_{src} \times g_e \to s_{des}) \in E_{Ctr}$ **do**
15 **for** $n = 1 : \#_{model}$ **do**
16 **if** $\nexists (l_{src} \times g_\delta \to l_{des}) \in \Delta$ ***and*** $l_{src} \in s_{src}, l_{des} \in s_{des}$ **then**
17 $\Delta_{Ctr} \mathrel{+}= \{(l_{src} \times g_{\delta,Ctr} \to l_{des})\}$;
 /* the expression of $g_{\delta,Ctr}$ is assigned in line 21 */
18 $\#_{Ctr} := |\Delta_{Ctr}|$;
 /* the number of C-guards */
19 $C := \left\{ctr_1, ctr_2, \cdots, ctr_{\#_{Ctr}}\right\}$;
 /* a set of C-guards */
20 **foreach** $\delta_{Ctr} \in \Delta_{Ctr}$ **do**
21 $g_{\delta,Ctr} := C(i)$;
 /* i is the index of δ_{Ctr} in Δ_{Ctr} */
22 $\exists! l_{src} \times g_{\delta,Ctr} \to l_{des}$;
23 **foreach** $g_\delta \in G_\Delta$ **do**
24 **if** $\exists l'_{des} \mid l_{des} \times g_\delta \to l'_{des}$ **then**
25 $G_{Ctr} \mathrel{-}= \{g_\delta\}$;
26 $g_\delta := g_\delta \wedge \neg C(i)$;
27 $G_{Ctr} \mathrel{+}= \{g_\delta\}$;

Algorithm 4: Pseudo-code of *EvoCalc*

Data: $Path_S$
Result: $Path_{L-new}, e_{new}$

Initialization: $max_{sum-r} := 0$; $max_{sum-c} := 0$;
    $min_{sum} := \infty$;
begin
    foreach $(s_i, s_j) \in S^2$ **do**
        $D_{sum-s_i} := \sum_{s_j} Path_S(s_i, s_j)$;
        /* Sum of path costs between states in row */
        $D_{sum-s_j} := \sum_{s_i} Path_S(s_i, s_j)$;
        /* Sum of path costs between states in column */
        if $D_{sum-s_i} > max_{sum-r}$ **then**
            $max_{sum-r} := D_{sum-s_i}$; $s_{max-i} := s_i$;
        if $D_{sum-s_j} > max_{sum-c}$ **then**
            $max_{sum-c} := D_{sum-s_j}$; $s_{max-j} := s_j$;
    if $max_{sum-r} > max_{sum-c}$ **then**
        foreach $(s_i, s_j) \in L^2, s_i \neq s_{max-i}$ **do**
            $Path_{sum} := \sum_{s_j} Path_S(s_i, s_j)$;
            if $Path_{sum} < min_{sum}$ **then**
                $min_{sum} := Path_{sum}$; $s_{min-i} := s_i$;
        $e_{new} := e(s_{min-i}, s_{max-i})$;
    else
        foreach $(s_i, s_j) \in L^2, s_j \neq s_{max-j}$ **do**
            $Path_{sum} := \sum_{s_i} Path_S(s_i, s_j)$;
            if $Path_{sum} < min_{sum}$ **then**
                $min_{sum} := Path_{sum}$; $s_{min-j} := s_j$;
        $e_{new} := e(s_{max-j}, s_{min-j})$;
    $Path_{L-new} := Floyd\text{-}Warshall(Path_S, e_{new})$;

new evolution will then be built to this state from a state that takes the least cost to be reached by other states, i.e., s_{min-i} (lines 11 to 16).

If max_{sum-c} is larger than max_{sum-r}, it means that there is a *most critical source state*, i.e., s_{max-j}, which takes the largest path cost to reach other states. A new evolution will then be built from this state to a state that takes the least cost to reach other states, i.e., s_{min-j} (lines 17 to 22).

After adding a new evolution, indirect path costs between all pairs of states might also be shortened, which will again be calculated with the Floyd-Warshall algorithm (line 23).

5.4 Design, testing & normal execution

In brief, applying the DTT approach in the design phase, the specification models will be checked if they encounter any of the SIC-testability, observability, and controllability issues. If yes, the models will be modified by adding T-guards, O-actions and C-guards to the proper transitions, outputs, and pairs of states.

5.4.1 Test cost through design

Adding additional guards and actions means requiring additional input and output ports of programmable controllers for testing.

Considering the fact that the hardware of programmable controllers usually has limited number of input and output ports, e.g., a common off-the-shelf input/output module has 8 or 16 ports, the additional 'port-cost' of the DTT approach should also be evaluated/estimated.

The T-guard method will always generate only one additional guard, and therefore the port-cost is always only 1, independent from the size and complexity of a system under test.

The number of added O-actions is the logarithm with the base of 2 to the number of stable states that can not be directly distinguished[3]. Using the synchronous composition, the total number of stable states is polynomial to the size of the system models, i.e., the number of inputs and locations. In the worst case, i.e., when all stable states are not directly distinguishable, the theoretical upper bound of the number of O-actions would be polylogarithmic to the system size. Of course, in a meaningful practical system, the result is much lower than the upper bound. Applications on case studies are presented in section 5.6.

The number of C-guards is influenced by two factors: the structure of stabilized composed automaton, and the required controlability by users. In the worst case, all locations need to be directly connected, which can be understood as a mathematical combination problem, i.e., choose two from all locations in each model. Similar to the O-actions, in practice, the number of necessary C-guards is much lower than the theoretical upper bound. Applications on case studies are also presented in section 5.6.

5.4.2 Influence of added guards and actions on state-space

T-guards, O-actions, and C-guards are added to the specification models after the composition. They are only used as support to a better realization of complete testing, but they are never involved in the system functionality.

Therefore, in the test generation which reaches a full coverage of transition/evolution conditions from all states, T-guards, O-actions, and C-guards are not considered. As a result, the added guards and actions do not change the system state-space of test generation.

5.4.3 Settings of added guards and actions in testing

During the testing phase, T-guards can be set to the value '0' accordingly so that original MIC test steps can be handled as SIC test steps.

[3]In practice, the chosen value is the celling of the calculated logarithm.

O-actions can be assigned values in accordance with the O-action method results so that all locations will be directly distinguishable.

C-guards can be accordingly set to the value '1' to enable the control of evolutions.

5.4.4 Settings of added guards and actions in normal execution

Once testing is completed, i.e., before running the IUT in its normal mode, the input signal Tg will be connected to the logic 1 level (3.3V, 5V or 24V depending on the implementation architectures) so that it will not affect the original transition guards ($g \wedge 1 = g$).

Similar to T-guards, the input signals ctr_i will be connected to the logic 0 level (0V or below 1V for most of the architectures) so that they will never enable the firing of these transitions ($g \wedge 0 = 0$).

O-actions are only observable output signals which are not used in the transition conditions. They do not affect the internal system behavior throughout testing and normal executions.

In summary, the DTT approach does not change the nominal behavior of IUT with regard to their specifications during normal execution.

5.5 DTT-MAT: MATLAB Toolbox for the DTT approach

A toolbox DTT-MAT has been developed to realize the proposed DTT approach. It is available from the homepage of my research group: www.ses.mw.tum.de.

5.5.1 Workflow of DTT-MAT

The workflow of DTT-MAT is briefly depicted in Fig. 18.

To use DTT-MAT, the user should firstly model the system in MATLAB Stateflow. Usually, a complex system is split into several individual models for reasons of simplicity.

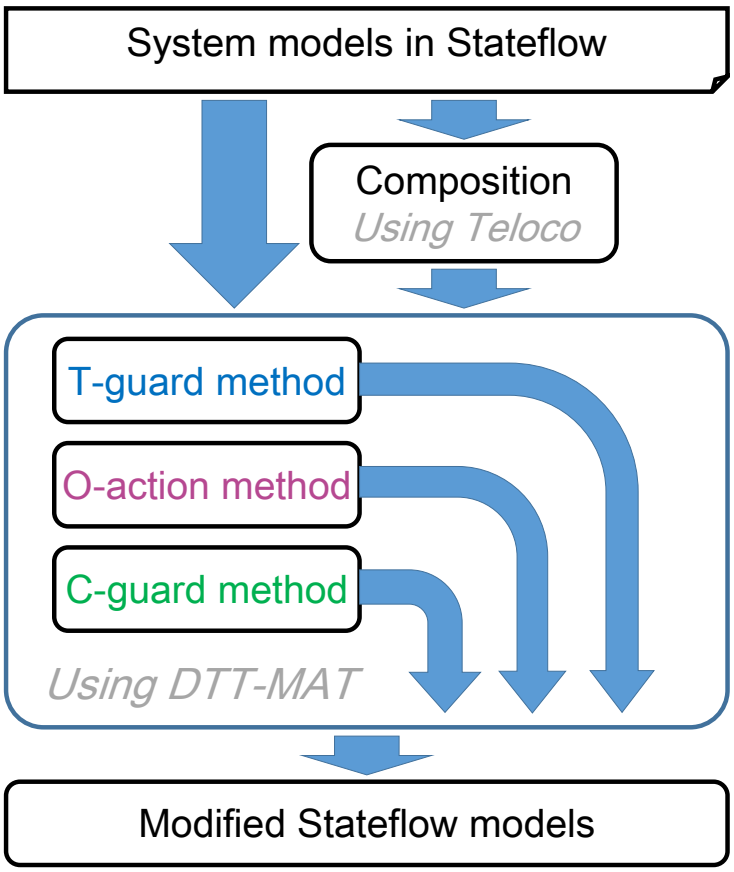


Figure 18: Workflow of DTT-MAT

Stateflow models are first transformed into Moore machines, which will next be read by Teloco [22]. Then, all the individual models will be composed with Teloco, and SIC-testability results will also be generated.

The Stateflow models and the composed model from Teloco will then be analyzed by the proposed DTT methods implemented in MATLAB. The three DTT methods can be executed one after another, which is recommended when modifying a new system; but is also possible to run them separately, e.g. to compare the impact of different values of $Limit_{Ctr}$ on the system structure.

Based on the results from the DTT methods, the Stateflow models will be automatically updated with T-guards, O-actions and C-guards, in order to fulfill the observability, controllability and SIC-testability requirements.

Additionally, automatic code generation has been implemented for programmable logical controller (PLC), using IEC 61131-3 Structured Text format.

5.5.2 Limitations for applicable Stateflow models

MATLAB Stateflow offers rich possibilities to build models. For example, signals and events are both accepted as inputs, actions can be linked to states, transitions or transition conditions [111].

However, only Moore machine models can be handled with the current version of Teloco. Besides, only Boolean signals are accepted as valid inputs for the current version of DTT-MAT. Detailed instructions as well as some examples are available together with the toolbox.

5.6 Case Studies

In this chapter, two case studies are presented to illustrate the DTT approach and the DTT-MAT toolbox: the first one is a critical application while the second one is a larger scale system. For more information upon the application of the DTT approach, [108] provides two other case studies on industrial applications.

5.6.1 A cooling water system

The first case study is a cooling-water system for a gas turbine slightly adapted from [112].

System Description

As shown in Fig. 19, the cooling-water system is made up of three pumps (P_1, P_2 and P_3), two fuel gas control valves (V_1 and V_2), a compressor, a combustion chamber, and

a turbine.

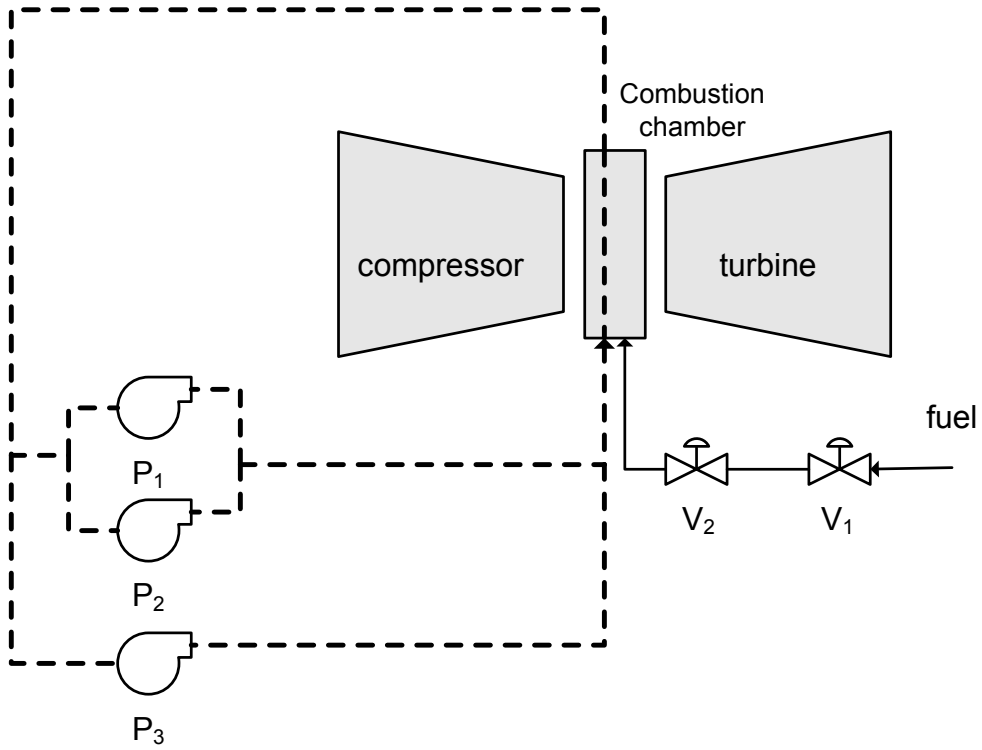


Figure 19: Case study: a cooling-water system

Fuel is injected into the combustion chamber, mixed with air and burned at a high temperature. The flow of fuel is controlled by V_1 and V_2. The combustion produces a high temperature and high pressure gas steam that expands through the turbine and spins the blades. The more fuel is injected, the more power will be generated by the turbine blades, and the higher the temperature in the chamber will be.

The cooling-water is pumped through the three pumps to cool down the lubrication oil that lubricates the blades and shaft in the combustion chamber. Similar to the principle of fuel flow, the more cooling-water is pumped, the better cooling effect will be achieved. Among the pumps, P_1 is operated under normal conditions, while P_2 is a standby pump and P_3 is an emergency pump used when emergency action is needed.

The system contains in total 7 inputs and 11 outputs, as listed in Tab. 3.

Table 3: Inputs & outputs of the cooling-water system

Input	Description
Sys_on	activated when the System is turned on
*V*1*_stuck*	activated when the valve-1 is stuck (not visible from outside)
Mnt_req	activated when the request of a maintenance is sent
Mnt_done	activated when a maintenance is done
*T*1*_above*	activated when the temperature is above T1
*T*2*_above*	activated when the temperature is above T2
*T*3*_above*	activated when the temperature is above T3

Output	Description
*V*1*_close*	close the valve-1
*V*1*_openC*	open the valve-1 with control
*V*1*_openS*	open the valve-1 without control (stuck)
*V*2*_close*	close the valve-2
*V*2*_openF*	open the valve-2 fully
*V*2*_openC*	open the valve-2 with control
Pi_off	turn off the pump-i ($i \in \{1,2,3\}$)
Pi_onC	turn on the pump-i with control ($i \in \{1,2,3\}$)
Pi_onF	turn on the pump-i fully ($i \in \{1,2,3\}$)
Sys_fail	the system fails
Sys_mnt	the system maintenance takes place

Modeling of system

The cooling-water system can be modeled with 9 individual Moore machines. The models for V_2 and *System Status* are selected as illustrative examples for this section (Fig. 20). Again, the initial models are depicted in black. Blue, green and purple drawings and text correspond to the elements added by DTT approach.

When the system is idle, the pumps are off and valves are closed, i.e., the models are in *close* or *off* states.

When the system is turned on, V_1 is opened with control while V_2, as an emergency valve, is kept fully open (state *V*2*_openF*). The cooling-water is circulated by P_1 to take the excess heat away from the combustion. Under normal conditions, P_1 works alone

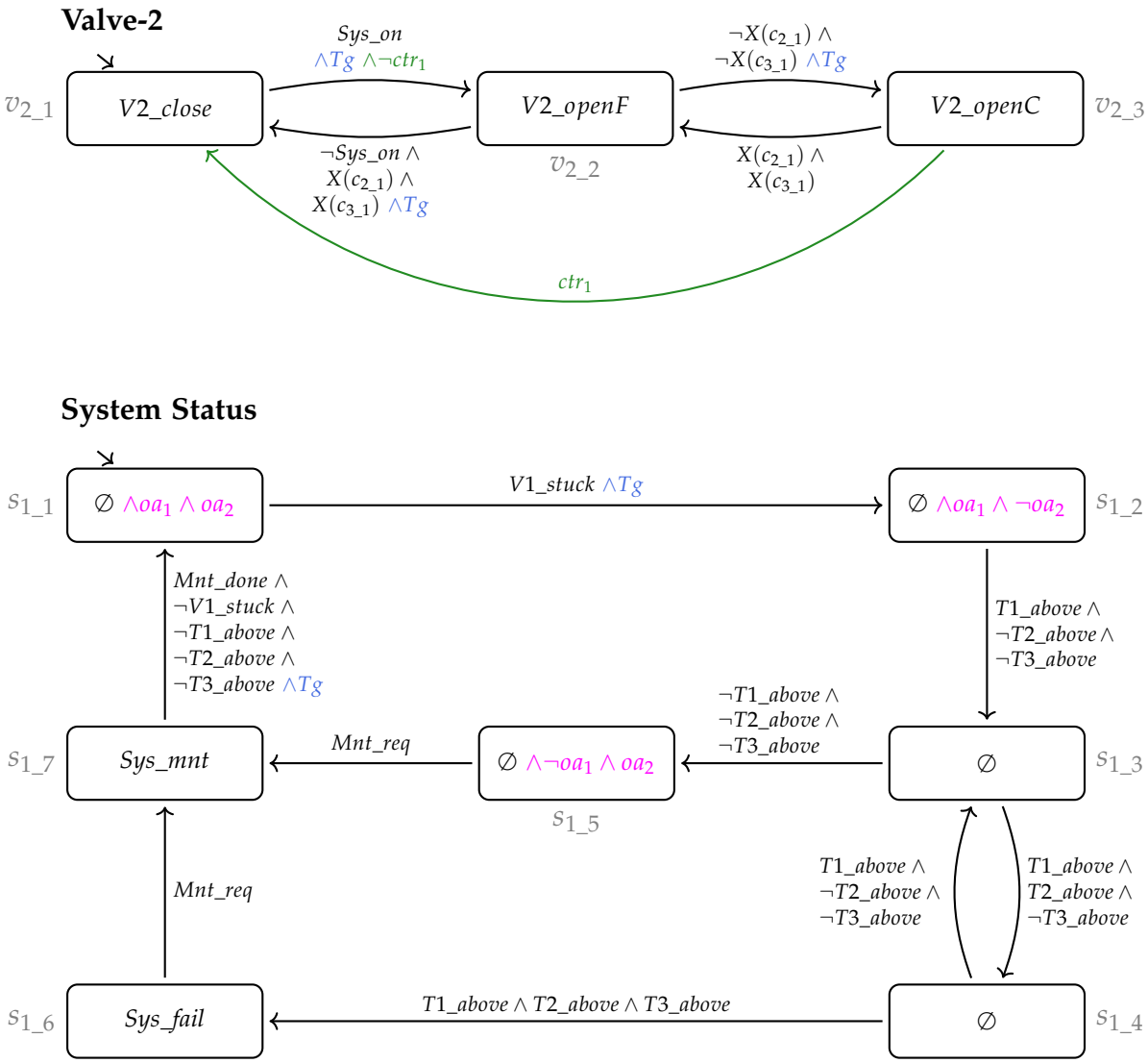


Figure 20: Specification model of V_2 and *System Status*

to regulate the temperature in the combustion chamber.

In this application, the fuel control valve V_1 may get stuck open, meaning that much more fuel than required can be sent to the combustion chamber. This error causes the generation of excessive heat in the combustion process. When the chamber temperature exceeds a certain limit $T1$, P_2 is turned on with control to bring the temperature to normal while P_1 is kept working at full speed.

If P_2 reaches its capacity limit and the temperature continues to rise and surpasses a higher limit $T2$, emergency pump P_3 is turned on with control to further help cool down the chamber. Meanwhile the emergency valve V_2 will be used to control the fuel flow (state $V2\_openC$). P_2, P_3 and V_2 will finish their operations i.e., return to initial states only when the temperature is cooled down below $T1$.

During the time the temperature is rising, if both pumps are not turned on within a certain time, the system may fail (state Sys_fail) when a certain temperature limit $T3$ is surpassed.

It is assumed that once a valve gets stuck, it cannot work at high performance until being restored or replaced. Therefore in both cases, whether the system fails or recovers from an error i.e., the temperature returns below $T1$, a maintenance request will be raised. The system can be restarted after maintenance (state *Sys_mnt*).

Qualitative analysis

In this case study, several subsystems can be executed concurrently. Thus, multiple signals may change simultaneously, which may lead to SIC-testability issues.

By observing the individual models, it can be readily found that some states have the same output actions. For example, some states in the model *System Status* do not have an observable action. This implies that after composition there might be some locations that have same actions, leading to an observability issue.

This system is neither of large scale nor of complex structure, so controllability might not be problematic. However, the controllability performance can always be improved

according to user requirements.

Applying the DTT approach

The stabilized composed automaton (SCA) of this system contains 35 stable states and 323 evolutions.

Analyzed by the DTT approach, 34 out of the 35 states are not fully SIC-testable. To solve this issue, 9 T-guards are added to the specification models. In Fig. 20, the added T-guards are drawn in blue.

In the SCA, 9 states share the same output actions with other states. Analysis done with the DTT approach shows that the observability issue was caused by 3 locations in the individual models. After adding 2 O-actions (drawn in purple in Fig. 20), all states are directly distinguishable.

Finally, according to the C-guard method results, the initial controllability of this system is 5. Although this seems not a very bad value for controllability, the DTT approach can help to reach a better performance. After adding 1, 8 and 19 C-guards, the controllability is improved to 4, 3 or 2 steps, respectively. The one C-guard (for the controllability of 4 steps) is drawn in green in Fig. 20.

Influences on executable code generation

The automatically generated PLC structured text (ST) code from the initial models of the case study contains 177 lines. Adding the 9 T-guards only increases the code length by 1 line: declaration of the Tg variable. In addition, 9 lines of code are modified by adding '*AND Tg*' to existing guards. An example of ST code for T-guard assignment is given as follows:

- $tS11 := X(s_{1\_1})$ *AND V1_stuck AND Tg ;*

Adding the two O-actions increases the code length by 4 lines: 2 lines to declare outputs variables and 2 lines to assign the conditions when the O-actions are activated.

An example of ST code for O-action assignment is given as follows:

- $oa1 := X(s_{1\_2})\ OR\ X(s_{1\_1})$;

Adding 1 C-guard increases the code length by 2 lines: 1 line to declare the added input variable, and 1 line to add the new transition with 'ctr_1'. In addition, another line is modified by adding '*AND NOT* ctr_1'. An example of ST code for a new transition with C-guard is given as follows:

- $tV21 := X(v_{2\_3})\ AND\ ctr_1$;

Thus, concerning the design overhead added by the DTT methods, it can be positively concluded that the length of added/modified code is linear to the number of inserted O-actions, C-guards, and T-guards.

5.6.2 A manufacturing cell

The second case study is a manufacturing cell adapted from [15].

Description of system

As presented in Fig. 21, this system contains nine machines: four robots, two fixtures, two turntables and a conveyor.

This cell does welding tasks in three phases. At the beginning, *Conveyor-1* delivers a car body into the cell. *Robot-1* begins to weld the parts, which are loaded by previous systems. This is the first weld job (J1). Meanwhile, *Robot-2* picks plates from *Turntable-1* and places them in *Fixture-1*. *Turntable-1* turns when two plates have been taken. In the second job (J2), *Robot-1* and *Robot-2* work together to weld the plates held by fixture to the car body. After they finish J2, *Fixture-1* moves away from its workstation, to enable *Robot-1* and *Robot-2* weld the parts which were blocked by *Fixture-1*. This is the third Job. On the opposite side, the same work will be executed by *Robot-3*, *Robot-4*, *Turntable-2* and *Fixture-2*. As soon as the weld jobs are completed, the robots also move

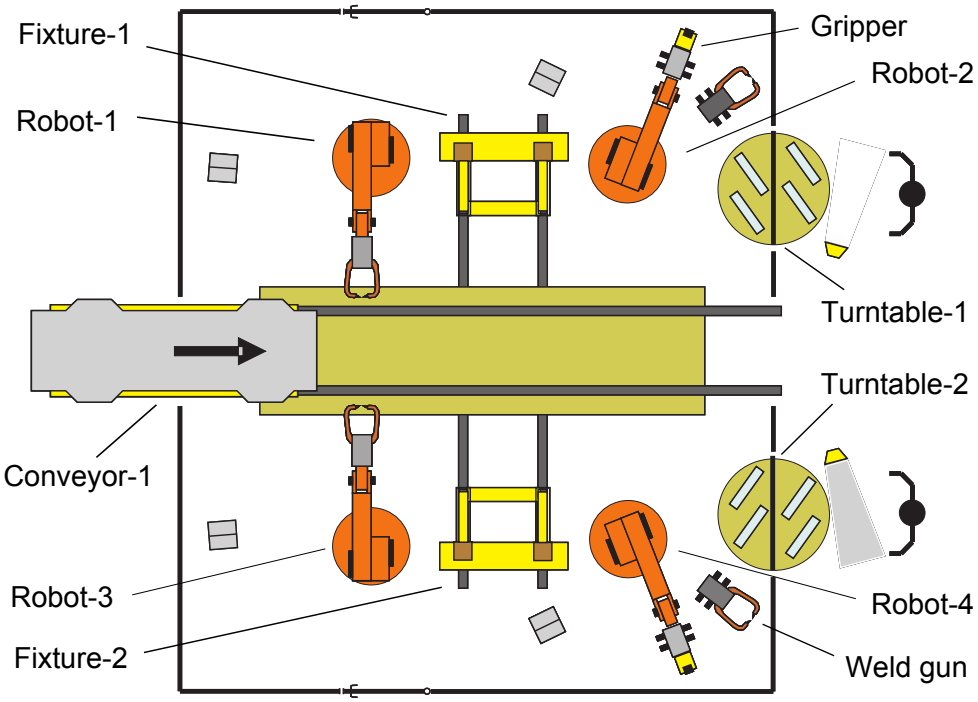


Figure 21: Case study: a welding and material handling cell

away from their workstation. *Conveyor-1* delivers then the car body out of the cell. Afterwards, the robots and fixtures move back to their workstations, making the cell ready for next round.

Modeling of system

To synchronize the different subsystems, a few coordinators have been set up to control the correct operation of all individual machines and robots. The complete system can thus be modeled with 12 individual Stateflow models with 34 Boolean inputs and 33 Boolean outputs. A selection of inputs and outputs for *Robot-2* is given in Tab. 4.

The model for *Robot-2* and *Turntable-1* are selected as illustrative examples and presented in Fig. 22. Again, the initial models are depicted in black; blue, green and purple drawings and text correspond to the guards/outputs added by the DTT approach.

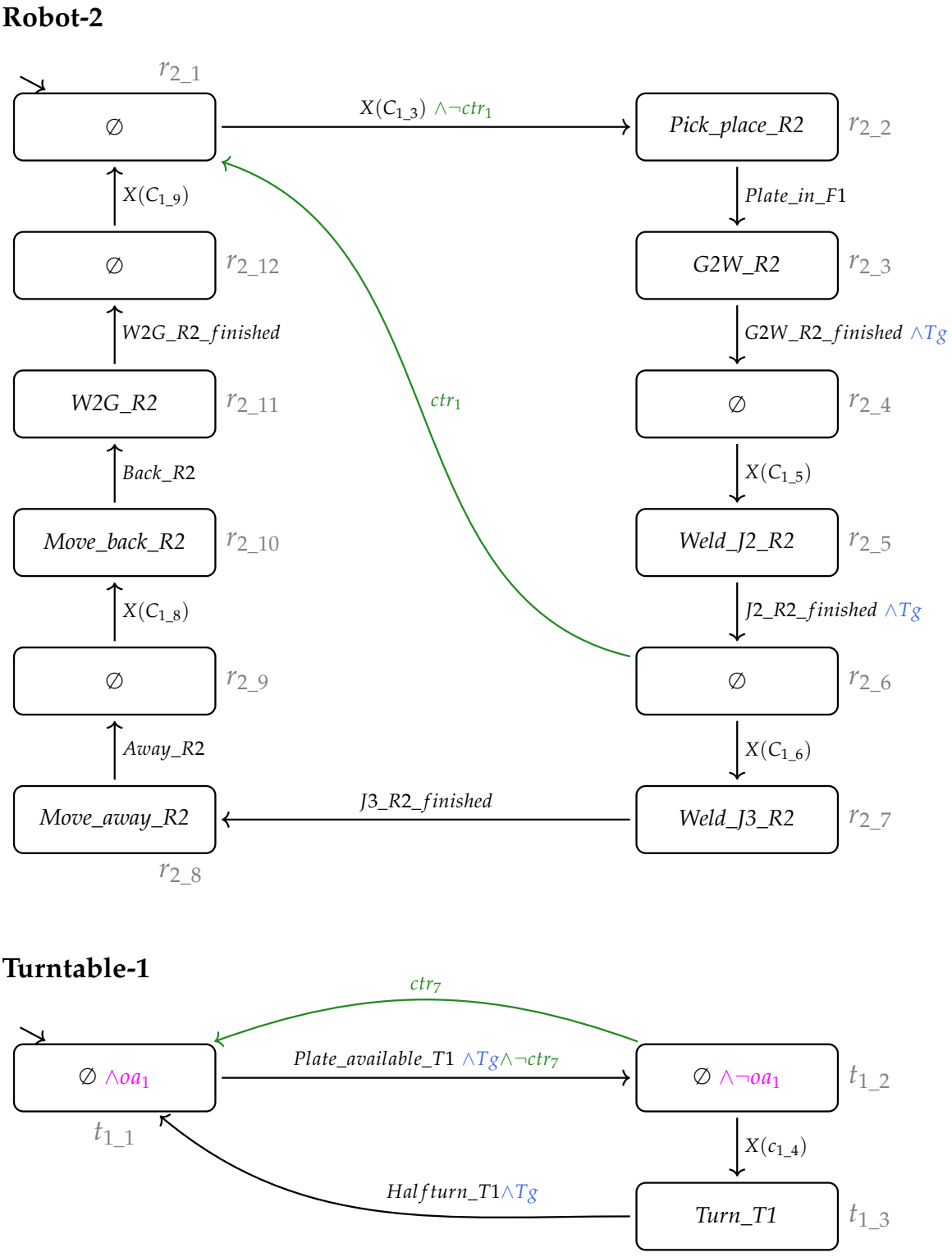


Figure 22: A Moore machine model for Robot-2 and Turntable-1

Table 4: Inputs & outputs for the models *Robot-2* and *Turntable-1*

Input	**Description**
*Plate_in_F*1	activated when a plate is placed in Fixture-1
G2W_R2_finished	activated when Robot-2 finishes tool change G2W
W2G_R2_finished	activated when Robot-2 finishes tool change W2G
J2_R2_finished	activated when Robot-2 finishes Job2
J3_R2_finished	activated when Robot-2 finishes Job3
*Away_R*2	activated when Robot-2 is away from workstation
*Back_R*2	activated when Robot-2 is back to workstation
*Plate_available_T*1	activated when a plate is available at Turntable-1
*Halfturn_T*1	activated when Turntable-1 finishes the turn by half circle
Output	**Description**
*Pick_place_R*2	pick the plate and place it in Fixture-1
*G2W_R*2	change tool from gripper to weld gun
*W2G_R*2	change tool from weld gun to gripper
*Weld_J*2_*R*2	do the Weld-Job2
*Weld_J*3_*R*2	do the Weld-Job3
*Move_away_R*2	move away from workstation
*Move_back_R*2	move back to workstation
*Turn_T*1	turn Turntable-1 by half cirlce

Qualitative analysis

When observing the individual models, it can easily be found that some locations have the same output actions. For example, the locations t_{1_1} and t_{1_2} in model *Turntable-1*, and r_{1_1}, r_{1_4}, r_{1_6}, r_{1_9}, and r_{1_9} in model *Robot-2* don't have any observable action. This implies that, after composition, it is possible that some states have the same outputs, which leads to the observability issue.

Since the system contains many subsystems which run in parallel, multiple signals are likely to change at the same time. Thus, the SIC-testability might be an issue.

With so many subsystems, the SCA can contain a large number of states, and therefore the distance from some states to other states could be very long, i.e., this system might suffer from the controllability issue.

Applying the DTT approach

The SCA of this case study contains 792 stable states and 93,587 evolutions. With the help of DTT-MAT, quantitative results can be automatically obtained for the testing issues.

Out of 792 states, 777 of them are not fully SIC-testable. Applying T-guard method, one feasible solution is found, 12 Boolean inputs out of 34 are involved in non-SIC-testable transitions guards. After updating the original specification models with 14 T-guards, the composed machine reaches a full SIC-testability. Two of the T-guards have been drawn in blue in Fig. 22.

Then, 537 states have the same output action with at least one of the other states. With traditional methods, a distinguishing sequence may be 537 steps long (in the worst case). Analysis with the O-action method shows that the observability issue of 537 states was caused by 4 locations in the individual models. After adding two O-actions (oa_1 is drawn in purple in Fig. 22), all states in the SCA are directly distinguishable in one step.

Finally, according to the C-guard method results, some states are not reachable from some other states, i.e., the path cost between a pair of those states is infinitely large. After adding 14 C-guards on individual models (two of them have been drawn in green in Fig. 22), any state can be reached within maximum 4 steps from any other state in the composed model. It is worth noting that the number of added C-guards is calculated with the proposed optimized algorithm according to user requirements. For example, after adding 12 or 22 C-guards in this case study, users can obtain the controllability of 5 or 3 steps, respectively.

Result of executable code generation

The automatically generated ST code for PLC from the initial models of the case study contains 349 lines. After adding 14 T-guards, 2 O-actions, and 14 C-guards, 33 lines have been added and 28 lines have been modified. The code is generated in the same way as in the first case study, and thus not explicitly presented here.

This is another practical evidence that with the DTT approach, the length of added/-modified PLC ST code is linear to the number of inserted O-actions, C-guards, and T-guards.

5.7 Summary of the DTT approach

This chapter has presented a design-to-test (DTT) approach for programmable controllers in critical automation systems, which aims at improving the testability and reducing the testing overheads with limited design overhead.

Firstly, system specifications are modeled as Moore machines extended with Boolean signals. By running the T-guard, O-action and C-guard methods, the specification models are modified so that they fulfill the requirements of full SIC-testability, full observability and better controllability. Two case studies have been used to illustrate the application.

It is worth underlining that with the proposed DTT approach, during normal execution, all T-guards and C-guards can be inhibited (by connecting them to the logic 1 and 0 levels, respectively), and all O-actions are only additional output signals that can be ignored. Thus, none of the added T-guards, C-guards and O-actions affect the nominal behavior of the system in its normal mode.

6 Testing with plant features

6.1 Introduction

In this chapter, we propose a model-based test generation approach for programmable controllers that aims at reducing the length of a test sequence by applying plant features (PFs). The PF approach does not require detailed or full knowledge of the plant behavior of a system under test (SUT), but it can achieve 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 PF approach is illustrated with two large scale case studies in this chapter.

6.2 Core idea

Test generation with plant features is a model-based test generation approach that guarantees full coverage of nominal system behavior with a shortened test sequence. Additionally, faulty system behavior can also be included upon need. The core idea is to involve not only specification models but also plant features in the test generation.

In an automation system, physical elements such as sensors and actuators are usually considered as *plants* while controllers are implemented according to *specifications*, i.e., formal descriptions of user requirements. Plant features are extracted from simplified plant models and thus require a limited design effort. As a result, the number of

generated test cases and the length of a test sequence could be significantly reduced, and therefore, large-scale systems can be tested efficiently. Compared to coverage-oriented testing, the obtained set of test cases is not reduced stochastically but is selected in a way that guarantees full coverage of nominal behavior of a system.

The idea of modeling and using plant features to reduce test cases was proposed in [1], while the algorithms of applying plant features in test generation, more specifically, in its late phase, were presented in [2]. Compared to [1] and [2], the main contribution of this chapter is: applying plant features in very early phase of test generation, so that the state space throughout the whole computation is also significantly shrunk, and the length of a final test sequence is further shortened.

6.3 Description of signal relations

Signal relations can be expressed through different formal and informal languages. In the following parts three methods are used to describe two basic types of signal relations that are presented in Fig. 23: $i_1 - i_2$ and $i_3 - i_4$.

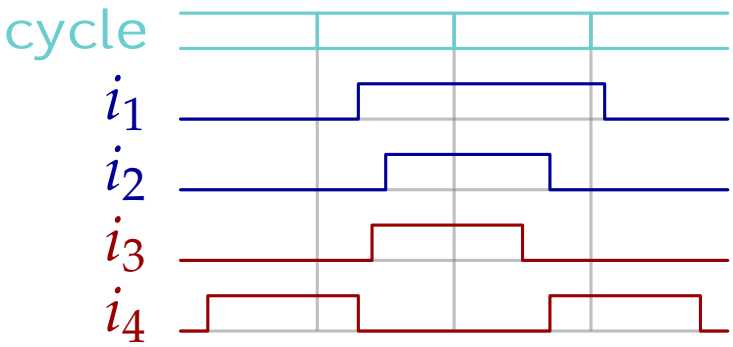


Figure 23: Two basic types of signal relations

Natural language

With natural language, the two types of signal relations can be organized as follows:

- Signal i_2 can only be *True*, when i_1 is *True*, i.e., i_1 is premise of i_2.

- Signal i_3 and i_4 are mutually exclusive, i.e., at the same time, only one of the signals i_3 and i_4 can be *True*.

Temporal logic

Tw popular forms of temporal logic languages, linear temporal logic (LTL) and computational tree logic (CTL), are applied to depict the signal relations:

- *LTL*: $G(\neg i_1 \rightarrow \neg i_2)$
  CTL: $AG(\neg i_1 \rightarrow \neg i_2)$

- *LTL*: $G\neg(i_3 \wedge i_4)$
  CTL: $AG\neg(i_3 \wedge i_4)$

Finite state machine

The modeling language finite state machine (FSM) can also be used to formalize the signal relations, as introduced in chapter 4 and also presented in Fig. 24.

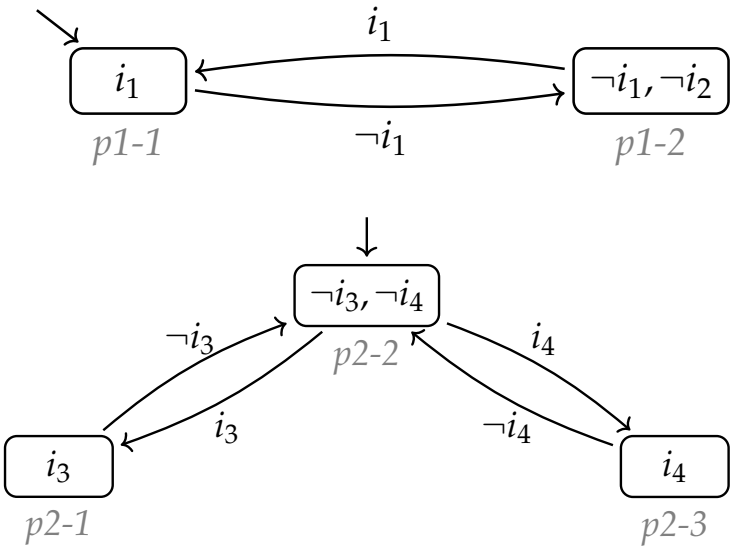


Figure 24: Representation of the premise relation and mutual exclusion of signals with FSM

It is worth noting that signals values can be freely assigned if they do not appear in the initial state, i.e., the output of location *p1-1* can either be (i_1, i_2) or $(i_1, \neg i_2)$.

In fact, the two basic types of signal relations, premise and mutual exclusion, can be combined to construct complex signal relations when involving several signals.

For example, signals in a system can have such behavior: if a is *True* and remains *True*, and b becomes *True*, then c can be *True*; once a becomes *False*, c turns to *False* as well. With CTL the signal relations can be expressed as: $AG\big((\neg a \rightarrow \neg c) \wedge (a \wedge b \rightarrow AFc)\big)$. The same signal relations can be modeled as FSM, as presented in Fig. 25.

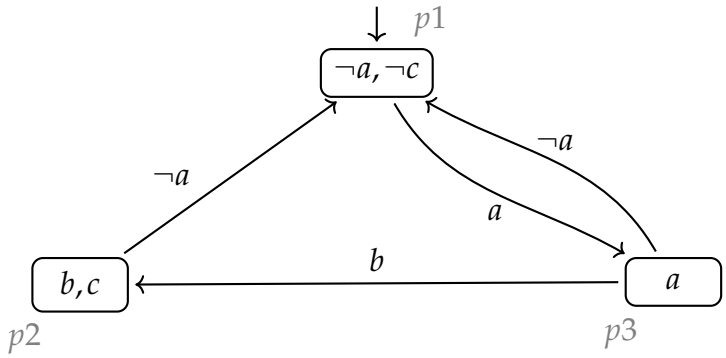


Figure 25: A simple example of multiple signal relations

In brief, natural language is well capable to handle small scale systems with a limited number of signals. For large scale systems, it is recommended to use one of the formal methods, i.e., temporal logic and/or FSMs.

6.4 Framework of test generation with plant features

As introduced in chapter 4, a model-based testing process for a programmable controller consists of four steps: test generation, feed the input sequence, execute the program on the controller, observe and compare the output sequence. The focus of this chapter lies in the first step: construction of a *test sequence*, which is presented with more details in Fig. 26.

The yellow blocks in Fig. 26 correspond to a classic process of test generation, i.e., complete conformance testing, which has been presented in [22], while the gray, blue,

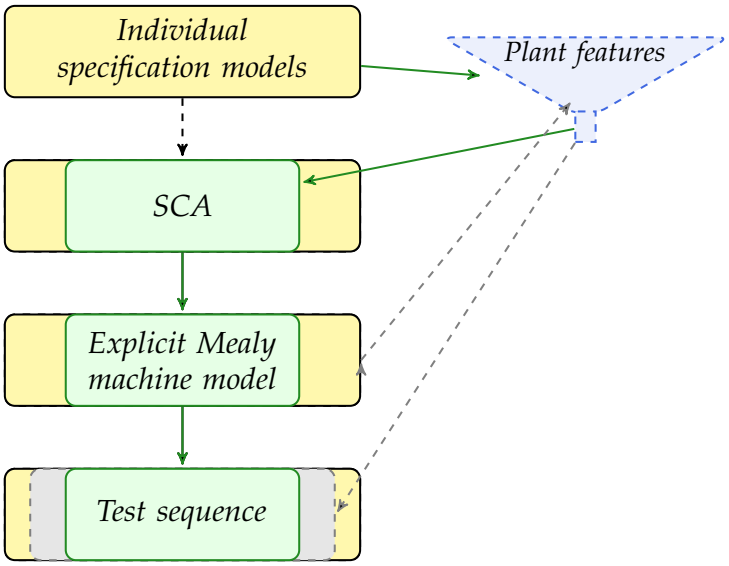


Figure 26: Framework of involving plant features in the test generation. Yellow blocks: generation of complete testing; Gray block and arrows: earlier version of test generation with plant features ([1], [2]); Green blocks and arrows: current version of test generation with plant features.

and green blocks correspond to the test generation with plant features.

As discussed in chapter 4, the length of a test sequence is determined by two factors: the number of test cases, and the ordering and repetition of test cases.

Regard to the first factor, in a large scale system, when the number of inputs of an SUT grows linearly, the sizes of stabilized composed automaton (SCA) and Mealy machine model grows exponentially, and therefore the number of test cases also grow exponentially. This fact leads to the well-known *state space explosion* issue, which was the motivation of involving plant features in the test generation.

The first version of test generation method with plant features was proposed in [1] and [2], in which plant features are used after the Mealy machine model has been generated (see Fig. 26). As a result, the number of test cases is remarkably reduced, and consequently the length of the generated test sequence is also remarkably shortened.

In the current version, plant features are applied early in the generation of SCA (see Fig. 26, and the algorithms are presented in the next section). The new and additional advantages with regard to [1] and [2] 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 mentioning that, this method does not require very detailed or full plant models, but only fragments of knowledge from plant models. Of course, the more plant features can be modeled, the greater reduction to the length of the test sequence can be achieved.

Additionally, this method can also be combined with the idea of *fault injection*. Users can insert a set of selected faults into the target behavior of an SUT by modifying plant models. Examples can be found in [2].

6.5 Test case generation with utilization of plant 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 27, specification and plant constitute a closed-loop. More specifically, plant is controlled by specification, directly as for actuators and indirectly as for other parts; while the reachable space of specification is also influenced by plant on the other hand.

In this chapter, 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), 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.

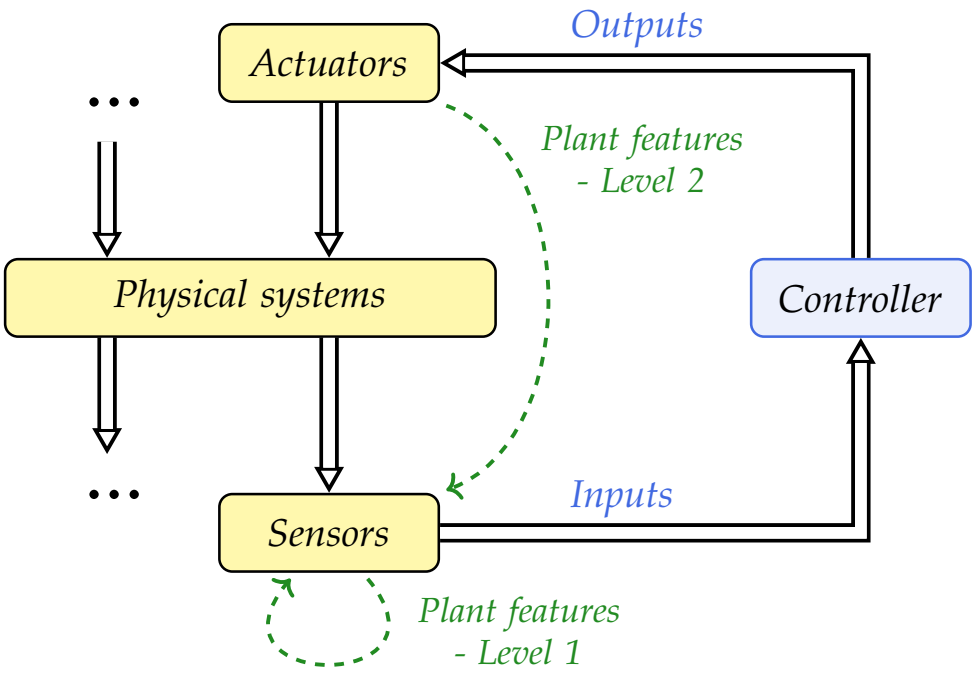


Figure 27: Specification and plant in an automation system

6.5.1 Level 1: Signal relations among sensors

Alg. 5 presents the algorithm to consolidate plant features from plant models of level 1. The plant model given in Fig. 28 is used as a simple example to help illustrate the algorithm.

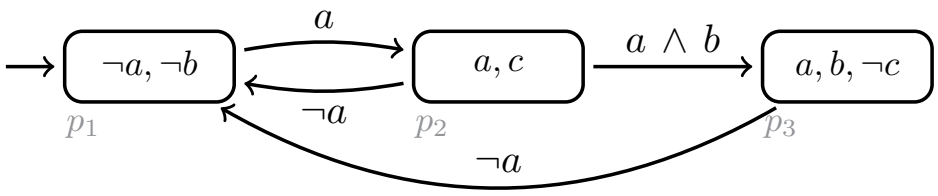


Figure 28: Example: plant model of level 1

L_P and Λ_P are the inputs of the algorithm, and represent the set of locations and outputs in a plant model, respectively. It is worth noting that the outputs and transition guards[1] of plant models are constituted by inputs from specification models.

[1]For the current version, transition guards in plant models level 1 are not used.

Algorithm 5: Consolidating plant features of level 1, i.e., signal relations among sensors

Input: L_P, Λ_P
Output: PF

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

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. 28, for location p_1, a and b should be both *False*. This model contains another input c, 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 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. 28, it applies that in a nominal behavior, at least one of the following three conditions should be fulfilled: a and b be both *False*; a and c be both *True*; a be *True*, b be *True*, and c 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 of level 1 is assigned $GLOBAL$ (line 9).

6.5.2 Level 2: Signal relations among sensors and actuators

The algorithm for consolidating plant features from plant models of level 2 is presented in Alg. 6. The plant model given in Fig. 29 is used as a simple example to help illustrate the algorithm.

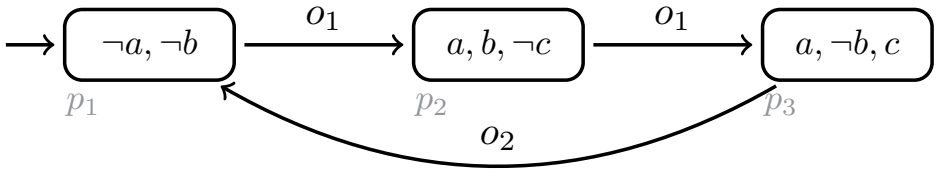


Figure 29: Example: plant model of level 2

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 model, respectively. The outputs of plant models are also constituted by inputs from specification models, same as for level 1. The difference is that, transition guards in plant 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. 29, for location p_1, a and b should be both *False* while c can be either *True* or *False*.

A pair of location and transition in a plant 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. 29, the first plant feature candidate, i.e., for location p_1, has the condition $\neg a \wedge \neg b$ and the

Algorithm 6: Consolidating plant features of level 2, i.e., signal relations among sensors and actuators

Input: $L_P, \lambda_P, \Delta_P, G_{P,\delta_P}$
Output: PF

1 **begin**
2 $PF\_temp := \emptyset$; $PF\_rm := \emptyset$; /* initialization */
3 **foreach** $\delta_P \in \Delta_P \mid l_{P,src} \times g_{P,\delta_P} \rightarrow l_{P,des}$ **do**
4 $\lambda_{PF,l_P} := True$; /* initialization */
5 **foreach** $\lambda_P \in \Lambda_P(l_{P,des})$ **do**
6 $\lambda_{PF,l_P} := \lambda_{PF,l_P} \wedge \lambda_P$;
/* the signal constraints in one location need to be fulfilled at the same time */
7 $pf.cond := \lambda_{PF,l_P}$;
8 $pf.scope := g_{P,\delta_P}$;
/* the plant features of level 2 are valid only for the states of SCA which hold the relevant actions */
9 $PF\_temp := PF\_temp \cup \{pf\}$;
10 **foreach** $pf\_ref \in PF\_temp$ **do**
11 **foreach** $pf\_cpr \in PF\_temp \backslash PF\_rm$ **do**
12 **if** $pf\_ref.scope = pf\_cpr.scope$ ***and*** $pf\_ref.cond \neq pf\_cpr.cond$ **then**
13 $pf\_ref.cond := pf\_ref.cond \vee pf\_cpr.cond$;
/* if an action can leed to different pf conditions, merge them with `OR' (ref: reference; cpr: compare) */
14 $PF\_rm := PF\_rm \cup \{pf\_cpr\}$;
/* save the used and redundant plant features in the set PF_rm (rm: remove) */
15 **foreach** $pf \in PF\_temp \backslash PF\_rm$ **do**
16 $PF := PF \cup \{pf\}$;

scope as o_2.

The second part of Alg. 6 deals with issue that a scope of a 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. 29, two plant feature candidates have the same scope o_1, and different condition, $a \wedge b \wedge \neg c$ and $a \wedge \neg b \wedge c$. The two candidates are merged into a final plant feature that has the scope o_1 and the condition $a \wedge (b \wedge \neg c \vee \neg b \wedge c)$.

6.5.3 Test case generation with fault injection

Fault injection is a class of testing techniques which involves faulty behavior supplementary to nominal behavior testing. The faults to be tested are usually selected based on expert knowledge and practical experience. For example, some components might be more error-prone in some environment, and some sensors might have physical interference with other sensors or actuators. More fault injection knowledge and techniques in the field of testing can be found in [113].

In this chapter, fault injection can be realized conveniently by modifying plant models. By definition, fault models are in conflict with the plant feature models described in Sec. 6.5.1 and Sec. 6.5.2. Thus, to consider these fault models, plant feature models have to be modified to be less restrictive. This implies that more test cases outside of the nominal behavior will be considered for testing.

6.5.4 Applying plant features in the generation of SCA

The generation of SCA is done by synchronous composition of individual specification models with stability search. The detailed process has been presented in [22] and is not repeated in this thesis.

With the new method proposed in this chapter, plant features are applied in the generation of SCA as introduced in Sec. 6.4. The difference compared to the process in [22] 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. 7.

Algorithm 7: 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 ∈ PF do
4          if pf.scope = GLOBAL then
5              g_{S,e,wP} := g_{S,e,wP} ∧ 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} ∧ pf.cond;
```

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

For plant features of level 2, first 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 $\{o_1, o_2\}$.

6.6 Case studies

In this chapter, two large scale case studies are presented to illustrate the test generation approach with PFs.

6.6.1 A logistics system

The first case study is a logistics system adapted from the didactic platform presented in [114]. The modules of interest in our case study are displayed in Fig. 30.

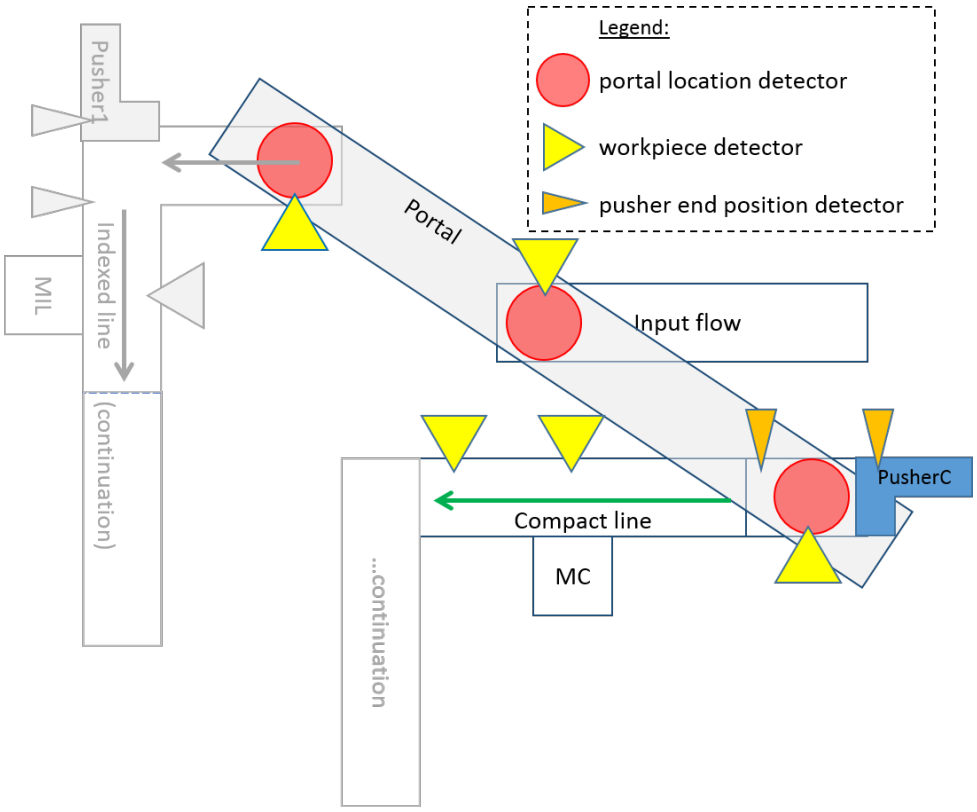


Figure 30: Case study: a logistics system containing a portal and two subsequent lines *(top view)*

System description

The *portal* transports workpieces from the *input buffer* to either the *compact line* or an indexed line. In this case study, the specification and plant behavior of the portal and

compact line are analyzed and presented. The compact line contains a vertical buffer with a pusher, a conveyor belt and one machine station. Several location sensors are used to sense the position of the workpiece (yellow triangles), the pusher (orange longish triangles) and the portal (red circles).

Five FSM models have been used for the specification models of the system under consideration. In Fig. 31, three specification models for the portal and compact line are given as examples. It is displayed, that the portal can move horizontally between three positions *In*, *IL* and *CL*. Only in those positions it can move up and down. Finally, only in the down end position it can activate the electromagnetic gripper to lift a workpiece or deactivate it (ungrip) in order to release a workpiece, respectively. On the compact line, a workpiece is brought to a machine via the belt; after the machining the workpiece is delivered to the output.

In total, 15 inputs and 9 outputs are considered, as listed in Tab. 5.

Complete test case generation

Applying Teloco [22], the SCA of the five specification models contains 221 states and 8,524 evolutions. Since the system has 15 inputs, the Mealy machine of the SCA contains $221 * 2^{15} = 7{,}241{,}728$ evolutions.

Based on that, an executable test sequence is obtained with 9,647,120 steps.

Test generation with plant features

Plant models are built based on domain knowledge of the system under test, i.e., through identifying the relations among sensors and actuators. For this case study, 10 plant models have been built. As illustrative examples, four plant models related to the motion of the portal, and signal relation between a workpiece and the machine, are presented in Fig. 32.

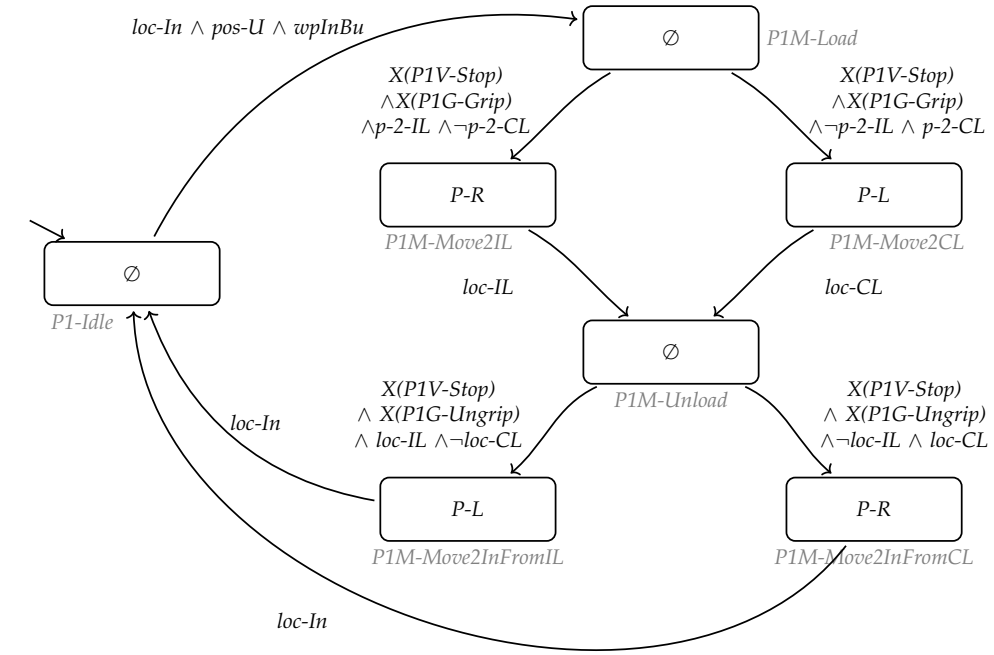
The first model in Fig. 32 presents a *mutual exclusion* relation between two input signals, i.e., plant features of level 1. In a nominal behavior, *pos-U* and *pos-D* should not be true

Table 5: Table of inputs & outputs for the portal, belt and machine on the compact line

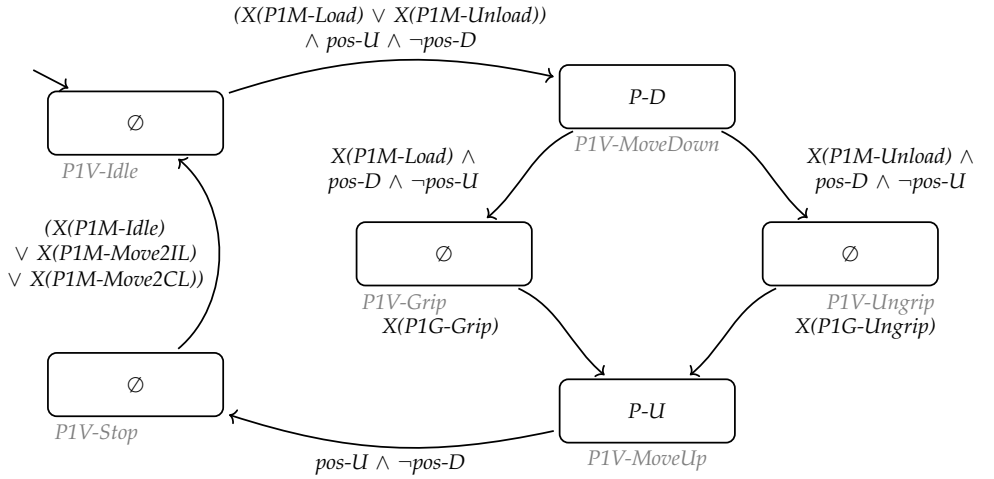
Input	Description
wpInBu	*True* when a workpiece is in the input buffer
loc-IL	*True* when the portal is at the drop location of indexed line
loc-In	*True* when the portal is at the input buffer location
loc-CL	*True* when the portal is at the drop location of compact line
pos-U	*True* when the portal is in its up end position
pos-D	*True* when the portal is in its down end position
p-2-IL	*True* when the current work piece should be brought to indexed line
p-2-CL	*True* when the current work piece should be brought to compact line
wpMC	*True* when the workpiece reaches the expected position in front of the machine
MC-done	*True* when the machine finishes its machining
wpCOut	*True* when the workpiece reaches the output position of compact line
wp-Output	*True* when the command of outputting the workpiece is received
wpCLBu	*True* when a workpiece is in the buffer of compact line
pos-E-PC	*True* when the pusher of compact line is in its extended position
pos-R-PC	*True* when the pusher of compact line is in its retracted position

Output	Description
P-G	activate the gripper
P-U	move the portal upwards
P-D	move the portal downwards
P-R	move the portal to the right (towards IL)
P-L	move the portal to the left (towards CL)
BC-P	run the belt of compact line in positive direction
MC	run the machine of compact line
PC-F	move the pusher of compact line forwards
PC-B	move the pusher of compact line backwards

Specification model - Portal Horizontal



Specification model - Portal Vertical



Specification model - Belt

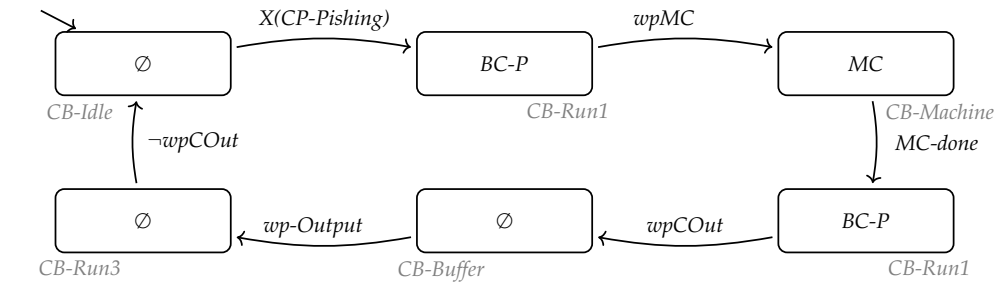
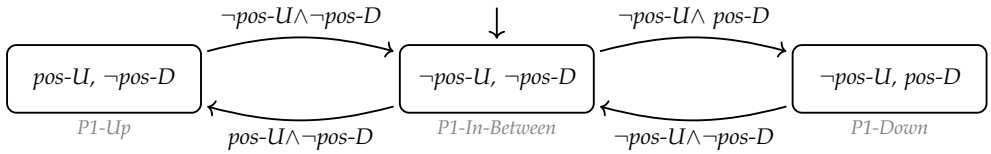
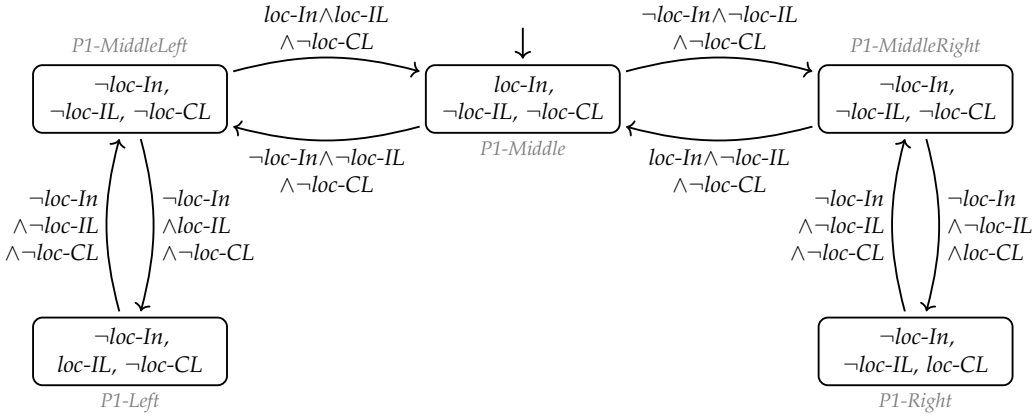


Figure 31: Specification models for the horizontal portal movement, the vertical portal movement and the belt of the compact line

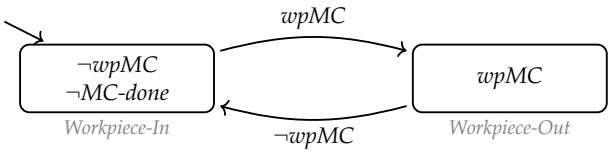
Plant model - Portal-vertical (Level 1)



Plant model - Portal-horizontal (Level 1)



Plant model - Machine-Workpiece (Level 1)



Plant model - Portal-Up-Actuator (Level 2)

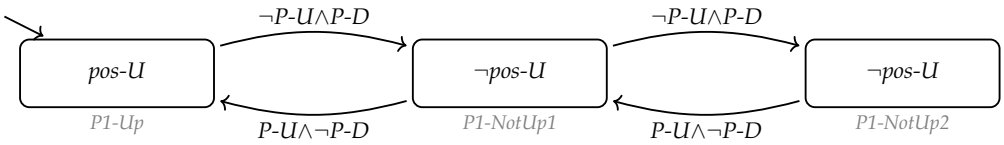


Figure 32: Plant models for the nominal behavior of the vertical and horizontal portal movement and the machine on the compact line

at the same time, since a portal cannot be simultaneously in its *up* and *down* position.

The second model presents a similar *mutual exclusion*, but among three input signals instead of two. At one moment, the portal can physically only be in one location, i.e., only one or none of the three signals *loc-In*, *loc-IL* and *loc-CL* can be *True*.

The third model presents a *premise* relation between two input signals. The input *MC-done* can only become *True* when the input *wpMC* is *True*, which means the machine does only operate when the workpiece is at the expected position in front of the machine.

The fourth model presents a relation among input and output signals, i.e., plant features of level 2. The input *pos-U* remains *True* unless the output *P-D* is activated (and *P-U* is not active). At the same time it is stated that - reading the model from right to left - *pos-U* will not instantaneously but eventually be *True* when *P-U* is activated. Note that it is explicitly not stated that *pos-U* will be true directly after the portal movement has been activated, i.e. from location *P1-NotUp2*, *pos-U* will first remain *False* (as in location *P1-NotUp1*), and will eventually become *True* (as in location *P1-Up*). Analogously, a feature for the down movement can be found, which is also used in the test generation but not presented here.

It is worth mentioning that modeling of plant features is done with human effort. Then, the rest process from composition of specification models to the generation of test sequences are all executed by the tool automatically. In addition, even some simple fragments of the nominal behavior of the system under test contribute to the reduction of test cases; of course, the more plant features are modeled, the higher reduction is obtained.

Combining the ten plant models with five specification models in the test generation, the newly obtained SCA contains 204 states and 3,036 evolutions. The newly generated Mealy machine contains 341,504 evolutions. The final executable test sequence is generated with 448,752 steps.

Comparison of results

The test generation results of the two methods are presented in Tab. 6.

It can be stated that integrating knowledge about signal relations into the generation process drastically reduces the length of generated test sequences. The cycle time of a programmable controller in practice is supposed to be approximately 10ms (can vary from 1ms to 100ms in various applications). By applying plant features in the test generation, the test execution time can be reduced from 26.8 hours to 1.2 hours.

Table 6: Results and comparison of test generation methods on the case study of compact line

Generation method	Size of SCA		#evol in the Mealy machine	Length of test sequence
	#state	#evol		
Complete testing	221	8,524	7,241,728	9,647,120
With plant features	204	3,036	341,504	448,752
- Comparison -	-7.7%	-64.4%	-95.3%	-95.3%

6.6.2 A flexible manufacturing system

The second case study is a flexible manufacturing system (Fig. 33) originally presented in [115].

Description of the system

As presented in Fig. 33, 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. 7.

New products enter the system with C1 and C2. C1 supplies blocks and C2 supplies

Table 7: 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 forward direction
Run_BW_C3	the conveyor belt C3 runs in 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

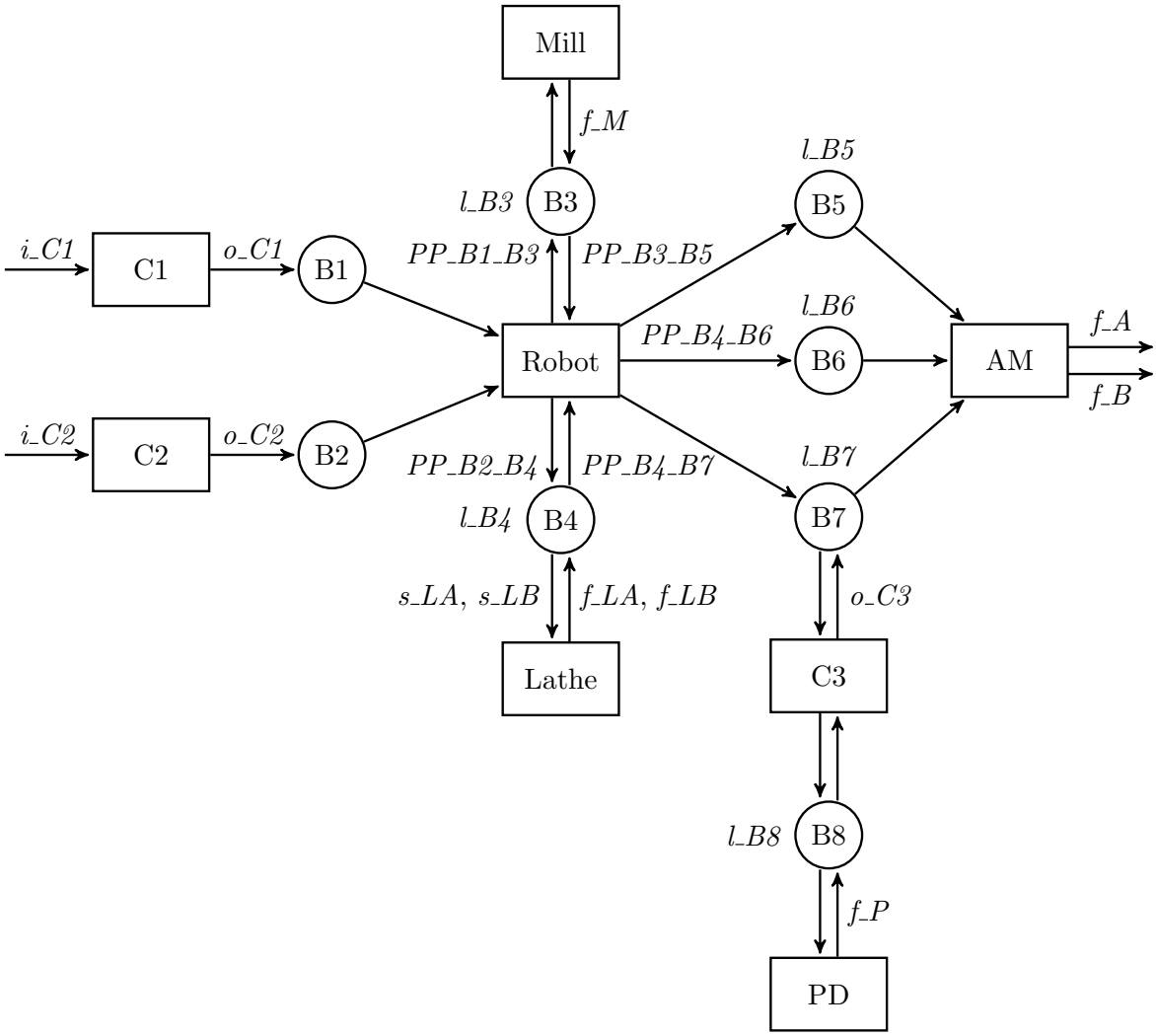
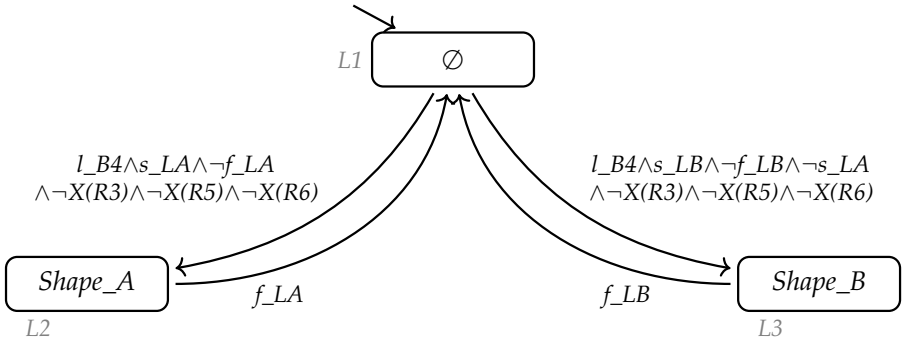


Figure 33: Case study: a flexible manufacturing system

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 also 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.

Seven Moore machines have been modeled for the specifications. For the sake of brevity, two models for the lathe and B4, and the robot are selected as illustrative examples and presented in Fig. 34.

Specification model - Lathe & Buffer4



Specification model - Robot

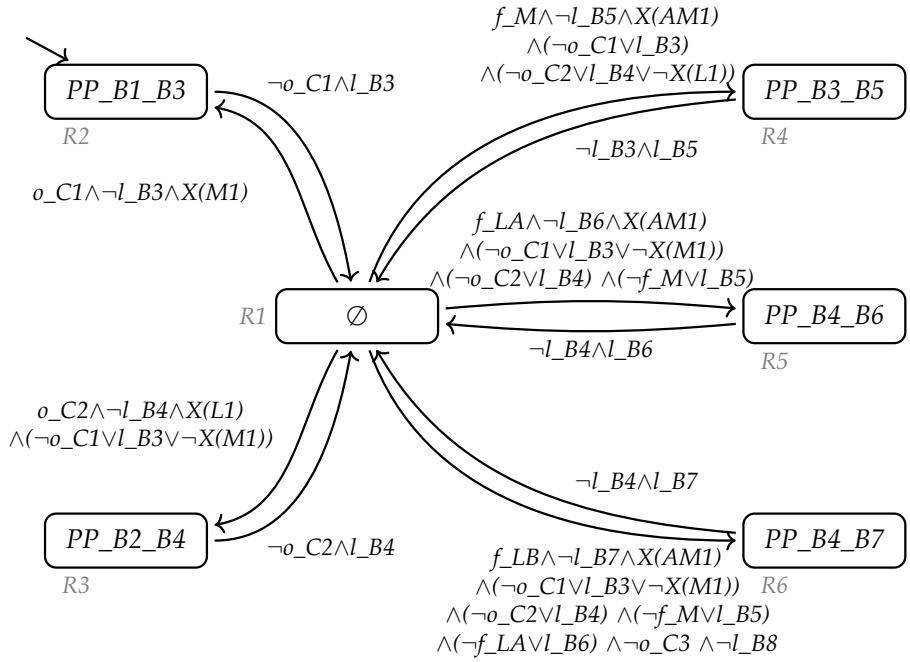


Figure 34: Specification models for two subsystems: *Lathe-Buffer4* and *Robot*

Complete test generation

Applying Teloco [22], 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 $1170 * 2^{19} = 613{,}416{,}960$ evolutions.

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

Test generation with plant features

Plant features are modeled by inspecting the physical structures and functional relations of the system. For the case study of FMS, 17 plant models have been built. As illustrative examples, the plant models for the lathe and B4, and the robot are presented in Fig. 35.

In *pl1*, i.e., the first plant 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 bewteen *f_LA vs. s_LA* and *f_LB vs. s_LB*, respectively.

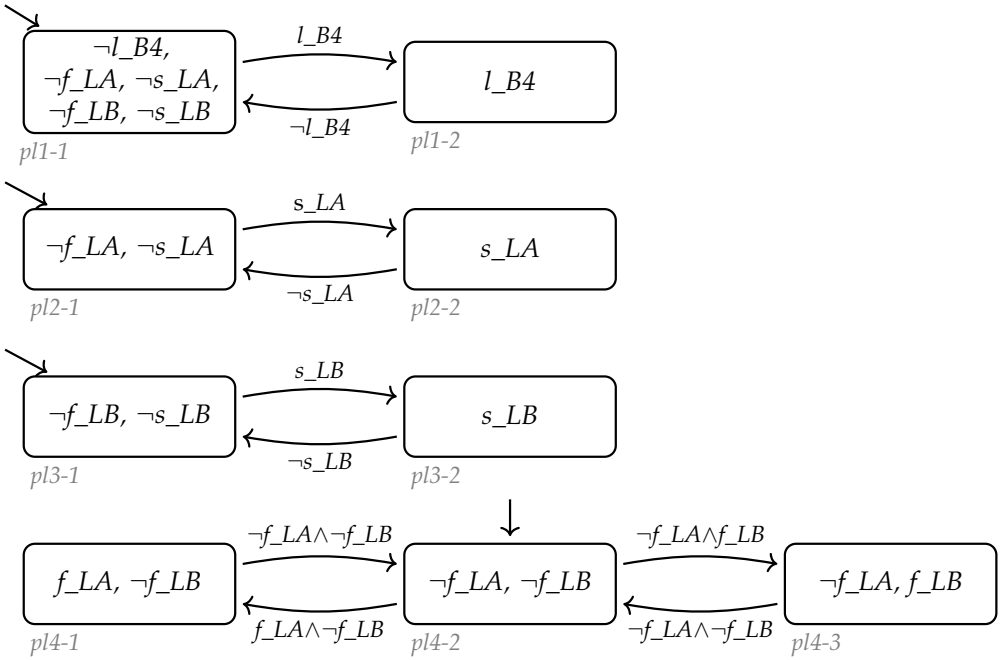
In *pl4*, *f_LA* and *f_LB* are mutually exclusive, since the lathe can not do both types of shaping operations simultaneously.

In *pr1*, i.e., the first plant 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 models with 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 evolutions. The final executable test sequence is

Plant models - Lathe & Buffer4 (Level 1)



Plant models - Robot (Level 2)

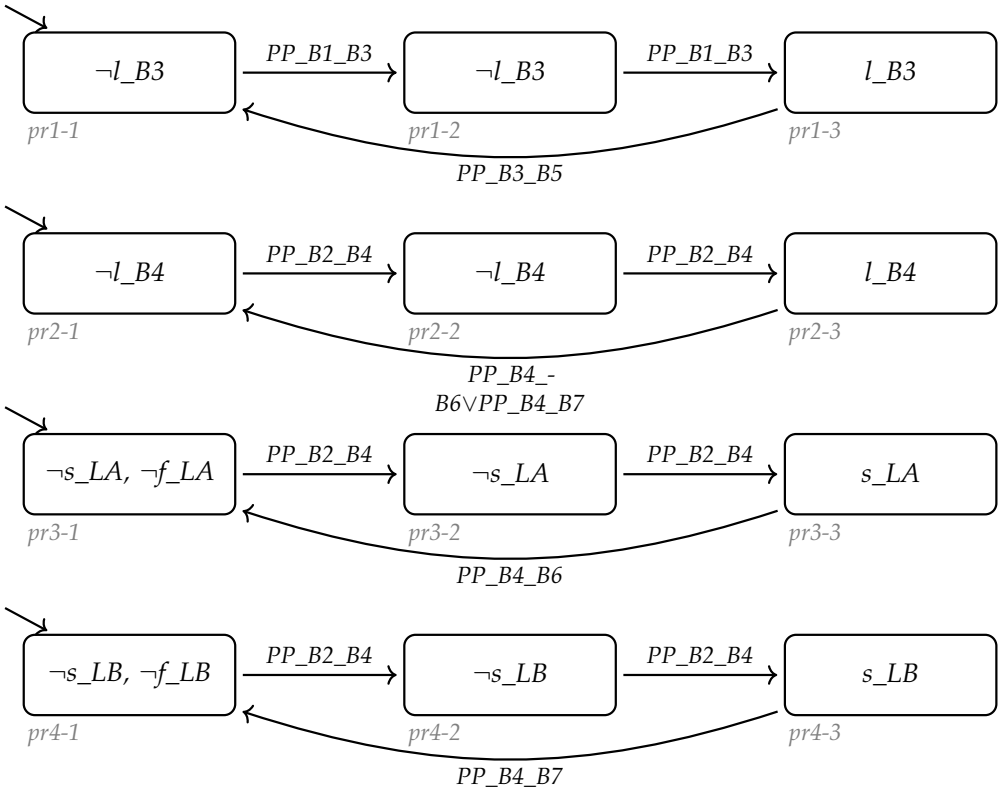


Figure 35: Plant models for two subsystems: *Lathe-Buffer4* and *Robot*

generated with 18,363,192 steps.

Comparison of results

The test generation results of the two methods are presented in Tab. 8. 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 complete testing. The cycle time of a programmable controller in practice is supposed to be approximately 10ms (can vary from 1ms to 100ms in various applications). By applying plant features in the test generation, the test execution time can be reduced from 2348 hours to 51 hours.

Table 8: Results and comparison of test generation methods on the case study of flexible manufacturing system

Generation method	Size of SCA		#evol in the Mealy machine	Length of test sequence
	#state	#evol		
Complete testing	1170	368,626	613,416,960	845,525,235
With plant features	970	134,637	12,514,080	18,363,192
- Comparison -	-17.1%	-63.5%	-98.0%	-97.8%

6.7 Summary of the PF approach

This chapter has presented a plant feature (PF) approach which aims at reducing the number of test cases / length of test sequence in test generation, 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.

Plant features are signal relations among sensors, and signal relations among sensors and actuators, which can be modeled as finite state machines or other formal languages. It is worth mentioning that this approach does not require detailed or full plant models. Any fragment of plant knowledge can contribute to the reduction. Additionally, users

can insert a selected set of faults into the target behavior to be tested by modifying the plant features.

The PF approach can be a good remedy for large scale systems where 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.

7 Conclusion and outlook

7.1 Conclusion

In this thesis, two innovative model-based test generation approaches have been presented: design-to-test (DTT) approach and plant feature (PF) approach. The two approaches are graphically reminded in Fig. 36, together with the classic complete conformance testing (CCT) approach as a reference.

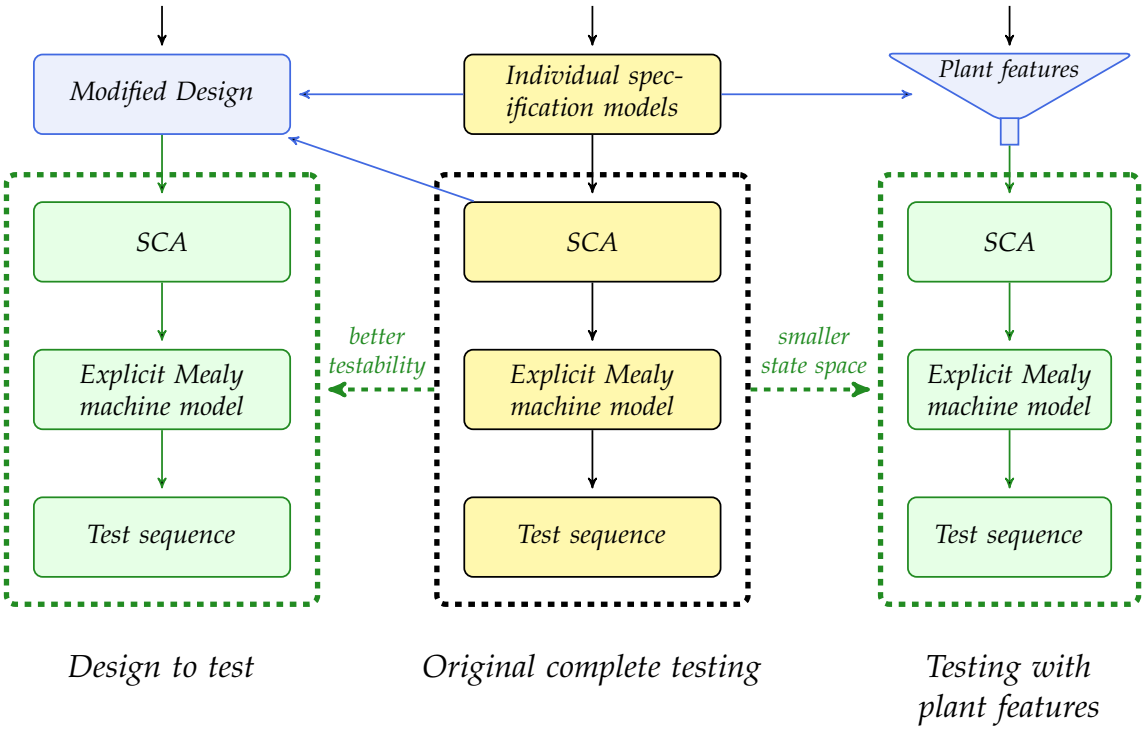


Figure 36: Frameworks of complete testing, design-to-test, and test generation with plant features

The CCT approach is structured and performed as follows. First, user requirements are modeled as individual Moore machine models with Boolean signals, which are then

composed into one stabilized composed automaton (SCA). Afterwards, an equivalent Mealy machine is derived from the SCA, which explicitly represents all Boolean conditions of evolutions by a set of minterms over the Boolean input set. In the last step, a test sequence is generated by solving the Transition Tour problem of the set of minterms from all states and all input values.

The DTT approach aims at improving the testability and reducing the testing overhead with limited design overhead. Applying the DTT approach, the SCA obtained from complete testing is analyzed in terms of single-input-change (SIC)-testability, observability and controllability. Based on the analysis result, a minimum number of C-guards, O-actions, and T-guards are automatically calculated, and added to the models. Then, the specification models as well as the finally obtained test sequence fulfill the requirements of full SIC-testability, full observability and better controllability.

From a global view of testing, the DTT approach focuses on 'how to test' aspect rather than 'what to test'. Though a complete exhaustive testing for a large/super large scale system is rarely scalable (e.g., billions of test steps), users can apply this approach on the most critical/important parts. A design-to-test MATLAB tool box (DTT-MAT) has also been presented in this thesis. To use it, specification models should be built in MATLAB Stateflow. Two industrial case studies are presented to illustrate the approach and the toolbox DTT-MAT.

It is worth mentioning that, during normal execution, all the added T-guards and C-guards can be inhibited (by connecting them to the logic 1 and 0 levels, respectively), and all the added O-actions are purely additional output signals that can be ignored for the control logic. Thus, none of the T-guards, C-guards and O-actions will affect the behavior of the system in its normal mode. Of course, when necessary, all these T-guards, O-actions and C-guards can be easily used again for testing purpose by properly setting their values. In maintenance and inspection, they can be very helpful to identify the problems occurring in the system.

The second approach, PF approach, aims at reducing the number of test cases / length of test sequence, by utilizing PFs extracted from a system under test (SUT). In the test generation, specification and plant features are both involved in the generation of SCA. In this way, the size of SCA, Mealy machine and test sequence can be much smaller, and test cases which represent unrealistic/unmeaningful situations are filtered out.

Meanwhile, the obtained shortened test sequence still achieves full coverage of the nominal behavior of the system under test.

For most industrial applications, objective of testing is not only to validate the conformance relation between implementation and specification, but also to determine the capability of a software product to adhere to standards, conventions and regulations. In this context, testing of nominal behavior can be a good remedy for large scale systems where complete testing is hardly realistic due to system complexity; or serve as a first validation step for safety critical systems, which enables to detect faults earlier.

It is worth mentioning that this method does not require detailed or full plant models. Any fragment of plant knowledge can contribute to the reduction. Additionally, users can insert a selected set of faults into the target behavior to be tested by modifying the plant features.

7.2 Limitations and outlook

7.2.1 Extension of signals in models

In the current versions of the two approaches, only Boolean signals are taken into account for the control logic. This restricts the applicability of the presented tool. Consequently, further investigation on extending the capability of handling other types of signals, e.g., digital signals with integer values or analog signals, is needed. It is obvious that the state space of test would be even larger when the signals can have multiple values.

To cope with the state-space explosion issue for these systems, equivalence class partition techniques [33] can be used to help reduce large and possibly infinite input data types and ranges into a limited set of equivalence partitions. The executed test cases are representatives selected from each equivalence partition. For example, a model-based black-box equivalence partition testing strategy as well as a formal proof of its completeness properties have been presented in [116].

In the future, equivalence class partition techniques might be of interest to be combined the two approaches presented in this thesis.

7.2.2 Extension of plant features

In this thesis, plant features have been classified into two levels: signal relations among sensors, and signal relations among sensors and actuators. The two levels only deal with current values of sensors and actuators.

However, other factors such as 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, if a belt is turned off at the beginning and the end, but it has been turned on for a while in-between, then the position of a product on the belt should have been changed. Future work can include more such types of plant features in order to gain a better description of nominal system behavior, and therefore a more efficient test sequence.

These extensions will be considered in the continued research. Some above mentioned ideas have been already realized by group colleagues of the author of this thesis and presented in [117].

7.2.3 Reuse of plant features in diagnosis

In this thesis, plant features are modeled and used in the test generation. The benefit is huge, as presented in this thesis, in reducing the number of meaningful test cases, the length of test sequences, and the duration of test execution.

Nevertheless, plant features can also be useful after the testing has been finished. Nominal plant features can serve as diagnosers in the normal execution of a system, since these plant features represent the nominal behavior of system elements such as sensors and actuators. Faulty plant features can serve as 'error-catchers' since many typical faults and errors that can occur in a system have been modeled in these plant

features.

Therefore, the plant features that were initially modeled for testing purpose can also support fault detection in diagnosis. More details and recent results of fault diagnosis techniques can be found in [118].

7.2.4 Modular approach

As presented in Fig. 36, the approaches in this thesis all need to compose the models of subsystems of an SUT to be a single automaton, i.e., an stabilized composed automaton (SCA). As for complete testing and the DTT approach, only specification models are composed, while for the PF approach, plant features are also involved. In both cases, test sequences can be generated only after the SCA is obtained.

Although we achieved to improve our implementation, the computation is still primary memory intensive.

The monolithic composition hinders the application of the two approaches on very large scale systems, since the size of SCA grows exponentially with the number of inputs in the SUT, especially with the DTT approach. With the PF approach, even though the resulting size is reduced due to the plant features, those extra models have to be considered during calculation. This leads to the question whether and how modular approaches can be applied and to what extent they reduce the computational effort.

For future work, we would like to replace the monolithic composition with modular approaches to enhance the whole process.

Recent research results such as a modular supervisor synthesis for extended finite-state machines subject to controllability [119] and a modular plant model synthesis from behavior traces and temporal properties [120] would be inspiring to the modular implementation of the approaches presented in this thesis.

7.2.5 Extended application on hybrid systems

The two approaches presented in this thesis all deal with discrete systems. The methodology can be however generally applicable.

Research work has been done to combine controllers which are usually specified with discrete models and plant which are continuous systems. One recent example that has been presented in [121] is a transformation and emulation framework of continuous physical processes for testing of discrete controllers without using the actual plant.

An extension to such hybrid systems with continuous dynamics would be also of interest for the approaches presented in this thesis in the future.

8 List of publications

8.1 Peer-reviewed journal publications

1. C. Ma and J. Provost, "Design-to-Test Approach for Programmable Controllers in Safety-Critical Automation Systems," IEEE Transactions on Industrial Informatics, pp. 1–10, **submitted**.

2. C. Ma and J. Provost, "Introducing Plant Features to Model-Based Testing of Programmable Controllers in Automation Systems," Control Engineering Practice, pp. 1–15, **submitted**.

8.2 Peer-reviewed conference publications

1. C. Ma and J. Provost, "Design-to-test approach for black-box testing of programmable controllers," in IEEE International Conference on Automation Science and Engineering (CASE), 2015, pp. 1018–1024.

2. C. Ma and J. Provost, "DTT-MAT: A software toolbox of design-to-test approach for testing programmable controllers," in 2016 IEEE International Conference on Automation Science and Engineering (CASE), Fort Worth, Texas, USA, 2016, pp. 878–884.

3. C. Ma and J. Provost, "Design-to-test: an approach to enhance testability of programmable controllers for critical systems – two case studies," in European Conference on Safety and Reliability - ESREL 2016, Glasgow, Scotland, 2016, pp.

2622–2629.

4. C. Ma and 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.

5. C. Ma and 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.

6. C. Ma, C. Jordan, and J. Provost, "SATE: Model-Based Testing with Design-to-Test and Plant Features," in 14th Workshop on Discrete Event Systems, Sorrento Coast, Italy, 2018, pp. 310–315.

8.3 Other peer-reviewed publications (not directly relevant to this thesis)

1. C. Jordan, C. Ma, and J. Provost, "An educational toolbox on supervisory control theory using MATLAB Simulink Stateflow: From theory to practice in one week," in 2017 IEEE Global Engineering Education Conference (EDUCON), 2017, pp. 632–639.

Bibliography

[1] C. Ma and 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. 11 655–11 660.

[2] ——, "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.

[3] D. Li, Z. Zhai, Z. Pang, V. Vyatkin, and C. Liu, "Synchronous-Reactive Semantic Modeling and Verification for Function Block Networks," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 6, pp. 3389–3398, 2017.

[4] I. Buzhinsky and V. Vyatkin, "Automatic inference of finite-state plant models from traces and temporal properties," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 4, pp. 1521–1530, 2017.

[5] B. F. Adiego, D. Darvas, E. B. Viñuela, J. C. Tournier, S. Bliudze, J. O. Blech, and V. M. G. Suárez, "Applying model checking to industrial-sized PLC programs," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 6, pp. 1400–1410, 2015.

[6] B. Boehm, "Software Risk Management," in *European Software Engineering Conference*. Spinger, 1989, pp. 1–19.

[7] H. Pham, *System software reliability*. Springer Science & Business Media, 2007.

[8] R. Isermann, "Supervision, fault-detection and fault-diagnosis methods—an introduction," *Control Engineering Practice*, vol. 5, no. 5, pp. 639–652, 1997.

[9] ISO/IEC/IEEE 24765, *Systems and software engineering — Vocabulary*. IEEE, 2010.

[10] B. Vogel-Heuser, D. Schütz, T. Frank, and C. Legat, "Model-driven engineering of Manufacturing Automation Software Projects - A SysML-based approach," *Mechatronics*, vol. 24, no. 7, pp. 883–897, 2014.

[11] J. Provost, J.-M. M. Roussel, and J.-M. M. Faure, "Generation of single input change test sequences for conformance test of programmable logic controllers," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 3, pp. 1696–1704, 2014.

[12] IEC61508, *Functional safety of electrical / electronic / programmable electronic safety-related systems*, 2nd ed. International Electrotechnical Commission, 2010.

[13] IEC61511, *Functional safety – Safety instrumented systems for the process industry sector*, 1st ed. International Electrotechnical Commission, 2003.

[14] ISO26262, *Road vehicles — Functional safety*, 1st ed. International Organization for Standardization, 2011.

[15] J. Richardsson and M. Fabian, "Modeling the control of a flexible manufacturing cell for automatic verification and control program generation," *International Journal of Flexible Manufacturing Systems*, vol. 18, no. 3, pp. 191–208, 2007.

[16] A. Dubey, "Evaluating software engineering methods in the context of automation applications," in *2011 9th IEEE International Conference on Industrial Informatics*, jul 2011, pp. 585–590.

[17] S. Rösch, S. Ulewicz, J. Provost, and B. Vogel-heuser, "Review of model-based testing approaches in production automation and adjacent domains — Current challenges and research gaps," *Journal of Software Engineering and Applications*, vol. 8, no. 9, pp. 499–519, 2015.

[18] J. Mcgregor, "Testing a software product line," Carnegie Mellon University, Tech. Rep., 2001.

[19] E. Jee, D. Shin, S. Cha, J.-s. Lee, and D.-h. Bae, "Automated test case generation

for FBD programs implementing reactor protection system software," *Software Testing, Verification and Reliability*, vol. 24, no. 8, pp. 608–628, 2014.

[20] V. Vyatkin and S. Member, "Software Engineering in Industrial Automation: State-of-the-Art Review," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1234–1249, aug 2013.

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

[22] 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.

[23] W. Bolton, *Programmable logic controllers*, 4th ed. Elsevier, 2006.

[24] IEC61131-3, *Programmable Controllers—Part 3: Programming Languages*. International Electrotechnical Commission, 2014.

[25] H. Dierks, "PLC-automata: A new class of implementable real-time automata," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1231, no. April 1999, pp. 111–125, 1997.

[26] G. Cutts and S. Rattigan, "Using Petri nets to develop programs for PLC systems," in *International Conference on Application and Theory of Petri Nets*, 1992, pp. 368–372.

[27] G. Frey, "Automatic implementation of Petri net based control algorithms on PLC," in *Proceedings of the IEEE American Control Conference (ACC)*, vol. 4, no. June, 2000, pp. 2819–2823.

[28] B. Vogel-Heuser, D. Witsch, and U. Katzke, "Automatic code generation from a UML model to IEC 61131-3 and system configuration tools," in *International Conference on Control and Automation*, vol. 2. IEEE, 2005, pp. 1034–1039.

[29] F. Chiron and K. Kouiss, "Design of IEC 61131-3 function blocks using SysML,"

in *2007 Mediterranean Conference on Control and Automation, MED*, 2007, pp. 3–7.

[30] M. Obermeier, S. Braun, and B. Vogel-Heuser, "A model-driven approach on object-oriented PLC programming for manufacturing systems with regard to usability," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 3, pp. 790–800, 2015.

[31] J. L. Luo, H. J. Ni, and M. C. Zhou, "Control Program Design for Automated Guided Vehicle Systems via Petri Nets," *IEEE Transactions on Systems Man Cybernetics-Systems*, vol. 45, no. 1, pp. 44–55, 2015.

[32] N. Delgado, A. Q. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[33] ISTQB, *Standard glossary of terms used in Software Testing*, version 2. ed., E. Van Veenendaal, Ed., 2014, vol. 3.

[34] S. Cheon, J. Lee, K. Kwon, D. Kim, and H. Kim, "The software verification and validation process for a PLC-based engineered safety features-component control system in nuclear power plants," in *30th Annual Conference of IEEE Industrial Electronics Society, 2004. IECON 2004*, vol. 1, Busan, Korea, 2004, pp. 827–831.

[35] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[36] S. A. Cook, "The Complexity of Theorem-Proving Procedures," in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, 1971, pp. 151–158.

[37] Y. Bertot and P. Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer Science & Business Media, 2013.

[38] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski, "SPASS version 3.5," in *International Conference on Automated Deduction*. Berlin, Heidelberg: Springer, 2009, pp. 140–145.

[39] H. Wan, G. Chen, X. Song, M. Gu, H. Wan, G. Chen, X. Song, M. G. Formalization, and P. L. C. Timers, "Formalization and Verification of PLC Timers in Coq," in *33rd Annual IEEE International Computer Software and Applications Conference*, 2011, pp. 315–323.

[40] J.-m. Roussel, B. Denis, J. Europ, and J.-m. R. B. Denis, "Safety properties verification of ladder diagram programs," *Journal Européen des Systemes Automatisés (JESA)*, vol. 36, no. 7, pp. 905–917, 2002.

[41] S. Ray, "Overview of Formal Verification," in *Scalable Techniques for Formal Verification*. Boston: Springer, 2010, pp. 9–23.

[42] S. Lampérière-Couffin, O. Rossi, J. Roussel, and J. Lesage, "Formal validation of PLC programs: a survey," in *European Control Conference (ECC)*, 1999, pp. 2170–2175.

[43] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, pp. 279–295, 1997.

[44] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *International journal on software tools for technology transfer*, vol. 1, no. 1-2, pp. 134–152, 1997.

[45] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[46] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuXmv symbolic model checker," in *International Conference on Computer Aided Verification.* Cham: Springer, 2014, pp. 334–342.

[47] O. Pavlovic and H.-D. Ehrich, "Model Checking PLC Software Written in Function Block Diagram," *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 439–448, 2010.

[48] D. Beyer and T. Lemberger, "Software Verification: Testing vs. Model Checking - A Comparative Evaluation of the State of the Art," in *Haifa Verification Conference.*

Springer, 2017, pp. 99–114.

[49] P. Emanuelsson and U. Nilsson, "A Comparative Study of Industrial Static Analysis Tools," *Electronic Notes in Theoretical Computer Science*, vol. 217, pp. 5–21, 2008.

[50] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[51] H. Prahofer, F. Angerer, R. Ramler, H. Lacheiner, and F. Grillenberger, "Opportunities and challenges of static code analysis of IEC 61131-3 programs," in *17th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2012, pp. 1–8.

[52] D. Singh, V. R. Sekar, K. T. Stolee, and B. Johnson, "Evaluating how static analysis tools can reduce code review effort," in *Proceedings of IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC*, 2017, pp. 101–105.

[53] T. Ball and S. K. Rajamani, "The SLAM project: debugging system software via static analysis," *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 1–3, 2002.

[54] N. Ayewah and W. Pugh, "The Google FindBugs fixit," in *Proceedings of the 19th international symposium on Software testing and analysis - ISSTA '10*, 2010, pp. 241–252.

[55] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: A static analysis tool for detecting web application vulnerabilities," in *Security and Privacy, 2006 IEEE Symposium on*, 2006, pp. 258–263.

[56] M. Obster and S. Kowalewski, "A live static code analysis architecture for PLC software," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2017, pp. 1–4.

[57] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[58] M. Leucker, "Teaching runtime verification," in *International Conference on Runtime Verification*. Berlin, Heidelberg: Springer, 2011, pp. 34–48.

[59] K. Havelund and G. Rosu, "An overview of the runtime verification tool Java PathExplorer," *Formal methods in system design*, vol. 24, no. 2, pp. 189–215, 2004.

[60] F. Chen and G. Roşu, "Mop: an efficient and generic runtime verification framework," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 569–588, 2007.

[61] L. Garcia, S. Zonouz, D. Wei, and L. P. De Aguiar, "Detecting PLC control corruption via on-device runtime verification," in *Proceedings Resilience Week, RWS 2016*, 2016, pp. 67–72.

[62] M. E. Khan and F. Khan, "A Comparative Study of White Box, Black Box and Grey Box Testing Techniques," *International Journal of Advanced Computer Science and Applications*, vol. 3, no. 6, pp. 12–15, 2012.

[63] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng, "Generating test cases from UML activity diagram based on gray-box method," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2004, pp. 284–291.

[64] G. A. Di Lucca and A. R. Fasolino, "Testing web-based applications: The state of the art and future trends," *Information and Software Technology*, vol. 48, no. 12, pp. 1172–1186, 2006.

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

[66] P. Mani and 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.

[67] H. T. Park, J. G. Kwak, G. N. Wang, and S. C. Park, "Plant model generation for PLC simulation," *International Journal of Production Research*, vol. 48, no. 5, pp. 1517–1529, 2010.

[68] A. Pretschner, W. Prenninger, S. Wagner, C. Kuhnel, M. Baumgartner, B. Sostawa, R. ZoIch, and T. Stauner, "One evaluation of model-based testing and its automation," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, no. 2, 2005, pp. 392–401.

[69] 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, aug 2012.

[70] M. Shafique and Y. Labiche, "A systematic review of model based testing tool support," Tech. Rep. May, 2010.

[71] I. Stürmer, D. Weinberg, and M. Conrad, "Overview of existing safeguarding techniques for automatically generated code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1–6, 2005.

[72] H. Shokry and M. Hinchey, "Model-based verification of embedded software," *Computer*, vol. 42, no. 4, pp. 53–59, 2009.

[73] E. Bringmann and A. Krämer, "Model-Based Testing of Automotive Systems," in *1st International Conference on Software Testing, Verification, and Validation*, 2008, pp. 485–493.

[74] D. Bohlender, H. Simon, N. Friedrich, S. Kowalewski, and 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.

[75] S. Ulewicz and B. Vogel-Heuser, "Increasing system test coverage in production automation systems," *Control Engineering Practice*, vol. 73, pp. 171–185, 2018.

[76] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, pp. 803–819, 2015.

[77] R. Ramler, W. Putschögl, and D. Winkler, "Automated testing of industrial

automation software: practical receipts and lessons learned," in *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation - MoSEMInA 2014*, 2014, pp. 7–16.

[78] A. Bulajic, S. Sambasivam, and R. Stojic, "Overview of the Test Driven Development Research Projects and Experiments," in *Proceedings of Informing Science & IT Education Conference (InSITE)*, 2012, pp. 165–187.

[79] A. Causevic, D. Sundmark, and S. Punnekkat, "Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review," in *2011 4th IEEE Int. Conf. on Software Testing, Verification and Validation*, 2011, pp. 337–346.

[80] J. Grenning, *Test-Driven Development for Embedded C*. The Pragmatic Programmers, 2011.

[81] R. Hametner, D. Winkler, T. Östreicher, S. Biffl, and A. Zoitl, "The Adaptation of Test-Driven Software Processes to Industrial Automation Engineering," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, 2010, pp. 921–927.

[82] C. Schotten and H. Meyr, "Test point insertion for an area efficient BIST," in *Test Conference, 1995. Proceedings., International*, 1995, pp. 515–523.

[83] P. Girard, L. Guiller, C. Landrault, and H.-J. Pravossoudovitch, S. Wunderlich, "A Modified Clock Scheme for a Low Power BIST Test Pattern Generator," in *19st IEEE VLSI Test Symp.*, 2001, pp. 306–311.

[84] F. Liang, L. Zhang, S. Lei, G. Zhang, K. Gao, and B. Liang, "Test patterns of multiple SIC vectors: Theory and application in BIST schemes," *IEEE Trans. on VLSI Syst.*, vol. 21, no. 4, pp. 614–623, 2013.

[85] Y. J. Huang, J. F. Li, J. J. Chen, D. M. Kwai, Y. F. Chou, and C. W. Wu, "A built-in self-test scheme for the post-bond test of TSVs in 3D ICs," in *29th IEEE VLSI Test Symp.*, 2011, pp. 20–25.

[86] A. Koneru, S. Kannan, and K. Chakrabarty, "A Design-for-Test Solution Based on

Dedicated Test Layers and Test Scheduling for Monolithic 3D Integrated Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. accepted, 2018.

[87] S. M. Saeed and O. Sinanoglu, "A Comprehensive Design-for-Test Infrastructure in the Context of Security-Critical Applications," *IEEE Design and Test*, vol. 34, no. 1, pp. 57–64, 2017.

[88] J. Wang, M. Ebrahimi, L. Huang, X. Xie, Q. Li, G. Li, and A. Jantsch, "Efficient Design-for-Test Approach for Networks-on-Chip," *IEEE Transactions on Computers*, p. accepted, 2018.

[89] M. Rausch and B. H. Krogh, "Formal verification of PLC programs," in *Proceedings of the American Control Conference*, vol. 1, no. June, 1998, pp. 234–238.

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

[91] J. M. Machado, B. Denis, J. J. Lesage, J. M. Faure, and J. C. L. Ferreira Da Silva, "Logic controllers dependability verification using a plant model," *IFAC Proceedings*, vol. 39, no. 17, pp. 37–42, 2006.

[92] J. Machado, B. Denis, J.-J. Lesage, J.-M. Faure, and J. F. 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.

[93] P. Ovsiannikova, D. Chivilikhin, V. Ulyantsev, and 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.

[94] I. Hegny, M. Wenger, and 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.

[95] C.-h. Yang, G. Zhabelova, C.-w. Yang, and S. Member, "Cosimulation environment for event-driven distributed controls of smart grid," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 3, pp. 1423–1435, 2013.

[96] 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, 2012.

[97] 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*, 2011, pp. 57–62.

[98] E. A. Lee and S. A. Seshia, *Introduction to embedded systems: A cyber-physical systems approach*, 2011.

[99] C. K. Chang and H. Huang, "On transforming Petri net model to Moore machine," in *14th Annual Int. Computer Software and Applications Conf.*, no. 052. Chicago: IEEE, 1990, pp. 267–272.

[100] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[101] M.-k. Kwan, "Programming method using odd or even pints," *Acta Mathematica Sinica*, vol. 10, pp. 263–266, 1960.

[102] M. Jamro, "POU-Oriented Unit Testing of IEC 61131-3 Control Software," *IEEE Transactions on Industrial Informatics*, vol. 11, no. 5, pp. 1119–1129, 2015.

[103] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *Computer Journal*, vol. 52, no. 5, pp. 589–597, 2007.

[104] S. Fischmeister and P. Lam, "Time-aware instrumentation of embedded software," *IEEE Transactions on Industrial Informatics*, vol. 6, no. 4, pp. 652–663, 2010.

[105] Z. Gu, C. Wang, M. Zhang, and Z. Wu, "WCET-Aware Partial Control Flow Checking for Soft Error Protection in Resource-Constrained Real-Time Embedded

Systems," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 10, pp. 5652–5661, 2014.

[106] C. Ma and J. Provost, "Design-to-test approach for black-box testing of programmable controllers," in *IEEE Int. Conf. on Automation Science and Engineering (CASE)*, 2015, pp. 1018–1024.

[107] ——, "DTT-MAT: A software toolbox of design-to-test approach for testing programmable controllers," in *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, Fort Worth, Texas, USA, 2016, pp. 878–884.

[108] ——, "Design-to-test: an approach to enhance testability of programmable controllers for critical systems – two case studies," in *European Conference on Safety and Reliability - ESREL 2016*, Glasgow, Scotland, 2016, pp. 2622–2629.

[109] I. K. Voyiatzis and D. J. Kavvadias, "On the generation of SIC pairs in optimal time," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2891–2901, 2015.

[110] J. Provost, J.-M. Roussel, and J.-M. Faure, "Technical report on Conformance Test of Programmable Logic Controllers – Execution of Minimum-Length Test Sequences," LURPA, ENS Cachan, France, Cachan, Tech. Rep., 2014.

[111] Mathworks, "User ' s Guide," Tech. Rep., 2017.

[112] Q. Wen, R. Kumar, and J. Huang, "Framework for optimal fault-tolerant control synthesis: Maximize prefault while minimize post-fault behaviors," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, no. 8, pp. 1056–1066, 2014.

[113] S. Rösch and B. Vogel-Heuser, "A light-weight fault injection approach to test automated production system PLC software in industrial practice," *Control Engineering Practice*, vol. 58, pp. 12–23, 2017.

[114] C. Jordan, C. Ma, and J. Provost, "An educational toolbox on supervisory control theory using MATLAB Simulink Stateflow: From theory to practice in one week," in *2017 IEEE Global Engineering Education Conference (EDUCON)*, 2017, pp.

632–639.

[115] M. H. De Queiroz and 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.

[116] W.-l. Huang and J. Peleska, "Complete model-based equivalence class testing," *International Journal on Software Tools for Technology Transfer*, vol. 18, no. 3, pp. 265–283, 2016.

[117] C. V. Jordan, J. C. Herrero, and J. Provost, "Extension of the Plant Feature Approach Introducing Temporal Relations," in *Automation Science and Engineering (CASE), 2018 IEEE International Conference on*, 2018, p. published.

[118] Z. Gao, C. Cecati, and S. X. Ding, "A survey of fault diagnosis and fault-tolerant techniques - Part I: Fault diagnosis with model-based and signal-based approaches," *IEEE Transactions on Industrial Electronics*, vol. 62, no. 6, pp. 3757–3767, 2015.

[119] R. Malik and M. Teixeira, "Modular supervisor synthesis for extended finite-state machines subject to controllability," in *Discrete Event Systems (WODES), 2016 13th International Workshop on*, 2016, pp. 91–96.

[120] I. Buzhinsky and V. Vyatkin, "Modular plant model synthesis from behavior traces and temporal properties," in *22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2017, pp. 1–7.

[121] A. Malik, P. S. Roop, N. Allen, and T. Steger, "Emulation of cyber-physical systems using IEC-61499," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 1, pp. 380–389, 2018.