Chapter 11
Formal Verification of Evolutionary Changes

Bernhard Beckert, Jakob Mund, Mattias Ulbrich, and Alexander Weigl

In this chapter, we elaborate how formal verification techniques can be used to ensure safety properties of automated production systems during their evolution. First we discuss the opportunities that formal methods offer, particularly when dealing with the evolution of automated production systems, but also which special needs this particular domain requires from the formal methods to be applied. We argue that evolution allows the seamless combination of experiential knowledge with formally founded reasoning.

We exemplarily present three approaches that successfully incorporate a formal verification technique for analysis, modelling, or reasoning, into the system evolution process, namely, *regression verification*, *generalised test tables*, and *model checking* of holistic (multidomain) models.

All three approaches contribute to the guiding theme *Methods and Processes for Evolution* of the priority programme.

While formal verification methods have the potential of being used in several application fields, we concentrate on the aspect of ensuring **correctness** (in the form of maintaining safety properties or **consistency** with earlier versions). We focus on techniques that operate on the actual implementation (the code executed on a plant) rather than on more abstract behavioural descriptions. Here, we describe the logical foundations and technical aspects of the applied formal verification techniques and their applications; their benefits for the user, as far as system and model comprehensibility are concerned; and the embedding into development processes are discussed in Sect. 10.

B. Beckert · M. Ulbrich (✉) · A. Weigl
Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
e-mail: beckert@kit.edu; ulbrich@kit.edu; weigl@kit.edu

J. Mund
Institute of Informatics L4, Technical University of Munich, Garching, Germany
e-mail: mund@in.tum.de

R. Reussner et al. (eds.), *Managed Software Evolution*,
https://doi.org/10.1007/978-3-030-13499-0_11

11.1 Verifying Production Systems: Assessment of Opportunities

System analyses based on formal methods are powerful techniques to ensure that a system has desired properties. Formal methods provide a versatile toolbox that can be used for many reliability- or safety-enhancing tasks like formal verification, advanced testing, modelling, formal specification and design, etc. Formal methods are known for being very thorough analysis techniques since they import mathematical rigour into the analysis process.

As in the case for many very general notions, the question of when a technique is to be called formal has no definite answer. Moreover, different people from different communities are likely to give very different answers. Within this chapter, by "formal verification method", we denote a formal technique that mathematically proves that a system or component satisfies its specified requirements [IEE90]. Such a formal technique usually comprises a formal description of the system (i.e. a model of the system expressed using a formal notation), a formal specification of the requirements, and rigorous (formal) rules that allow one to reason that the system satisfies the requirements. In addition, we focus on techniques that allow automated verification, where the actual verification step is conducted by a computer program requiring as little guidance to user input as possible.

11.1.1 Peculiarities of Automated Production Systems

For the remainder of this chapter, instead of discussing the application of formal verification in general, we will focus on the application of a particular kind of systems, namely, automated production systems. Distinctive characteristics of such a system are as follows:

1. They are *long running*. Oftentimes, the plants that a software drives are designed to run for several decades, which makes a thorough design-time analysis worthwhile that takes potential evolutionary developments into consideration.
2. They are often mission or safety *critical*. Due to immense forces and speeds that can build up in a plant, a malfunctioning automated production system may cause considerable damage to products or production systems and may even bring people to harm. Damaged systems may cause immense costs if plants stand still.
3. They are *multidisciplinary* in the sense that their design spans several engineering disciplines that must work together to achieve the desired system behaviour and *heterogeneous* in the sense that they comprise analogue as well as digital components. For instance, a software engineer may be responsible for developing the software that controls a conveyor belt installed by a mechanical engineer. The controller actions are based on sensor information obtained from a bus system designed by an electrical/electronical engineer.

4. While automated production systems remain in service for a long time, their requirements often are not cast in stone, but *change over time*: New types of products are to be manufactured, systems are upgraded to increase throughput or to keep up with technological development, etc. Moreover, flaws in the controlling software or the hardware design may have to be fixed. Production systems therefore frequently *evolve* during their lifetime. Thus, methods and means that accompany the transition induced by evolution must be put into action. One has to ensure that a revision does not break existing intended behaviour while achieving the intended change effect.

Based on these peculiarities, we subsequently identify and describe both the general intricacies and the opportunities for formal verification in the domain of automated production systems on an abstract level and come back to these points in the sections on the individual approaches.

11.1.2 Intricacies of the Application of Formal Verification

Formal methods have been the subject of scientific investigation for decades. However, the industry is very reluctant to incorporate them into their development processes. Only in recent years have formal methods gained reputation, for instance, by being added as acceptable verification techniques for avionics [GP12] or for the automotive industry [ISO11a]. Based on our experience in the priority programme, we see the following intricacies (or challenges) for formal verification of evolutionary changes in automated production systems:

Specification efforts One of the main reasons for reluctance to adopt formal methods in industrial contexts is that many of their use cases have in common that they require a formal description of the properties to be established (a "formal specification"). Obtaining for formal specifications or models is hard [Pak+16], as this requires training in the formal system and due to the additional workload it puts on industrial-sized projects. During evolution, specifications have to be consistently co-evolved alongside the code, which increases the required overhead even more.

Cyber-physical systems Automated production systems have an interdisciplinary nature which combines discrete software-driven controllers with continuous physical dimensions. Hence, hybrid systems that combine models for both types of behaviours are a natural fit to represent automated production systems. For instance, the geo-spatial translation of workpieces may be modelled in terms of a function from continuous time to a continuous variable, i.e. the position, by means of differential equations. Continuous behaviour inherently induces that the system state space becomes infinite. Checking correctness thus becomes a more difficult problem and inaccessible for explorative techniques like many model-checking approaches. Hence, finding suitable (finite) abstractions

that model physical phenomena both correctly and sufficiently precisely becomes a key challenge for verification.

Large state space As a reactive system running for long periods of time, automation software must always be validated by analysing traces of system responses; it does not suffice to analyse individual cycles of the software individually. The state space that needs to be considered during verification grows exponentially in the number of steps that are analysed. Moreover, the output of the software also depends on the behaviour of the hardware on which it operates. The (often nondeterministic) hardware models which are often used in the validation make the state space grow even larger.

Specific languages/tools Control software for automated production system is typically written in languages fairly different from commonplace programming languages used for other embedded systems, e.g. C/C++. The IEC61131 [Com02] standard defines five different textual and graphical languages to program automated production systems. As a consequence, the use of existing approaches and tools requires adaptation.

11.1.3 Opportunities for the Application of Formal Verification

Based on our experiences from our research on formal methods within the priority programme, we also see potential for the application of formal methods in the practice of automated production system development and evolution.

Existing older system versions Due to the evolutionary aspect, we can assume the existence of older revisions of the system (the plant and the software). Such existing system versions can be leveraged for formal verification in several ways. For instance, the analysis can be restricted to investigating the *difference* (structural or behavioural difference) of the new revision w.r.t. the old revision. Furthermore, old revisions allow obtaining precise models of the system efficiently using observations and model learning techniques.

Limited structural complexity Typically, due to the cyclic behaviour of the programmable logic controller and the imposed timing restrictions, the structural complexity of the software of an individual controller is rather limited compared to other software programs such as database management systems. For instance, program loops with complex exit conditions and algorithmic traversal of complex data structures are rarely found in control software of automated production systems.

Economically justifiable efforts Due to the longevity of automated production systems, initial efforts put into formalisation have a longer period to break even. Hence, higher efforts typically associated with formal verification are more likely to be economically justified for automated production systems.

Infeasibility of alternatives Common alternatives to verification, first and foremost testing, are often economically or technically infeasible since neither the

actual testing environment nor the system under test can be used, since that would require to either hold production at the customer site or install a prototypical machine for testing purposes. Hence, the value of testing, i.e. the ability to find bugs in an efficient way, is diminished. In turn, formal verification operates solely on the controller software code and, thus, does not suffer from these drawbacks and may therefore be a suitable addition or alternative (especially in early stages of the development process) to verification by testing of automated production systems.

In conclusion, while the applicability of formal verification depends on the specific system and engineering context, the above opportunities suggest that it is particularly well suited for engineering automated production system, compared to engineering software systems in general.

11.1.4 Addressed Software Evolution Challenges

To illustrate the applicability and benefits of formal verification, this chapter reports on three formal approaches that each verifies a distinct aspect of the correctness of automated production systems. They address two complementary questions from the collection of general challenges regarding software evolution in Chap. 3.

How to model, specify, and verify that a system retains desired behaviour during system evolution? In Sect. 11.2, we present an approach to verify that defined aspects of the behaviour of the system software are preserved during system evolution.

The approach takes the code of two versions of the system software as input and a formal condition under which the two should behave equivalently and a formal definition of when two behaviours are considered equal. Using a state-of-the-art model checker, the verification approach then asserts that for all admissible input sequences, the two revisions satisfy the required equivalence condition.

This kind of equivalence checking is called *regression verification* and particularly helpful since it reduces the need for specification: The old software versions serve as (partial) specification for the new version. The verification transfers the trust in the correctness of the old software revision onto the new one. Regression verification does not require formally specified system properties: The old revision defines the functional property to be verified for the new revision.

How to model, specify, and verify intentionally changed behaviour during system evolution? In Sect. 11.3, a novel temporal specification language is introduced which allows a comprehensible specification of reactive systems like the software of automated production systems. For those parts of the software behaviour which are intended to change, this temporal specification language can be used to describe the new behaviour.

Sometimes, a specification of the software changes alone does not suffice. To answer the question for cases in which not only the software but entire systems evolve, we present in Sect. 11.4 an approach to verify that the integrated plant behaviour, i.e. the composition of software, automation platform, and mechanical components, satisfies the system requirements. Specifically, the approach is based on creating multidomain models of automated production systems in a coherent model that represents the results of several involved engineering disciplines by using a common (formal) modelling language. By translating those models into formal representations, model-checking tools give us qualitative or quantitative results that can be used to decide whether the system meets the specified requirements.

11.2 Regression Verification

One of the main bottlenecks for using formal methods in practice is coming up with suitable system and requirement specifications. This problem is particularly severe in the domain of automated production systems as formal specifications are even less common in this domain than in other software disciplines. In the following, we give a brief introduction to the concept of regression verification which exploits existing software revisions as specifications of new releases of the system—thus severely reducing the need to formulate specifications. In this section, we explain our application of regression verification to PLC software (more details are given in [Bec+15]). The embedding into the software development process is outlined in Sect. 10.2.2. The idea of regression verification is to formally prove that a version of code driving a plant after an evolution step shows the same reactive input/output behaviour as the code version before evolution. Only desired deviations that are explicitly stated are allowed. Thus, the original code serves as a formal specification for the new implementation, and formal verification techniques like model checking or deductive theorem proving can be applied to prove that the behavioural effect of the code remains the same. Regression verification covers all parts of system behaviour that are intended to remain untouched during an evolution step.

In this and the following sections, we consider PLCs to be reactive systems with a periodic cyclic data processing behaviour, repeating the same control procedure indefinitely. A PLC cycle consists of the following steps: (1) read input values (input space I), (2) execute task(s), (3) write output values (output space O), and (4) wait till next cycle starts. This leads to the following formal definition using infinite sequences over inputs and outputs (I^ω and O^ω):

Definition 11.2.1 The semantics of a PLC program P is a causal-deterministic function $b(P) : I^\omega \to O^\omega$. That is, $i_1{\downarrow_n} = i_2{\downarrow_n}$ implies $b(P)(i_1){\downarrow_n} = b(P)(i_2){\downarrow_n}$ for all $n \in \mathbb{N}$, where $x{\downarrow_n}$ denotes the finite initial subsequence of x of length n.

PLCs are modelled as causal-deterministic systems as we assume they are stateful, deterministic programs whose output is a function of the inputs received since system start—but that cannot depend on the input which is still to come.

The aim of regression verification is to formally prove that the existing (good) behaviour of PLC code is retained during system evolution—which is (as a verification goal) different to proving that PLC code satisfies a functional specification and different to showing that the whole production system works correctly. We assume that the old software revision has proved its value during its lifespan and has thus gained "trust by experience". Regression verification formally transfers this trust to the new revision. The main advantage of regression verification is that no functional/behavioural specification is required for the part of the behaviour to be retained (besides the old code version). The application area of software in automated production systems is particularly well suited for a treatment with regression verification for the following reason: During the lifespan of a plant, its software usually needs to adapt to changing requirements. As a rule, the requirements for the machine behaviour do not change entirely but only in certain well-defined aspects while most parts are to be retained in an evolution step.

In an ideal verification scenario, regression verification and regression testing should go side by side as both approaches have their particular advantages. Regression verification provides a formal equivalence proof for all considered input sequences and not only for the (usually restricted) set of selected test cases. Also, while regression testing of PLC software requires either a hardware test bed or an executable hardware model, this is not needed for regression verification. It suffices to provide a formal relational description of how the hardware has changed during the evolution step (if it has changed at all). Testing, on the other hand, is not limited to an analysis of the software alone but allows comprising the physical entities of the machine.

We define a notion of reactive conditional and reactive relational equivalence together with a proof methodology, also in the presence of environment models. Our method concentrates on the PLC software that runs on the controller and for now disregards all effects outside the software (in particular the context and the platform). Some additional measures for incorporating models of effects outside the software into the verification are discussed below.

A core element of our verification method is a translation of PLC code into the input language for model checkers. Using this translation on both the old and the new software revision, we can specify the retained behaviour. Our technology targets PLC code written in Structured Text (ST) and Sequential Function Chart (SFC), two languages of the IEC 61131 standard [Com02]; an adaptation to other languages is easily possible. A further core element is the use of a model checker supporting invariant generation. It is an important insight that this allows the automatic generation of *coupling invariants*, which in many cases make regression verification more efficient than symbolic or explicit state model checking. Accordingly, we have adapted the concept of coupling invariants to the world of reactive systems. We have implemented our approach in a tool chain using the model checker nuXmv [Cav+14]. It supports techniques for predicate abstraction and invariant generation by interpolant inspection [Bra11, McM03].

The first notion of equivalence we define is that of perfect, bit-wise equivalence of two PLC programs in which both systems always answer with the same response to sensor stimuli:

Definition 11.2.2 (Trace Equivalent PLC Programs) Two PLC programs P, Q whose variable declarations contain the same input/output variables are called *perfectly equivalent* if they produce the same output sequence when presented with the same input sequence, i.e. $b(P)(\bar{i}) = b(Q)(\bar{i})$ for all $\bar{i} \in I^{\omega}$.

When considering the semantics of programs to be sets of traces, this definition is equivalent to requiring that $b(P) = b(Q)$.

This first definition of trace equivalence is too strong a condition in most cases since software re-factorisation is the answer to changed requirements and the software is indeed intended to behave differently. Hence, we introduce a second notion of equivalence: conditional equivalence:

Definition 11.2.3 (Conditionally Equivalent PLC Programs) Two PLC programs P and Q are called *conditionally equivalent* modulo the condition $\varphi : I^{\omega} \to bool$ if they produce the same result for all input sequences that satisfy condition φ, i.e. if $\varphi(\bar{i})$ then $b(P)(\bar{i}) = b(Q)(\bar{i})$ for all $\bar{i} \in I^{\omega}$.

During evolution, the behaviour of the system's sensors and actors may be changed in addition to software changes. Then, the notion of *conditional* equivalence may still be too strong and needs to be further relaxed. This leads to the notion of *relational* equivalence:

Definition 11.2.4 (Relationally Equivalent PLC Programs) Two PLC programs P, Q are called *relationally equivalent* modulo relations $\sim_{in} \subseteq I_P^{\omega} \times I_Q^{\omega}$ and $\sim_{out}$: $O_P^{\omega} \times O_Q^{\omega}$ if they produce related output sequences when presented with related input sequences, i.e.

$$\text{if } \bar{i} \sim_{in} \bar{i}' \text{ then } b(P)(\bar{i}) \sim_{out} b(Q)(\bar{i}') \text{ for all } \bar{i} \in I_P^{\omega}, \bar{i}' \in I_Q^{\omega}.$$

With these notions we have established the different proof obligations for regression verification. Figure 11.1 shows how the approach is realised. After processing the program code of the two revisions to be compared into formal models ("SMV"), these two models are combined into one product automaton, which is then—together with the properties to be checked—encoded into a combined model that is sent to a model checker tool. The program code is translated into an automaton by first normalising the code to a restricted programming language ST_0 with limited feature set and then symbolically executing it. The model checker either proves the equivalence property (✓), finds a counterexample that exposes that the two versions are not equivalent (✗), or times out (🕒).

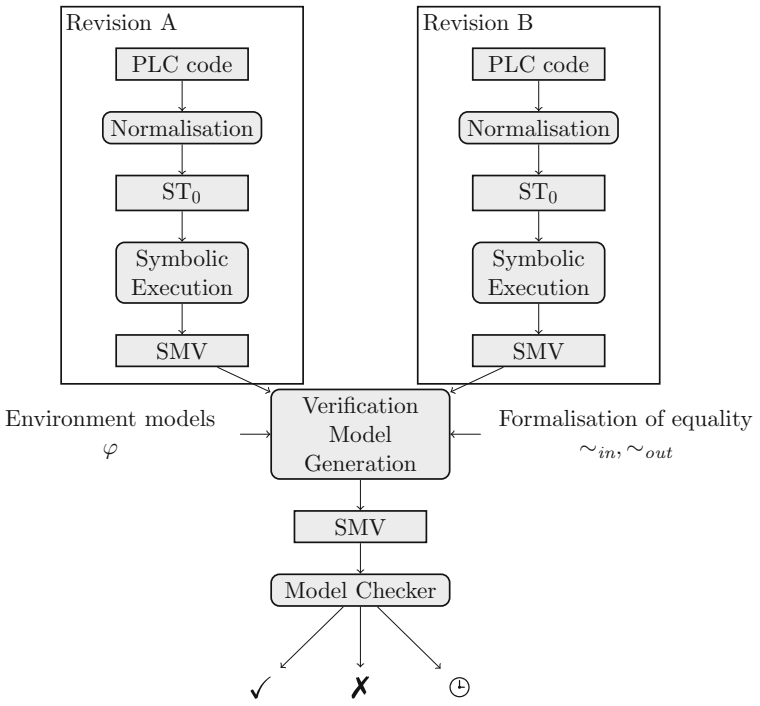

Fig. 11.1 Overview of the regression verification method

11.2.1 Environment Models

There are realistic software evolution steps which would, in general, change the behaviour of an automated Production Systems (aPS), but which do not change the behaviour of a particular plant since not every sequence of input signals is possible. For instance, two software versions may behave differently, if two signals `movingLeft` and `movingRight` are simultaneously set. Since this will never be the case in reality (at least in normal operation), the software revisions can still be called equivalent if they behave equivalently in all other cases.

To make the approach more precise by allowing such verification cases, we include a mechanism to incorporate models of the plant environment into the verification chain. By definition, equivalence needs to hold for all conceivable sequences of input values, which is a very strong requirement. However, it suffices that the systems behave equivalently for all input sequences that can occur in practice. It is therefore sensible to add knowledge on the possible sensor inputs as assumptions to the process and perform a conditional regression verification, where the condition not only excludes inputs for which the systems are intended to behave differently but also inputs that cannot occur in practice. Our methodology allows

incorporating environment models either as LTL formulas or as automata. In both cases, the models are added as assumptions.

For example, the crane of the Pick-and-Place Unit (PPU) (Sect. 4.3) can never be in more than one of the positions Magazine (M), Stamp (S), and Conveyor Belt (C) at the same time. Assuming correctly working sensors, at most one of the Boolean input variables M, S, C can be true at the same time. Thus, it is irrelevant whether the two program revisions react differently in case, e.g. M and S are signalled simultaneously, but still are equivalent for all other inputs. This assumption adds the implicit precondition that sensors never fail. If they fail, no guarantees are made. The regression verification approach is flexible in the sense that it allows one either to add such assumptions or to show equivalence also for failure cases.

11.2.2 Case Study: PPU

For evaluation, we have applied our regression verification tool chain to the PPU case study introduced in Sect. 4.3 in this book. The case study covers different aspects of evolution, containing pure software changes as well as changes that incorporate adaptations to the mechanics and automation hardware in their 16 different variants. In the largest of the original scenarios, the PPU has 22 digital input, 13 digital output, and 3 analogue output signals and defines a number of simple discrete event automation tasks.

We discovered some unintentional regressions in the PPU using our approach. In four cases, a regression by delaying the system answer one cycle for each workpiece has been caused by newly added code blocks. Due to the short cycle time of the PPU (4 ms), the discrepancy between the programs was not revealed during testing. Moreover, regression verification discovered that a fix for a safety violation had not been ported to an earlier version in the PPU evolution sequence. It is possible that the crane tries to grab a workpiece while it is still in motion which might under very unfortunate circumstances cause damages.

In the following, we discuss two evolution scenarios from the PPU and show how they can be subject to regression verification. More details can be found in [Wei15]; see Table 11.1[1] for the time required for verification. Not all evolution scenarios include a software modification or have an intentional behaviour difference. The scenarios for which the equivalence verification is trivial have been omitted from the table.

In the evolution scenario Ev3, the new stamping hardware for metallic products brings with it a new emergency stop button E_2 (triggering the same emergency logic as the existing button E_1) and a new start switch S_3 (complementing S_1 and S_2 already present). Only after *all* three start switches have been pressed does the plant

[1] Verification with nuXmv in version 1.0.1 on an Intel Dual-Core with 2.7 GHz and 4 GB RAM running OpenSUSE 12.2; see [Wei15, Bec+15] for detail information.

Table 11.1 Results of the experiments

Scenario	In	State	Min	Max	Scenario	In	State	Min	Max
Ev1	10	140	4 s	8 s	Ev6+EM	11	299	2 min	21 min
Ev1+EM	12	146	7 s	12 s	Ev8	20	289	13.7 min	20.9 min
Ev2	11	141	4 s	8 s	Ev9	20	305	50.5 min	1.3 h
Ev3	19	246	9 s	17 s	Ev10	23	365	13 s	24 s
Ev6+A	19	284	15.1 min	155.4 h	Ev11	28	576	3.5 h	6.3 h
Ev6+A_m	19	284	8.9 min	9.1 h	Ev12	34	860	22.2 h	56.4 h
Ev6+A_{nm}	19	284	18.1 min	13 h	Ev13	34	1225	21.9 h	21.9 h
Ev6+AEM	11	299	25.7 min	104.1 h	Ev14	47	1663	22.1 h	22.1 h

"Scenario" is the name of the evolution scenario in [Vog+14b], "In" is the size of the sensor input space in bits, "State" is the size of the state space in bits, and "Min/Max" show the minimum and maximum time needed for verification using nuXmv in seconds (s), minutes (min), or hours (h). +EM indicates that an environment model has been used

start processing workpieces. Trace equivalence between the two revisions of this evolution step can only be shown for traces where these new components do not influence the flow of signals already present in the old software. This is the case if (1) no metallic workpiece is ever detected in the plant, (2) button E_2 is only pressed if simultaneously E_1 is also pressed, and (3) S_3 is not activated after the other switches S_1 and S_2 have been pressed. Under these assumptions, conditional equivalence can indeed be proved by our tool chain.

In evolution scenario Ev14, the three position sensors at Crane A, Magazine B, and Stamp C are replaced by a single angle transmitter that reports the angular position of the crane (in degrees). Now, the PLC programs take their input from two different value domains such that we need to express the relationship between these input spaces by a predicate $\sim_{in}$ which relates each Boolean position switch (A, B, and C) to a 5° interval in the angular input space represented by the continuous value α:

$$(A, B, C) \sim_{in} \alpha = (A \leftrightarrow 0 \leq \alpha \leq 5) \wedge (B \leftrightarrow 90 \leq \alpha \leq 95) \\ \wedge (C \leftrightarrow 180 \leq \alpha \leq 185) \ .$$

11.3 Generalised Test Tables

In the last section, an approach is presented which permits one to prove that a software revision behaves (partially) equivalently to an earlier revision. But when a system evolves, regression verification presented in the last section can cover validation for inputs where system behaviour does not change. But how to deal with the part of the behaviour which is *intended to change*? For those inputs where different behaviour is expected, we cannot simply specify by reference to the old version. A formal specification is needed to fill this unspecified gap.

We fall back to assurance of functional correctness, with the same application issues as mentioned in Sect. 11.1.2, especially the specification efforts. A specification language and methodology are needed which are accessible and applicable to engineers. To address the challenge of lacking languages and tools for formal specification of automated production systems, we introduced *generalised test tables*, a practical specification methodology with which PLC systems can be conveniently formally specified and verified. We present syntax and semantics of the specification technique and show how they can be used to model check reactive system behaviour.

Test cases are commonly written in the form of *test tables*, in which each row contains the input stimuli for one cycle and the expected response of the reactive system. Thus, the whole table captures the intended behaviour of the system (the sequence of actuator signals) for one particular sequence of input signals. Generalised test tables extend the concept of test tables, which are already frequently used in quality management of aPS. The main idea is to allow more general table entries, thus enabling a table to capture not just a single test case but a family of similar behavioural cases.

In Sect. 10.2.2, the shape of generalised test tables and their generalisation concepts have already been introduced. Here, we describe the formal foundations of generalised test tables and reports about their principal suitability for formal specification and automatic verification.

11.3.1 Formal Syntax

Formally, a generalised test table is a finite sequence of rows. Each row consists of three constraining formulas: *symIn* for the inputs, *symOut* for the outputs, and *symDur* for the duration (the number of repetitions) of that row. The constraints are formulated in a generalisation of the expression language of Structured Text (see Sect. 10.2.2 for details).

Definition 11.3.1 (Generalised Test Table) Let T be a generalised test table with m rows; let $\mathscr{I}_T$ and $\mathscr{O}_T$ be the set of input variables resp. the set of output variables of T; and let $\mathscr{G}_T$ be the set of global variables occurring in T. Then T is identified with the sequence

$$(symIn_1, symOut_1, symDur_1) \cdots (symIn_m, symOut_m, symDur_m) \ ,$$

where $symIn_i$ is the conjunction of all constraints contained in cells in row i that correspond to *input* variables, $symOut_i$ is the conjunction of all constraints contained in cells in row i that correspond to *output* variables, and $symDur_i$ is the interval contained in the duration column at row i.

The constraint $symIn_i$ on the input values is the precondition of the ith row, and analogously $symOut_i$ is its post-condition. The duration constraint $symDur_i$ describes how often the ith row is allowed to be repeated successively.

11.3.2 Semantics

The semantics of generalised test tables is discussed in detail in [Bec+17a]; here, we give a summary. The definition of the semantics is based on a two-party game—between a challenger and the system—for which a generalised test table describes the allowed moves. The challenger tries to force the system to violate the generalised test table, whereas the system tries to conform to the generalised test table.

Like any two-party game, this game is played alternately. At each turn, the challenger provides a set of input values, and the system replies with output values. If the challenger has played an invalid input value not allowed by the generalised test table, then the system wins. Analogously, the challenger wins if the system provides an output that is in conflict with the generalised test table. In addition, the system wins if the generalised test table has been played to the end, such that there are no more valid input values available.

We define the conformance of a system to a generalised test table based on the outcome of the plays against any possible challenger.

Definition 11.3.2 (Conformance) The reactive system P *strictly conforms* to the generalised test table T if it wins against every possible challenger for all instantiations of the global variables in T. The reactive system P *weakly conforms* to T if its strategy never loses.

11.3.3 Model-Checking Generalised Test Tables

The first step towards formally verifying the conformance of a reactive system to a generalised test table T is the normalisation of T such that the normalised version T' represents the same family of concrete test tables, but the duration column in T' only contains the constraints $[0, 1]$ (at most one cycle), $[1, 1]$ (exactly one cycle), and $[0, -]$ (arbitrary number of cycles).

The second step then is to generate input for a model checker that represents the game to be played w.r.t. T'. The system is, in particular, modelled using the set $\tilde{R}$ of rows of T' to which a given system state can correspond To keep track of $\tilde{R}$, in every move of the game, the constraint pairs $(symIn_i, symOut_i)$ for $i \in \tilde{R}$ need to be considered in the current state of the game. After each move of the challenger or the system, the row set $\tilde{R}$ is adapted. Rows that violate the pre- or post-condition are removed. Rows that can be reached by the system in this move are

added. If $\tilde{R}$ becomes empty, the last party to have moved has violated the generalised test table and loses the game.

Technically, we use the state-of-the-art model checker nuXmv [Cav+14] to verify that $\tilde{R}$ becomes empty only through the violation of the input constraint. For this we encode both, the software and the generalised test table automaton, into the SMV format. On the concept level, a product automaton is built. For *strictly conformance* we assert via an LTL that the software reaches the end of the table, represented by the sentinel, under assumption of a fair challenger. The checking of *weak conformance* is more efficient, as it can be asserted with an invariant.

11.3.4 Application Example

As an example, we consider a function block `MinMaxWarning` that is commonly used in safety-critical applications (more details may be found in [Bec+17a]). The purpose of this function block is to watch over the input values and to raise a warning if they repeatedly, for a certain number of cycles, exceed a range of allowed values that is fixed during an initial learning phase.

More precisely, the system under test is a function block `MinMaxWarning`, written in ST, with input variables `mode`, `learn`, *and* `I` and output variables `Q` *and* `W`. It operates in two modes, `Active` and `Learn`, as selected by the caller via input `mode`. During the learning phase, it learns the minimum and the maximum of the input `I` that occur while the `learn` flag is activated. When switched into the active phase, the function block checks that the input `I` stays within the previously learned interval. The output `Q` is equal to `I` if `I` is within the learned interval; otherwise, the nearest value from the interval is returned. If the input value keeps being out of range for a specified number of cycles, then the function block raises an alarm via the variable `W`. The alarm is reset after a certain cooldown time if the input value falls back into the learned interval. An unlearned function block always signals a warning. The expected behaviour of `MinMaxWarning` is partially described in Fig. 11.2.

#	Input			Output		⏲
	mode	learn	I	Q	W	
1	Active	–	–	0	TRUE	–
2	Learn	TRUE	q	0	FALSE	1
3	Learn	TRUE	p	0	FALSE	1
4	Active	–	$[p,q]$	$[p,q]$	FALSE	*
5	Active	–	$>q$	q	FALSE	5
6	Active	–	$<p$	p	FALSE	5

#	Input			Output		⏲
	mode	learn	I	Q	W	
1	Learn	TRUE	q	0	TRUE	1
2	Learn	TRUE	p	0	TRUE	1
3	Active	–	$>q$	q	FALSE	10
4	Active	–	$>q$	q	TRUE	≥ 1
5	Active	–	$[p,q]$	$[p,q]$	TRUE	5
6	Active	–	$[p,q]$	$[p,q]$	FALSE	≥ 1

Fig. 11.2 Two generalised test tables for the specification of the function block `MinMaxWarning`

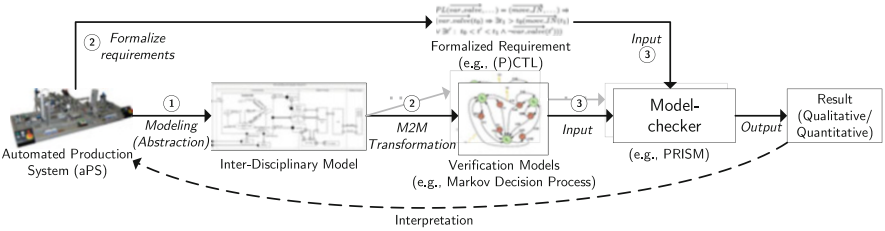

Fig. 11.3 MoDEMMiCAS approach for model-checking interdisciplinary systems (simplified)

Using the implementation of our approach, we were able to prove that a given implementation of `MinMaxWarning` conforms to the tables shown in Fig. 11.2. The `MinMaxWarning` function block consists of 61 lines of source, translated into a space of 131 bits (state and input variables) in the model checker. The verification needs 0.53 CPU seconds for proving weak conformance of the first generalised test table and 0.63 CPU seconds for the second one (median, $n = 6$). With the same setup, the verification of strict conformance takes 1.35 resp. 1.39 CPU seconds. Proving strict conformance requires an additional fairness condition to avoid infinite stuttering on the nondeterministic input variables.[2]

11.4 Model-Checking Changes in Multidisciplinary Systems

aPS are cyber-physical systems which can be best formally analysed by not only considering there software. In this section, we present an overview of techniques to verify such interdisciplinary systems by means of model checking. The general idea behind those approaches is illustrated in Fig. 11.3. First, we model the aPS by means of a formal modelling language and leveraging model abstractions (step 1). Second, for a property of interest, we select an alternative model and specification language and apply a model-to-model transformation (step 2). Finally, we run the model checker on the resulting model and for the property under consideration and obtain a quantitative or qualitative verification result (step 3).

In the remainder of this section, we present the modelling approach in more detail and exemplarily illustrate its use for verifying the system's availability based on the probabilistic model checker PRISM [KNP11] for the PPU case study [LFV13].

[2]The experiments were run on a 3.20 GHz system with Intel Core i5-6500 and 16 GB RAM with version 1.1.1 of the model-checker nuXmv. The files are available the companion website: https://formal.iti.kit.edu/ifm2017.

11.4.1 Modelling Interdisciplinary Systems

Formal System Model

The fundamental system model used in our approach is based on the FOCUS theory [BS01] to provide a strict formal semantics. Central to the FOCUS system model is the notion of components and their interfaces. Firstly, a component's *static interface*, i.e. its *typed* input and output ports, defines what signals the component may receive and send. The interfaces can be used to ensure structural compatibility between composed components. Secondly, the behaviour observable at a component's interface regarding those ports is called *semantic interface*. It is defined in terms of *behavioural functions* that map input streams to output streams. Intuitively, a stream is a sequence of messages sent or received over time on an input and output port, respectively.

Originally, the FOCUS theory was primarily conceived for modelling distributed embedded systems based on a discrete time execution. Modelling automation systems holistically by considering software and mechanical aspects originate the need of a common language for the description of physical processes and phenomena. For this reason, the FOCUS theory was extended to support continuous time elaboration and data types [Cam13, Bro12]. The behaviour of hybrid components is defined by a modified version of the hybrid automaton, called I/O hybrid state machine [Cam13]. Components that have discrete as well as continuous interfaces are referred to as hybrid components.

Formally, given a set of (typed) input ports I and output ports O with $I \cap O = \emptyset$ and types $T_{i \in I \cup o \in O}$, a continuous stream $\overrightarrow{c} \in \overrightarrow{I}$ is a function $\overrightarrow{c} : \mathbb{R}_+ \to T_i \cup \{\square\}$ that maps logical time instants to messages of type T_i. The symbol $\square \notin T_i$ denotes that no message occurred. In contrast, discrete streams are represented as partial functions $\mathbb{R}_+ \cdot\!\!\to\ T_i \cup \{\square\}$. The interface behaviour is then defined as a function $F : \overrightarrow{I} \to \overrightarrow{O}$, and we denote the set of all behaviour functions with input ports I and output ports O as $[I \rhd O]$. Input/output state machines (e.g. Mealy machines) are one particular means to specify behavioural functions which is commonly regarded as suitable for describing the system's behaviour [De +09].

Furthermore, as presented in more detail in [Mun+17], we use the probabilistic extensions developed in [Neu12] to model faulty behaviour of individual components. To this end, we generalise behaviour functions to $F : \overrightarrow{I} \to \wp(\mathbf{Pr}(\overrightarrow{O}))$, where $\mathbf{Pr}(\overrightarrow{O})$ denotes probabilistic spaces of output streams. Intuitively, it refers to a set of possible outputs and their associated probability. As input streams are mapped to sets of probabilistic spaces, generalised behaviour functions enable both nondeterministic (due to the superset) and probabilistic (due to the probabilistic spaces) behaviour specifications. To specify those behaviour functions, we extend the state machine transitions with probability values.

Finally, individual components are connected by input and output ports via *channels*. The respective interface types must be compatible, i.e. $T_i = T_o$ for input interfaces $i \in I$ and output interfaces $o \in O$. The well-defined *composition* operator

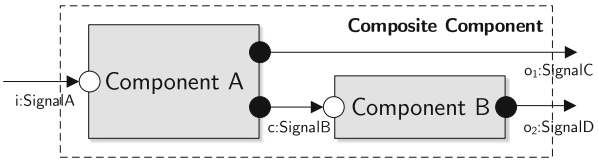

Fig. 11.4 FOCUS in a nutshell: two components connected via channel c forming a composite with semantic interface $F : \{\vec{i}\} \to \{o_1, o_2\}$

$\otimes$ ensures that a set of interfaces and channels form a composite component, where the composite component's interface is derived from the member components and their channels, thus enabling hierarchical specifications (see Fig. 11.4).

To avoid the anomalies described in [Kel77], for any set of mutually recursive components, we demand at least one component to be strongly history-deterministic,[3] i.e. require its output at time $t + 1$ to be solely determined by inputs up to time t for any $t \geq 0$.

This formal modelling extends from the one in Definition 11.2.1. In the former definition input, sequences are infinite sequences of signal values I^ω, with one value for each PLC cycle. In this section, to model physical effects more adequately, the inputs are not given as discrete traces but as continuous streams $\vec{I}$ in which every point in time may provide a value.

Model Abstractions

We apply the above formalism to model interdisciplinary systems such as automated production systems as outlined in Sect. 5.3.3. However, to cope with the inherent complexity of (continuous) physical processes, we require model abstractions which are suitable to find design errors, on the one hand, but are also amenable to automated verification on the other hand.

To this end, we model the (mechanical) context and the automation platform (e.g. bus systems, programmable logic controllers (PLCs)) using discrete abstractions obtained by combining two techniques. First, discrete time and variables behaviour can be obtained by prior simulation (see [Vog+15b]) and the use of non-uniform sampling techniques (see [Cam13]). Second, given a specific component that can be precisely specified by a function S_2, we may rely on a (more abstract) function S_1, if S_2 is a behaviour refinement of S_1. This is denoted as $S_1 \rightsquigarrow S_2$ and formally defined as:

$$S_1 \rightsquigarrow S_2 \Leftrightarrow \forall \vec{i} \in \vec{I} \,.\, S_2(\vec{i}) \subseteq S_1(\vec{i}) \,.$$

[3] See Definition 11.2.1, also called *causal*.

In addition, our model may also incorporate more abstract input and/or output ports (data types) by means of interface abstractions, defined as:

$$S_1 \overset{(D,U)}{\rightsquigarrow} S_3 \Leftrightarrow S_1 \rightsquigarrow (D \otimes S_3 \otimes U)$$

with $S_1, S_2 \in [I \rhd O]$ and $S_3 \in [I' \rhd O']$, $D \in [I \rhd I']$, $U \in [O' \rhd O]$. For more details on those refinement relations, see [BS01].

While those techniques can be used to obtain a model of various size (and accuracy), the level of detail of the resulting model we relied on can be seen in, e.g. [Leg+14].

11.4.2 Verifying Availability Requirements Using Probabilistic Model Checking

Based on the modelling approach described in the previous section, we now outline how model-to-model transformations can be applied to automated verification in terms of an example. In this example, we translate the interdisciplinary model to Markov Decision Processes (MDP), as supported by the probabilistic model checker PRISM [KNP11], to verify the specified system satisfies its availability constraints.

Translation to PRISM

A specification of an MDP consists of a set of global variables and modules. Each module defines a set of variables and a set of commands consisting of guards and probabilistic actions, i.e. updates on variables associated with a probability distribution. The standard composition of modules in PRISM is composition by interleaving; from the set of commands, at most one action is executed in each step. However, synchronous composition can be achieved by attaching a common label to commands that should always synchronise. In addition, PRISM supports to associate a number, called reward, with transitions and states (specified by logical formulas over the model's variables). Rewards can be used to quantitatively query the model.

We automatically translate our system model (extended with availability models; see also [Mun+17]) into an MDP as follows: We translate each software, platform, and context component into a PRISM module. For the syntactic interface, we introduce variables for input and output ports. For the behaviour, we encode state machines in PRISM using internal variables and commands to represent their current state and state transitions, respectively. The same translation is applied to the components of the availability models of our approach, with the sole exception of availability metrics, for which the output is mapped to rewards instead of variables. For instance, the uptime metric associates a failure-free state with a reward of 1 and a failure with 0. Finally, to achieve synchronisation among all modules, we introduce a common action label.

Verification of Availability Constraints

Based on this translation, we can use the `Rmin=?[C<=t]` query in PRISM to compute the cumulative reward up until t steps of the system have been executed. This corresponds to the system's uptime, i.e. the expected time the system is operating without failure in the time interval $[0; t[$.

11.4.3 Application to PPU Case Study

We now illustrate the approach on a concrete example of the PPU case study. For a more comprehensive presentation of this case study, the reader is referred to its publication [Mun+17].

Model As an example for the microswitch sensors used for crane positioning, we describe the particular sensor that observes whether a workpiece is pushed out of the stack and is ready for pickup by the crane. Therefore, the sensor has an input port that specifies whether a workpiece is indeed located there (as a consequence of the mechanical processes of the context model) and a single output port that outputs an electrical voltage. If a workpiece is present, the sensor outputs 24 V. Otherwise, it outputs 0 V. Consequently, in this mode, the sensor is perfectly reliable and available all the time.

To account for availability issues, we extend this with a deviation model that models two failures, namely, temporal unavailability (e.g. due to pollution) and permanent unavailability (e.g. permanent damage due to wear out). This deviation model is illustrated in Fig. 11.5. Therein, we introduce an activation function (`Micro-switch Failure Activation`), an input filter (`IF: Identity`), and an output filter (`OF: Missing WP`). The activation function signals whether the microswitch failed to the input and output filter by means of a probabilistic state machine illustrated at the bottom of the figure. Initially, it is in the "Available" state. The output filter outputs 0V in case of a failure or behaves as specified above otherwise. Therefore, the failure activation causes the sensor to potentially miss workpieces located at the stack. In contrast, the physical phenomenon of the workpiece position is not altered by the deviation model. The input filter is modelled as an identity function for the sake of illustration which merely forwards the workpiece position.

Verification To analyse the availability of the crane's transportation function, the model can be translated to the PRISM model checker. Essentially, we provide a component that represents an uptime metric which associates rewards of 1 and 0 with a timeliness or (potentially infinitely) delayed transportation, respectively. Then, we can verify that the required availability is achieved by querying

```
Rmin=?[C<=36000],
```

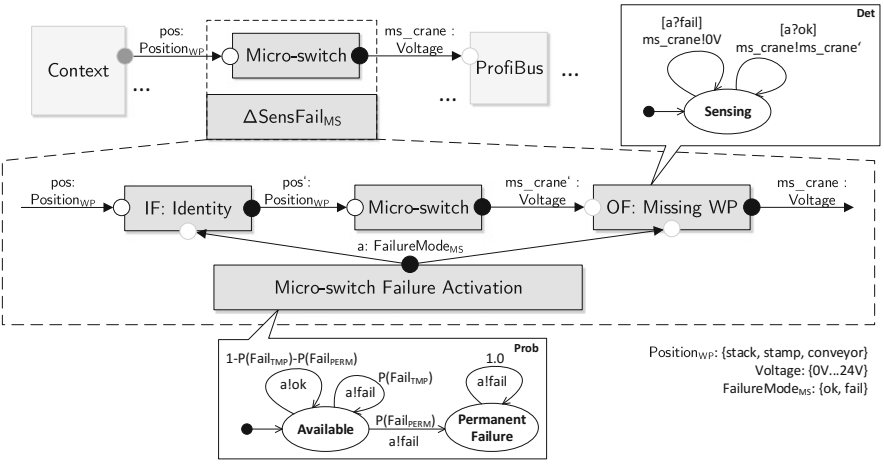

Fig. 11.5 The extended system model of the microswitch sensing whether a workpiece is available for pickup by the crane at the stack position using a probabilistic state machine for activation

where `Rmin` refers to the cumulative rewards associated with the uptime metric within 1 h (i.e. 36,000 ticks with a 100 ms cycle time), and comparing the resulting value to the availability requirement, e.g. 0.9965 if an 99.65% availability of the transportation function is required.

11.5 Related Work

The verification of PLC programs w.r.t. temporal logic specifications (for safety, liveness, and time properties) has been subject of a number of publications already. The paper [YF03a] gives an overview of the field, and the survey [Lam+99b] discusses transformation processes for program languages to verifiable models. Various translations from IEC 6113-3 languages into the input languages of model checkers have been presented: Brinksma et al. [BMF02] present a translation of SFCs into Promela input for the SPIN model checker [Hol97]; De Smet et al. [Sme+00] translate all languages within IEC 61131-3 into input for the symbolic model checker Cadence-SMV [Bur+92]; and Bauer et al. [Bau+04b] translate SFCs into timed automata to be used with UPPAAL [Beh+01]. This model checker is also used to verify properties of continuous function charts (CFC) in [WFV09]. In [Bau+04a, BHL00] a unifying semantics for SFC is given where the ambiguities of the standard are addressed in a formal fashion.

Süflow and Drechsler [SD08] present a framework to verify that the *same* program behaves equivalently on *different* PLC platforms, a scenario closely related to ours. The authors employ a SAT solver to verify the arising proof conditions.

Strichman and Godlin [GS13] coined the term *regression verification* and presented a verification methodology based on replacing function calls by uninterpreted

function symbols within a bounded software model-checking framework for C programs. In [GS08] they define "reactive equivalence", which is closely related to our notion of perfect trace equivalence. In earlier work [Fel+14], we presented an automated approach to regression verification based on invariant generation using Horn clauses. Many other approaches [Ver+10, VJB12, Haw+13, BCK11, WP12] exist on regression verification for imperative programming languages.

Equivalence checking is an established issue for the verification of hardware circuits. In *sequential equivalence checking*, the perfect trace equivalence between clocked circuits is analysed; see [HC98] or [KE02] for an overview. Lu and Cheng [LC09] present an approach based on inferred invariants, in which conditional or relational equivalence is not considered.

Table-based languages for visualisation of mathematical are common. For example, *Parnas tables* are a tabular representation of relations. Lorge et al. [PMI94] use them in addition to first-order logic for the specification of procedure contracts.

Also, *Software Cost Reduction* approach (SCR) [Hei+05] claims to be understandable and comprehensible by exploiting table-based syntax. SCR is a method for managing formal requirements, which was successful applied in practice, e. g. mission-critical systems from the NASA [HJ07]. It bases synchronous state machines to describe the behaviour of a system. The state machine is specified by tables, similar to the Parnas tables, to define the transition relation and the output relation. SCR benefits from a various tools that are built upon the formal semantics: the simulation and validation of specifications, the generation of system invariants and source code, and the formal verification of application properties. A commonality between an SCR specification and generalised test table is that both describe an automaton. For generalised test tables, this automaton is given by the transformation rules and is therefore restricted. Otherwise, generalised test tables are optimised for specification of sequential stimulus and responses. They allow the direct access to past values via back references or global variables; SCR requires an encoding of these values into the state.

CocoSpec [Cha+16] is a specification language for reactive programs that are written in the *Lustre* programming language. *CocoSpec* follows *assume-guarantee* paradigm using Boolean expression for specifying assumptions and assertion on the current in every time step. Like SCR, *CocoSpec* exploits a state machine to make these assertions and assumptions time-dependent. The state machine is written in *Lustre*. In contrast, the assumptions (input) and assertions (output) of a generalised test table are always time-dependent, i.e. they depend on the table rows.

Moszkowski [Mos85] follows with his Interval Temporal Logic (ITL), a different approach to the classical temporal specification languages CTL and LTL. ITL bases regular expression and therefore it is ω-regular. For concatenation, ITL introduces the chop operator $(r_1; r_2)$ which – similar to our concept of rows – divides a given trace into a suffix and prefix, where r_1 has to be valid on the suffix, resp. r_2 on the suffix. An unbounded repeated application concatenation is denoted by star operator r^*, identical to our "–" in the duration column. A generalised test table can be expressed as an ITL formula, under the costs of an exponential blow-up [Bec+17a].

The idea of using regular expression can be combined with Linear Temporal Logic (LTL) as *ForSpec Temporal Logic* (FTL) proves. FTL was developed by

Intel [Arm+02]. In addition to LTL operators (until, always, eventually), it supports the corresponding past operators, *regular events* and *time windows*. A regular event is a finite regular language in a similar fashion as ITL or generalised test tables. Time windows are helpful to specify that certain events need to occur with a defined time window (bounded LTL operators). Additionally, FTL allows the composition with temporal connectives (a composition of generalised test tables is possible on the automata level). In [Lju+10], Ljungkrantz et al. propose ST-LTL, which enriches LTL with the arithmetical operators of Structured Text, syntactical abbreviations for specifying the rising or falling edges of variables, and access to previous variable value.

Becker et al. [Bec+12] present the MechatronicUML language for modelling and analysing component-based software for mechatronic systems, which supports links between engineering disciplines. SysML4Mechatronics [KV13] is a language for interdisciplinary modelling, which addresses mechanical, electrical/electronic, and software aspects explicitly. A formal semantics for automatic verification of structural compatibility has been proposed [FKV14], but verifying functional conformance is not considered yet. Shah et al. [Sha+10] present a multi-discipline modelling framework based on SysML.

Various lines of research are meant to analyse the reliability and availability of technical systems. Several established modelling techniques, such as Reliability Block Diagrams (RBD) [Bir10, DP09] and Fault Trees [DBB92], are based on combinatorial models. In both cases, the diagrams model which elementary faults lead to a failure of the whole system or a system function. A problem with these kinds of models is that only rather simple scenarios can be captured [Bir10]. Besides those combinatorial approaches, several approaches for an architecture-based availability prediction have been proposed (see, e.g.[Kub89, Lap84, Lit79]). More recently, availability analysis also gained widespread attention for the domain of aPS. In [Lai+02], a general model is built based on the Markov model to predict the availability of distributed software and hardware systems. The authors use the Kolmogorov differential equations to calculate the probability that a system process is in a certain state and then derive the availability function for the respective system.

11.6 Conclusion

In this chapter, we have presented the opportunities and challenges for the application of formal verification during system evolution of automated production systems that we identified from our experiences in the projects MoDEMMiCAS and IMPROVE APS within the priority programme.

In addition, we have presented three approaches for formal verification of evolutionary steps that exploit the opportunities that automated production systems provide and address the challenges that arise:

Regression verification uses an older revision of the PLC software as specification for a newer release and allows one to prove that desired aspects of the system

behaviour are retained by the evolution step. It thus leverages evolution by using the old code revision as specification. Model checking is feasible due to the limited structural complexity.

Generalised test tables allow specifying desired system behaviour as tables. They thus address the challenge of reducing the specification efforts by providing a user-friendly specification technology.

Model-checking interdisciplinary models translates multidomain models of automated production system, composed of software, automation hardware, and mechanical components into representations amenable to model checking. To verify that the software achieves the intended behaviour at the system level, we rely on a common formal modelling approach for all system components, which may also include continuous behaviour and make extensive use of model abstractions. We claim that the initial modelling effort may be justified by the longevity of such systems and the lack of effective alternatives, while the software's limited structural complexity benefits model checking despite the potentially large space state.

All techniques perform a full verification of the properties they claim to be true by using modern model-checking tools. Automated production systems have a limited state space by design and are thus suitable targets for such formal verification systems.

One cross-cutting challenge for all three presented techniques is that they need to make assumptions about the behaviour of the physical plant on which the software is deployed. The interdisciplinary approach relies on an explicit model for the environment as part of the verification input. Generalised test tables have columns for input signals such that signal sequences can be restricted to those occurring in practice. Thus, a plant model is specified implicitly. Regression verification often works without environment models, but not always. There are cases where parts of the plant behaviour need to be added as an explicit.

The presented approaches exemplarily demonstrate the ability of formal verification to provide valuable support and feedback in engineering long-living systems, especially automated production systems, thus suggesting a promising field of application for future research and motivating transfer into engineering practice.

11.7 Further Reading

The interested reader is invited to find more and more detailed information about the presented verification approaches in the following scientific publications:

The idea of regression verification for PLC programs developed in the project IMPROVE APS within the priority programme has originally been presented by Beckert et al. [Bec+15]. Ulewicz et al. [Ule+15] have shown how the presented regression verification approach can be extended to comparing different variants of PLC software in order to reduce unneeded variant diversity. Moreover, we show

in [Ule+16] how the regression verification approach for PLC code can be embedded into the development process for aPS software.

Regression verification can not only be applied to PLC software, but a similar approach has been presented by Kiefer et al. [Fel+14, KKU16] for the automatic regression verification of C programs. The tool LLRÊVE compares two C routines for various types of equivalence. It can be applied to programs with certain heap data structures [KRU17] and combines static and dynamic analyses to extend the reach of the regression verification approach [KKU17]. LLRÊVE can be accessed as a publicly available web application: https://formal.iti.kit.edu/projects/improve/reve/. The tool *semantic slicer* [Bec+17b] employs LLRÊVE to produce very precise slices that a syntactical analysis cannot find. In [BKU15] Beckert et al. reduce Java regression verification problems to equivalent secure information flow problems on the JML* specification language and the KeY prover [Ahr+16].

Generalised test tables were first introduced by Weigl et al. [Wei+17], and their formal semantics was defined by Beckert et al. [Bec+17a]. The automatic verification tool GETETA that proves that a PLC program behaves as specified in a generalised test table is an open source project hosted at github. Current releases and more information can be found on the companion webpage https://formal.iti.kit.edu/geteta/.

The *ST Verification Studio* (STVS) is a tool that provides a user-friendly frontend for the specification and verification of PLC software using generalised test tables. It is presented in Fig. 10.11, and details can be found on the companion webpage https://formal.iti.kit.edu/stvs/.

The multidisciplinary modelling approach of our verification is based on a model-based development for cyber-physical systems [Bro97, Hub+98, Sch+02, Bro+10], which is extended to automated production systems by [Leg+14]. Essentially, it extends the FOCUS theory described in [Bro86, BS01] with notions of discrete and dense time [Bro12], spatio-temporal systems [Hum09, Bot+09], and using dynamic sampling [Cam13]. Finally, those concepts were implemented in the AutoFOCUS 3 tool, which is described in more detail in [Hub+96, SHT12, Ara+15]