Suhyun Cha*, Alexander Weigl, Mattias Ulbrich, Bernhard Beckert and Birgit Vogel-Heuser

Applicability of generalized test tables: a case study using the manufacturing system demonstrator xPPU

Anwendbarkeit von Generalized Test Tables: Eine Fallstudie anhand des xPPU-Demonstrators

Abstract: With recent trends in manufacturing automation, control software in automated production systems becomes more complex and has more variability to keep pace with customer and market requirements. Quality assurance also becomes more and more important to ensure that the systems live up to expectations. However, correct-ness of automation software is rarely verified using for-mal techniques in spite of their high coverage. One of the main reasons is the lack of specification languages suit-able for this application area that are both comprehensible and sufficiently expressive. Generalized test tables (GTTs), which are a specification language for reactive systems, were presented recently as an accessible representation for application engineers. This formalism achieves both the comprehensibility of concrete test tables and the cov-erage of formal methods. In our approach, the specifica-tion provided by GTTs is used for formal verification, es-pecially model checking. In this paper, we present four new features for GTTs: the progression flag, strong repetition, row grouping, and specification on internal variables. We demonstrate the applicability and evaluate the comprehensibility of GTT-based specification and verification using a range of diverse scenarios from the community demonstrator, the extended Pick & Place Unit.

Keywords: formal verification, formal specification, functional specification, software engineering, manufacturing system engineering

Zusammenfassung: Steigende Kunden- und Marktanforderungen in der Fertigungsautomatisierung erfordern komplexere Steuerungssoftware und kürzere Entwicklungszyklen. Um zukünftig Korrektheit und Zuverlässigkeit sicherstellen zu können, ist eine Anpassung an der Qualitätssicherung erforderlich. Formale Methoden können hierfür nachprüfbare Garantien bieten, aber obwohl Automatisierungstechnik in unternehmenskritischen Bereichen eingesetzt wird, werden formale Methoden dort selten verwendet. Einer der Gründe ist der Mangel an geeigneten Spezifikationssprachen für die Automatisierungsdomäne, die sowohl nachvollziehbar als auch ausreichend aussagekräftig sind. Generalized Test Tables (GTTs) sind eine formale tabellen-basierte Spezifikationssprache für reaktive Systeme für den Entwicklungsingenieur. GTTs erhöhen die Aussagemächtigkeit und Testabdeckung konkreter Testtabellen unter Beibehaltung ihrer Verständlichkeit. In diesem Beitrag analysieren wir die Anwendbarkeit und Verständlichkeit von GTTs. Dazu spezifizieren wir Teile des Anlagenverhaltens diverser Szenarien aus dem Community Demonstrator Pick&Place-Unit (PPU).

Schlagwörter: formale Verifikation, formale Spezifikation, funktionale Spezifikation, Software-Engineering, Fertigungssystem-Engineering

***Corresponding author: Suhyun Cha,** Technical University of Munich, Institute of Automation and Information Systems, Boltzmannstr. 15, 85748 Garching near Munich, Germany, e-mail: suhyun.cha@tum.de
Alexander Weigl, Mattias Ulbrich, Bernhard Beckert, Karlsruhe Institute of Technology, Application-oriented Formal Verification, Am Fasanengarten 5, 76131 Karlsruhe, Germany, e-mails: weigl@kit.edu, ulbrich@kit.edu, beckert@kit.edu
Birgit Vogel-Heuser, Technical University of Munich, Institute of Automation and Information Systems, Boltzmannstr. 15, 85748 Garching near Munich, Germany, e-mail: vogel-heuser@tum.de

1 Introduction

Automated Production Systems (aPS) and their engineering become more and more complex, following current trends such as, e. g., increasing customer flavor variety [21]. The proportion of system functionality realized

by software is increasing [18]. A malfunctioning software may cause damage to the system itself, the payload, or even harm persons within the reach of the system. Therefore, effective software quality assurance is essential [8]. aPS are usually automated with Programmable Logic Controllers (PLCs), and these computing devices are expected to control aPS with assured quality in dependable or safety-critical real time environments [20]. The set of guidelines defined in the Good Automated Manufacturing Practice (GAMP) by the ISPE (www.ispe.org) is an example for quality criteria for the validation of aPS (in the pharmaceutical domain).

In today's industrial practice, software quality is commonly achieved by dynamic validation either through manual step-by-step testing or automatically generated test cases [14]. However, the main weakness of traditional testing is that one test case covers only a single, particular run of the aPS software. Moreover, developers are often under pressure to meet a delivery deadline [21]. This implies that the full system behavior is usually not covered during validation, and that some scenarios remain untested. Thus, software correctness cannot be proven completely. Unpredictable and rare malfunctions may remain undiscovered, which can have severe consequences.

In contrast to testing, formal verification achieves full coverage by proving the correctness of an implementation mathematically and exhaustively with respect to its formal specification. Moreover, formal verification allows for an easy re-validation, which is, e. g., required by the GAMP guidelines for each change of a safety-critical aPS. Thus, there is a need to support formal verification of PLC software [8].

But formal verification is not commonly used, yet, to verify the correctness of implementations. One of the main reasons is that it is difficult to specify the desired temporal properties of a system, since that requires expert knowledge on formal specification languages [13]. Even worse, in many cases not even an informal description of the requirements is available that could be used as a basis for a formal specification.

To tackle this problem, we have extended the concept of test tables, which are commonly used to describe test cases in table form [19]. In recent works [24, 3], we suggested an approach to support quality assurance by generalizing test tables such that they can be used for formal verification purposes in addition to testing.

While non-formal behavior verification, e. g., using a conventional testing process, is largely based on the actual execution of a system, formal methods provide proofs ensuring software correctness with respect to the specification without actually executing the system [12]. Formal verification uses logic-based deductive techniques for these proofs. In our approach, we use model checking, which checks for desired behavioral properties by systematically exploring all states of the system's model [15].

In this paper, we demonstrate the applicability and evaluate the comprehensibility of specification and verification using generalized test tables (GTTs). Their expressiveness is analyzed using a range of diverse scenarios from the community demonstrator, the extended Pick & Place Unit (xPPU) [21]. A further contribution are additional features for GTTs that allow to better express repetitive behavior.

2 Related work

Though formal verification of aPS behavior is a recent research topic, there has been work on verifying PLC software using model checkers, e. g., [2, 23, 5]. Regression verification is a variant of verification focusing on ensuring that no regressions, such as software bugs or undesired behavior, are introduced by changes to an evolving system. Starting from [17], many approaches have been developed for such relational proofs. In recent work, these ideas have been adapted to PLC software [4].

Software Cost Reduction (SCR) is a formal requirements method, that was applied to mission-critical systems by NASA [9]. SCR uses synchronous state machines to describe the behavior of a system. State machine specifications use a "user-friendly" table-based notation for the transition relation and the output relation. SCR provides various tools for the simulation and validation of specifications, the generation of system invariants and source code, and the formal verification of application properties.

CocoSpec [7] is a specification language for reactive programs that are written in the Lustre programming language. Similar to GTTs, CocoSpec is based on an *assume-guarantee* paradigm using constraints on input values (assumptions) and output values (assertions) in every time step. The constraints are Boolean expressions following the semantics of Lustre. Using a state machine, assertions and assumptions become time-dependent in CocoSpec. In GTTs, in contrast, assumptions and assertions are always time-dependent, i. e., they depend on the table-rows.

The ForSpec Temporal Logic (FTL) is an extension of Linear Temporal Logic (LTL) developed by Intel [1]. In addition to LTL operators (until, always, eventually), it supports the corresponding past operators. Moreover, FTL adds some features that are of interest w.r.t. GTTs. For example, FTL supports the specification of time windows, in

which certain events need to occur (bounded LTL operators). And, FTL allows the description of *regular events*, which are sets of finite state sequences described by regular expressions.

In [13], several user-friendly specification languages are presented and compared. Unfortunately, formal specification languages such as temporal logic are still a barrier for the application developers to understand and use. Most of the languages for describing the relative order of events, such as Computational Tree Logic (CTL) and LTL, require a level of expertise that is currently not found in developers that have little experience in using formal specifications [10]. This lack of expertise in using logical formulas is a main motivator for the development of GTTs.

3 Introduction to GTTs

In this section, we introduce the syntactical concepts of GTTs, explain what it means for a system to conform to a GTT, and describe how conformance is proven.

3.1 Concrete test tables

A (non-generalized) *concrete* test table describes a test protocol consisting of a sequence of input values (provided by the environment) and expected output values (computed by the PLC software under test). Every input and output variable has its dedicated column in the table. Every step in the protocol is represented by a row in the table. A system can be checked against the test case described by a concrete table by providing the input stimuli to the system row by row and checking that it responds as specified with the expected output values.

Instead of repeating a row several times, the number of repetitions can be annotated in the special table column DURATION (⏲).

3.2 Abstractions in GTTs

GTTs follow the same principles as concrete test tables, but go beyond them by introducing three means to abstract from concrete values: (1) abstraction using constraint expressions, (2) using references to other cells in constraint expressions, and (3) using generalization in the duration column (see Fig. 2 for a simple example GTT).

Though a GTT covers a (possibly infinite) set of concrete behaviors, it usually does not fully specify a system but only its behavior for a certain situation or scenario.

Abstraction using constraints. Whereas cells of concrete test tables contain concrete values, the cells in GTTs describe constraints, such as "$X \geq 0$", "$X - 1 = 0$", or "$X > 3 \wedge X < 10$" that may be satisfied by many values. These constraints are formulas over the input and output variables. Their syntax and semantics follow those of expressions in the programming language STRUCTURED-TEXT [11].

To make GTTs more readable, we allow a number of abbreviations. They are shown in Fig. 1. Furthermore, we use a vertical line (cf. Column IN, Fig. 2) over consecutive cells of a column to mark the repetition of the upper constraint in the consecutive cells.

Abbrev.	Constraint	
n	$X = n$	
$< n$	$X < n$	(same for $>, \leq, \geq, \neq$)
$[m, n]$	$X \geq m \wedge X \leq n$	
—	$X = X$	(don't care)

Figure 1: Constraint abbreviations (X is the name of the variable that the cell corresponds to; n, m are arbitrary expressions of type integer) [24].

References to other cells. A reactive system may possess an internal state, i. e., its behavior may depend not only on the current but also on previously observed input values. GTTs have two expressive means to formulate such dependencies: global variables and past references.

Global variables can be used in constraints. They are denoted by lower-case letters and have a fixed value which does not change throughout a run. For example, if v is a global variable, $A = v$ occurs in one cell and $X = v + 1$ in another cell, then this requires that $X = A + 1$. While the value of a global variable v does not change in a single run, there may still be several values for v that satisfy all constraints. Using $X = v$ in a cell for X is equivalent to "don't care" if this is the only occurrence of v.

Past references are relative references to previous values of variables. A past reference "$X[-n]$" refers to the value of variable X which it had $n \in \mathbb{N}$ cycles before the current one. A past reference refers to the system iteration n cycles ago, not to the nth row above the current row (this may differ because rows may be repeated). Absolute references to particular cells can be expressed using global variables.

Generalization in the duration column. The entry in the DURATION column determines the number of repetitions for each row. In this column, only intervals with natural numbers as bounds are allowed as constraints [3], i. e.,

constraints of the form "$[n, m]$", "$\geq n$", and "—" ("don't care") are the only possibilities (with m, n natural numbers). If a duration constraint includes the value 0, then that row is optional and can be skipped. Note that for systems with a constant cycle time, duration can also be specified as time intervals (in ms) instead of as number of repetitions: time intervals are converted to repetitions by dividing them by the cycle time.

Example: PULSE TIMER. Figure 2 shows an example GTT specifying the PULSE TIMER function block from IEC 61131-3 [11]. As long as the timer receives the input `IN = FALSE`, it keeps waiting and signals constant values (Row 0). When the PULSE TIMER is started, which happens when `IN` is set to `TRUE` (Row 1), the software must output `Q = TRUE` (signalling that the timer is running) for a period whose duration is specified by the pulse time provided as input in `PT`. Note that the pulse time is measured in milliseconds, but that the value shown in the table actually represents an integer, namely the time shown divided by the cycle time. In this GTT, we require that the input `IN` remains `TRUE` and `PT` remains constant as long as the timer runs. The latter is expressed using `=PT[-1]` in Rows 2–4, which requires `PT` to have the same value as in the previous cycle.

#	INPUT		OUTPUT		⏲	
	IN	PT	ET	Q		
0	FALSE	—	0 ms	FALSE	≥ 1	⊤
1	TRUE	$\geq$1 ms	$\geq$ET[−1], $<$PT	TRUE	1	
2	∣	=PT[−1]	$\geq$ET[−1], $<$PT	TRUE	≥ 0	$\_\_\__\infty$
3	∣	∣	=PT	—	1	
4	⊥	⊥	=PT	FALSE	≥ 1	⊥

Figure 2: A simple GTT for the PULSE TIMER function block defined in IEC 61131-3 [11]. The vertical lines in columns IN and PT denote a repetition of "TRUE" resp. "=PT[−1]".

Rows 2 and 3 correspond to the state in which the timer is running and the elapsed time `ET` has not reached the pulse time. The entries "1" resp. "≥ 0" in the duration column specify that the timer must be in this state for at least one cycle. While the timer runs, the elapsed time output `ET` must monotonically increase ("$\geq$ `ET[-1]`").

When the elapsed time reaches the pulse time, the output `Q`, signalling that the timer is running, must switch from `TRUE` to `FALSE` (Row 4). Interestingly, the GTT leaves the exact switching behavior unspecified, allowing the PLC software to signal `TRUE` or `FALSE` in the first cycle where the pulse time is reached (`Q` is "don't care" in Row 3).

The GTT contains a strong repetition "$\_\_\__\infty$", which means that repeatedly starting the timer is possible.

3.3 Conformance

The behavior of a reactive system S is the set of its possible runs, modeled by a set $\mathcal{B}(S)$ of infinite sequences of inputs and outputs. These sequences can also be interpreted as (infinite) concrete test tables. Similarly, a GTT G describes a set $\mathcal{T}(G)$ of concrete test tables, which arise from G by unwinding rows according to the DURATION column and by instantiating all cells with values that satisfy the constraints. The set $\mathcal{T}(G)$ may comprise both finite and infinite instances of G. See [24] for details.

Intuitively, S conforms to G if all its possible runs $b \in \mathcal{B}(S)$ conform to G. A single run b conforms to G iff one of the following cases holds: (1) $b \in \mathcal{T}(G)$, (2) there is an instance $t \in \mathcal{T}(G)$ which is an initial sub-sequence of b, or (3) there is *no* concrete test table in $\mathcal{T}(G)$ whose input sequence is the input sequence of b or an initial subsequence of it. In the first two cases, b is covered by G: The behavior b is a possible (finite or infinite) concretization of G. The third case covers the situation in which G does not include the scenario to which b belongs at all. If the sequence of input values in b is not present in $\mathcal{T}(G)$, then the table does not make a statement about how the software should react to these inputs.

3.4 Verification

The question of whether a PLC program conforms to a GTT is decidable as the state space of PLC programs is always bounded.

Based on the state-of-the-art model checker NUXMV [6], we have built an automatic decision procedure for conformance checking (Fig. 3). Our tool translates the PLC program, given as STRUCTUREDTEXT or SEQUEN-

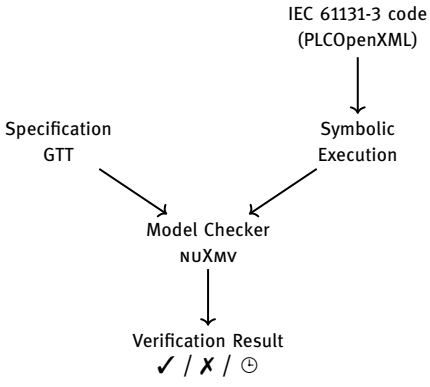

Figure 3: A schematic view of the procedure deciding conformance of PLC software to GTTs. The result of checking conformance is either (a) that the software indeed conforms to the GTT, (b) a counterexample, or (c) a time out due to limitations on resources.

tial Function Chart, and the GTT into a combination of two automata and formulates conformance as an invariant on them. The program automaton describes the transition relation between two PLC cycles and is obtained from the PLC source code using symbolic execution and a transformation into single static assignment form. The GTT is translated into an automaton that keeps track of the table rows to which the current state may correspond (which can be more than one). The program conforms to the GTT if this automaton either reaches an end state or if no row is active any more. It violates the specification if the output values violate the output variable constraints of all currently possible rows.

We have implemented two tools that support the construction and verification of GTTs. The backend GETETA constructs automata for the software and the GTT and launches the model checker. The graphical user interface STVS provides a user frontend with support for the visualization of counter-examples and the generation of concrete test tables.[1]

4 Extensions for GTTs

In addition to the concepts of GTTs presented earlier [24], we introduce in this paper four extensions, mainly possibilities to formulate DURATION constraints: the progress flag, strong repetition, row grouping, and columns for internal state variables. Although the expressiveness is not extended by the new features, they enable better comprehensibility and efficient expressions.

4.1 The progress flag

The progress flag is a supplementary annotation to a duration interval, denoted by a subscript $\cdot_p$. If a row is annotated with the progress flag, then the test must progress to a subsequent row if possible. For example, "$—_p$" is, like "—" (don't care), a repetition of arbitrary length, but unlike "—", the flag requires that the execution continues with a successor row if possible. Only if that is not possible, the current row is repeated. Using the progress flag ensures that the test does not get stuck unnecessarily.

Moreover, the progress flag helps to specify the deterministic software requirements concisely and leads to more comprehensible test tables. A typical pattern in specification is waiting for a trigger event to occur, and then to proceed as specified. In a GTT, this is expressed by two successive rows: the first row allows all input and output values for an arbitrary duration. Its successor row specifies the trigger event by input constraints (See Rows 0 and 1 in Case 2 in Sect. 6.2). When the second following row is satisfied, the first row needs to be vacated in favor of the second. Without the progress flag, any system would be weak conform to this table because the system can not violate the first row.

An equivalent specification without "$—_p$" can be obtained by including the negation of the input constraints of the following rows to the current row and using "—" as duration. But this leads to tables that are unnecessarily difficult to read.

4.2 Strong repetition

Strong repetition (indicated by using "$—_\infty$" instead of "—" in the DURATION column) denotes that a row is to be repeated infinitely often. This is needed to specify that a desired pattern must *never* be violated. Strict safety requirements are typical use cases of strong repetition. The don't-care duration "—" is not suited for specifying such properties since a table row with "—" is already satisfied if the software conforms to a finite number of times – or even zero repetitions.

The strong repetition in a row prevents a system from being *strict* conform to the table, since the end of the GTT cannot be reached. Therefore, if all possible paths through a GTT end up in strong repetition, there does not exists a *strict* conform system to this table. The App. B gives a formal explanation.

For every GTT T_∞ that contains strong repetitions, we can construct a GTT T_* with equal (*strict* and *weak*) conformance for every system. T_* is obtained by replacing every strong repetition with a "don't-care", and adding a sentinel row that prevents the row is ever left. The construction and proof are given in App. C.

4.3 Repetition of multiple rows

The third extension is the grouping of consecutive rows to be repeated as a row group. Every row group has its own additional DURATION constraint. Row groups can be nested, i. e., a group may contain other groups and rows (but they cannot partially overlap). With row groups, we

1 Both tools are available online at https://formal.iti.kit.edu/geteta resp. https://formal.iti.kit.edu/stvs

can express specifications composed of repetitive or optional sections that span over multiple rows. The sections can represent different state of the software, e. g., initialization, error recovery, or automatic and manual operation.

As an example, an error handling function of a conveyor belt is shown in Fig. 4. This shows complicated nested row groups on a conveyor belt behavior. Row 2 defines the triggering the conveyor and an error might directly occur without this conveyor trigger. The row group covering only Row 2 makes this row optional (see the duration defined as $[0, 1]$). Rows 3 to 5 form the error recovery by moving the work piece back and forth up to three times. The error recovery is exited when the work piece reappears at the beginning and the error is reset as specified in Row 6.

We present the revised definitions in App. A.

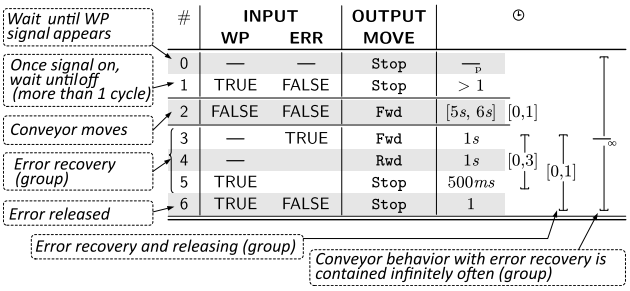

Figure 4: Example of a nested GTT for a conveyor belt.

4.4 Expression over state variables

We allow a column to designate not only an input or output variable, but also an (internal) state variable of the software. This enables the specification of internal behavior of the system, e. g., the specification of invariants or changes of global variables.

The state column can either be categorized as an input or output, resulting in different interpretations in the semantics. As an output column, the state column behaves equally as if it would be an output of the system: a violation of the corresponding constraint leads nonconformance of the system. The constraint is an assertion. If the column is categorized as an input, its constraints are a prerequisite or assumption for the application of the corresponding row. The value of the state variable is determined by the system and can not be chosen by an environment, as any normal input variable. But a constraint violation on the input state column does not lead to nonconformance of the system.

Beside the conformance, the categorization also effects the evaluation of the corresponding constraint. By default, a state variable refers to the last value. As an input column the state variable refers to the previous cycle (because the input constraints are evaluated before the system is executed). As an output column, the state variable refers to the value after the execution of the system. To make a clear distinction, we use $X_{\mathtt{pre}}$ or $X_{\mathtt{post}}$ to refer to the value of a state variable before and after the execution of the system. Technically, an input column for state variable $X_{\mathtt{pre}}$ has always an implicit constraint $= X_{\mathtt{post}}[-1]$. Their value in the first cycle is determined by the standardized default values or user-defined in the variable declaration (cf. [11]).

5 Demonstrator: The xPPU

The Pick & Place Unit (PPU) is a lab-size manufacturing plant demonstrator [21] to benchmark evolution scenarios for aPS, and the extended PPU (xPPU) [22] is a recent extension of that demonstrator (see Fig. 5). It has been established within the DFG priority programme SPP 1593 in Germany as a common case study for evolution in plant and machine automation. Although the xPPU is quite simple, it realizes the basic functionalities representative in intralogistics systems as identified by Spindle et al. [16]. The original PPU consists of four modules: a stack as a loader for work pieces, a stamp as a manipulation demonstrator, a conveyor as a sorting unit, and a crane as a transporter to transfer work pieces between the other modules. The extension adds more modules and features, e. g., a reordering module for logistic flexibility named PicAlpha, reinforced security and safety, product recognition, and more. A wide range of evolution scenarios have been defined and implemented for the (x)PPU with different moti-

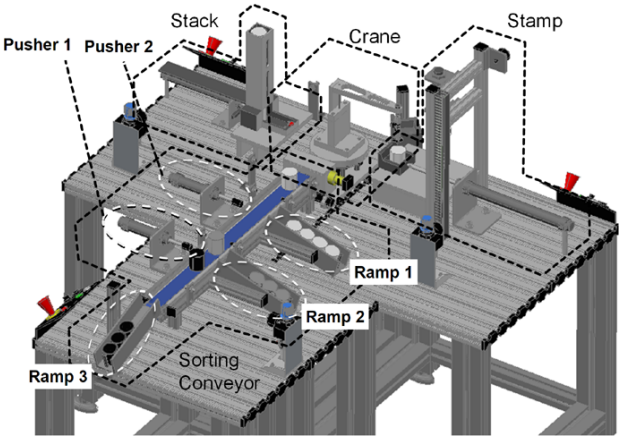

Figure 5: Community demonstrator: the Pick & Place Unit.

vations (e. g., changing requirements, fixing failures, and unanticipated situations on site).

In the xPPU, work pieces are processed differently depending on their type (black plastic, white plastic, and metal). Work pieces that are to be processed are loaded into the stack. From there, a cylinder separates an individual work piece which is then on standby to be delivered by the crane either (a) directly to the sorting module or (b) first to the stamp and then – after stamping – to the sorting module.

Transportation by the crane module requires a series of actions: turning to the target, stopping, lowering the arm, gripping the work piece with pneumatic pressure, raising the arm, turning, stopping, lowering the arm, releasing the pneumatic pressure, and raising the arm again. Turning/stopping and lowering/raising are implemented by a motor and a cylinder. In the stamping module, once a work piece has been placed into the container at the end of the slider, the slider cylinder is retracted down to the stamping point where a vertical cylinder with a stamp at its tip is installed. As soon as the work piece has been stamped, the slider is extended to move the piece back to its arrival position. In the sorting module, once a work piece has been placed at the end of the conveyor belt, it is taken to the pusher point by a motor. The pusher is implemented by a cylinder that pushes out the work piece onto the ramp on the opposite side.

The case study which we specified using GTTs and verified is a collection of three xPPU scenarios that are variations of this basic setup. They differ in their mechanic, electric, electronic and/or software configuration.

6 Case study: Using GTTs for specification

In this section, we present three xPPU scenarios, which we use as the application cases of a case study, and show how behavioral patterns can be specified using GTTs. The formal verification of the PLC software w.r.t. the presented GTTs is discussed in Sect. 7. The application cases were selected such that a variety of GTT features is covered, but also to demonstrate that various behavioral patterns of aPS can be specified and verified using GTTs. For printing we omit variables in the presented GTTs. A detailed version of the GTTs can be found on the companion website[2] along with the verification models and results.

2 Companion Website: https://formal.iti.kit.edu/at2018

6.1 Application Case 1: Emergency stop

Once an unexpected or abnormal situation is detected by the software, it has to initiate the right halting actions immediately to avoid damage. After stopping in a safe configuration, the xPPU is switched into manual mode, and the operator may be asked to operate the aPS manually to allow an automatic restart, or to prevent further damage. For the purposes of initiating an emergency stop, the xPPU has three emergency-stop push-buttons at the stack, the stamp, and the sorting module. As soon as one of these buttons is pressed, the emergency-stop process is initiated and the following actions are taken: The pneumatic cylinders in the stack are retracted, the rotation of the crane is stopped, the conveyor of the sorting module is stopped, and the pushers on the conveyor are all retracted.

The emergency-stop procedure for the pneumatic pressure on the crane's gripper is more delicate, as turning off the vacuum without taking the current state of the gripper into consideration may allow a work piece to fall to the ground. Therefore, if a work piece is currently being gripped, then the gripper must continue to hold onto that piece until the emergency procedure has terminated (or the work piece is removed manually).

The GTT in Fig. 6 specifies this emergency stop behavior. Note that this particular table only describes the emergency routine; to fully specify the PLC behavior, other GTTs for non-emergency situations would need to be added. Row 0 is active as long as the emergency-stop procedure is not activated (`EmergencyStop = TRUE`, active-low). For that case, this GTT does not prescribe any particular action, so that all output columns in Row 0 contain a don't-care symbol. Once an emergency stop is initiated (`EmergencyStop = FALSE`), the GTT advances to Row 1. Without delay, the routine needs to ensure that the crane gripper does not drop any work piece. Thus, if the input variable `Sucked` is `TRUE`, which indicates that the gripper holds a piece, the output variables `VaccuumOn` and `VacuumOff` are set accordingly in Row 1. If the operator removes the work piece during the emergency routine, `Sucked` becomes `FALSE` and the vacuum is switched off. For all other emergency actions (besides making sure that no piece is dropped by the crane), a delay of up to 250 ms is admissible. Because of that, the procedure may remain for up to 250 ms in Row 1, which has don't-care symbols for the corresponding output variables. Then, after at most 250 ms, the procedure must move on to Row 2, where all output variables are set to constant values that move all actuators into a safe position. The symbol $—_{\infty}$ (strong repetition) in the duration column of Row 2 indicates that as long as the emergency is active (indicated

#	INPUT		OUTPUT						⏲
	EmergencyStop	Sucked	StackSlider	CraneLower	VacuumOn	VacuumOff	...	StampPusher	
0	TRUE	—	—	—	—	—	...	—	≥ 0
1	FALSE	⌊	—	—	Sucked	=¬Sucked	...	—	$\leq 250\,\mathrm{ms}$
2	FALSE	⌊	FALSE	FALSE	Sucked	=¬Sucked	...	FALSE	$-_{\infty}$

Figure 6: A GTT for specifying the emergency-stop behavior (Application Case 1). For readability, we use a single input variable *EmergencyStop* combining the three active-low signals for the emergency buttons (in the original version, there are three input variables, one for each button). Also, 11 output variables are not shown that control further emergency actions.

by `EmergencyStop = FALSE`), the PLC software needs to maintain the specified output values.

6.2 Application Case 2: Crane as a buffer for improved throughput

This application case is based on Scenario 5 in the xPPU description [21], where the behavior of the crane is changed to achieve a higher overall throughput. In the unchanged version, the plant processes only one single work piece at a time, i. e., a new piece is taken from the stack only after completely processing the previous piece and storing it in the ramp. The idea of the change in Scenario 5 is that now the crane delivers a new piece from the stack to the conveyor instead of waiting for the old piece to be stamped at the stamp module. That is only possible if the piece that happens to be at the front of the stack does not need to be stamped. After moving the new piece to bypass the stamp module, the crane continues with the original behavior by delivering the work piece that has been stamped in the meantime from the stamp to the conveyor.

A GTT specifies the behavior required to correctly carry out the stamp-bypass maneuver in Fig. 7 . The GTT exemplifies a typical pattern: it specifies that, if a certain event occurs that is modeled by a condition on the input variables, then a particular detailed behavior sequence is carried out. In this scenario, if the bypass maneuver becomes possible (Row 1), then the crane correctly carries out the maneuver (Rows 2–17).

Row 1 specifies the condition for starting the maneuver by requiring a certain combination of input variable values (including `TRUE` for the internal variable `Go_Up.X`). At first, the GTT can stay in Row 0 indefinitely (all columns are "don't care"), but once the conditions on input variables in Row 1 are met, the GTT – and, thus, any PLC software conforming to this GTT – must move on to Row 1 and then Rows 2–17, because the progress flag is set in the duration column of Row 0.

The maneuver, as specified by Rows 2–17, consists of the following steps: (1) the crane turns to the stack, (2) picks up the new work piece (lower, vacuum-on, raise), (3) turns to the conveyor, (4) release the piece (lower, vacuum-off, raise), and (5) turns back to the stamp.

#	INPUT							OUTPUT			⏲	
	CraneUp	Metallic	WPReady	Go_Up.X	OnStack	OnConveyor	OnStamp	TurnCCW	Lower	VacuumOn		
0	—	—	—	—	—	—	—	—	—	—	$-_{\mathrm{p}}$	
1	TRUE	FALSE	TRUE	TRUE	—	⌊	⌊	—	⌊	⌊	1	⊤
2	—	—	—	—	FALSE	⌊	⌊	FALSE	⌊	⌊	≥ 0	⌊
3	⌊	⌊	⌊	⌊	TRUE	⌊	⌊	⌊	⌊	⌊	1	⌊
4	⌊	⌊	⌊	⌊	—	⌊	⌊	⌊	TRUE	TRUE	≥ 0	⌊
5	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	TRUE	⌊	1	⌊
6	FALSE	⌊	⌊	⌊	⌊	⌊	⌊	⌊	FALSE	⌊	≥ 0	⌊
7	TRUE	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	1	⌊
8	—	⌊	⌊	⌊	⌊	FALSE	⌊	TRUE	⌊	⌊	≥ 0	⌊
9	⌊	⌊	⌊	⌊	⌊	—	⌊	TRUE	⌊	⌊	≥ 0	⌊
10	⌊	⌊	⌊	⌊	⌊	TRUE	⌊	FALSE	TRUE	⌊	1	$-_{\infty}$
11	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	1	⌊
12	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	FALSE	1	⌊
13	FALSE	⌊	⌊	⌊	⌊	⌊	⌊	⌊	FALSE	⌊	≥ 0	⌊
14	TRUE	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	⌊	1	⌊
15	—	⌊	⌊	⌊	⌊	⌊	FALSE	TRUE	⌊	⌊	≥ 0	⌊
16	⌊	⌊	⌊	⌊	⌊	⌊	TRUE	TRUE	⌊	⌊	1	⌊
17	⌊	⌊	⌊	⌊	⌊	⌊	TRUE	FALSE	⌊	⌊	1	⌊
18	⌊	⌊	⌊	⌊	⌊	⌊	—	—	—	—	$-_{\mathrm{p}}$	⌊

Figure 7: A GTT for the crane-as-buffer maneuver to bypass the stamp and improve overall throughput.

After finishing the maneuver, the GTT goes back to waiting for the conditions of Row 1 to become true (Row 18). Note that Rows 1–18 are grouped together and can be repeated so that the behavior specified by the GTT can contain the bypass-maneuver infinitely often. Alternatively, the GTT can remain in Row 18 forever from a certain point onwards and not repeat the maneuver.

6.3 Application Case 3: Sorting the work pieces

This application case is known as Scenario 11 in the xPPU description [21]. The scenario is concerned with sorting work pieces according to their type into three different ramps: white non-metal (plastic) pieces are put into ramp 1, metal pieces into ramp 2, and black pieces into ramp 3. Sorting is done by placing each work piece on the conveyor, which first moves the piece into a sensor position to detect its type. There are two sensors: a light sensor for distinguishing black and non-black, and an inductive sensor that detects metallic pieces. The conveyor then moves the work pieces towards two pushers. If the piece is to be sorted into ramps 1 or 2, one of the pushers must extend at the right moment to push the work piece onto the correct ramp. If no pusher extends, the piece moves on to ramp 3.

We present two GTTs for the sorting application case. Figures 8 and 9 show GTTs specifying the correct behavior for handling black and white non-metal pieces, respectively.

#	INPUT			OUTPUT		⏲
	SFCReset	Lightness	StartVar	Push1	Push2	
0	FALSE	—	FALSE	—	—	≥ 1
1	↓	↓	TRUE	—	—	1
2	↓	⊥	—	FALSE	FALSE	1
3	⊥	FALSE	—	FALSE	FALSE	$-_\infty$

Figure 8: A GTT specifying the PLC software's behavior for sorting black work pieces.

Handling black pieces is easy because they are sorted into ramp 3 independently of their inductivity (Fig. 8). First, the GTT waits in Row 0. Then, at some point, it moves on to Row 1 and indicates that the piece has arrived (indicated by `StartVar = TRUE`). After that, the two pushers must be retracted by the PLC software (Row 2). Then, the input variable `Lightness` is set to `FALSE` by the GTT corresponding to the case of a black piece (Row 3). In reaction to that, the software must keep the pushers retracted indefinitely so that the piece moves into ramp 3.

The GTT in Fig. 9, specifying the case of sorting a white non-metal piece, is more complex as it must consider the sensor input variable `Metallic` (in addition to `Lightness`). Moreover, it must specify the activation of the first pusher, which needs to be extended at the right moment. Rows 0–2 are similar to the GTT for sorting black pieces: After starting the process (Rows 0 and 1), the pushers are retracted (Row 2). But then, when a white non-metal piece is detected (`Metallic = FALSE` and `Lightness = TRUE` in Row 3), the behavior starts to diverge from the one for black pieces. After waiting for 340 ms (Row 4) and checking that the first pusher is indeed retracted (`PusherIn1 = TRUE` and `PusherOUT1 = FALSE`, Row 5), the PLC software must set the output variable `Push1` to `TRUE` for 200 ms to activate the pusher and move the work piece into ramp 1. Finally, the software must retract the pusher again (Row 7).

7 Case study: Verification

In the previous section, we have shown that GTTs can be used to specify the behavior of PLC software. To demonstrate that these GTTs can also be the basis for formal verification, we have applied our tool chain (see Sect. 3.4) to generate input for the model checker NuXmv and prove that the xPPU [21] PLC software is correct w.r.t. the GTTs from Sect. 6.

For Application Case 1, we verified the complete software of PPU Scenario 13 w.r.t. the GTT from Fig. 6. For

#	INPUT					OUTPUT		⏲
	Metallic	Lightness	PusherIn1	PusherOut1	StartVar	Push1	Push2	
0	—	—	—	—	FALSE	—	—	≥ 1
1	↓	↓	↓	↓	TRUE	—	—	1
2	⊥	⊥	↓	↓	—	FALSE	FALSE	1
3	FALSE	TRUE	↓	↓	↓	↓	↓	≥ 1
4	—	—	⊥	⊥	↓	⊥	↓	340 ms
5	↓	↓	TRUE	FALSE	↓	TRUE	↓	1
6	↓	↓	—	—	↓	TRUE	↓	200 ms
7	⊥	⊥	TRUE	FALSE	⊥	FALSE	⊥	≥ 1

Figure 9: A GTT specifying the PLC software's behavior for sorting white non-metal work pieces.

Application Cases 2 and 3, we applied the verification tool chain to prove the correctness of individual function blocks. The function block Crane (Scenario 5) was shown to be correct w.r.t. the GTT from Fig. 7 (Application Case 2); and the function block Pusher (Scenario 11) was shown to be correct w.r.t. the GTTs from Fig. 8 and 9.

Statistics for the verification are shown in Fig. 10. Experiments were carried out with an Intel i5-6500 (3.2 GHz) CPU and 16GB RAM, and with the NUXMV model checker version 1.1.1.

GTT used for specification	Implementation being verified	Verification	
		cpu time	model size
Appl. Case 1, Fig. 6	Scen. 13	0.88 s	784 bit
Appl. Case 2, Fig. 7	Scen. 5, Crane	0.26 s	114 bit
Appl. Case 3, Fig. 8	Scen. 11, Pusher	0.66 s	127 bit
Appl. Case 3, Fig. 9	Scen. 11, Pusher	32.30 s	186 bit

Figure 10: Statistics for verification of PLC software w.r.t. the specifications given by the GTTs from Sect. 6. The verified implementations are taken from the xPPU [21]. CPU times are the median for $n = 5$ runs.

As can be seen from the table, all required proofs could be done in reasonable time. Because formal verification is based on carrying out a single symbolic proof for each GTT and not on running concrete test cases, the verification effort is independent of the number of test cases that the GTT represents (as long as they are similar to each other). Also, proof time is independent of the specific PLC hardware.

Our experiments show that proof time and complexity mainly depend on the number of rows in the GTT and their minimum duration, as this influences the search depth of the model checker.

8 Discussion and future work

Software verification is independent of specific hardware and can be deployed early in the software development process. In contrast to software testing, verification covers all input sequences included in a GTT – even if there are infinitely many of them. The GTT and the PLC software are exhaustively explored by the model checker. For example, in Application Case 1, the GTT covers all situations where an emergency stop is activated. We verified that the software handles the emergency stop correctly in all reachable software states.

In this paper, we have presented four extensions of the GTTs and demonstrated the applicability. The applicability of GTT-based specification and formal verification is investigated using diverse-ranged community demonstrator scenarios. The syntax of GTTs is aligned with that of concrete test tables, which are state of the art for validation in industry and, therefore, are accessible for application developers. Generalizing concepts allow GTTs to represent a family of test cases, and, therefore, allow the quality assurance activity to have more coverage. As GTTs have the potential to be an understandable formal specification method for describing the desired behavior of reactive systems, they can help to establish formal verification as a validation technique for aPS and make their coverage strength available to application developers.

In future research, we plan to apply our GTT-based specification and verification methodology on practical industry cases to evaluate its practical efficiency and complexity. In particular, food and pharmaceutical packaging industry cases are under consideration, in which aPS must be validated according to regulations to ensure both safety and behavioral integrity. Also, to evaluate the comprehensibility and usability of GTTs, we will perform expert and junior interviews with subjects that are application engineers and representative for real future users.

Funding: Research supported by the DFG (German Research Foundation) in Priority Programme SPP1593: Design for Future – Managed Software Evolution (VO 937/28-2, BE 2334/7-2, and UL 433/1-2).

Appendix A. Definition for the new features

In this section we present the updated definitions for the structure of GTT ([3, Def. 3]), and the definition of unrolled instances ($D1(T)$ cf. [3, Def. 3]).

In earlier version the duration interval is an interval $[m, n]$, where $m \in \mathbb{N}$ and $n \in \mathbb{N} \cup \{\bot\}$.

In the previous work, the structure of a GTT is defined as a sequence of triples (input, output and duration constraint). Strong repetition and progress flag requires extension on the domain of the duration constraints τ:

Definition 1 (Duration interval τ). *A duration interval is either ω (strong repetition), an (top-open) interval $[m, n]$ with or without the progress flag:*

$$\tau \in \{\omega\} \cup \{[m,n] \mid m \in \mathbb{N}, n \in \mathbb{N}_\infty\} \cup \{[m,n]_p \mid m \in \mathbb{N}, n \in \mathbb{N}_\infty\}$$

with $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$.

We define a function $val(\ \tau\ )$ that evaluates a duration constraint into a set of possible repetitions.

Definition 2 ($val(\ \tau\ )$).

$$val\colon \tau \mapsto \begin{cases} \{\omega\} & \tau = \omega \\ \{x \mid n \geq x\} & \tau = [n,\infty] \vee \tau = [n,\infty]_p \\ \{x \mid n \leq x \leq m\} & \tau = [n,m] \vee \tau = [n,m]_p \end{cases} \quad (1)$$

With the introduction of nested blocks, this sequential structure becomes obsolete. The new mathematical structure is defined inductively to capture the recursive nature of GTTs.

Definition 3 (GTTs as a Tree of Constraints). *Let InVar be the set of input variables, OutVar be the set of output variables, and GVar be the set of global variables; then the set of all GTTs $\mathfrak{T}_\Sigma$ over signature $\Sigma = InVar \cup OutVar \cup GVar$. is defined as*

base *The empty table ϵ and a single row (ϕ, ψ, τ) are GTTs ($\epsilon, (\phi, \psi, \tau) \in \mathfrak{T}$, ϕ is the conjunction of all input column constraints of a row, resp. ψ for the output columns and τ defines the duration interval).*

step *Let $T \in \mathfrak{T}$ a GTT, then*

- **seq** *the sequential composition $\langle T, T' \rangle \in \mathfrak{T}$ is also a GTT, where $T, T' \in \mathfrak{T}$.*
- **nest** *the repetition of a GTT T is also a GTT: $[T, \tau] \in \mathfrak{T}$ with an arbitrary duration interval τ and $T \in \mathfrak{T}$.*

The set *InVar* and *OutVar* also includes the categorized state variables. The constraints formulas ϕ and ψ are Boolean formulas over the signature Σ. The sequential composition with the empty table cover the GTTs as defined in [3]. We use angle brackets to denote the sequence, and parentheses to denote a row. ϵ denotes the empty table. The nest rule creates the nested row groups, denoted by brackets.

Example 1. *The GTT in Fig. 4 is represented by the following structure for a cycle of 100 ms.*

$$\begin{aligned} &[\ (\neg WP \vee ERR, true, -_p), \\ &\ (WP \wedge \neg ERR, MOVE = \texttt{Stop}, [1,\infty]) \\ &\ [\ (\neg WP \wedge \neg ERR, MOVE = \texttt{Fwd}, [50,60]),\ [0,1]\ ] \\ &\ [\ \langle\ [\ (ERR, MOVE = \texttt{Fwd}, [10,10]) \\ &\quad (ERR, MOVE = \texttt{Rwd}, [10,10]) \\ &\quad (WP \wedge ERR, MOVE = \texttt{Stop}, [5,5]),\ [0,3]], \\ &\ (WP \wedge \neg ERR, MOVE = \texttt{Stop}, [1,1])\rangle, [0,1]], \omega] \end{aligned}$$

The game [3, Fig. 4] maintains the set $S = D1(T)$, which represents the possible remaining test table. $D1(T)$ defines a infinite language over the alphabet $\Sigma = Fml_\Sigma \times Fml_\Sigma$, where Fml_Σ is the set of all Boolean formulas over signature Σ. We define $D1'(T)$—a version for GTTs after Def. 3. $D1'(T)$ transform the recursive data structure into an (infinite) set of (in)finite words.

$$D1'(T) =_{\text{def}} \begin{cases} \{\epsilon\} & \text{if } T = \epsilon \\ (\phi, \psi)^\tau & \text{if } T = (\phi, \psi, \tau) \\ D1'(T') \cdot D1'(T'') & \text{if } T = \langle T'; T'' \rangle \\ D1'(T')^\tau & \text{if } T = [T', \tau] \rangle \end{cases} \quad (2)$$

Note, ϵ denotes the empty word, and $L^\tau = \bigcup_{k \in val(\tau)} L^k$. With $D1'(T)$ we are back into the framework given in [3] with one exception: $D1'(T)$ could contain infinite words. The game remains applicable, but an infinite word in $D1'(T)$ prevents the win of the system by reaching the table end.

Appendix B. Semantic impact: No strict conformance

The following proposition rephrase the termination of the game:

Proposition 1. *Termination of the game [3, Fig. 4] The game terminates in round $k > 0$*

1. *if $S = \emptyset$ after the turn of the challenger (System wins, Line 13)*
2. *if $S = \emptyset$ after the turn of the system (Challenger wins, Line 17)*
3. *if the end of the table is reached $\exists D \in S.|D| = k$ (System wins, Line 21).*

S denotes the set of the currently remaining valid unwound GTTs.

The next proposition is generalization of the statement, that a strong repetition prevents strict conformance.

Proposition 2. *Let S_k be set of remaining test tables in round k. If all remaining test tables are infinite $\forall w \in S_k.|w| = \omega$ then the system can not win by Line 21 (end of table).*

Proof. The first implication is a direct consequence that k is always finite: $k \in \mathbb{N}$ and $\forall n \in \mathbb{N}.n < \omega$. □

Moreover for the prevention, we need to require the existence of a challenger that can survive infinitely long. Therefore all input constraints ϕ, especially these the strong repeated rows, need always be satisfiable.

Proposition 3. *If there exist infinite valid challenger stimuli in, i. e.*

$$\exists w \in S^k.\forall k \in \mathbb{N}.in[k] \vDash \phi[k]$$

then there does not exists a strict *conform system. $\phi[k]$ is the input constraint given $w[k]$.*

Proof. Let there be a *strict* conform system. Also, the system wins against every challenger. The win by end-of-table is disabled and there exists a challenger which has a never-ending valid play. The system can not win. Contradiction! □

Appendix C. Proof: Strong repetition does not extend the expressiveness

After defining the structure in Def. 3. We can define the transformation of T_∞ to T_* formally.

Definition 4 (Construction of T_*). *T_* for a given GTT T_∞ is constructed by the following recursive function $trans(T_\infty)$:*

$$trans(T) =_{def} \tag{3}$$

$$\begin{cases} \epsilon & \text{if } T = \epsilon \\ \langle(\phi,\psi,\tau), trans(T')\rangle & \text{if } T = \langle(\phi,\psi,\tau), T'\rangle \wedge \tau \neq -_\infty \\ [trans(T'), \tau] & \text{if } T = [T', \tau] \wedge \tau \neq -_\infty \\ \langle(\phi,\psi,-), (\phi, \texttt{false}, 1), trans(T')\rangle & \text{if } T = \langle(\phi,\psi,-_\infty), T'\rangle \\ \langle[trans(T'),-], (\Phi(T'), \texttt{false}, 1)\rangle & \text{if } T = [T', -_\infty] \end{cases} \tag{4}$$

$\Phi(T)$ is defined by the disjunction $\bigvee_{(\phi,\psi,\tau)\in succ(0)} \phi$ of all immediately reachable rows $succ(0)$ in T from the start (cf. function succ in [3]).

We introduce $\mathbb{I}$ and $\mathbb{O}$ as finite sets that denote the value domain of the input and output variables of the reactive system, determined by the system's interface (variable type).

Our goal is to prove the equal *weak* and *strict* conformance.

Proposition 4 (Equal Conformance). *A reactive system is* weak *conform to T_∞ iff it is conform to T_*, analog for* strict *conform.*

The conformance of a test table is based on the outcome of all possible plays, i. e. a system is *strict* conform iff it is a winning strategy and *weak* conform iff its strategy never loss.

The next lemma reduces the equal conformance on equal game outcome on all plays.

Lemma 1. *Two GTT T, T' have equal conformance if the game outcome for all possible plays is equal.*

In rest of the proof for Prop. 4, we need to show that for an arbitrary play, without assumption on the table or the system, the outcome (winner and loser) is equal on T_* and T_∞. The outcome is determined by the algorithm [3, Fig. 4]. We show that by defining a coupling between both run of the algorithm for T_* and T_∞. More formally, the algorithm based on a set S (cf. Lemma 1), that holds the remaining possible unwound GTT. S is determined by $D1'(T)$ in the beginning. The outcome is decided within one round of the game, especially if S becomes empty. We established a coupling relation between the set S from the run over T_∞ and T_*: S^k_∞ and S^k_* for round $0 < k$. The coupling relation established a bisimulation.

To define the coupling relation, we need following property on the structure of S^k_∞ and S^k_*.

Lemma 2 (Structure of S^k). *In every round $k \geq 0$, S^k_∞ and S^k_* can be separated:*

$$S^k_\infty = L^{fin} \cup \bigcup_l \alpha_l \cdot \beta_l^\omega \tag{5}$$

$$S^k_* = L^{fin} \cup \bigcup_l \alpha_l \cdot \beta_l^*(\Phi, \texttt{false}) \tag{6}$$

for a language α_l and β_l.

L^{fin} contains the finite words defined of a test tables, e. g., if a strong repetition is avoidable. It is also possible to select a separation, s. t. α_l is the path into an strong repetition, and β_l is the strong repetition. There are only a finite amount of strong repetition, the set union is finite. Moreover in the separation S^k, β represents the block (or row), that is strong repeated, which is implicitly described in Lemma 3. This connects the β_l with the construction T_*, especially Φ_l.

Further, we define the languages $V_C, V_S \subseteq S^k$, that represent the words in S^k that are violated by turns of the challenger or the system.

Definition 5. *Let S be language of (ϕ,ψ), $p \in (\mathbb{I},\mathbb{O})^*$ a play, and $k > 0$:*

$$V_C(S,p,k) = \{w \in S^k \mid (\phi,\psi) = w[k-1] \wedge p \not\models \phi\} \tag{7}$$

$$V_S(S,p,k) = \{w \in S^k \mid (\phi,\psi) = w[k-1] \wedge p \not\models \phi \wedge \psi\} \tag{8}$$

(Proof by induction over k). For $k = 0$, the separation follows immediately from the Definition of $D1'$, S^0 is actually regular.

For $k > 0$, in every round S^k is filtered by the current play p with $|p| = k$, formally $S^k = (L^{fin} \cup S^{k-1}) \setminus V_S(S,p,k)$. From induction hypotheses, we know that S^{k-1} can be separated and by definition of the game the words in S^{k-1} have at least the length k, otherwise the game has terminated.

Assume w. l. g. that every α_l in the separation of S^{k-1} contains only words with length k, otherwise we would unwind β as many times as needed.

$$S^k_\infty = (L^{fin} \setminus V_S(S,p,k)) \cup \bigcup_l (\alpha_l \setminus V_S(S,p,k)) \cdot \beta_l^\omega \tag{9}$$

$$S^k_* = (L^{fin} \setminus L_k(v)) \cup \bigcup_l (\alpha_l \setminus V_S(S,p,k)) \cdot \beta_l^*(\Phi_l, \texttt{false}) \tag{10}$$

□

We established a relation between both sets, which follows from the Def. 4.

Lemma 3 (Coupling of S^k_∞ and S^k_*). *There exists a separation of S^k_∞ and S^k_*, s. t. both L^{fin} are equal and for*

$$\forall l \,.\, \alpha_l \cdot \beta_l^\omega \in S^k_\infty \Leftrightarrow \alpha_l \cdot \beta_l^*(\Phi, \texttt{false}),$$

We show the coupling between both sets are maintain after each, additionally prove a little bit more: The coupling is maintained after the choice of the challenger and the system.

(Proof by induction over k). The lemma is immediately valid for $k = 0$.

Let S^{k-1}_∞ and S^{k-1}_* are be coupled, we need to show that the coupling ensured after the round k for S^k_∞ and S^k_*. We also assume an arbitrary play $p \in (\mathbb{I}, \mathbb{O})$ s. t. $p = p' \cdot (a,b) \wedge |p| = k$ and there is no winner of the play in all rounds $k' < |p|$, otherwise the game would have terminated (both sets are empty and the coupling relation would immediately hold).

The coupling is *stable* under set difference: Stable means, that the a coupled $\alpha_l\beta_l^\omega$ and $\alpha_l\beta_l^*(\Phi_l, \texttt{false})$ will stay coupled in all rounds.

In detail: By induction hypotheses the languages of the finite words in $D1'(T_\infty)$ are coupled, hence $L^{fin,k-1}_\infty = L^{fin,k-1}_*$. Applying the set difference keeps equivalence:

$$L^{fin,k-1}_\infty \setminus V_{k,\xi} \;=\; L^{fin,k}_\infty = L^{fin,k}_* \;=\; L^{fin,k-1}_* \setminus V_{k,\xi}.$$

The same holds for any coupled $\alpha_l\beta_l^\omega$ and $\alpha_l\beta^*(\Phi_l, \texttt{false})$ of the separation. W. l. g. we can pump up α_l with β_l s. t. all words $w \in (\alpha_l \cdot \beta_l)$ are at least greater or equals to k.

Leaving one open case, $\alpha_l(\Phi, \texttt{false})$ (we choose $\epsilon \in \beta^*$). Obviously, this word is removed by subtracting with $V_S(S,p,k)$ (ψ in position k is `false`), but this is not valid $V_C(S,p,k)$. Here it is possible, that $\alpha_l\beta_l^\omega \subseteq V_C(S^{k-1}_\infty, p, k)$, but $\alpha_l(\Phi_l, \texttt{false}) \not\subseteq V_c(S^{k-1}_*, p, k)$. Note, β_l represents the corresponding block for which Φ_l is built for. By definition, Φ_l is the disjunction of all ϕ of the first symbols in β_l. Therefore, the play p violates Φ_l iff it violates all *first* ϕ in β. □

Prop. 4. From coupling in Lemma 3 it follows that $S^k_\infty = \emptyset \Leftrightarrow S^k_* = \emptyset$. (Proof in contraposition) □

Example 2. *For a better auditability, we show the equal conformance on the special case with*

$$T_\infty = (\phi, \psi, -_\infty) \qquad T_* = \langle (\phi, \psi, -), (\phi, \mathit{false}, 1) \rangle. \tag{11}$$

The row $(\phi, \mathit{false}, 1)$ prevents the strict conformance of the system—the system cannot fulfill false, the same holds for the strong repetition in T_∞. Therefore, we only need to consider weak *conformance. Let $p \in (\mathbb{I}, \mathbb{O})^*$ be an arbitrary play. The system wins in T_∞ if $p \not\models \phi$. At the same time it would win in T_*, as p is not allowed in both rows. Otherwise, the system loses in T_∞ ($p \models \phi$ and $p \not\models \psi$) iff it also violates T_* as none of both rows is adhered ($p \not\models \psi \vee p \not\models \mathit{false}$).*

References

1. Roy Armoni et al. “The ForSpec Temporal Logic: A New Temporal Property-Specification Language.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 296–311.
2. Nanette Bauer et al. “Verification of PLC Programs Given as Sequential Function Charts.” In: *Integration of Software Specification Techniques for Applications in Engineering: Priority Program SoftSpez of the German Research Foundation (DFG), Final Report*. Ed. by Hartmut Ehrig et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 517–540. DOI: 10.1007/978-3-540-27863-4_28.
3. Bernhard Beckert et al. “Generalised Test Tables: A Practical Specification Language for Reactive Systems.” In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 129–144. DOI: 10.1007/978-3-319-66845-1_9.
4. Bernhard Beckert et al. “Regression verification for programmable logic controller software.” In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. LNCS 9407. November (2015), pp. 234–251. ISSN: 16113349. DOI: 10.1007/978-3-319-25423-4_15.
5. Sebastian Biallas, Jörg Brauer and Stefan Kowalewski. “Arcade.PLC: A Verification Platform for Programmable Logic Controllers.” In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: ACM, 2012, pp. 338–341. DOI: 10.1145/2351676.2351741.
6. Roberto Cavada et al. “The nuXmv Symbolic Model Checker.” In: *CAV*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 334–342. ISBN: 978-3-319-08866-2.
7. Adrien Champion et al. “CoCoSpec: A Mode-Aware Contract Language for Reactive Systems.” In: *Software Engineering and Formal Methods*. Ed. by Rocco De Nicola and Eva Kühn. Cham: Springer International Publishing, 2016, pp. 347–366, ISBN: 978-3-319-41591-8.

8. 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. DOI: 10.1109/ICSMC.2000.884356.
9. L. Heitmeyer and R. D. Jeffords. “Applying a Formal Requirements Method to Three NASA Systems: Lessons Learned.” In: *2007 IEEE Aerospace Conference*. 2007, pp. 1–10. DOI: 10.1109/AERO.2007.352764.
10. Gerard J. Holzmann. “The Logic of Bugs.” In: *Proceedings of the 10th ACM SIGSOFT Symposium on Foundations of Software Engineering*. SIGSOFT ’02/FSE-10. Charleston, South Carolina, USA: ACM, 2002, pp. 81–87. ISBN: 1-58113-514-9. DOI: 10.1145/587051.587064. URL: http://doi.acm.org/10.1145/587051.587064.
11. International Electrotechnical Commission, *IEC 61131: Programmable controllers – Part 3: Programming languages*. Tech. rep. International Electrotechnical Commission, Feb. 2002.
12. S. Kowalewski et al. “Verification of logic controllers for continuous plants using timed condition/event-system models.” In: *Automatica* 35.3 (1999), pp. 505–518. ISSN: 0005-1098. DOI: 10.1016/S0005-1098(98)00179-4.
13. Antti Pakonen et al. “User-friendly formal specification languages – conclusions drawn from industrial experience on model checking.” In: *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2016)*. Vol. 2016-Novem. Berlin, Germany, 2016. ISBN: 9781509013142. DOI: 10.1109/ETFA.2016.7733717.
14. Susanne Rösch and Birgit Vogel-Heuser. “A Light-Weight Fault Injection Approach to Test Automated Production System PLC Software in Industrial Practice.” English. In: *Control Engineering Practice* 58.Complete (2017), pp. 12–23. DOI: 10.1016/j.conengprac.2016.09.012.
15. Doaa Soliman and Georg Frey. “Verification and validation of safety applications based on PLCopen safety function blocks.” In: *Control Engineering Practice* 19.9 (2011). Special Section: DCDS’09 – The 2nd IFAC Workshop on Dependable Control of Discrete Systems. pp. 929–946, ISSN: 0967-0661. DOI: 10.1016/j.conengprac.2011.01.001.
16. Markus Spindler et al. “Erstellung von Steuerungssoftware für automatisierte Materialflusssysteme per Drag & Drop.” In: *Logistics Journal: Proceedings* 2017.10 (2017). DOI: 10.2195/lj_Proc_spindler_de_201710_01.
17. Ofer Strichman. “Regression Verification: Proving the Equivalence of Similar Programs.” In: *Computer Aided Verification* Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, p. 63. DOI: 10.1007/978-3-642-02658-4_8.
18. Kleanthis Thramboulidis. “The 3+1 SysML View-Model in Model Integrated Mechatronics.” In: *Journal of Software Engineering and Applications* 03.02 (2010), pp. 109–118. ISSN: 1945-3116. DOI: 10.4236/jsea.2010.32014.
19. Sebastian Ulewicz and Birgit Vogel-Heuser. “Automatisiertes Testen von Sondermaschinen - von der Modulbibliothek bis zur Anlage.” In: *Tagungsband Automation Symposium*. 2015, pp. 53–65.
20. Sebastian Ulewicz et al. “Proving equivalence between control software variants for Programmable Logic Controllers: Using Regression Verification to Reduce Unneeded Variant Diversity.” In: *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*. Vol. 2015-October. 2015, pp. 1–5. DOI: 10.1109/ETFA.2015.7301603.
21. Birgit Vogel-Heuser et al. “Evolution of software in automated production systems: Challenges and research directions.” In: *Journal of Systems and Software* 110 (2015), pp. 54–84. DOI: 10.1016/j.jss.2015.08.026.
22. Birgit Vogel-Heuser et al. “Fault Handling in PLC-Based Industry 4.0 Automated Production Systems as a Basis for Restart and Self-Configuration and Its Evaluation.” In: *Journal of Software Engineering and Applications* 9.1 (2016), pp. 1–43. DOI: 10.4236/jsea.2016.91001.
23. A. N. I. Wardana, J. Folmer and B. Vogel-Heuser. “Automatic program verification of continuous function chart based on model checking.” In: *2009 35th Annual Conference of IEEE Industrial Electronics*, 2009, pp. 2422–2427. DOI: 10.1109/IECON.2009.5415231.
24. A. Weigl et al. “Generalized test tables: A powerful and intuitive specification language for reactive systems.” In: *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*, 2017, pp. 875–882. DOI: 10.1109/INDIN.2017.8104887.