Generalized Test Tables: A Powerful and Intuitive Specification Language for Reactive Systems

Alexander Weigl*, Franziska Wiebe†, Mattias Ulbrich*, Sebastian Ulewicz†, Suhyun Cha†, Michael Kirsten*, Bernhard Beckert*, Birgit Vogel-Heuser†

*Karlsruhe Institute of Technology, Institute of Theoretical Informatics, Karlsruhe, Germany

†Technical University of Munich, Institute of Automation and Information Systems, 85748 Garching near Munich, Germany

***Abstract*—With recent trends in manufacturing automation, such as Industry 4.0, control software in automated production systems becomes more and more complex and volatile, complicating and increasing importance of quality assurance. Test tables are a widely used and generally accepted means to intuitively specify test cases for automation software. However, each table only specifies a single software trace, whereas the actual software behavior may cover multiple similar traces not covered by the table.**

Within this work, we present a generalization concept for test tables allowing for bounded and unbounded repetition of steps, "don't-care" values, as well as calculations with earlier observed values. We provide a verification mechanism for checking conformance of an IEC 61131-3 PLC software with a generalized test table, making use of a state-of-the-art model checker. Our notation is inspired by widely-used paradigms found in spreadsheet applications. By an empirical study with mechanical engineering students, we show that the notation matches user expectations. A real-world example extracted from an industrial automation plant illustrates our approach.

I. Introduction

Automated production systems (aPS), like industrial manufacturing plants, often need to adapt to changed requirements, and are thus subject to constant evolution—which also affects the software [2]. Recent trends in manufacturing automation, such as Industry 4.0 and Cyber Physical Production Systems (CPPS), exacerbate this problem drastically. Volatility of the aPSs and their environment increases tremendously, permitting flexible changes of the system and interacting systems dynamically during runtime. Since aPS are often safety-critical systems where faults can cause severe damage to the system, the payload, or persons within the reach of the system, effective software quality assurance measures during evolution are of utmost importance. Because of the changing environment of the systems, exactly corresponding to specifications becomes imperative to allow for safe interaction and predictable changes.

In today's industrial practice, software quality is achieved by dynamic verification and validation, either through manual step-by-step testing or by running automatically generated test cases [3].

The main weakness of traditional testing is that one test case covers only a single, particular run of the aPS software; many scenarios remain uninvestigated during testing. Full test coverage can rarely be achieved. Systematic testing is good for detecting typical and expected faults, but unpredictable and rare malfunctions (which also can have severe consequences) are less likely to be discovered using testing. In contrast to testing, formal verification achieves full coverage by mathematically proving the correctness of an implementation with respect to its formal specification. Moreover, for quickly evolving systems like aPS, a benefit of verification is that during development of a new function block, potential faults can be removed early on.

In engineering of aPS—besides safety issues—formal methods are not commonly applied, yet. As analyzed by Pakonen et al. in [4], one reason for this is that adequate formal specifications are not easily obtained and require a deep understanding of the underlying formal concepts. This makes the application of formal methods often unduly labor-intensive.

This work aims for combining the comprehensibility of test cases with the coverage of formal verification by providing an understandable—yet powerful—extension to a widespread test case notation technique: *Test tables* are widely used by our industry partners in machine and plant manufacturing to specify test cases for aPS software [5]. Each table consists of a sequence of sensor inputs with their expected software responses, and its rows are a natural way to denote the successive steps of input and output needed for a reactive system like aPS software.

To lower the threshold of applying formal verification—in particular for coming up with formal specifications—we propose an approach making use of existing concrete test tables as a starting point when writing formal specifications. The degree of generalization and the choice of advanced specification features are individually decided by the engineer. This allows for a gradual progress from specification-by-example to a fully systematic specification. The extensions proposed support both dynamic and static verification.

Test tables are commonly written using spreadsheet software and executed within test automation tools like the CODESYS Test Manager. The specification of the system under test has a mandatory aspect of time as the system behaves differently for different sequences.

The main contribution of this paper is the concept of generalized test tables, which allow for describing an entire family of test cases instead of a single test case. For ensuring the practical applicability of generalized test tables, the concept was developed and evaluated together with two companies from

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 1-2).

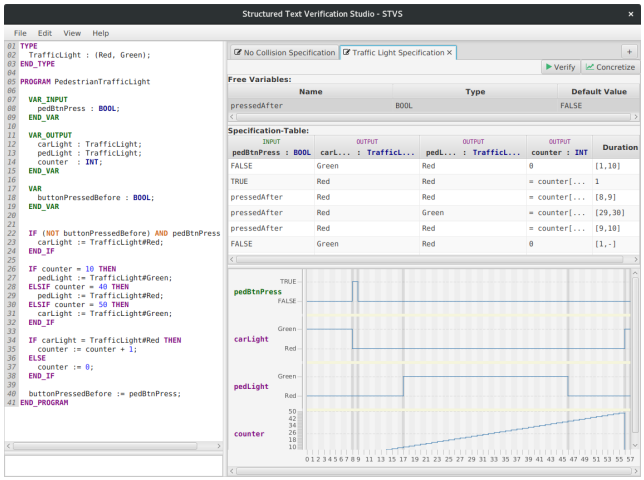

Fig. 1: The GUI for generalized test tables shows the source code, the generalized test tables, and corresponding counter example or generated concrete test tables within a timing diagram.

the domain of food and pharma plant manufacturing and from both PLC hardware and software manufacturing. Moreover, we provide a graphical tool[1] for the verification of Structured Text against generalized test tables (Fig. 1). All the tests described by a generalized test table can be checked simultaneously.

Furthermore, generalized test tables exhibit two important properties: (1) They are a natural extension of the well-known concept of test tables, which are an expressive, natural, and intuitive means for a test design easily understandable by engineers. (2) They are effectively formally verifiable using state-of-the-art model checking tools. Generalized test tables can also serve as a basis for the generation of concrete test tables that can be used to complement formal verification with testing.

The generalizations we consider in this paper are the following:

Abstraction. Instead of concrete values, cells in test tables contain constraint expressions such as "$X > 0$" or "$X + 1 = 4$". Any value satisfying the constraint is a possible cell entry.

References to other cells. Cells in tables can contain a reference to values encountered in other cells.

Generalization of row durations. A table row may be repeated for more than one cycle; the number of repetitions is specified by an interval constraint.

Apart from being more expressive, generalized test tables have the additional benefit that they can be specified to be less sensitive to changes in system design. They allow for specifying schematic conditions like "after between 10 and 20 waiting cycles" or "when observing a value that is greater than the input given in the previous cycle" whereas concrete test cases would impose a choice of concrete instances in such cases. In the light of software evolution, it is important to note that a generalized table may still be valid after a software update, whereas its concrete instances are generally not.

Vogel-Heuser et al. has shown in [6] that faults mostly occur on lower software architecture levels. Subsequently, errors must often be handled on the atomic or basic module layer. Hence, we demonstrate the feasibility of our specification and verification approach on an atomic function block and on one example composed of multiple function blocks, both from an industrial context.

To substantiate the claim that the notation following paradigms found in spreadsheet applications, is intuitively understandable and useful, we present the results of an empirical survey consulting mechanical engineering students.

Overview. The remainder of this paper is structured as follows: Sect. II introduces the concept of generalized test tables intuitively and formally. We present a case study on the verification of industrial code against generalized test tables in Sect. III. An empirical study (Sect. IV) evaluates our approach with respect to its applicability and comprehensibility. Finally, the related work (Sect. V) and Sect. VI concludes the paper.

II. Generalized Test Tables

In this section, we introduce the concept of generalized test tables. Generalized test tables are an extension of traditional, non-generalized tables (in the following, non-generalized tables are called *concrete* tables).

A. Concrete Test Tables

A concrete test table describes a single test case for a reactive system (e.g., a PLC function block). The rows of a concrete table correspond to the successive steps performed by the system under test. The columns correspond to the system's variables. These are partitioned into *input* variables and *output* variables. In addition, there is a special column resp. variable named DURATION.

The reactive systems we consider are executed cyclically, where each cycle is one step in the test. Cycles consume a fixed period of time, the *cycle time*. In each cycle, the concrete input values contained in the table row corresponding to that step are the stimuli for the system; and the system is expected to react with the output values contained in the same row. If the observed system response is different from the expectation for one or more of the rows in the test table, then the system violates the test case. The value of DURATION determines how long the system is to remain in the step, in particular how many cycles the input values are provided. DURATION is given as a number of cycles (it can also be given as a time constraint, which is transformed into cycles by division with the system's specific cycle time).

Note, that a table row with a duration of n is equivalent to repeating that same row n times with a duration of 1.

Fig. 2 shows an example for a simple concrete table. The table has three input variables A, B, C and three output variables X, Y, Z, and describes a test case of 10 cycles (as the durations of the three rows add up to 10). In this example, all variables are of type integer; whereas in general, other types, such as boolean variables, are also possible.

[1] https://formal.iti.kit.edu/stvs

#	Inputs A	B	C	Outputs X	Y	Z	DURATION
0	1	1	2	0	0	5	1
1	0	3	3	6	6	5	7
2	1	4	2	2	8	5	2

Fig. 2: Example for a concrete test table.

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)

Fig. 3: Constraint abbreviations (X is the name of the variable that the cell corresponds to; n, m are arbitrary expressions of type integer).

B. From Concrete to Generalized Test Tables

Generalizing a test table and its specified test case is done by substituting concrete values in the table's cells by *constraint* expressions. Intuitively, a system satisfies a generalized test table if it responds to input values, that adhere to the input constraints, with output values, that adhere to the output constraints. This generalized the meaning of concrete test cases were the constraints are unique values. Thus, a generalized test table specifies a—possibly infinite—set of test cases. A detailed explanation of the semantics of generalized test table is given in Sect. II-D.

In the following, we explain three generalization concepts: (1) abstraction using constraint expressions (which is the basis of generalization), (2) using references to other cells in constraint expressions, and (3) using generalization in the duration columns of tables.

Abstraction using constraints. Instead of concrete values, we allow cells to contain constraints such as "$X > 0$", "$X + 1 = 4$", or "$X > 3 \wedge X < 10$." Besides the name of the variable that the cell corresponds to (e.g., X), the expressions can be built using all operators of the appropriate type ($+, *$ etc.), constant values ($0, 1, 2, \ldots$), and predicates such as $=, >, \geq$ etc. In addition, logical operators ($\wedge, \vee$, etc.) can be used to combine several atomic constraints.

For convenience, we allow abbreviations (see Fig. 3): "$X < n$" can be written as "$< n$" and "$X = n$" simply as "n". Moreover, we allow interval constraints $[n, m]$, which stand for "$X \geq n \wedge X \leq m$." Finally, "–" is the constraint satisfied by all values ("don't care"). Abbreviations can be combined conjunctively using commas; the expression "$[n, m]$" is, e.g., equivalent to "$\geq n, \leq m$".

References to other cells. A reactive system's behavior depends both on the current and the previous input stimuli. Therefore, the expected values in the cells of a generalized test table are not independent of each other. We may want to specify that, e.g., for the value of input A being n, the value of output X is $n + 1$. For that purpose, we introduce two additional syntactical concepts to be used in constraints: global variables and references to other cells.

Global variables, denoted by lower-case letters, can be used in all constraints in any place where an expression of the corresponding type is expected. The value of a variable v is globally the same in all cells, in which v occurs. Thus, we can write v in a cell with input A (short for $A = v$) and $v + 1$ in a cell with output X (short for $X = v + 1$) to express that the value of output X is equal to $v + 1$ for the input A being of value v. Besides being the same in all cells, the value of a global variable is only restricted by the constraints, in which it occurs. Thus, for example, $X = v$ is equivalent to "don't care" if v does not occur in any other cell.

In addition to global variables, we allow references to other cells using the form "$X[-n]$" and "$X[n]$", where X is a variable name and n is a concrete number. $X[-n]$ refers to the value of X n cycles before the current cycle, while $X[n]$ refers to X evaluated n cycle in the future (as we evaluate generalized test tables statically, future references are possible). For references to other variables in the current cycle, we just write "X" as an abbreviation for "$X[0]$".

Thus, we can write "$A + 1$" in an X-cell to express that the output X is by one greater than the input A. To express that the value of Y increases by one in each cycle, we write $Y[-1] + 1$ in each Y-cell except for the first one.

References to other cycles are always relative to the current cycle—they are not given w.r.t. the start or end of the table. Absolute references to particular cells can be expressed using global variables.

Generalization in the duration column. The DURATION variable defines the number of cycles for which a row is repeated. As a further generalization concept, we allow the concrete values in the DURATION column to be replaced by constraints. However, in contrast to the columns for input and output variables, we only allow the DURATION column to contain constraints describing intervals; and they must not refer to other cells. Thus, constraints of the form "$[n, m]$" and "$\geq n$" are the only possibilities. We use "$*$" as a special "don't care" symbol for the duration column; it is equivalent to "≥ 0".

C. Example: A Simple Generalized Test Table

Fig. 4 shows an example of a simple generalized test table, incorporating the generalization concepts described above. Note that the concrete table depicted in Fig. 2 is one of the possible instances of the generalized test table given in Fig. 4, achieved

#	Inputs A	B	C	Outputs X	Y	Z	DURATION
0	1	1	2	0	0	–	1
1	–	p	p	$2{*}p$	X	$Z[-1]$	>5
2	–	$p{+}1$	–	$[0,p]$	$>Y[-1]$	$2{*}Z{>}Y$	$*$

Fig. 4: Example for a generalized test table.

by instantiating the global variable p with the value 3. The instantiated generalized test table for $p = 3$ still covers more than one conrete test case.

The first row expresses a cycle, which is executed once. It provides three concrete input values for the sensor inputs A, B, C, and expects the outputs X, Y to both be equal to 0, whereas the output value for Z can be of arbitrary value.

The input values for the second row are applied repeatedly for strictly more than five scan cycles (there is no upper bound). The input A is a "don't care" value, i.e., it can potentially be different for each cycle. The input values for B and C may also be arbitrary; however, they are bound to be equal to the global variable p. Hence, the values of B and C are the same in each of the cycles of the second table row. The output value of X is required to be identical to $2 * p$, i.e., twice the value of the input values for B, C. Moreover, Y is also required to be equal to $2 * p$, enforced by the reference to the X-cell. Finally, the value of input Z is equal to the one of the first row, as is ensured by the back-reference $Z[-1]$, requiring the value in each cycle to be the same as that of the previous one.

For the third row—which does not correspond to the third cycle, but at least to the eighth cycle, as the second row is repeated at least six times—the inputs for A, C are arbitrary and B is equal to $p + 1$. The output value for X is an arbitrary one between 0 and p. The output Y contains a back reference to Y from the previous cycle. Thus, in the first cyle of the third row, Y is greater than $2 * p$ (as $Y = 2 * p$ from the second row's last cycle). The value of Y must then increase in each further cycle. The value of Z must be more than half the value for Y in order to satisfy the constraint $2 * Z > Y$. The third row may be repeated arbitrarily often, as indicated by the symbol $*$ in column DURATION. Note, that no real system is able to fulfill the last row for an arbitrarily large number of steps, since the enforcement of strict monotonicity in Y must lead to an integer overflow at some point.

D. Semantics

A generalized test table can be unrolled to a family of test tables with only value 1 in the duration column. To this end, each row is repeated a number of times according to the origin duration constraint. In general, the set of unrolled tables may be infinite, containing test tables of arbitrary (but finite) length. A system fails if there exists a run and a number $n \in \mathbb{N}$, such that all unrolled test tables which satisfy the input constraints for the first n cycles fail the first n output constraints. A system passes the test cases described by a generalized test table iff the system does not fail it.

This definition may seem unnecessarily complicated, but it is not: There are two aspects we need to consider. First, a generalized test tables covers only those situations where the sequence of input stimuli is described in the generalized test case. All uncovered input stimuli can not reveal a failure. Second, there may be multiple unrolled test cases for one generalized test table, that match the input stimuli of a run. In this case, only one unrolled test case needs to match the system response to let the run pass the generalized test table. Through the evaluation on prefixes with length n, we prevent two cases: (a) We can apply the prefixes of a test case until length n if the $n + 1$th input constraint is contradictory, and (b) if the run matches the shortest unrolled test case of length n, it can not fail on longer unrolled test cases, even the run is longer than n.

For example, consider a very simple generalized table with just one row. In that row, there is one input variable fixed to have the constant value 1 and one output variable specified with a constraint $[1, 2]$. Then, the value pairs $1/1$ and $1/2$ are the possible concrete instances. The intuition of the test is that the system may answer 1 or 2 to the input 1. With a naive definition, where the concrete system behavior must pass all concrete tests, an observable behavior of $1/1$ would pass the test $1/1$ but fail the test $1/2$. On the other hand, using the more complex definition given above, the behavior $1/1$ satisfies the generalized test table as it passes the test case for $1/1$. Also, the behavior $2/3$ passes the test as the input 2 does not occur in the concrete tests. However, the behavior $1/3$ does not satisfy the generalized table, which corresponds to the intuition.

We give a deeper description of the formal semantics in [1].

III. Application Example

In this section we exemplarily show how generalized test tables can be used to increase test coverage for an industrial piece of code. The approach's feasibility is shown by analyzing (1) a typical function block from an industrial context and (2) an application scenario which combines this block with a second one. According to Vogel-Heuser et al. [6], software faults are mainly caused on these levels of system design: the atomic and the basic level.

Function of the Software. Interpolation blocks which map actual sensor values to a defined range of physical values in software provide one of the most commonly needed functions in software for automated systems. The function block examined here can be operated in two different modes: Before the actual operation, the calibration function (mode "Teach") expects two independent reference points to learn the linear relationship between sensor and physical values. In practice, these points are provided by the application engineer who manually calibrates the system by supplying default work pieces to the system. After calibration, the mapping function (mode "Op") performs linear interpolation where it translates a sensor reading into the physical value according to the learned data.

A schematic view of the function block is shown in Fig. 5. Besides the mode selector (Mode) and the sensor value input (X), the block has two additional inputs needed during calibration: (TPy) is used as the physical reference value during teaching and (TPSet) indicates that teaching is in progress if set to 1. The block's single output is the physical value (Y). In normal operation, after two reference points (x_1, y_1) and (x_2, y_2) have been learned, the input value X results in

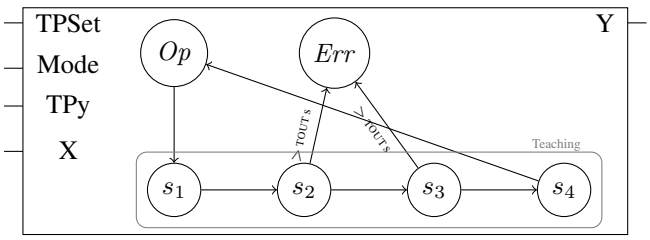

Fig. 5: Schematic view of the investigated function block with a state machine describing its operation.

#	Inputs TPy	TPSet	Mode	X	Outputs Y	DURATION
1	60	0	Op	0	0	10
2	60	0	Teach	1.2	0	10
3	60	1	Teach	1.2	0	1
4	2108	0	Teach	4.0	0	10
5	2108	1	Teach	4.0	0	1
6	0	0	Teach	0	0	1
7	0	0	Op	28	.438	1
8	0	0	Op	316	3.65	1
9	0	0	Op	3132	4.20	1

Fig. 6: Concrete test table of analog sensor function block

an output value $Y = L(x_1, y_1, x_2, y_2, X)$ which the linear regression between the two points defined as

$$L(x_1, y_1, x_2, y_2, X) =_{\text{def}} \; y_1 + \frac{y_2 - y_1}{x_2 - x_1}(X - x_1) \; . \qquad (1)$$

If the two reference points make interpolation impossible (e.g., if $x_1 = x_2$), the function block enters an error state. If no reference point is presented for more than TOUT cycles while in teaching mode, the block also goes into the error state. Before finishing calibration or if the function block is in the error state, the output Y is always 0. Fig. 5 includes a state chart for the block which contains the normal operation state, the error state and the teaching state which is subdivided into four substates s_1 to s_4.

A concrete test table. One concrete test case for this block is shown in Fig. 6. It covers the calibration and normal operation. The block is brought into teaching mode and two reference points $(60, 1.2)$ and $(2108, 4.0)$ are used for block calibration from step 2 to 5. Afterwards the normal operation of the block is tested in steps 7–9 which send inputs (X) of 28, 316, and 3132. The expected physical values (Y) are 0.438, 3.650, 4.200 according to linear interpolation. The software fails this test case if it does not produce the expected output in one ore more steps.

Generalization. This test case can be generalized by the means introduced in Sect. II-B to increase the test coverage. The resulting generalized table does not only cover the few concrete test values mentioned in the concrete table but includes all possible input sequences, thus comprises infinitely many test cases. If the software can be formally verified to adhere to the generalized test table, this is a more far-reaching validation result than running concrete tests.

#	Inputs TPy	TPSet	Mode	X	Outputs Y	DURATION
1	–	–	Op	–	0	*
2	–	0	Teach	–	0	$[1, \text{TOUT}]$
3	y_1	1	Teach	x_1	0	1
4	–	0	Teach	–	0	$[1, \text{TOUT}]$
5	y_2	1	Teach	$x_2, \neq x_1$	0	1
6	–	–	Teach	–	0	1
7	–	–	Op	–	L	*

Fig. 7: Generalized test table of analog sensor function block, where L is the linear regression , see (1).

The generalization of the concrete test begins in step 1 where prior to calibration, Y is always 0. In step 2, the mode is set to "Teach" (entering s_1 in Fig. 5), and in step 3, the first teaching point (x_1, y_1) is provided to the block as described (s_2). In step 5, the second point (x_2, y_2) is sent to the system (s_3). Steps 2, 4 and 6 are waiting phases between the teach points. The waiting time is not fixed and only limited by the maximum waiting time TOUT. After calibration, the block is set back to normal mode in step 7 where an arbitrary sensor value X is sent to the function block. The expected output is the linear interpolation value according to (1). This last step is repeated indefinitely often.

This generalized test table represents infinitely many individual finite test cases for all possible reference points and queries. However, it is still not a complete behavioral specification for the block. The sequence of steps is fixed and does not cover all cases. Fig. 7 only represents the normal operation. All situations where the block is to go to the error state are not covered (e.g., if three reference points are given or if more than TOUT waiting cycles have occurred).

Verification process. To verify that the function block conforms to the specification given by the generalized test table, we encoded both the software and the test table as state transition systems and submitted them to a state-of-the-art model checker. To encode the examined software block in model checker logic, we reuse toolchain we presented in [7]. Like the software, the generalized test table is converted into a (non-deterministic) state transition system. Each step in the table correponds to one state in the system.

The proof for an industrial implementation against the table in Fig. 7 takes 8:50 minutes (median, n = 5) with nuXmv [8] (Version: 1.1.1) and IC3 on an Intel Core i7 860 with 2.80 GHz. We needed some restriction on the value domain of some variables. We used fixed-point number instead of floating-point numbers, with a precision three fractional digits. We limited the range teaching points values $[0, 5]$, set the waiting times TOUT and the margin to 1000. Any other variable, especially the sensor values X are only restricted by the given type domain (16 bit). The scalability of our approach mainly depends on the performance of the model checker. The generation of the logical representation is polynomially bounded in size of the generalized test table and the length of the given program. We

#	Inputs TPy	TPSet	Mode	X	Y'	Outputs Y	W	DURATION
7	–	–	Op	–	$< M, = Y$	L	0	1
8	–	–	Op	–	$\geq M, = Y$	L	0	$[0, Ton]$
9	–	–	Op	–	$< M, = Y$	L	0	1
10	–	–	Op	–	$\geq M, = Y$	L	0	Ton
11	–	–	Op	–	$\geq M, = Y$	L	1	*
12	–	–	Op	–	$< M, = Y$	L	1	$[0, Toff]$
14	–	–	Op	–	$< M, = Y$	L	1	$Toff$
13	–	–	Op	–	$\geq M, = Y$	L	1	1
15	–	–	Op	–	$< M, = Y$	L	0	*

Fig. 8: Extension of the generalized test table in Fig. 7 of a warning message

provide the verification artifacts on the companion web page[2].

Expandability. We analyzed a scenario with interconnected function blocks to evaluate the expandability of the approach on a basic software level. The verified interpolation block is integrated in a function block composed of four blocks: the interpolation block, a timer-off block (TOF), a timer-on block (TON) and a SR flipflop (see Fig. 9). The goal of the composed function block is that a warning is triggered, when the interpolated value is above a margin M for a certain time Ton. The warning is revoked only if the sensor value falls below the margin for another certain time $Toff$. The example is based on industrial code of an aPS. The generalized test table of Fig. 8 is an extension of the specification in Fig. 7, adding 9 steps and two columns: the input variable Y' for specifying the margin value M and the output warning (W). Steps 7–9 verify the behavior of the software in the case the interpolated value is above the margin for less than the time Ton. According to the specification, a warning must not be issued. The second case (steps 10 and 11) enforces a warning if the margin is exceeded for period longer than Ton. Steps 12–15 verify the that the warning is reset once the interpolated value falls below M again.

We verified the industrial software under the same conditions as above and created three different scenarios. In the first scenario, we used the steps 7–9 to ensure, that no warning is given, if Y' does not exceed the margin M for Ton cycles. The proof takes 15:37 minutes (median, $n = 5$). In the second scenario, we check the signaling behavior of warning, given the protocol given in the rows 9–14 (13:32 minutes, median $n = 5$). For the third scenario, we validate the error case of both teaching points have sames x value ($x_1 = x_2$). As a result the Y should remain 0 and no warning is given. The model checker takes 46.77 seconds (median, $n = 32$), without any restrictions on value domain of the teaching points. Like TOUT in the previous example, we set Ton and $Toff$ to 5 and the margin is 1000.

IV. Empirical Study

To assess how comprehensible the generalized table notation is for engineers, we conducted a survey with 22 students in the mechanical engineering department at TUM. The students

[2]https://formal.iti.kit.edu/indin17

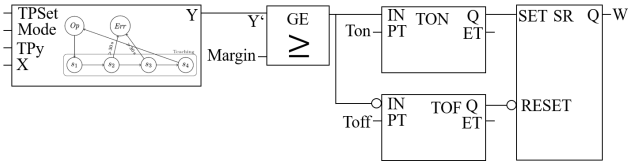

Fig. 9: Composed function block

have taken PLC coding courses, but were not experienced in practice. The questionnaire comprised four parts which had to be answered within 20 minutes. The concept of table generalization, its notations (see Sect. II) and the motivation for this survey were explained in the first part. In the second part, a basic function block description, a Timer-off function (TOF), was given together with a concrete test table and a generalized test table for comparison. To evaluate the clarity and understandability of the notation, the students were asked to decide whether the generalized table covered several different behavior descriptions. The third part provided a detailed description of an interpolation block, similar to the one of Sect. III but more complex due to additional physical values. A concrete test table was given and the participants were required to come up with a corresponding generalized test table for this given function block. This part was developed for measuring accessibility and comprehensibility of the notation for users without expert knowledge. Lastly in part 4, we asked the participants to rate the usability (i.e., simplicity) and usefulness (i.e., effectiveness) on a scale from one (positive) to ten (negative) for some examples using different expression types (global variables, "don't-care", comparisons, and references, as explained in Sect. II-B). Apart from the questionnaire sheet, no additional information or explanation was given during the survey. In spite of the considerable time restriction, 81% of the respondents were able to answer the questions on the simpler generalized tables correctly and 60% were correct on the most difficult one. These figures indicate that the concept of generalized test tables is easy to apprehend. As far as the different notational extensions are concerned, global variables, "don't-care" values and intervals were used correctly by 80% of the participants while references to other cells and constraints in form of general formulas were used correctly by less than 40%. Fig. 10 shows the survey result for part 4. Dots and crosses indicate average for the usefulness and simplicity for four different kinds of extension. Even though a larger number of subjects would be required for more general results, the positive result of this survey with students suggests that the concept of generalized test tables could be beneficial and accessible, and these degree of usability will be more for professional engineers.

V. Related Work in APS Verification

Software quality assurance methods aim to support developers in identifying software faults and to fix them efficiently. In the field of factory automation, research for software quality assurance is prevalently directed towards model-based testing

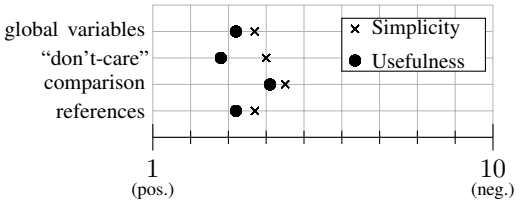

Fig. 10: The result of the survey on the comprehensibility of generalized test tables

and formal verification. Both testing and formal verification allow the developer to diagnose system faults and fix them. While formal verification covers a wide range of scenarios and unexpected events, testing validates system behavior within pre-defined scenarios and an expected scope of events, i.e., it does not go beyond the specified test cases. However, testing is the established means in production industry regarding quality assurance and enjoys broader industry acceptance [2]. Formal verification, on the other side, proves software correctness with respect to its specification by exhaustively exploring all reachable states within the model [9], [10]. As formal verification reaches a much higher coverage, automatic verification can already be applied earlier in the design phase [11]. The established formal verification technique of model checking has been used in several approaches in verification of PLC code [12], [13], [14]. Other methods focus on combining testing with formal methods in order to increase the test coverage. Buzhinsky et al. [15] use a formal model for Net Condition/Event Systems in order to generate and execute test cases. According to Pakonen et al. [4], formal methods are –besides safety issues– not widely used in production industry. Challenges are the state space explosion, the necessary knowledge in formal description languages for writing formal specifications, and a potentially high modelling effort. Duschl et al. [16] analyse the error types which can appear in code written in other modeling languages such as UML and SysML or IEC 61131-3 Function Block Diagrams (FBD). They conducted a precise analysis of error causes and concluded the most common reason for faults to be an inadequate understanding of used notations. As a means to reduce the complexity of formal specifications and hence overcome the barrier to establish formal methods in PLC-based production industry, several interviews with experts on quality assurance and experts on formal methods were conducted. They indentified the following six important requirements on specification language in two categories:

- Formal methods:
  - R1: Formalization of the specification
  - R2: Transformation into temporal logics such as CTL
- Industrial applicability:
  - R3: Consideration of timing aspects (continuous or discrete)
  - R4: Well-known language in industry of aPS
  - R5: Reuse of existing testing data
  - R6: Possibility of Automated verification

However, most similarly-targeted verification approaches rely on manually formalized mathematical artefacts or unfamiliar notations, e.g., petri nets [17]. Thus, petri nets do neither confirm to R4, nor to R3 which also considers timing aspects. Translation methods from the Sequential Function Chart (SFC) into formalized automata are also introduced in [13], [18], and [19]. A review on user-friendly formal specification languages is given in [20] and [4]. One proposed specification is a natural pattern language which combines several keywords to a natural sentence. This method is also used by PLCverif [21], [22], a tool for PLC code verification. The goal of this tool is to hide the unfamiliar formal methods and provide a platform for model generation, model abstraction and specification description in natural language instead. The tool supports the IEC 61131-3 language Structured Text (ST). With this approach, patterns must be defined new which is not in line with R5. In a second step, In [20] Pang et al. review visual formalisms such as UML and Timing diagrams. In addition, investigations on the generation of test cases from formal specifications in UML models are proposed in [23], [24], and [25]. UML is a wide-spread modelling language, for which many approaches are given [2]. Disadvantages are the complex handling of UML models and a gap of UML-models in aPS testing (cf. R5). Timing diagrams as modelling and specification languages in production industry have been evaluated for generating test cases [26], [5] as well as a means for formal specification [27], [28]. Their benefit has been demonstrated in existing industrial know-how and in the consideration of time. However, test cases in the form of timing diagrams are rarely used. A main reason is that available data cannot be reused that it focusses only on one scenario.

Darvas et al. present AND/OR tables for describing simple specification descriptions in [29]. Their advantage is that hardly any knowledge from the specification domain is required. Test cases as such often already exist in production industry and are usually written in the form of tables. Timing behavior is not mentioned in this approach (R3).

Hence, many considerd specification languages such as CTL, Petri Net, UML, Pattern Language, Timing Diagrams and concrete test tables are transferable to CTL logic (R1) and enable a formal description (R2). Thus, also an automation of the verification process (R6) is possible. But no specification languages complies with all requirements.

Test tables as a means for specification cover all six requirements R1-R6. The gap lies within the missing flexibility for covering a number of test cases. To the best of our knowledge, there is no approach which reuses existing test tables as a means for formal specification. This publication provides generalized test tables as an extension to test tables by allowing for notions of abstraction to cover a wide range of behavioral cases including flexible timing specifications.

VI. Conclusion

Formal verification is not yet an established validation technique in the field of aPS because of a barrier imposed by the requirement to formulate formal specifications. Yet, this

technique becomes increasingly interesting regarding recent developments such as Industry 4.0, with its increasingly flexible, cooperating systems, requiring close compliance with specifications and robustness.

In this paper, we presented generalized test table which are an understandable approach to provide such formal specifications of complex behavior of reactive systems. (Concrete) test tables are state of the art for validation in industry. The approach presented here makes test tables more general and thus allows them to cover many test cases at once. Three generalization features were introduced: abstraction, references, and generalized durations. Generalized test tables are particularly well-suited for evolving software systems: Concrete test tables require fixed values in all cells such that they are likely to become invalidated if, e.g., an extra wait cycle is introduced. The symbolic nature of the input/output specification in generalized tables is more flexible and thus more resilient to system changes during system evolution.

We demonstrated that proving the conformity of a software system against a generalized test table is feasible and reasonable fast by means of an example extracted from an industrial piece of code. Even though we showed the usability assessment, a study with more number of field engineers will be conducted in the near future to achieve more general usability results.

Moreover, our concept is extendable and can be adopted to the needs of software engineers and software evolution.

REFERENCES

[1] B. Beckert, S. Cha, M. Ulbrich, B. Vogel-Heuser, and A. Weigl, "Generalised test tables: A practical specification language for reactive systems," in *Integrated Formal Methods - 13th International Conference, IFM 2017, Torino, Italy, Sept. 18-22, 2017*. Springer, 2017.

[2] B. Vogel-Heuser, A. Fay, I. Schäfer, and M. Tichy, "Evolution of software in automated production systems - challenges and research directions," *Journal of Systems and Software (JSS)*, pp. 54–84, 2015.

[3] S. Rösch and B. Vogel-Heuser, "Model-based test case generation based on timing diagrams to test plc software error handling routines for technical process faults," *Journal of Systems and Software (JSS)*, 2016.

[4] A. Pakonen, C. Pang, I. Buzhinsky, and V. Vyatkin, "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)*, Berlin, Germany, 2016.

[5] S. Rösch, "Model-based testing of fault scenarios in production automation," Dissertation, Technische Universität München, München, 2016.

[6] B. Vogel-Heuser, J. Fischer, S. Rösch, S. Feldmann, and S. Ulewicz, "Challenges for maintenance of plc-software and its related hardware for automated production systems: Selected industrial case studies," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept 2015, pp. 362–371.

[7] B. Beckert, M. Ulbrich, B. Vogel-Heuser, and A. Weigl, "Regression verification for programmable logic controller software," in *17th International Conference on Formal Engineering Methods (ICFEM 2015)*, ser. LNCS, vol. 9407. Springer, 2015, pp. 234–251.

[8] R. Cavada, A. Cimatti, M. Dorigatti, A. Griggio, A. Mariotti, A. Micheli, S. Mover, M. Roveri, and S. Tonetta, "The nuxmv symbolic model checker," in *CAV*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, 2014, pp. 334–342.

[9] T. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, P. Schnoebelen, and P. McKenzie, *Systems and Software Verification*. Berlin, Heidelberg: Springer Berlin Heidelberg., 2001.

[10] E. Clarke, O. Grumberg, and D. Peled, "Model checking," *MIT Press*, 1999.

[11] J. Lahtinen, J. Valkonen, K. Björkman, J. Frits, I. Niemelä, and K. Heljanko, "Model checking of safety-critical software in the nuclear engineering domain," *Reliab. Eng. Syst. Saf.*, pp. 104–113, 2012.

[12] N. Bauer, S. Engell, R. Huuck, S. Lohmann, B. Lukoschus, M. Remelhe, and O. Stursberg, "Verification of plc programs given as sequential function charts," *Integration of Software Specification Techniques for Applications in Engineering*, pp. 517–540, 2004.

[13] A. N. I. Wardana, J. Folmer, and B. Vogel-Heuser, "Automatic program verification of continuous function chart based on model checking," *35th Annu. Conf. IEEE Ind. Electron.*, pp. 2422–2427, 2009.

[14] S. Biallas, J. Brauer, and S. Kowalewski, "Arcade.plc: A verification platform for programmable logic controllers," *Conference on Automated Software Engineering (CASE)*, pp. 338–341, 2012.

[15] I. Buzhinsky, C. Pang, and V. Vyatkin, "Formal modeling of testing software for cyber-physical automation systems," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 3, 2015, pp. 301–306.

[16] K. C. Duschl, D. Gramß, M. Obermeier, and B. Vogel-Heuser, "Towards a taxonomy of errors in plc programming," *Cogn. Technol. Work*, pp. 417–430, 2015.

[17] T. Mertke and G. Frey, "Formal verification of plc programs generated from signal interpreted petri nets," *2001 IEEE Int. Conf. Syst. Man Cybern. e-Systems e-Man Cybern. Cybersp.*, pp. 2700–2705, 2001.

[18] M. P. Remelhe, S. Lohmann, O. Stursberg, S. Engell, and N. Bauer, "Algorithmic verification of logic controllers given as sequential function charts," *IEEE International Conference on Robotics and Automation*, pp. 53–58, 2004.

[19] J. Provost, J.-M. Roussel, and J.-M. Faure, "Translating grafcet specifications into mealy machines for conformance test purposes," *Control Eng. Pract.*, pp. 947–957, 2011.

[20] C. Pang, A. Pakonen, I. Buzhinsky, and V. Vyatkin, "A study on user-friendly formal specification languages for requirements formalization," in *14th IEEE International Conference on Industrial Informatics (INDIN)*, 2016.

[21] D. Darvas, B. Fernández Adiego, and E. Blanco Viñuela, "Plcverif: A tool to verify plc programs based on model checking techniques," in *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, Melbourne, Australia, 2015, pp. 911–914.

[22] D. Darvas, E. Blanco Viñuela, and I. Majzik, "Plc code generation based on a formal specification language," in *14th IEEE International Conference on Industrial Informatics (INDIN)*, 2016, pp. 389–396.

[23] B. Kormann, D. Tikhonov, and B. Vogel-Heuser, "Automated plc software testing using adapted uml sequence diagrams," *14th IFAC Symposium of Information Control Problems in Manufacturing*, pp. 1615–1621, 2012.

[24] D. Winkler, S. Biffl, and T. Östreicher, "Test-driven automation: Adopting test-first development to improve automation systems engineering processes," *EuroSPI*, pp. 1–13, 2009.

[25] D. Winkler, S. Biffl, and T. Östreicher, "Model based TTCN-3 testing of industrial automation systems - First results," *IEEE Conference on Emerging Technologies in Factory Automation*, pp. 1–4, 2011.

[26] S. Rösch, D. Tikhonov, D. Schütz, and B. Vogel-Heuser, "Model-based testing of plc software: Test of plants' reliability by using fault injection on component level," *IFAC World Congress*, p. 3509–3515, 2014.

[27] S. Ulewicz, M. Ulbrich, A. Weigl, M. Kirsten, F. Wiebe, B. Beckert, and B. Vogel-Heuser, "A verification-supported evolution approach to assist software application engineers in industrial factory automation," in *IEEE International Symposium on Assembly and Manufacturing (ISAM)*, Fort Worth, USA, 2016, pp. 19–25.

[28] B. Fernández Adiego, D. Darvas, E. Blanco Viñuela, J.-C. Tournier, V. M. González Suárez, and J. O. Blech, "Modelling and formal verification of timing aspects in large PLC programs," in *Proceedings of the 19th IFAC World Congress*, 2014, pp. 3333–3339.

[29] D. Darvas, E. Blanco Viñuela, and I. Majzik, "A formal specification method for PLC-based applications," in *Proceedings of the 15th International Conference on Accelerator and Large Experimental Physics Control Systems*, 2015, pp. 907–910.