Generalised Test Tables: A Practical Specification Language for Reactive Systems[⋆]

Bernhard Beckert[1], Suhyun Cha[2], Mattias Ulbrich[1], Birgit Vogel-Heuser[2], and Alexander Weigl[1]

[1] Karlsruhe Institute of Technology, Germany
beckert@kit.edu, ulbrich@kit.edu, weigl@kit.edu
[2] Technical University of Munich, Germany
suhyun.cha@tum.de, vogel-heuser@ais.mw.tum.de

Abstract. In industrial practice today, correctness of software is rarely verified using formal techniques. One reason is the lack of specification languages for this application area that are both comprehensible and sufficiently expressive. We present the concepts and logical foundations of generalised test tables – a specification language for reactive systems accessible for practitioners. Generalised test tables extend the concept of test tables, which are already frequently used in quality management of reactive systems. The main idea is to allow more general table entries, thus enabling a table to capture not just a single test case but a family of similar behavioural cases. The semantics of generalised test tables is based on a two-party game over infinite words.
We show how generalised test tables can be encoded into verification conditions for state-of-the-art model checkers. And we demonstrate the applicability of the language by an example in which a function block in a programmable logic controller as used in automation industry is specified and verified.

1 Introduction

Complex industrial control software often drives safety-critical systems, like automated production plants or control units embedded into devices in automotive systems. Such controllers have in common that they are *reactive systems*, i.e., that they periodically read sensor stimuli and cyclically execute the same piece of code to produce actuator signals.

Usually, in practice, the correctness of implementations of reactive systems *is not* verified using formal techniques. What *is* used instead in industrial practice today is testing, where individual test cases are used to check the reactive system under test [11]. Main reasons why formal methods are not popular are: (a) It is difficult to adequately formulate the desired temporal properties. (b) There is

[⋆] 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).

a lack in specification languages for reactive systems that are both sufficiently expressive and comprehensible for practitioners.

Test cases are commonly written in the form of *test tables*, in which each row contains the input stimuli for one cycle and the expected response of the reactive system. Thus, the whole table captures the intended behaviour of the system (the sequence of actuator signals) for one particular sequence of input signals.

In this paper, we present a novel specification language called *generalised test tables* (gtts) which lifts the principle of test tables to an expressive means for temporal specification of reactive systems. With a gtt one can describe an *entire family* of test cases with a single table.

The specification language comprised of gtts is designed to preserve the intuitiveness and comprehensibility of (non-generalised) concrete test tables – in particular for system design engineers who are experts in test case specification but are not familiar with formal temporal specification. To this avail, the generalisations are defined as conceptional extensions of notation already present in concrete test tables. The features that go beyond the concrete case are chosen such that essential characteristics of concrete test tables are preserved. Moreover, concrete test tables are a special case of gtts. We argue that, thus, gtts are still intuitive for an engineer. The characteristics of concrete test tables that we deem essential and that are preserved in gtts are:

1. Every signal/actuator cycle corresponds to one row in the test table.
2. Rows in the test table are traversed sequentially (no jumping around).
3. Every row formalises a local implication of the form: "If the signal values adhere to the input constraint, then the actuator signals adhere to the output constraint."

The main features of generalised test tables that go beyond concrete tables are (a) generalisations of notational elements known from concrete tables and (b) concepts adopted from other well-known table formalisms like spreadsheets.

The main contributions of this paper are: (1) the concept of gtts as a practical specification language for reactive systems (Sect. 3); (2) a formal semantics for gtts (Sect. 4), defined by means of a semi-deterministic input/output game; (3) a sound encoding for gtts into Büchi automata, together with optimisations (Sect. 5), which has been implemented (4) an extended example in which a realistic min/max function block, which is a typical example for the software driving automated production systems, is specified and verified using a gtt (Sect. 6).

2 The Basis: Concrete Test Tables

Concrete test tables – of which generalised test tables are an extension – describe a single test case for a reactive system. The rows of a concrete test 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 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, i.e., how often the row is to be repeated. 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). A table row with a duration of n is equivalent to repeating that same row n times with a duration of 1.

	Inputs			Outputs			
#	A	B	C	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. 1: Example for a concrete test table.

Example 1. Fig. 1 shows an example for a simple concrete test 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.

There is no restriction on the types of variables and their values that can be used in the tables. In the following, we use variables of type Boolean and integer; and the example in Sect. 6 uses bounded bit vector types.

3 The Concept of Generalised Test Tables

Generalising 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 generalised test table if it responds to input values that adhere to the input constraints with output values that adhere to the output constraints. This generalises the meaning of concrete test cases were the constraints are unique values. Thus, a generalised test table specifies a – possibly infinite – set of concrete test tables. A detailed explanation of the semantics of generalised test table is given in Sect. 4.

In the following, we explain three generalisation concepts: (1) abstraction using constraint expressions (which is the basis of generalisation), (2) using references to other cells in constraint expressions, and (3) using generalisation 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. 2): In the column for variable X, the constraint "$X < n$" can be written as "$<n$" and "$X = n$" simply as "n". We allow interval constraints $[n, m]$, which stand for "$X \geq n \wedge X \leq m$." And "–" is the constraint satisfied by all values ("don't care") .

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

References to other cells. A reactive system's behaviour depends both on the current and the previous input stimuli. Therefore, the expected values in the cells of a generalised 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 p in a cell with input A (short for $A = p$) and $p + 1$ in a cell with output X (short for $X = p + 1$) to express that the value of output X is equal to $p + 1$ for the input A being of value p. 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, $X = p$ is equivalent to "don't care" if p does not occur in any other cell.

In addition to global variables, we allow references to other cells using the form "$X[-n]$", where X is a variable name and $n \geq 0$ is a concrete number. $X[-n]$ refers to the cell in the X-row n cycles before the current one. For references to other cells in the current cycle, we just write "X" as an abbreviation for "$X[-0]$", which refers the value of column X on the same row.

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. References to future cycles (both relative and absolute) could also be added – at least for static analysis – but are not covered in this paper.

Generalisation in the duration column. The DURATION variable defines the number of cycles for which a row is repeated. As a further generalisation 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 “$\geq$0”.

Example 2. Fig. 3 shows an example of a simple generalised test table, incorporating the generalisation concepts described above. Note that the concrete table depicted in Fig. 1 is one of the possible instances of the generalised test table given in Fig. 3, achieved by instantiating the global variable p with the value 3.

	Inputs			Outputs			
#	A	B	C	X	Y	Z	DURATION
0	1	1	2	0	0	–	1
1	–	p	p	$=2*p$	X	$Z[-1]$	≥ 6
2	–	$p+1$	–	$[0,p]$	$>Y[-1]$	$2*Z>Y$	$*$

Fig. 3: Example for a generalised test table with a global variable p.

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 output Z is equal to the one of the first row, as it 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 inclusively. The output Y contains a back reference to Y from the previous cycle. Thus, in the first cycle 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 fulfil 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.

This paper introduces and describes the formal foundations of gtts and shows their principal suitability for formal specification and automatic verification. In a companion paper [12], the presentation focuses more on the adequacy and usability of the approach for engineers. A thorough empirical study which analyses the accessibility of the individual features by field engineers remains as future work.

4 Semantics of Generalised Test Tables

A gtt is a sequence of rows. Each row corresponds to three constraints: one for the input variables, one for the output variables, and one for duration of that row. Which part of a constraint is written in exactly which column is only relevant as long as abbreviations and syntactic sugar is used. For example, writing Y in the X-column of a table is expanded to $X = Y$, so the column is relevant. But it is irrelevant in which column we write a constraint like $X = Y$, that is not further expanded. This gives rise to the following definition, which basically just fixes notation:

Definition 1 (Generalized Test Table as Sequence of Constraints). *Let T be a generalised test table with m rows; let $InVar_T$ and $OutVar_T$ be the set of input variables resp. the set of output variables of T; and let $GVar_T$ be the set of global variables occurring in T. Then T is identified with the sequence*

$$(\phi_1, \psi_1, \tau_1) \cdots (\phi_m, \psi_m, \tau_m) \ ,$$

where ϕ_i is the conjunction of all constraints contained in cells in row i that correspond to input *variables, ψ_i is the conjunction of all constraints contained in cells in row i that correspond to* output *variables, and τ_i is the interval contained in the duration column at row i.*

The reactive systems whose behaviour is being specified by test tables can be formalised as functions from sequences of inputs to sequences of outputs. The possible inputs are elements of $I = I_1 \times \cdots \times I_k$ where the I_r are the value spaces of the input variables. And the possible outputs are elements of $O = O_1 \times \cdots \times O_l$ where the O_s are the value spaces of the output variables.

Definition 2 (Reactive system). *A reactive system is a history-deterministic function $p : I^\omega \to O^\omega$. That is, $i_1{\downarrow_n} = i_2{\downarrow_n}$ implies $p(i_1){\downarrow_n} = p(i_2){\downarrow_n}$ for all n, where $x{\downarrow_n}$ denotes the finite initial sub-sequence of x of length n.*

In the following, we often identify a reactive system p with the set of its possible traces, i.e., $p \subseteq (I \times O)^\omega$.

4.1 Unrolled Instances of Generalised Test Tables

The rows of gtts have a duration and may be repeated more than once. In a first step towards defining the semantics of gtts, we eliminate the indeterminism w.r.t. the repetition of rows and define the set of *unrolled instances* of a gtt by making the repetitions explicit. At the same time, we also instantiate the global variables contained in gtts with all their possible values.

Definition 3 (Unrolled Instances). *Let $G = (\phi_1, \psi_1, \tau_1) \cdots (\phi_m, \psi_m, \tau_m)$ be a gtt without global variables. The set $D1(G)$ of* unrolled instances *of G consists of all gtts*

$$(\phi_1, \psi_1, 1)^{T_1} \cdots (\phi_m, \psi_m, 1)^{T_m}$$

such that T_i is in the interval τ_i ($1 \leq i \leq m$). [3]

In the unrolled instances, the duration constraint is redundant (as it is always 1); in the following, we therefore write (ϕ_i, ψ_i) instead of $(\phi_i, \psi_i, 1)$.

Global variables are not considered in unrolled instances, as their semantics is defined via universal quantification: A system has to conform to the test table for all their instances (see Def. 5).

In general, the set of unrolled instances for a generalised test table is infinite. This does not pose a problem as the notion of unrolled instances is only used as a theoretical concept for defining the semantics of gtts.

4.2 Evaluation of expressions

The evaluation of constraints that appear in unrolled instances of gtts is straight forward. Note that they do not contain global variables anymore as these have been instantiated during unrolling.

Definition 4. *Let $v \in (I \times O)^*$ be a partial trace of length $n \geq 1$. And let $v{\downarrow_n} = (i, o)$ be the last element of the trace. Then, the valuation function $[\![e]\!]_v$, which assigns a value to every expression or formula e, is inductively defined by:*

$$\begin{aligned}
[\![e \circ f]\!]_v &= [\![e]\!]_v \circ [\![f]\!]_v \quad \textit{for } \circ \in \{+, -, \leq, \wedge, \vee, \ldots\} \\
[\![X]\!]_v &= i(X) \quad \textit{if } X \in \mathit{InVar} \\
[\![X]\!]_v &= o(X) \quad \textit{if } X \in \mathit{OutVar} \\
[\![X[-k]]\!]_v &= [\![X]\!]_{v\downarrow_{(n-k)}} \quad \textit{if } k < n \\
[\![X[-k]]\!]_v &= [\![X]\!]_{v\downarrow_1} \quad \textit{if } k \geq n
\end{aligned}$$

4.3 Two-Party Game for Defining Test Conformance

The intuition behind the following definitions is the following: A reactive system p conforms to a gtt G if every trace $t \in p$ conforms to G, where a trace conforms to G if one of the following conditions holds: (a) the input/output pairs of t satisfy *all* rows of *at least one* unrolled instance of G, or (b) t fails to satisfy the input constraints of *all* unrolled instance of G. In the former case, the trace is covered by the specification described by G, in the latter case, the input sequence triggers an application scenario which is not covered by the specification.

Formally, we define the semantics of a gtt G by means of a game played between a challenger (that chooses the inputs) and the reactive system p under test (that chooses the outputs). The challenger can be identified with the environment of the system. The game is played operating on a set S of unrolled instances of G from which in every round inconsistent and conflicting instances are removed.

The player removing the last consistent instance from S loses the game. In addition, the system can win by successfully reaching the end of one of the non-eliminated table instances.

[3] We use the notation $(\phi, \psi, \tau)^n$ do denote that the row (ϕ, ψ, τ) is repeated n times.

```
    Input: A gtt $T$
    $S \leftarrow D1(T)$
    $v \leftarrow \epsilon$
    $k \leftarrow 1$
 5: loop
        Challenger chooses $i \in I$.
        System computes $o \in O$.
        $v \leftarrow v \cdot (i, o)$

10:     $S \leftarrow \{D \in S \mid v \models \phi_k$ for the $k$-th row $t_k = (\phi_k, \psi_k)$ in $D\}$
        if $S = \emptyset$ then
            terminate: System wins                    ▷ Chosen input not covered by $T$
        end if

15:     $S \leftarrow \{D \in S \mid v \models \psi_k$ for the $k$-th row $t_k = (\phi_k, \psi_k)$ in $D\}$
        if $S = \emptyset$ then
            terminate: Challenger wins                ▷ Chosen output violates $T$
        end if

20:     if $\exists D \in S.\ |D| = k$ then
            terminate: System wins                    ▷ Unrolled instance $D$ has finished
        end if
        $k \leftarrow k + 1$
    end loop
```

Fig. 4: Game between challenger and system w.r.t. a gtt T

Fig. 4 shows the game's rules in algorithmic form. During the course of a game, S holds the set of unrolled instances of G which have not been eliminated, v holds the so far observed partial trace up to and including the current move, and k counts the iterations. Initially the set $S = D1(G)$ contains all unrolled instances of the gtt T (Line 2). In each round, the challenger chooses input values (Line 6), and the program under test computes its output from its internal state and the input values (Line 7). The functions which choose the input/output values depending on the observed partial trace are called *strategies*. Since reactive programs are deterministic, there is only one strategy for the program, which is encoded in its implementation. The challenger is not confined in its choices; there are many possible strategies for the challenger. Whenever S becomes empty, i.e., no unrolled instance of G satisfies the partial trace, the respective player loses the game: If this is caused by the input constraint ϕ_k being violated, the challenger loses and the system wins (Line 12). If S becomes empty because the output constraint ψ_k is violated, the system loses and the challenger wins (Line 17). If S contains a consistent unrolled instance which has been fully traversed (its length is the current iteration counter), then the partial trace v is a witness for the system conforming to the gtt. The system wins (Line 21).

A single game has three possible outcomes: Either (a) the challenger wins, or (b) the system wins, or (c) neither party wins (draw). In the case of a draw, the game is infinite, while a game where one player wins ends after a finite number of iterations.

A strategy for one party is called a winning strategy if it wins every possible game regardless of the other party's strategy. The definition of conformance to a gtt can now be defined based on who wins the games:

Definition 5 (Conformance). *The reactive system $P : I^\omega \to O^\omega$* strictly conforms *to the gtt T iff its strategy is a winning strategy for the game shown in Fig. 4 for all instantiations of global variables, i.e., it is winning w.r.t. $\sigma(T)$ for all instantiations σ that replaces each global variable occurring in T by an element of its value space.*[4] *The reactive system P* weakly conforms *to T iff its strategy never loses.*

I	O	⏲
−	1	*
−	2	[1,1]

Fig. 5: Gtt illustrating the difference between strict and weak conformance

The difference between weak and strict conformance is that of whether the analysis of a system w.r.t. a test table successfully finishes after finitely many steps or whether the system is under consideration for infinitely many steps. For example, consider the very simple gtt shown in Fig. 5. Intuitively, it requires that – independently of the input – the output must eventually be 2 after an arbitrary number of cycles with output 1. The reactive system that always returns 1 (and never 2) does not have this property. Correspondingly, it does not strictly conform to the table (it does not have a winning strategy). But it weakly conforms (it never loses either). This corresponds to the fact that by inspecting finite partial traces, one cannot decide whether or not this system violates the test table.

Any analysis that only considers partial traces (like run time monitoring or testing) can, in general, only test weak conformance. A static analysis, however, is able to analyse a reactive system w.r.t. strict conformance.

The definition of conformance (Def. 5) can be lifted to the case of non-deterministic reactive systems by requiring that *all* possible strategies of P must be winning strategies.

This semantics definition seems unnecessarily complicated, but an attempt to define it on the program traces is bound to fail as the implication of a violated constraint is different depending on whether it occurs on the input or on the output values: A gtt with constraint false on the input side is trivially satisfied, while false on the output side makes it unsatisfiable. Yet, both describe the same set of traces: the empty set.

[4] In fact, the global variables are replaced by constants representing values. We assume that every value can be represented by a constant.

	I	O	DURATION
1	...	...	$[0, \infty]$
2	...	...	$[1, 1]$
3	...	...	$[0, 1]$
4	...	...	$[1, 1]$

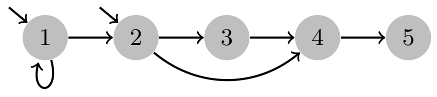

Fig. 6: A normalised table and the successor relation on its rows.

5 Transforming Generalised Test Tables into Automata

In this section, we describe the construction of a Büchi automaton that logically encodes conformance to a gtt. In order to ease the presentation of this construction, we assume a normalised form of test tables which allow only a restricted form of duration constraints:

Definition 6 (Normalised gtt). *A gtt $G = (\phi_1, \psi_1, \tau_1) \cdots (\phi_m, \psi_m, \tau_m)$ is* normalised *if $\tau_i = [1, 1]$, $\tau_i = [0, 1]$, or $\tau_i = [0, \infty]$ $(1 \leq i \leq m)$.*

The syntactical restriction of normalised tables does not pose a limitation on the expressiveness of gtts due to the following observation:

Proposition 1. *For every gtt T there is a semantically equivalent normalised gtt T_0.*

The construction of such a normalised table T_0 for T is canonical: Every row with a finite duration interval $\tau = [a, b]$ is unrolled into b rows with the first a repetitions having duration $[1, 1]$ and the remainder having duration $[0, 1]$. If $\tau = [a, \infty]$, then the row is repeated a times with duration $[1, 1]$ and once with duration $[0, \infty]$. Note that if T has m rows and the largest number in the duration constraints is n, then the normalised table has at most $m \cdot n$ rows.

Due to the intervals in the duration constraints, it is not automatically clear to which row a system cycle has to conform, and which the successor row of each row is (as intermediate rows may have zero duration). The set of possible successor rows $succ(k)$ for row k in a normalised gtt can be represented as

$$\begin{aligned} succ(k) = \quad & \{k + 1\} & (1) \\ & \cup (\text{if } k < m \wedge 0 \in \tau_{k+1} \text{ then } succ(k + 1) \text{ else } \emptyset) & (2) \\ & \cup (\text{if } \tau_k = [0, \infty] \text{ then } \{k\} \text{ else } \emptyset) \ , & (3) \\ succ(0) = \quad & \{1\} \cup (\text{if } 0 \in \tau_1 \text{ then } succ(1) \text{ else } \emptyset\}) \ . \end{aligned}$$

In a normalised table, the next row is always a possible successor row (1), but rows may be leapt over (2), or repeated (3). $succ(0)$ is the set of rows in which the table may begin. Fig. 6 illustrates the row successor relation (right) for a normalised gtt (left).

Alphabet. The Büchi automata will accept ω-traces in $(I \times O)^\omega$ produced by a reactive system (Def. 2). The alphabet of the automata is defined over the domains of input and output variables of the reactive system. In the following, we use Boolean formulas to describe subsets of the alphabet.

States. A gtt $G = (\phi_1, \psi_1, \tau_1) \cdots (\phi_m, \psi_m, \tau_m)$ with m rows results in an automaton with 2^{m+2} states. The states are characterised by vectors $(s_1, \dots, s_{m+1}, \mathit{fail})$ of Boolean variables, one for each row in G (s_1 to s_m), one indicating termination s_{m+1}, and one indicating failure (fail). Intuitively, s_k is true in a state iff the table is in a situation where the test table may have been executed by the trace up to the k-th row. The initial state $(s^0_1, \dots, s^0_{m+1}, \mathit{fail}^0)$ is defined by

$$s^0_k = \text{true iff } k \in \mathit{succ}(0) \text{ and } \mathit{fail}^0 = \text{false} \ . \tag{4}$$

State Transition. Given a state $(s_1, \dots, s_{m+1}, \mathit{fail})$, its successor state $(s'_1, \dots, s'_{m+1}, \mathit{fail}')$ is deterministically computed according to these equivalences:

$$\bigwedge_{k=1}^{m} \big(s'_k \leftrightarrow \bigvee_{i=1}^{m} (s_i \wedge k \in \mathit{succ}(i) \wedge \phi_i \wedge \psi_i)\big) \tag{5}$$

$$s'_{m+1} \leftrightarrow \big(s_{m+1} \vee \bigvee_{i=1}^{n} (s_i \wedge m{+}1 \in \mathit{succ}(i) \wedge \phi_i \wedge \psi_i)\big) \tag{6}$$

$$\mathit{fail}' \leftrightarrow \Big(\mathit{fail} \vee \big(\bigvee_{i=1}^{m} (s_i \wedge \phi_i \wedge \neg\psi_i) \wedge \bigwedge_{i=1}^{m+1} \neg s'_i\big)\Big) \tag{7}$$

The equivalences in (5) encode that the k-th row is active in the next step (variable s'_k) if there is an active row i preceding k such that both its input constraint ϕ_i and output constraint ψ_i are satisfied. The same applies to the virtual row $m + 1$ behind the table in (6). Here, additionally, once true, the variable s_{m+1} never falls back to false again. The *fail* flag indicating a specification violation is defined in (7). It is triggered whenever there is one active row i such that its input constraint ϕ_i is satisfied while the output constraint ψ_i is violated and there is no active row in the next step. Note that the equivalences above ensure the state transition system is always deterministic.

The acceptance condition remains to be described. By definition, a Büchi automaton accepts an infinite word if one state from the set of final states is traversed infinitely often. We construct two different such accepting sets of states: condition A_{WC} for weak conformance and A_C for strict conformance (the following formulas are identified with the set of states that satisfy them):

$$A_{WC} := \neg \mathit{fail} \qquad\qquad A_C := \big(\bigwedge_{i=1}^{m} \neg s_i \wedge \neg \mathit{fail}\big) \vee s_{m+1}$$

For weak conformance, the automaton accepts any trace that never have set the flag *fail* to true. For strict conformance, the automaton accepts a trace if it reaches a state in which s_{m+1} (the flag for finishing a table) is true or there is no active row anymore without failing (i.e., the challenger has lost).

The automata. Based on the above constructions, we can now define two Büchi automata. They share the state space, initial states (4), and transition function (5)–(7). The automaton $\mathcal{A}_{WC}$ for weak conformance uses A_{WC} as set of final states, the one for strict conformance uses A_C.

Proposition 2. *Let T be a normalised gtt (Def. 6).*

A reactive system P weakly conforms to T iff all traces of p are accepted by $\mathcal{A}_{WC}(T)$, i.e., $P \subseteq \mathcal{L}(\mathcal{A}_{WC}(T))$.

A reactive system P strictly conforms to T iff all traces of p are accepted by $\mathcal{A}_C(T)$, i.e., $P \subseteq \mathcal{L}(\mathcal{A}_C(T))$.

Extension for back references. The automata construction described above does not cover gtts with back-references of the form $v[-k]$.

To handle back reference, the state space needs to be enriched by additional variables. For any input or output variable v for which a back-reference $v[-k]$ occurs in a table, the state variables $v_1, \ldots, v_k$ are added. Moreover, the following equivalencies are added to the state transition:

$$v'_1 = v \;\wedge\; \bigwedge_{i=2}^{k} v'_i = v_{i-1}$$

The expression $v[-c]$ then refers to the variable v_c for any constant $c \in \{1, ..., k\}$. The same construction is applied for each global variable (as global variables have the same value in all states).

Verifying system conformance. We provide two tools: the backend *geteta* for conformance verification of software for automated production systems, and *stvs* a graphical frontend for the creation of gtts and the inspection of counter examples. The implementation of *geteta* takes a gtt (encoded in XML), and a reactive system in Structured Text (ST), a textual programming language for automated production systems within the IEC standard 61131-3, and translate both to SMV file format. For the translation of ST source code, we use symbolic execution to compute the state relation of one system cycle into single static assignment form. For verification, we combine the model of the reactive system and the automaton representing the gtt into a product automaton, in which the inputs are chosen non-deterministically by the model checker. Links and further information are available on the companion website[5].

6 Experiment

We demonstrate the suitability of gtts for the specification and verification using a realistic example from the domain of automated production systems.

System under test. We consider an example system whose purpose is to watch over the input values and to raise a warning if they repeatedly exceed the previously learned range of allowed values. Such diagnosis functionality is common in safety-critical applications. More precisely, the system under test is the function block `MinMaxWarning` written in ST. A function block declares its input,

[5] Companion page: `https://formal.iti.kit.edu/ifm17/`

	Input			Output		🕒
#	`mode`	`learn`	`I`	`Q`	`W`	
1	`Active`	–	–	0	`true`	–
2	`Learn`	`true`	q	0	`false`	1
3	`Learn`	`true`	p	0	`false`	1
4	`Active`	–	$[p, q]$	$[p, q]$	`false`	*
5	`Active`	–	$>q$	q	`false`	5
6	`Active`	–	$<p$	p	`false`	5

(a)

	Input			Output		🕒
#	`mode`	`learn`	`I`	`Q`	`W`	
1	`Learn`	`true`	q	0	`true`	1
2	`Learn`	`true`	p	0	`true`	1
3	`Active`	–	$>q$	q	`false`	10
4	`Active`	–	$>q$	q	`true`	≥ 1
5	`Active`	–	$[p, q]$	$[p, q]$	`true`	5
6	`Active`	–	$[p, q]$	$[p, q]$	`false`	≥ 1

(b)

Fig. 7: Two gtts for the specification of the `MinMaxWarning`'s behaviour

output and local variables. In the case of `MinMaxWarning`, the input variables `mode`, `learn`, `I` and the output variables `Q`, `W` are declared. `MinMaxWarning` learns the typical input values and warns the caller for subsequent outliers.

`MinMaxWarning` operates in two modes, `Active` and `Learn`, as selected by the caller via `mode`. During the learning phase, the function block learns the minimum and maximum values of the input values (`I`), if the `learn` flag is activated. When switched into the active phase, the function block checks that the input value (`I`) stays within the previously learned interval. The output value `Q` is equal to `I` if `I` is within the learned interval; otherwise, the nearest value from the interval is returned. If the input value keeps being out of range for a specified number of cycles, then the function block raises an alarm via the variable `W`. The alarm is reset after a certain cool down time if the input value falls back into the learned interval. An unlearned function block always signals a warning.

Test tables. The required functionality is partially described by the two gtts shown in Fig. 7. These tables have two global integer variables p, q. As p should represent the minimum input value, resp. q the maximum, we specify the constraint $p \leq q$ in the model checker. The waiting time before an alarm is raised is fixed to ten cycles, and the cool-down time to five cycles.

The first gtt (Fig. 7a) specifies a behaviour without warning. In the beginning, it is checked that the unlearned system returns the default constants ($Q = 0$ and $W = \texttt{true}$; Row 1). This phase can be interrupted for switching into the learning mode (Rows 2 and 3). During learning, the system learns the minimum p and the maximum q input values. Subsequently, the system response is only allowed to be within this range. In Row 4, we test the non-warning case, in which only inputs between p and q are supplied. Rows 5 and 6 test for input values outside the range, and ensure that no warning is risen too early.

The second gtt (Fig. 7b) targets the case where warnings need to be given. We use the same initialisation, but require a warning due to a too high input (Rows 3 and 4). Rows 5 and 6 specify the cool-down within five cycles.

Verification. The verification system *geteta* that uses the construction from Sect. 5 and version 1.1.1 of the model-checker nuXmv [4], needs 0.53 CPU seconds for proving weak conformance of the first gtt and 0.63 CPU seconds for the second (median, $n = 6$). With the same setup, the verification of strict conformance takes 1.35 and 1.39 CPU seconds. Proving strict conformance requires an additional fairness condition to avoid infinite stuttering on the non-deterministic input variables. The experiments were run on a 3.20 GHz system with Intel Core i5-6500 and 16 GB RAM. The companion website provides the experiment files.

7 Related Work

A *Parnas table* is a tabular representation of a relation. Lorge et al. [10] use them in addition to first order logic for the specification of procedure contracts. In the *Software Cost Reduction* approach (SCR) [7], a collection of Parnas tables is used to specify a system's behaviour as a finite automaton. We follow a different approach with gtts: A system behaviour is specified as a sequence of admissible reactions to stimuli in the rows of a single table. Automata in SCR are deterministic while gtts are allowed to have non-deterministic transitions. Gtts allow the direct access to past values via back references or global variables; SCR requires an encoding of these values into the state. Both specification methods use tables as specification representation because of its accessibility for system engineers [7].

As an addition to the classical temporal specification languages CTL and LTL, Moszkowski [9], presents Interval Temporal Logic (ITL), which is ω-regular. ITL contains the chop-operator $(r_1; r_2)$ which – similar to our concept of rows – describes that there exists a point in time t s. t. until t the formula r_1 holds in all states and from t formula r_2 holds in all the following states. Obviously, we can encode a gtt T into an ITL by forming a disjunction of the generated normalised gtt of T (Def. 6). In general, the encoding results into an exponential blow-up. Armonie et al. [1] present *ForSpec Temporal Logic* (FTL) as an extension to LTL with logical and arithmetical operations and description of *regular events*. These regular events describes a finite regular language, similar to ITL and gtt. Additionally, FTL allows the composition with temporal connectives (a composition of gtts is possible on the automata level). Ljungkrantz et al. [8] propose ST-LTL, which enriches LTL with the arithmetical operators of Structured Text, syntactical abbreviations for specifying the rising or falling edges of variables, and access to previous variable value. To lower the obstacle for using formal specification in the development of critical software, like automated production systems, Dwyer et al. [6], Campos and Machado [3], and Bitsch [2] provide collections of specification patterns. The idea of specification patterns is that they cover the typical cases that arise from safety engineering. Additionally, their usage is simplified due to documentation and categorisations.

8 Conclusion

Gtts are a novel formal specification method for behavioural specifications of reactive systems. Their syntax is aligned with the concrete test tables and spreadsheet applications used in industry to ease the use of formal methods for software or mechanical engineers.

We have shown that it is possible to specify realistic software blocks from industry using gtts and verified them. Besides for verification at design time, gtts can also be used to generate checker code that monitors systems at runtime [5].

The concept of gtts is an important step towards the integration of formal methods into engineering automated production systems. Future work includes a user study on the accessibility of the features and an extension of the notation allowing the specification of software change during evolution.

References

1. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M., Zbar, Y.: The ForSpec temporal logic: A new temporal property-specification language. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2002)
2. Bitsch, F.: Safety patterns: The key to formal specification of safety requirements. In: Computer Safety, Reliability and Security (SAFECOMP). Springer (2001)
3. Campos, J.C., Machado, J.: Pattern-based analysis of automated production systems. IFAC Proceedings Volumes 42(4), 972–977 (2009)
4. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: Computer Aided Verification (CAV). pp. 334–342. LNCS 8559, Springer (2014)
5. Cha, S., Ulewicz, S., Vogel-Heuser, B., Weigl, A., Ulbrich, M., Beckert, B.: Generation of monitoring functions in production automation using test specifications. In: 15th IEEE International Conference on Industrial Informatics, INDIN 2017, Emden, Germany, July 24-26, 2017. IEEE (2017), to appear
6. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 1999 International Conference on Software Engineering (IEEE Cat. No.99CB37002). pp. 411–420 (May 1999)
7. Heitmeyer, C.L., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for constructing requirements specifications: The SCR toolset at the age of ten. International Journal of Computer Systems Science and Engineering 20(1), 19–35 (2005)
8. Ljungkrantz, O., Åkesson, K., Fabian, M., Yuan, C.: A formal specification language for plc-based control logic. In: 2010 8th IEEE International Conference on Industrial Informatics. pp. 1067–1072 (July 2010)
9. Moszkowski, B.: A temporal logic for multilevel reasoning about hardware. Computer 18(2), 10–19 (Feb 1985)
10. Parnas, D.L., Madey, J., Iglewski, M.: Precise documentation of well-structured programs. IEEE Transactions on Software Engineering 20(12), 948–976 (Dec 1994)
11. Rösch, S.: Model-based testing of fault scenarios in production automation. Phd thesis, Technische Universität München, München (2016)
12. Weigl, A., Wiebe, F., Ulbrich, M., Ulewicz, S., Cha, S., Kirsten, M., Beckert, B., Vogel-Heuser, B.: Generalized test tables: A powerful and intuitive specification language for reactive systems. In: 15th IEEE International Conference on Industrial Informatics, INDIN 2017, Emden, Germany, July 24-26, 2017. IEEE (2017), to appear