Runtime Verification of Generalized Test Tables*

Alexander Weigl[1][0000-0001-8446-4598], Mattias Ulbrich[1][0000-0002-2350-1831], Shmuel Tyszberowicz[2][0000-0003-4937-8138], and Jonas Klamroth[3]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
{ulbrich, weigl}@kit.edu
[2] Afeka Academic College of Engineering, Tel Aviv, Israel
tyshbe@tau.ac.il
[3] FZI Research Center For Information Technology, Karlsruhe, Germany
klamroth@fzi.de

Abstract. Runtime verification allows validation of systems during their operation by monitoring crucial system properties. It is common to generate monitors from temporal specifications formulated in languages like MTL or LTL. However, writing formal specifications might be an obstacle for practitioners. In this paper we present an approach and a tool for generating software monitors for reactive systems from a set of Generalized Test Tables (GTTs)—a table-based, user-friendly specification language specially designed for engineers. The tool is a valuable addition to the already existing static verifier for GTTs since assumptions made in specifications can thus be validated at runtime. Moreover, it makes software and specifications amenable for formal validation that cannot be verified statically. Moreover, the approach is particularly well-suited for the specification of workflows as a collection of tables since it supports dynamic, trigger-based spawning of monitors. The tool produces monitor code in C++ for tables provided in an existing table definition format. We show the usefulness of our approach using characteristic examples.

Keywords: Runtime Verification · Monitoring · Formal Specification

1 Introduction

Motivation. Safety-critical systems are usually validated using testing or static verification to ensure that they conform to their specification. Testing can usually only cover a small number of possible scenarios, and static verification is infeasible for many systems. One reason for that is that relevant information may not yet be available during static verification. Another potential problem is that the actual static verification engine may require too many resources (in terms of time, memory, or effort needed to come up with suitably strong environment models) to be feasible in practice. Runtime verification (or, synonymously, monitoring) [2],

* This work was funded by German Research Council (BE 2334/7-2, and UL 433/1-2), the state Baden-Wuerttemberg via CyberProtect project, and the KIT Alumni Visiting Grant.

on the other hand, does not suffer from these problems. Monitors are software systems, produced from specifications, that run in parallel to the production code and raise an alarm if the system runs (or potentially runs) into a bad state. They thus provide a sensible alternative to ensure the dependability and reliability of software systems at production time.

The generation of runtime monitors from temporal specifications is a well-studied problem (e.g., for Metric Temporal Logic [13], Bounded Linear Temporal Logic [10], HyperLTL [11]). However, the temporal logics used in these approaches were not originally designed with an engineer developing reactive systems in the field as the intended user. Beckert et al. [4] propose to use *Generalized Test Tables* (GTTs) as a practical temporal specification language for reactive systems (i.e., embedded systems driving cyber-physical systems (CPSs) through a periodically executed program that reads sensors and controls actuators). The specification language picks up specification concepts that practitioners in the engineering field are familiar with, and thus it is particularly user-friendly. By design, they are well-suited for specifying sequential processes.

We present the approach behind the monitor generator tool TTMonitor which generates efficient runtime monitors code in C++ from GTT specifications. It implements new features (which we describe in this paper) that make it particularly suitable for monitoring analysis for workflows where each process step is specified as an individual test table. Trigger-based mechanisms spawn monitors dynamically to allow this specification technique to work. The tool is part of our formal analysis toolbox for PLC verification code.

Generalized Test Tables. Generalized test tables are a table-based specification language for specifying reactive systems with a focus on practicability and comprehensibility. A reactive system, in the context of this paper, is a piece of software, which is periodically executed: reading input sensor signals and producing output actuator commands. A single repetition of the code is called an I/O cycle. The concept behind GTTs has been derived from concrete test tables—a description language used in industry to formulate test protocols which are written as sequences of concrete sensors and actuators signal values. A GTT is a generalization of a concrete test table in which concrete values (or durations) in table cells may be abstracted into constraints that can represent many values. Hence, a GTT covers not only a single (concrete) test case, but an entire *family of test cases*. Though a GTT thus covers a (possibly infinite) set of concrete behaviors, it keeps its exemplary character since all concrete behaviors are instances of the same ideal workflow description.

Usually a single GTT does not fully specify a system. It rather is a generalized example, covering a certain situation or scenario, and a comprehensive specification requires several tables. It is therefore important that the presented runtime verification approach can efficiently operate on sets of tables. GTTs are well-suited for an incremental specification process, where the specification grows over time as experience on the system behavior is gathered (be it during the design phase or later during testing, or even during production).

GTTs are stateful contract specifications that have assumptions (preconditions) and assertions (postconditions) in every I/O cycle. This distinction in the conditions allows us to distinguish a monitor terminating because of a failed assumption (uncovered case) from a monitor halting because of a failed assertion (specification violation). The contract design of GTTs allows us to distinguish four different modes of a monitor:

– *running*—system and monitor in operation, no violation;
– *extraneous*—the specification does not cover this concrete run;
– *failure*—the monitored run violates the specification;
– *finished*—the monitor has finished, the system continues, but cannot fail this specification any longer.

Contributions. Our contributions in this paper are:

(1) We present an approach by which GTT specifications can be verified dynamically using runtime monitors. It extends an earlier approach that was limited to fewer language constructs. In particular, the presented extensions include row groups, omega repetition, global parameters, and nondeterminism.
(2) We introduce the concept of *Dynamic Monitors*, by which monitors can be restarted, and can have multiple instances running at the same time.
(3) We present an approach for hierarchical combination of monitors. This approach allows adding and removing runtime monitors during operation. The hierarchical combination enables a flexible aggregation of monitor results using a variety of functions.
(4) We provide TTMonitor, a monitor-generation tool that creates monitors from GTT specifications. The C++ code of the monitor produced by TTMonitor is highly portable as it does not depend on libraries. The tool sources are publicly available under https://formal.iti.kit.edu/nfm2021.

This work extends and generalizes ideas of generating runtime monitors from GTTs presented by Cha et al. [7], where the approach was tailored to the specific needs of the domain of automated production systems and did not support row groups, omega repetition, global parameters, and nondeterminism.

Outline. In Sect. 2, we briefly explain the syntax and semantics of GTTs. The monitor generation and the supported features are presented in Sect. 3, followed by the application scenarios of these features in Sect. 4. In Sect. 5 we discuss some issues regarding our approach. Related work is presented in Sect. 6, and we conclude and present further potential optimizations in Sect. 7.

2 Generalized Test Tables

A GTT is a temporal specification in tabular form for a reactive system, such that every I/O cycle corresponds to one row in the table. In principle, the rows are executed from top to bottom, in their natural order, but the specification language possesses means to specify repeated lines or blocks.

#	ASSUME		ASSERT		⏲		
	T_c[°C]	T_b[°C]	P	B			
0	$(T_c - T_b) > d$	$[10, 60 + d]$	TRUE	FALSE	30s	[0, 1] (rows 0–1)	∞ (rows 0–4)
1	$> T_b, < T_c[-1]$	$> T_b[-1], < 60 + d$	TRUE	FALSE	—		
2	$\leq T_b$	$\leq 60 - d$	FALSE	TRUE	—	[0, 1] (rows 2–3)	
3	$\leq T_b$	$\leq 60 + d$	FALSE	TRUE	—		
4	—	$> 60 - d, \leq 60 + d$	FALSE	FALSE	$[1\text{min}, \text{—}]_p$		

Fig. 1. An example GTT for a solar thermal system

A full account on the syntax and semantics of GTTs can be found in [4, 8]; we will briefly summarize it in the following section. The introduction is guided by the concrete example in Fig. 1. The specified system is a solar thermal collector that uses energy of the sunlight to heat water. The system is equipped with an auxiliary gas burner which is activated when the solar energy is not sufficient. The GTT in Fig. 1 specifies how the system should control its water pump (P) and the gas burner (B) in response to the water temperature in the boiler (T_b) and in the collector (T_c).

2.1 Syntactical Elements

Every signal and every actuator variable has its column in a table. Since each table row describes a single step of the behavior, each cell constrains the value of a variable in the corresponding I/O cycle. The set of columns is divided into *assumption columns* and *assertion columns*. The former serve as *preconditions* for the cycle and the latter as *postconditions*, in the sense that all postconditions need to hold after the cycle if the preconditions were true before the cycle. Typically, the input variables of the system are assumption columns as these are generated by a physical environment and thus cannot be influenced by the system. The output variables are usually the assertion columns.

In contrast to concrete test tables, GTTs may contain constraints instead of concrete values in each cell. These constraints describe the set of admissible values for the corresponding cell. Thus GTTs are more expressive than concrete test tables. Syntactically, these constraints are a comma-separated list of Boolean constraints. GTTs support several abbreviations for the constraints. The contraints within a GTT may refer to global parameters which are placeholders for nondeterministically chosen, but fixed values. A system needs to conform to every possible instantiation of a global parameter (in this sense they are universally quantified over the entire GTT). The example has a global parameter d that is used to make the specification parametric in the temperature span. For example, the constraint "$[60 - d, 60 + d]$" (in Fig. 1) restricts the boiler temperature T_b to the depicted range and is an abbreviation for $T_b \geq 60 - d$, $T_b \leq 60 + d$ for any arbitrary d. A "don't-care" (—) constraint signals that the value may be chosen arbitrarily. References to values of past I/O cycles can be made using square brackets, e.g., "$< T_c[-1]$" specifies that the collector temperature is strictly decreasing compared to the last cycle. We denote global parameters with lowercase

letters to distinguish them from program variables, for which we use uppercase letters. To increase readability, we omit a cell constraint if it is identical to the constraint of the cell directly above.

To make GTTs more expressive than mere sequences of I/O cycles, an individual line or multiple lines (a block) may be annotated with a repetition scheme. The repetition of rows is defined in the special table column DURATION (🕒), and the repetition of blocks is marked by a vertical bar. For example, the duration specification of 30 seconds in the first row of Fig. 1 states that for the first 30 seconds the system should adhere to this row. The stated time spans are converted into equivalent numbers of I/O cycle iterations. For a cycle time of 10 seconds, the first row is repeated three times, and the last row for at least six times. For the specification of durations, the set of expressions is limited: the cells may contain concrete values, concrete intervals of natural numbers, "—" (nondeterministic, finite repetition) and "$—_\infty$" (infinite repetition). They specify the number of iterations that the respective row (or block) may be repeated.

2.2 Semantics: Table Conformance

The semantics of a GTT as a temporal specification is a set of admissible concrete behaviors. A generalized table T essentially corresponds to the set $B(T)$ of all concrete table expansions in which table rows (and blocks) have been rolled out in accordance to their duration annotations, and all table cells have been replaced with concrete values that satisfy the constraints.

We model a reactive system $S : (I \times \Sigma) \to (\Sigma \times O)$ as a function which takes a signal input in I and an internal state in Σ of the system, and computes the new state and the output in O. Thus, a reactive system is causal and deterministic. An (infinite) trace of S is a sequence $((i_1, o_1), (i_2, o_2), \ldots) \in (I \times O)^\omega$, such that the output values o_k are the result of the repeated application of function S to the input values i_k.

A trace tr of S conforms to T if there exists a concrete table c in the expansion set $B(T)$ such that the i-th element in tr satisfies all assumptions and all assertions in the i-th row of c. A trace violates T if there is no such satisfying witness c, but there exists a table $d \in B(T)$ whose assumptions are satisfied while at least one assertion fails. It is also possible that there is no concrete table for which all assumptions are satisfied by the trace. In this case the trace is not covered by the specification. A system S conforms to a GTT T if every trace of S conforms to T.

Beckert et al. [4] provide a formal definition of GTT conformance as a two-party game between the software system and its environment that also covers cases that we omitted here. In each turn, the environment of the system under test chooses the input values and the system responds with the computed output. A party loses if it emits a value that violates the current assumptions (for the environment) or assertions (for the system). This conformance condition can be encoded into an automaton which is described in the following section.

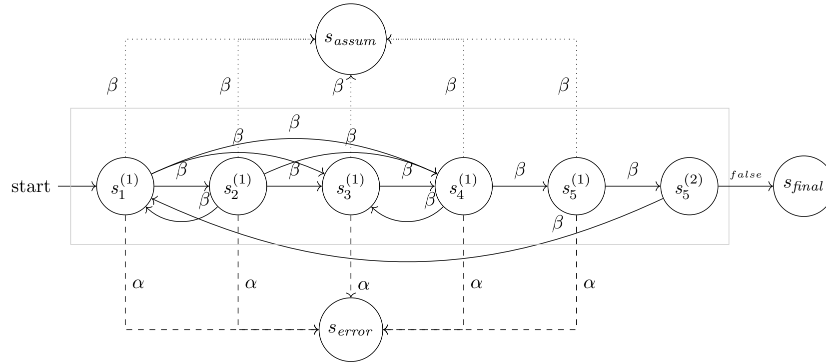

Fig. 2. Sketch of the automaton generated for the GTT of Fig. 1.

2.3 Automaton Generation

For (static) conformance verification, a GTT is translated into a nondeterministic automaton as described in the following such that when a trace tr of the system S is accepted by the automaton, it conforms to the GTT. Later this automaton is translated into a transition system encoded by Boolean formulas; see Sect. 3.

Automaton. Figure 2 sketches the generated automaton for the example shown in Fig. 1. A state $s_i^{(k)}$ represents the k-th iteration of the i-th row of the table and expresses that the i-th row is currently a possible step of the test protocol. Hence, if $s_i^{(k)}$ is active, the assumption and assertion of i-th table row define a valid turn for the challenger and the system in the conformance game. The state s_{error} represents a violation of a row assertion, and the state s_{assum} represents a violation of a row assumption. The state s_{final} represents the end of the table. If this state is reached, the system conforms to the GTT.

There are three kinds of transitions: An α edge from a state $s_i^{(k)}$ to the state s_{error} is triggered if the assumption of the i-th row is satisfied, but the assertion of the same row is violated. A β edge to the state s_{assum} is triggered if the assumption of the i-th row is violated. In this case it does not matter whether the assertion holds or not. A γ edge is taken when both the assumption and the assertion hold, leading to the next possible steps in the test protocol. Note that due to the strong-repeated row group in Fig. 1, the end-of-the-table and thus the final state s_{final} is not reachable. We model this situation by labeling the edge to s_{final} with the contradictory guard *false*.

Acceptance Condition. A state may have more than one successor state with transition γ which makes the automaton nondeterministic. The acceptance condition for a trace $tr \in S$ is that it must never reach a situation where *only* the error state s_{error} can be reached. This would imply that the assertion did not hold and there is no possible continuation.

More formally: Introducing a Boolean variable per state (and considering that multiple variables may be true at the same time due to nondeterminism), the condition that the system does not fail can be encoded as

$$s_{error} \rightarrow s_{final} \vee \bigvee_{i,k} s_i^{(k)} \quad . \tag{1}$$

which requires that at least one other state is possible (either a state $s_i^{(k)}$ inside the table or the final state s_{final}) whenever an error has been recognized.

3 Monitor Generation

We now explain how a runtime verification monitor is created from a GTT.

Monitor. A monitor is a software module that runs alongside the monitored reactive system and is executed at the end of each I/O cycle—after the output of the reactive system has been computed. It checks whether the trace (comprising of input, output, and internal state values) observed thus far (i.e., the current system state together with previously observed system states) satisfies the given specification. In the case of a monitor derived from a GTT T, the monitor can report one of four cases:

(1) The trace adheres to the specification, i.e., there is at least one sequence of rows in T such that all assumptions and assertions are satisfied (*running*).
(2) There is no sequence of rows in T such that all trace assumptions are satisfied (*extraneous input*), i.e., the specification does not cover the observed trace.
(3) There is a sequence of rows in T such that all assumptions of the trace are satisfied, but no sequence satisfies all assertions (*failure*).
(4) The trace adheres to the specification for a sequence of rows in T such that the end of T has been reached (*finished*).

The state *finished* is a special case of *running*, but it is particularly interesting since the monitor can idle as it can no longer change its state (in particular it can no longer fail the specification).

Definition 1 (Monitor). *Let $S\colon (I \times \Sigma) \rightarrow (\Sigma \times O)$ be a reactive system with input space I, output space O, and state space Σ. A monitor $\mathcal{M}$ with internal state space Σ_M is a reactive system $\mathcal{M}\colon (I \times O \times \Sigma) \times \Sigma_M \rightarrow \Sigma_M \times \{\text{RUN}, \text{EXTRA}, \text{FAIL}, \text{FIN}\}$ that takes as input the current input, output, and state values of S and returns as output a verdict. The verdict may be* RUN *(for* running*),* EXTRA *(for* extraneous*),* FAIL *(for* failure*), or* FIN *(for* finished*).*

From an Automaton to a Monitor. We use the automaton definition from Sect. 2.3 to build a monitor $\mathcal{M}(T)$ from a GTT T that realizes such an automaton. Since the automaton can be nondeterministic, $\mathcal{M}(T)$ needs to consider all possible runs, and hence has to maintain in its state space Σ_M a *set* of current automaton states

$\mathbf{S} \subseteq \{s_{error}, s_{assum}, s_{\mathit{final}}, \dots s_k^{(i)} \dots\}$. The automaton construction is suitable for GTTs which do not use global parameters. We derive the verdict $m_T(\mathbf{S})$ of $\mathcal{M}(T)$ from the current automaton states $\mathbf{S}$ as follows:

$$m_T(\mathbf{S}) := \begin{cases} \text{RUN} & : \mathbf{S} \cap \mathbf{S}_{row} \neq \emptyset \\ \text{EXTRA} & : \mathbf{S} = \{s_{assum}\} \ \lor \ \mathbf{S} = \emptyset \\ \text{FAIL} & : s_{error} \in \mathbf{S} \ \land \ \mathbf{S}_{row} \cap \mathbf{S} \neq \emptyset \\ \text{FIN} & : s_{\mathit{final}} \in \mathbf{S} \end{cases} \tag{2}$$

where $\mathbf{S}_{row} = \{s_{\mathit{final}}, \dots s_k^{(i)} \dots\}$ is the set of automaton states representing a table row or the end of the table. If the invariant (1) that encodes conformance to a GTT specification is violated by a system trace, then the verdict function (2) returns FAIL for that trace. In case that the invariant is satisfied for a (finite) trace, the verdict function can make a more fine-grained statement and return one of the three other verdicts, distinguishing between situations in which the specification does not cover the trace (EXTRA), the end of a specification has been reached (FIN), or the trace runs according to the specification (RUN). The monitor construction is designed to maintain conformance (Sect. 2.2).

Proposition 1 (Relation to Conformance). *Let S be a reactive system, T a GTT, and $\mathcal{M}(T)$ the generated monitor. S conforms to T if and only if $\mathcal{M}(T)$ does never produce the verdict* FAIL *in any I/O cycle step for any possible behavior of S.*

From [4], we know that S conforms to the GTT T if and only if the constructed automaton $\mathcal{A}_T$ (Sect. 2.3) never violates its invariant (1). As the generated monitor $\mathcal{M}(T)$ simulates the execution of the automaton $\mathcal{A}_T$ and the verdict FAIL corresponds to the violation of the invariant, the monitor $\mathcal{M}(T)$ will emit FAIL if and only if a system does not conform to the specification.

Challenges. One challenge of the monitor is to determine the instantiation of the global parameters from the observable system state. In contrast to static conformance verification, where a system needs to adhere to all global parameters' instantiations, the monitor supervises and assesses only the current trace, where the instantiations (along with the input and output values) of the global parameters are determined by the environment and by the system.

In the remainder of this section, we explain how we tackle the following challenges: handling global parameters, especially in combination with a nondeterministic row choice (Sect. 3.1); combining multiple GTTs into a single monitor (Sect. 3.2); restarting after bailing out (Sect. 3.3); and monitoring concurrent events and their effects (Sect. 3.3). These topics have solutions for static verification which cannot be transferred to the case of runtime verification.

3.1 Global Parameters and Nondeterminism

Global parameters within a GTT are considered universally quantified, which works out fine for static verification which can deal with uninterpreted symbols.

But during runtime monitoring, the monitor needs to determine the instantiation of global parameters by observing the current input-output trace. Hence, we need to decide *when* and *to which value* a global parameter is to be bound.

A global parameter may occur in a GTT at an arbitrary position. The first occurrence of a global parameter g could, for instance, be in the contraint $g \text{ DIV } 2 = \mathit{In}$ (where "In" is a program variable of the reactive system and DIV denotes integer division). In general, constraints could have zero, one, or multiple solutions for g, hence the value of g may be ambiguous. In our example, the constraints has two solutions for each given input value: $g = 2 * \mathit{In}$ and $g = 2 * \mathit{In} + 1$.

We tackle this problem by introducing a syntactical restriction: the first appearance of a global parameter g needs to be in a binding equation, where g stands alone on one side of the equation. In the above example, the user needs to rewrite the equation, and make the solution bound to g explicit, e.g., "$g = 2 * \mathit{In}$". In Sect. 5 we present two approaches to eliminate this syntactical restriction.

Since time constraints allow rows (and blocks) to be skipped, it cannot be guaranteed that the syntactically first occurrence of a global parameter is evaluated. However, it can be statically ensured that the first evaluation of a global parameter during a run is within a binding equation. Alternatively, this check can also be performed at runtime by the monitor.

Another challenge for global parameters is potential ambiguities induced by nondeterministic tables as multiple rows (automaton states) with different assignments for the same global parameter could be active at the same time and thus force a binding to different values. To resolve this challenge, we use a token-based evaluation of the automaton, where each token represents a possible run of the automaton. Each token carries an assignment of the global parameters together with its current automaton state. A token is always in a single state, and therefore the value bound to a global parameter is unambiguous. If there are multiple possible successor automaton states for a token, the token is duplicated and each copy obtains a different successor state. Because the automaton can be in multiple states, there might be multiple tokens. Furthermore, it is also possible that there are two tokens at the same automaton state with different assignments of the global parameters. Two tokens at the same state with identical assignments can be reduced to a single token as both behave identically.

3.2 Combined Monitors

Since GTTs are designed to describe a set of similar system behaviors, it is oftentimes not possible to describe the complete system behavior in one table. Hence, the specified behavior of a GTT is only a partial view of the complete system and a more comprehensive specification can be gained by using several GTTs to specify a system. To support such multi-table specifications, we need to support monitoring of several GTTs at the same time. We now show how the generated monitors of GTTs can be stitched together into one combined monitor.

A combined monitor $\mathcal{M}_{T_1,\ldots,T_n}$ is a reactive system which monitors a set $\{T_1, \ldots, T_n\}$ of GTTs by using the monitors $\mathcal{M}(T_i)$ for $1 \leq i \leq n$. The combina-

tion essentially runs the monitors in parallel, and the combined monitor state is the tuple of the states of the individual monitors: $\mathbf{S}_{1,\ldots,n} = (\mathbf{S}_1, \ldots, \mathbf{S}_n)$. The most relevant part of the combined monitor is the aggregation function $\overline{m}_{T_1,\ldots,T_n}(\mathbf{S}_{1,\ldots,n})$ which combines the verdicts $m_{T_i}(\mathbf{S}_i)$ of the sub-monitors $\mathcal{M}(T_i)$ (for $1 \leq i \leq n$) into a single verdict $\overline{m}_{T_1,\ldots,T_n}(\mathbf{S}_{1,\ldots,n}) = \text{agg}(m_{T_1}(\mathbf{S}_1), \ldots, m_{T_n}(\mathbf{S}_n))$. There are two canonical aggregation functions: $\text{agg}_\wedge$ and $\text{agg}_\vee$.

For the aggregation, we filter out the *bail-out* results from the sub-monitor verdicts ($\text{filter}_{\text{EXTRA}}(\cdot)$), and then the functions $\text{agg}_\wedge$ and $\text{agg}_\vee$ can be defined as the minimum and the maximum functions with respect to the order $\text{FAIL} < \text{RUN} < \text{FIN}$ on the results. Formally,

$$\text{agg}_\wedge(a_1, \ldots, a_n) = \min(\text{filter}_{\text{EXTRA}}(a_1, ..., a_n))$$
$$\text{agg}_\vee(a_1, \ldots, a_n) = \max(\text{filter}_{\text{EXTRA}}(a_1, ..., a_n))$$

with the special case that $\max(\emptyset) = \min(\emptyset) = \text{EXTRA}$. The aggregation functions $\text{agg}_\wedge$ and $\text{agg}_\vee$ correspond to the conjunction and disjunction in a three-valued logic with the given order.

The $\text{agg}_\wedge$ function corresponds to the conjunction of the monitors returning RUN if there is no sub-monitor that returns FAIL and at least one monitor is RUN. Similarly, $\text{agg}_\vee$ represents the disjunction returning RUN if at least one sub-monitor signals RUN. The value EXTRA expresses that a monitor has diverged, and this value is ignored in both aggregations.

In general, aggregation functions can be user-defined functions which are fine-tuned for the given tables and the automation system based on gained experience. For example, we allow complex aggregation functions which compute histograms of the given monitor results and aggregate their results based on a given threshold for each category (e.g., a combined monitor indicating RUN implies that at least a given percentage of the sub-monitors are fine (RUN) and the number of errors (FAIL) is below a threshold).

Note that combined monitors themselves can be subject to a combination, which allows the construction of sophisticated combinations. For example, imagine one GTT *emerg* which describes the emergency behavior of a system, and two mutually exclusive GTTs *man* and *auto* covering the manual and automatic operation modes. We can compose a comprehensive specification by logically combining the corresponding monitors for the GTTs, expressing that "*emerg* and *man* or *auto*" should always be satisfied. The corresponding combined monitor is $\mathcal{M}^\wedge(\mathcal{M}_{emerg}, \mathcal{M}^\vee(\mathcal{M}_{man}, \mathcal{M}_{auto}))$.

Performance Considerations. The monitor combination could have been implemented as a single product automaton construction combing all constraints of a set of GTTs. We decided against this product automaton construction, as the implementation effort would be higher and there are no clear performance benefits. States and tokens of and in the product automaton can be saved if the GTTs share initial rows, but this effect is negligible for long-running systems.

On the other hand, if a global parameters occur in the GTTs, the approach with several individual monitors (and, hence, a separate token for each GTT) is

more flexible as each monitor can consider a separate global parameter binding. Moreover, the approach of combining individual monitors allows the user to include handwritten monitors and supports dynamic monitors (Sect. 3.3).

3.3 Triggered Restarts and Dynamic Monitors

Triggered Restarts. If a monitor $\mathcal{M}$ runs into a situation where its monitored table does not cover the current run, i.e., the assumptions of all currently possible rows are violated, $\mathcal{M}$ does not need to be continued since it cannot recover from that state. Let us call such a monitor *diverging*. Consider a situation where a GTT describes the normal behavior of a system. If an (abnormal) emergency situation has been triggered for the system, the monitor diverges when the abnormal situation occurs since this behavior is not covered by the table. After recovery, it can no longer be used to monitor the system.

This problem was already identified in [7], and a solution which allows a simple and precise monitoring of event-triggered processes has been proposed there. An additional specification can be provided which triggers a restart of a monitor for a GTT. A restart trigger is a condition ϕ on the current state in the constraint language of the table cells. A monitor $\mathcal{M}$ restarts if it has diverged, i.e., once it results in a verdict of EXTRA, and the observed system trace meets ϕ. The restart resets the monitor to its initial state.

Dynamic Monitors. We generalize the idea of restarting further by allowing—beside a restarting condition—a starting condition ψ for a GTT T. Whenever ψ is met by the current system trace, a new instance of the monitor $\mathcal{M}_T$ is created and started. Note that, unlike the restart condition ϕ, the trigger ψ is not bound to another diverged monitor being stuck in the EXTRA state.

Dynamic monitors can be used to compose event-triggered specifications, where the expected system reaction to the event is described. For example, they can be used to specify the flow of work pieces and tracking the correct processing of each work piece in the software of production systems. Whenever a work piece appears at the beginning of the conveyor belt, this event triggers the spawning of a new monitor which monitors that particular work piece. With dynamic monitors, it is not necessary to globally formalize the entire work process chain, but rather one can focus locally on each process step for a single work piece.

A starting condition ψ is evaluated before the execution of the sub-monitors. Therefore, the newly created monitor instances start in the same I/O cycle in which ψ has been satisfied. At the end of a cycle, dynamic monitors which have diverged are discarded to avoid growing memory consumption. As a best practice to keep the memory consumption low, every dynamic monitor should eventually terminate, e.g., the end of specification is reachable.

The concept of dynamic monitors seems to subsume the concept of restarting monitors. But there is a subtle difference: with restarting, there always exists only one monitor instance which can be restarted after it has diverged, whereas a dynamic monitor can have multiple active instances at the same time.

#	ASSUME									ASSERT		⏲
	On BOOL	Off BOOL	Resume BOOL	Set BOOL	QuickDecel BOOL	QuickAccel BOOL	Accel BOOL	Brake BOOL	Speed FLOAT	CruiseSpeed FLOAT	CruiseState ENUM	
0	FALSE	FALSE	FALSE	—	—	—	—	—	—	0	OFF	≥ 0
1	TRUE			FALSE				—	$> SpeedMin$	$= Speed$	ON	1
2	—				FALSE	FALSE	FALSE	FALSE				≥ 0
	Restart: $CruiseState = Off$											

Fig. 3. Generalized Test Table for the cruise control system.

4 Application Scenarios

In this section we demonstrate the specification of reactive systems with GTTs and show how the TTMONITOR tool can generate monitors from the GTTs using the presented approach. The chosen examples demonstrate the benefits of the approach in different application contexts for reactive systems. Due to space restrictions, the table input files and monitors generated from them can be found on the companion website.[4]

4.1 Cruise Control System

A CCS is a driver assistance system found in cars that accurately maintains the speed set by the driver by controlling the throttle-accelerator pedal linkage without driver intervention. If the driver uses the accelerator or the brake pedals, the system releases its control over the velocity. CCSs have already been formally studied [1, 12, 15]. We follow the specification and Esterel implementation in [19]. There are nine input parameters to the system: *On*, *Off*, *Resume*, *Set*, *Speed*, *QuickDeccel*, *QuickAccel*, *Accel*, and *Decel*. The CCS returns three output values: the current operation mode (on, off, stand-by, disabled), the current target speed, and the value of the throttle. The GTT in Fig. 3 describes those scenarios in which the CCS is switched on and should maintain the current speed until either the brake or the accelerator pedal are pressed. This monitor becomes obsolete (i.e., it diverges) if the CCS is switched off, and restarts once the system is switched on.

4.2 Linear Regression

Here we demonstrate the feature of global parameter binding. The *Linear Regression* function block implements a commonly needed functionality for implementing CPSs, namely the calibration of sensor values. The evaluated software module origins from [18], where it is used to demonstrate static verification of GTTs using model checking. The state space of this function block, which uses floating point arithmetic, is relatively large and its state transition function relatively complex, which limits the applicability of static verification tools.

Linear Regression maps actual sensor values to a defined range of calibrated values. This mapping is internally represented as a linear interpolation curve

[4] https://formal.iti.kit.edu/nfm2021/

#	ASSUME				ASSERT	◷	
	TPy	TPSet	Mode	X	Y		
0	—	—	Op	—	0	—	
1	—	0	Teach	—	0	$[1, to]$	
2	y_1	1	Teach	x_1	0	1	
3	—	0	Teach	—	0	$[1, to]$	ω
4	y_2	1	Teach	$x_2, \neq x_1$	0	1	
5	—	—	Teach	—	0	1	
6	—	—	Op	—	$= y_1 + {}^{y_2 - y_1}/_{x_2 - x_1}(X - x_1)$	—	

Fig. 4. Generalized test table of a system which maps sensor values to their physical representation by a taught linear curve. Originally presented in [18].

#	ASSUME					ASSERT						◷
	CranePos ENUM	CraneWP BOOL	WP@Magazin ENUM	StampState ENUM	WP@Conveyor BOOL	Crane ENUM	Vaccum BOOL	Stamp BOOL	Conv.Belt BOOL	Pusher1 BOOL	Pusher2 BOOL	
0	MAGAZINE	FALSE	METAL_READY	—	—	STOP	—	—	—	—	—	1
1			—			PICKUP	TRUE					$[1, T_1]$
2		TRUE	EMPTY	FREE		STOP						1
3	—		—			MOVE_CW						$[1, T_2]$
4	STAMP					STOP						10
5						RELEASE						5
6						—	FALSE					1
7		FALSE		OCCUPIED				TRUE				1
8	—	—			—	FALSE		FALSE		—		—
9	STAMP	FALSE		READY		PICKUP	TRUE					$[1, T_3]$
10		TRUE		FREE		MOVE_CCW	TRUE					—
11	CONVEYOR					STOP	TRUE					—
12						RELEASE						5
13	—	FALSE				—	FALSE		TRUE	FALSE		1
14						—						T_4
15											TRUE	5

Fig. 5. A GTT for describing the material flow in the PPU plant, which is instantiated when a new work piece appears at the magazine ($WP@Magazin \neq$ `EMPTY`).

whose parameters are learned during operation. To this end, the function block can be operated in two modes: the calibration mode ("Teach") and the operation mode ("Op"). After learning, the block performs the linear interpolation in the operation mode, mapping the incoming sensor values to values according to the calibrated curve. The system receives the selected mode (*Mode*) and the sensor value input (X), and two additional inputs needed for the calibration: *TPy* for the reference value and *TPSet* to trigger teaching. The system has only a single output Y, which is zero during teaching or for improper reference points. If the reference points are proper values, then the output Y is defined by the linear curve at position X. This behavior is described by the GTT in Fig. 4.

4.3 Conveyor Belt Process

In this scenario we demonstrate the features of dynamic monitors by specifying the material flow inside an automated manufacturing plant. The example is based on the Pick-and-Place-Unit (PPU) developed at the TU Munich [17]. The PPU was developed to demonstrate methods to manage the evolution of long-running hard- and software. More than 20 scenarios have been designed, and they demonstrate a variety of evolution scenarios typical for an automated production system. We use one scenario (scenario number 13) in which the PPU picks up work pieces from a deposit with a crane. If a work piece is metallic,

it is transported to the stamp to be engraved. Then the engraved work piece is picked up again and is moved to the conveyor belt, where the work pieces are finally sorted on different ramps. Non-metallic work pieces are not engraved, and are directly moved to the conveyor belt. For optimization, the crane moves non-metallic pieces to the conveyor belt while a metallic piece is being stamped.

Due to the parallel processing (stamping, transporting, and sorting) within the plant, a global specification of the input and output variables is hard to achieve. Instead, we can describe the plant by following the work pieces individually.

Note that the assumptions in Fig. 5 encode the expected physical behavior of the environment. If they are violated, e.g., if a work piece is not detected in time, the monitor raises the signal (*bail-out*), and this should be interpreted as a flag for an error in the environment. One possibility to deal with this is to deliver more explanations why a monitor diverges, as discussed in Sect. 5.

5 Discussion: Generalizations

Counting Repetitions. The automaton for constructing the monitors is generated from a normalized (unrolled) test table. Therefore, a row with a duration $[m, n]$ ends up in an automaton with $n \cdot d$ states, where d denotes the number of unrolled overlying row groups. Basing the evaluation of automata on tokens would allow us to use integer counters in the tokens for counting the repetition of rows and row groups, thus reducing number of states in the automaton and the code and data size of the monitor. Moreover, we can get rid of the restriction of nonrigid duration constraints and allow the use of state or input variables in the duration column. Their use also enables using a clock time instead of I/O cycle numbers and makes the generated monitors applicable for interactive systems.

Symbolical Representation of Global Parameters. In Sect. 3.1 we restricted the first occurrence of global parameters to a form which describes an unambiguous value to bind. This restriction could be lifted, with a negative impact on the performance, by using a symbolic representation, e.g., a BDD or a CNF formula. Instead of a concrete value, a token would hold a symbolic representation for each global parameter. The constraints of a global parameter in the monitored table cells are added to the token's symbolic representation and limit the value range of the global parameter. The symbolic representation must be satisfiable (describing at least one possible value of the global parameter) during monitoring. Moreover, every monitored constraint needs to be checked symbolically.

A simpler solution can be the use of multiple tokens. Instead of forcing the user to decide on one solution, we create a token for each adhering binding of the global parameters of the equation. Back to our example of a quadratic equation, we know there are at most two possible solutions, thus we will create zero to two tokens with different assignments. Note, this solution is only possible if the number of solutions is limited and rather small.

Assumptions as Assertions on the Environment. The presented approach reports violated assumptions as extraneous situations and bails out without reporting an

error. There are situations in which an assumption violation is an indicator for a serious error occurring in the environment, not only a situation not covered by the specification. An error should be reported. We observed this in Sect. 4.3, where a disappearing work piece on a conveyor belt is an unexpected event and indicates either a broken sensor or a plant standstill. It needs to be distinguished from a violated assumption for a work piece not covered by the specification. To this end, the specification mechanism can be extended to support more assertion levels than the two presented in this paper.

6 Related Work

The generation of runtime monitors from formal specification is well-studied; see, e.g., [6]. The most closely related topics are the monitoring of reactive systems and the monitoring of engineer-friendly specification languages. Two prominent examples for the latter are LoLA [9] and Copilot [14]. Both are stream-based languages which allow for and claim to be more user-friendly than the underlying temporal logics. Copilot focuses more on the real-time aspect of the created monitors while LoLA can additionally provide statistical measurements for system profiling (rather than pure Boolean verdicts). Both monitoring approaches do not explicitly support dynamic spawning or restarting of monitors.

Bloem et al. [5] propose the construction of *shields*—runtime monitors with the ability to alter the output of the monitored system when a violation is detected. Their monitor construction therefore also requires the synthesis of a reactive system, which computes the alternative correct outputs. They introduce a new notion of k-stabilization which captures the idea that a system can alter the output of a system for k steps, to avoid the violation of given properties.

Bauer et al. [3] present a framework which allows to identify the faulty sub-component in a reactive system (in addition to monitoring). This is achieved by first monitoring components locally (according to a Timed LTL specification [16]) and then using first order logic to describe the overall system behavior. Thus, it is possible to detect which components may be responsible for an observed error.

7 Conclusion

In this paper we presented an approach for generating runtime monitors from GTTs, which are a table-based specification language for the behavior of reactive systems. In contrast to earlier work, the presented approach can deal with nondeterminism and global parameters in tables. Moreover, we introduced the concept of dynamic monitors which are created/launched at runtime whenever a specified trigger event occurs. They make possible a local specification of parallel and multi-step processes. We show the applicability of the monitoring approach on concrete examples from the domains of automated production systems and embedded controllers. The approach has been implemented in TTMonitor, an open-source tool which generates monitor code in C++ from GTTs specifications.

GTTs have two distinct kinds of constraints: assumptions and assertions. Depending on the type of constraints that fails, a failing trace is reported to either diverge (i.e., the specification does not cover it) or to reveal a flaw in the implementation. This principle can be refined further in future work that will allow the introduction of several different constraint categories. This will allow the monitor to elaborate the nature of failures even further, as feedback to the engineer. For instance, for each hardware component a category could be introduced for the assumptions on its physical response behavior. If a failure is reported in this category, this will directly indicate that the hardware component has failed. Analogously, also for the assertions on software components.

We plan to evaluate our monitor generation approach and the example monitors in simulation and in (real-time) operation in their environment.

References

[1] Jagannath Aghav and Ashwin Tumma. Esterel implementation and validation of cruise controller. In *Computer Science, Engineering And Applications (CCSEA)*, pages 128–141, 2011. https://doi.org/10.5121/csit.2011.1214.

[2] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018. https://doi.org/10.1007/978-3-319-75632-5_1.

[3] Andreas Bauer, Martin Leucker, and Christian Schallhart. Model-based runtime analysis of distributed reactive systems. In *Australian Software Engineering Conference (ASWEC)*, pages 243–252, 2006. https://doi.org/10.1109/ASWEC.2006.36.

[4] Bernhard Beckert, Suhyun Cha, Mattias Ulbrich, Birgit Vogel-Heuser, and Alexander Weigl. Generalised test tables: A practical specification language for reactive systems. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods (IFM)*, volume 10510 of *LNCS*, pages 129–144. Springer, 2017. ISBN 978-3-319-66844-4. https://doi.org/10.1007/978-3-319-66845-1_9.

[5] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield Synthesis:: Runtime Enforcement for Reactive Systems. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *LNCS*, pages 533–548. Springer, 2015. https://doi.org/10.1007/978-3-662-46681-0_51.

[6] Ian Cassar, Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. A survey of runtime monitoring instrumentation techniques. In Adrian Francalanza and Gordon J. Pace, editors, *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017*, volume 254 of *EPTCS*, pages 15–28, 2017. https://doi.org/10.4204/EPTCS.254.2. URL `https://doi.org/10.4204/EPTCS.254.2`.

[7] Suhyun Cha, Sebastian Ulewicz, Birgit Vogel-Heuser, Alexander Weigl, Mattias Ulbrich, and Bernhard Beckert. Generation of monitoring functions in production automation using test specifications. In *International Conference on Industrial Informatics (INDIN)*, pages 339–344. IEEE, 2017. ISBN 978-1-5386-0837-1. https://doi.org/10.1109/INDIN.2017.8104795.

[8] 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. *Automatisierungstechnik*, 66 (10):834–848, 2018. https://doi.org/10.1515/auto-2018-0028.

[9] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar

Manna. LOLA: runtime monitoring of synchronous systems. In *Temporal Representation and Reasoning (TIME)*, pages 166–174. IEEE, 2005. https://doi.org/10.1109/TIME.2005.26.

[10] Bernd Finkbeiner and Lars Kuhtz. Monitor circuits for LTL with bounded and unbounded future. In Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification (RV) 2009*, volume 5779 of *LNCS*, pages 60–75. Springer, 2009. https://doi.org/10.1007/978-3-642-04694-0_5.

[11] Bernd Finkbeiner, Christopher Hahn, Marvin Stenger, and Leander Tentrup. Monitoring hyperproperties. *Formal Methods in System Design*, 54(3): 336–363, 2019. https://doi.org/10.1007/s10703-019-00334-z.

[12] Constance L. Heitmeyer, James Kirby, and Bruce G. Labaw. Tools for formal specification, verification, and validation of requirements. In *Conference on Computer Assurance (COMPASS)*, pages 35–47, 2009. https://doi.org/10.1109/CMPASS.1997.613206.

[13] Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online Monitoring of Metric Temporal Logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification (RV)*, volume 8734 of *LNCS*, pages 178–192. Springer, 2014.

[14] Ivan Perez, Frank Dedden, and Alwyn Goodloe. Copilot 3. Technical report, Technical Report NASA/TM-2020-220587, National Aeronautics and Space Administration, 2020.

[15] Sorina-Nicoleta Predut, Florentin Ipate, Marian Gheorghe, and Felician Campean. Formal modelling of cruise control system using event-b and rodin platform. In *High Performance Computing and Communications (HPCC)*, pages 1541–1546. IEEE, 2018. https://doi.org/10.1109/HPCC/SmartCity/DSS.2018.00253.

[16] Jean-François Raskin. *Logics, automata and classical theories for deciding real time*. PhD thesis, Facultés universitaires Notre-Dame de la Paix, Namur, 1999.

[17] Birgit Vogel-Heuser, Christoph Legat, Jens Folmer, and Stefan Feldmann. Researching evolution in industrial plant automation: Scenarios and documentation of the pick and place unit. Technical report, Institute of Automation and Information Systems, Technische Universität München, 2014.

[18] Alexander Weigl, Franziska Wiebe, Mattias Ulbrich, Sebastian Ulewicz, Suhyun Cha, Michael Kirsten, Bernhard Beckert, and Birgit Vogel-Heuser. Generalized test tables: A powerful and intuitive specification language for reactive systems. In *Industrial Informatics, (INDIN)*, pages 875–882. IEEE, 2017. https://doi.org/10.1109/INDIN.2017.8104887.

[19] Mark Yep and Sylvain Bechet. Esterel cruise controller. Website, `https://github.com/ooksei/esterel-cruise-controller/`, 2018. access: 2019-10-16.